Computational Geometry: Lecture 3

Don Sheehy

January 25, 2010

1 Last Time

Last time, we talked about the $2D$ convex hull problem. As a group we came up with an algorithm that looked mysteriously like selection sort with the usual comparison operation replaced by a slope or angle comparison. To do such a sort, we don't need to compute angles, just compare them.

For this we introduce the counter-clockwise (CCW) predicate. Given three points a, b, c in the plane, $\text{CCW}(a, b, c)$ is defined as follows.

$$\text{CCW}(a,b,c) = \begin{cases} 1 & \text{if } \triangle(a,b,c) \text{ is oriented counter-clockwise.} \\ 0 & \text{if } a,b,c \text{ are collinear.} \\ -1 & \text{if } \triangle(a,b,c) \text{ is oriented clockwise.} \end{cases}$$

We'll take a closer look at what this means mathematically by the end of the lecture.

2 What if sorting were free?

The rough algorithm we came up with last time looked like sorting but took $O(n^2)$ time. Let's look at the extent to which we can pack the convex hull computation into a generic sorting algorithm. As before, let v_0 be the left-most input vertex. We can define a generic point comparison function as $\text{COMPARE}(v_i, v_j) = \text{CCW}(v_0, v_i, v_j)$.

Suppose the input is sorted using this comparison function and placed on a stack. If we also have a stack to hold the output, our goal is to have the convex hull vertices ordered on the output stack. Using only basic stack operations (push, pop, peek) and the CCW predicate, we will maintain the invariant that the output stack always contains the convex hull of all the vertices that have been popped off the input stack.

This gives us a natural algorithm known as the Graham Scan.

Algorithm 1 GRAHAM SCAN

Input: $S \in \mathbb{R}^2 : |S| = n$
Find the leftmost point in S and label it v_0.
Sort the points by their angle relative to v_0 and label them $v_1, \ldots, v_{n-1}$
Push the sorted list of vertices onto the Input stack.
Output.push(Input.pop)
Output.push(Input.pop)
Output.push(Input.pop)
while the Input stack is not empty **do**
  Let x = Input.pop
  while CCW($Output[1], Output[0], x) < 1$ **do**
    Output.pop
  end while
  Output.push
end while

3 How fast is Graham Scan?

Ignore the cost to do the original sort. A very naïve analysis would look at the two nested loops and observe that in the worst case, each loop could iterate $O(n)$ times. This leads to a $O(n^2)$ algorithm. But clearly, a better analysis is possible. One idea proposed is to count operations from the perspective of the points. Each point takes part in at most 3 stack operations so the total number of stack operations is $O(n)$. Since there is at least 1 stack operation per inner loop, the total running time is $O(n)$.

If sorting is free, then computing the $2D$convex hull can be done in linear time. Given a $O(n \log n)$-time comparison-based sorting algorithm, the total running time is $O(n \log n)$.

4 Lower Bounds

Up until this point, we have been dancing around this comparison between $2D$ convex hull and sorting. Let's do another turn in this dance and prove that computing convex hull is at least as hard as sorting.

We will give a linear time reduction from sorting to convex hull. Let $a_1, \ldots, a_n \in R$ be the input to a sorting algorithm. Let $f : \mathbb{R} \to \mathbb{R}^2$ be defined as $f(x) = (x, x^2)$. This function takes a collection of points on the real line and maps them onto a parabola. These points are in convex position and their order on the convex hull is the ordering on the line up to a cyclic rotation. The cyclic rotation back to linear ordering can be done in linear time.

If we are restricted to CCW as our only arithmetic operation on points, then the $\Omega(n \log n)$ sorting lower bound applies.

5 A closer look at CCW

Now, we will talk about how to compute CCW and we will find a tighter connection between CCW and comparison.

Let a, b, and c be three points in the plane. To make things simpler, let's assume that a is at the origin, i.e., $a = (0, 0)$. Then, computing CCW is just the dot product of c (treated as a vector) and the vector n normal to b (again, treated as a vector). In this case,

$$\begin{aligned}
\text{CCW}(a, b, c) &= \text{sign}(c \cdot n) && (1)\\
&= \text{sign}\left(\begin{bmatrix} c_x \\ c_y \end{bmatrix} \cdot \begin{bmatrix} -b_y \\ b_x \end{bmatrix}\right) && (2)\\
&= \text{sign}\left(b_x c_y - b_y c_x\right) && (3)\\
&= \text{sign}\det\begin{bmatrix} b_x & c_x \\ b_y & c_y \end{bmatrix} && (4)
\end{aligned}$$

Now, we need to make this work when a is not at the origin. We can just add a translation. The new CCW function can now be written as follows.

$$\det\begin{bmatrix} b_x - a_x & c_x - a_x \\ b_y - a_y & c_y - a_y \end{bmatrix} = \det\begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{bmatrix} \quad (5)$$

It is very important to use these nice matrix representations rather than trying to roll your own code haphazardly. These operations can be very unstable numerically (think about what happens when the points are *almost* collinear). This is a major source of headaches for anyone writing geometric software for the first time.