Advanced Algorithms

Class Notes for Monday, November 12, 2012

Bernard Moret

1 Divide-and-Conquer: Convex Hulls

Last lecture focused on sweep-based algorithms; this time, we will solve the same problem through both sweep-based and D&C algorithms. This problem is the problem of computing the convex hull of a set of points in the plane.

Definition 1. *The convex hull of a set S of points is a subset $C \subseteq S$ of minimum cardinality such that the polyhedron (polygon in 2D) defined by C encloses all of S.*

There is no need to specify that the polyhedron must be convex: the minimality condition suffices to enforce convexity.

So how do we compute a convex hull in 2D? The problem is rather simple and amenable to dozens of different approaches, so we will start by a simple sweep approach. Whereas we used sorting along an axis for past sweep-based algorithms, this time we shall used sorting by angle. We start by identifying in linear time the point of lowest ordinate (and, if a tiebreaker proves necessary, of lowest abscissa). As an extremal point, this point, call it P_0, must be part of the convex hull; moreover, we know that the convex hull lies entirely above it—within the halfplane determined by a horizontal line passing through P_0. So we will sort the remaining points in terms of the angle formed by a line passing through each of them and P_0 with respect to the horizontal axis passing through P_0, moving counterclockwise. In case multiple points lie on the same line passing through P_0, we break the tie using their distance to P_0, from nearest to farthest. This sorting pass will take $O(|S| \log |S|)$ time. In a second pass, we will add each successive point to the hull under construction: because the last point added is always an extremal point (with respect to the hull built so far), it always belongs to the hull. However, adding a new point may invalidate previously added points. Thus, as we add point P_i, we must check that P_{i-1} still belongs to the new, larger hull. We can simply verify that the internal angle formed by points P_{i-2}, P_{i-1}, and P_i is less than 180 degrees or, equivalently, that point P_{i-1} lies on the opposite side from P_0 of the line defined by P_{i-2} and P_i. (The latter is much preferable from a computational standpoint, as it only involves the four arithmetic operations, avoiding any trigonometric functions.) Crucially, if P_{i-1} is eliminated in this process, we must now check P_{i-2} (using P_i and P_{i-3}), and so on. In fact, the addition of a single point P_i could eliminate every point in the hull at the time except for P_0 and one other: just picture that the resulting hull is a triangle with P_0, P_1 and P_i as its three vertices, but that the previous hull had a long convex curve from P_1 to P_{i-1}, with small angles for all steps, all of it contained within that triangle. This second pass adds each point to the hull under construction exactly once, and removes each point from the hull at most once, taking constant time for each addition and for each removal. Thus the second pass runs in $\Theta(|S|)$ time, that is, in linear time. This second pass, on its own, is known as the *Graham scan*; it is a particularly fast way to construct a convex hull from a sorted list of points. Overall, then, our sweep-based algorithm is dominated by the sorting pass and takes $O(|S|\log|S|)$ time.

Can we run faster? If the convex hull ends up having k points, then we must run in $\Omega(n+k\log k)$ time, since we must read all n input points and since, by returning a polygon for the hull, we effectively sort the k vertices of the hull. Thus, if k is $\Theta(n)$, we cannot do any better; however, in many applications of convex hulls, k can be expected to be rather small, in which case there is plenty of room for improvement. We are going to get one such improvement through divide-and-conquer. Note, however, that we cannot use "geometric" divide-and-conquer (that is, partitioning through the use of vertical halfplanes), since we saw that this approach ends up sorting the points along the x axis. Instead, we will use what might be termed "set-based" divide-and-conquer: we will assume that the n points are given to us in some arbitrary order in an array indexed from 0 to $n-1$ and our division will simply compute indices in the array, which takes constant time at each step, as opposed to linear time for the geometric version. In standard D&C style, we carry out the division recursively until the sets have reached a size small enough for the problem to be trivial. In our case, any set of 3 points or less is trivial: we simply have to test, for sets of 2 points, whether the two points coincide, and, for sets of 3 points, whether the three points are collinear, but otherwise we just return the list of the three (or fewer) points. Thus, as is again typical of D&C, the real work is done during the merging. For us, the merging problem is simply: given two convex polygons (which could overlap or intersect), construct the convex hull of their union.

Now, since the two hulls are given as two convex polygons, we know that they are effectively already sorted. But we have just seen an algorithm that produces the convex hull of a sorted set of points in linear time (the Graham scan)! Thus we need only take the two convex polygons and, from them, produce a single sorted list of vertices for use by the Graham scan. This is quite simple: first, we identify the anchor for the ordering used in the Graham scan, the point (among the two hulls) of lowest ordinate and, in case of ties, of lowest abscissa. This step takes linear time. Assume that the anchor belongs to the first hull; then this first polygon is already in sorted order by angle with respect to the anchor—we need only ensure that we move in the correct counterclockwise order from the anchor. For the second polygon, however, the situation is different: the anchor is below it and (in general) does not touch it. In that case, traversing the perimeter of the polygon will see a reversal in the order of change of the angle, a reversal that occurs at the tangency points (the vertices on lines from the anchor that just bracket the polygon in a fast-enlarging cone). This is the same phenomenon we observed when looking at the intersection of two convex polygons: as we move along the perimeter, we go from increasing the abscissa to decreasing it to increasing it again, and so on. It suffices to find the two extremal points (the points defining with the anchor the lines of smallest and of largest slope) and cut the perimeter of the polygon at these two points to obtain two pieces of the perimeter, each of which is in sorted order with respect to the angle. Now we merge these two sorted lists, and merge the resulted list with the perimeter of the first polygon, to obtain, in linear time, a sorted list (by angle of the line passing through the anchor, with respect to the horizontal) of all vertices of the two hulls. This is precisely the data we need to run the Graham scan, which, also in linear time, returns to us the hull of the the union of the two sets of points. Thus the recurrence describing the running time of this D&C algorithm is

$$T(n) = \Theta(1) + 2T(n/2) + \Theta(k_1 + k_2)$$

where k_1 and k_2 are the numbers of vertices on the first and second hulls and the constant term is the cost of division. Writing $k = k_1 + k_2$ and dropping the constant term (covered by the other terms), we get

$$T(n) = 2T(n/2) + \Theta(k)$$

If k is $\Theta(n)$, we get the recursion for mergesort, with solution in $\Theta(n \log n)$. However, if k is $o(n)$ *at every recursive step,* then the solution is in $\Theta(n)$, which is clearly optimal.

One would be tempted to conclude that we have obtained the best possible convex hull algorithm, but in fact the required condition is unlikely to hold at every possible step. So, what else can be done? If we do not sort the output (i.e., if we return a set of points instead of a polygon), then our previous lower bound does not hold and it should become possible to run yet faster, perhaps in linear time. No such algorithm is yet known, but an algorithm exists that runs in $O(n \log k)$, which is pretty good—and optimal if k is a constant. We shall not examine this particular algorithm, but look at a pair of strategies (one of them is not really an algorithm) to compute a 2D convex hull in a very physical way. In both cases, we will need a sheet of plywood, a ruler and some chalk, a hammer and a box of nails. We will use these to plot the positions of the points on the sheet of plywood, then drive a nail, leaving some part sticking out, at each point position. Now, for the first, and most efficient strategy, we need a large, strong rubber band; we stretch it to make it fit safely in a large circle around the group of nails, and then let go. Kazoom! the rubber band tightens around the nails and describes the convex hull, which is, as noted at the beginning, a minimum-energy configuration. Regrettably, this does not appear easy to translate into a computational algorithm… So our second approach is to buy a spool of thread. We tie the thread to a nail that is sure to be on the convex hull, for instance the lowest nail (minimum ordinate—if there are ties, we take the leftmost of these); we then stretch the thread horizontally to the right—all of the nails are above the line this defined by the thread, and start rotating the thread counterclockwise. Eventually, the thread will hit a nail and, as we continue the rotation, will start rotating around that nail rather than around the one it was tied to. As we keep going, the thread hits successive nails, each of which is a point of the convex hull, before we complete a 360-degree rotation and the thread its the nail to which we had attached it. This approach is known as *package-wrapping* for rather obvious reasons (and because it proves rather more useful in 3D than in 2D). Unlike the rubber band approach, this one is easy to program. We have a loop that, until a 360 turn has been completed, looks for the remaining vertex that forms a minimum angle with the last vertex hit by the thread and the line defined by that last vertex and it predecessor on the hull. Since each search for a minimum takes linear time, and we must conduct such a search for each vertex on the final hull, the running time is $O(k \cdot n)$. When k is in $\Theta(n)$, this algorithm performs very poorly indeed (quadratic time!), but when k is a constant, the algorithm runs in optimal linear time. (Note that the package-wrapping algorithm sorts the output, as it returns vertices on the hull in perimeter ordering. It does so quite inefficiently, however: if we have $k = n$, say because all points lie on one circle, then package-wrapping reduces to selection sort: find the next smallest item and append it to the sorted list.)

There are several observations we can make from all of the above. First, D&C can be done both in geometric fashion, thereby sorting along an axis, or through simple index computations. The former incurs the overhead of sorting, while the latter does not, but must content with poorly defined, overlapping configurations. Second, it is clear that (2D) convex hulls and sorting are intimately related, at least as long as we expect the hull to be returned in the form of a polygon. The connection is very tight: given a collection of numbers to be sorted, say x_1, x_2, $\ldots$, x_n, we can use a convex hull algorithm to do the sorting as follows: for each value x_i, create the point on the plane $P_i = (x_i, x_i^2)$, then feed the n points to a convex hull algorithm. Because the n points lie on a parabola, they all belong to the convex hull; because a parabola in the first quadrant is a monotonically increasing curve, the perimeter ordering of the points on the hull is a sorted ordering of the points (once we have located the smallest value); and since it takes only linear time to transform the sorting problem into the convex hull problem and back, if we could always compute the convex hull in $o(n \log n)$ time, then we could sort in $o(n \log n)$ time, which we know cannot be done in the "comparison" model of computation (models in which, in effect, bitwise or bytewise computations are not allowed). Finally, we see that the same problem can be analyzed in different ways depending on the choice of parameters: without the introduction of k, the size of the *output* of the algorithm, we could not have differentiated among the various algorithms we discussed (except for package-wrapping). In analyzing algorithms that come close to perfect efficiency, it is often useful to introduce one or more additional parameters that enable us to characterize instances more precisely.