Advanced Algorithms

Class Notes for Monday, December 3, 2012

Bernard Moret

1 Randomized Incremental Construction

1.1 2D Hulls (cont'd)

Last time we discussed an incremental randomized construction algorithm for 2D convex hulls, but did not complete the details of the update of the conflict graph and did not carry out the analysis of the (expected) running time. We now complete this part. We noted last time that much of what happens in the algorithm takes linear time overall and need not concern us: the algorithm adds exactly two new edges at every step and thus $O(n)$ edges overall; at each step, it also removes some number of edges, but these edges were first added and, once removed, are never seen again, so the that total number of edges removed is also $O(n)$. All n points are put in the conflict graph initially; when the algorithm is done, all will have been removed and again this takes $\Theta(n)$ time overall. The problem is thus not with the vertices of the conflict graph, but with its edges. Recall that we put an edge $\{p,e\}$ if point p (outside the current hull) could see edge e (a part of the current hull); the number of such edges may thus reach prohibitively large values: we could have a hull in the form of a triangle, but where one "edge" of the triangle is in fact a convex chain of $n/2$ vertices, and such that the remaining $n/2$ points all can see every vertex on that chain—giving us a quadratic number of edges in the conflict graph. This would force quadratic behavior and storage, so we must avoid it. We avoid it by noting that we do not really need to know all of the edges with which a point is in conflict: if we have one such edge, it suffices, since the edges removed when we add this point to the hull will form a continuous part of the hull's perimeter. Hence from a single edge in the part to be removed, we simply traverse in both directions, processing each successive edge for removal, until we reach the points where the two new edges will get attached (the tangency points).

So here is the crucial modification: we will establish at the very beginning a point inside the initial hull (a triangle in the first step, so we can establish such a point in constant time) and use it as reference. Call that point p_c; thereafter our conflict graph will record only *canonical* conflicts, defined using p_c. A point p outside the hull is defined to be in canonical conflict with edge e of the hull exactly when e intersects segment $\overline{pp_c}$. Assuming that all points are in general position (no three points can be collinear), a point outside the hull must be in canonical conflict with exactly one edge of the hull, so the number of edges of the (canonical) conflict graph is $O(n)$. Now we are ready to proceed with the algorithm. We have already seen how to identify edges to remove and how to define the two new edges, e' and e'', which connect the new hull vertex to the old hull at the boundaries of the removed portion of the perimeter. All of this, as discussed takes linear time over the life of the algorithm. The only additional work is to set up the (canonical) conflict lists for he two new edges, e' and e''. Each point on the (canonical) conflict list of a removed edge that has not already been eliminated by our filtering process will appear in the conflict list of either e' or e'' and we can decide which in constant

time by testing for the intersection of e' against the segment defined by p_c and the point in question. Thus the cost of handling e' and e'' is directly proportional to the number of points on the conflict lists of the removed edges, i.e., those points that will be given a new conflict edge (one of e' or e''). By linearity of expectations, the expected number of such points is simply $n-i$ (the total number of possible candidates at step i) times the probability that any of these points does get assigned a new conflict edge. So we need to derive this last probability and then we can complete the analysis.

To derive this probability, we will reason backwards: instead of looking at the effect of adding a new point to the hull, which is very difficult to do because we do not know which point gets added and what the hull before it looked like (we have a serious problem of conditional probabilities), we will look at the effect of removing the last point added to the hull. This has two big advantages: one, we know that the last point added to the hull is any of the first i points with equal probability; and (ii) the hull after i points have been processed does not depend on the order in which the points were processed, so we do not have a serious problem with conditional probabilities. So consider the hull after i points have been processed and what happens if we undo the last addition of a point. One of the remaining $n-i$ points, call it point p, would have to be assigned a new conflict edge if and only its current edge is one of the two edges added to the hull by the last point processed. But that last point was chosen at random from the among the i points processed so far; and it affects the conflict edge of p only if it is one of the two endpoints of that edge. Thus the probability that p will be assigned a new conflict edge is simply $\frac{2}{i}$ and therefore the expected cost of handling edges e' and e'' is $O(\frac{2}{i}(n-i))$; this cost is at step i, so the expected total cost over all n steps is the sum

$$\sum_{i=4}^{n} \frac{2}{i}(n-i)$$

which is $O(n \log n)$. Thus our 2D convex hull algorithm runs in $\tilde{O}(n \log n)$ time.

1.2 2D Linear Programming

Linear programming is a major business optimization model and also finds use in heuristics for hard problems. Basically, the problem is modelled through a collection of real-valued variables, a collection of linear constraints on these variables, and a linear objective function. Thus, if we use variables $x_1, x_2, \ldots, x_n$, each constraint is an inequality of the form

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \leq a_0$$

Note that such an inequality defines a halfspace in n-dimensional space. Imposing a collection of such constraints is thus equivalent to constraining any solution to lie within the intersection of a collection of halfspaces, that is, within a convex polyhedron. We can use matrix notation to write the set of constraints—if we write the ith constraint as

$$a_{i1} x_1 + a_{i2} x_2 + \ldots + a_{in} x_n \leq b_i$$

then we can simply write

$$\underline{A} \cdot \underline{x} \leq \underline{b}$$

where $\underline{A}$ is an $m \times n$ matrix, $\underline{x}$ is our vector of n variables, and $\underline{b}$ is a vector of m values. In typical applications of linear programming, m is much larger than n. Finally, we have a linear objective function, i.e., something of the form

$$f(x_1, x_2, \ldots, x_n) = c_0 + c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$$

The problem is to find an assignment of values to our variables that (i) respects all constraints and (ii) optimizes the objective function.

Geometrically, as we noted, the constraints describe an intersection of halfspaces (halfplanes in 2D) and so a convex polyhedron (polygon in 2D); this polyhedron is usually bounded, simply because we do not want unbounded solutions to the problem. (Of course, with some constraints, the intersection may be empty, in which case the instance is *infeasible*—it has no possible solution.) The objective function defines a family of hyperplanes (lines in 2D, planes in 3D, etc.), all parallel to each other. Changing the values of the variables moves the hyperplane in a direction perpendicular to the hyperplane—in one direction such a move increases the value of the objective and in the other it decreases it. In consequence, the optimal solution is reached when the hyperplane just touches the convex polyhedron, typically at one vertex of the polyhedron. Solving the linear program thus amounts to finding the extremal point of the intersection polyhedron along a particular direction (the perpendicular to the hyperplane defined by the objective function).

Computationally, the challenge is that the intersection polyhedron, while it cannot have more than m faces (one per hyperplane), can have an exponential number of vertices. Given the vertices, it is completely straightforward to find the two extremal ones along the direction defined by the perpendicular to the hyperplane of the objective function—it takes just one linear pass through the vertices. But generating these vertices can be prohibitively expensive. Until about 30 years ago, it was not known whether linear programs could be solved in polynomial time: the main algorithm, called the *simplex* algorithm (dating from 1947!), walks the edges of the polyhedral boundary and so potentially takes exponential time (although it usually runs very fast). In 1984, the first so-called interior-point method was published (the name indicates that the method does not work by testing the polyhedral boundary); interior-point methods run in polynomial time (of sorts), although they can be slower than the simplex method on many datasets. Good LP (the universal shorthand for linear programs) solvers are a pretty big business, given that customers are usually large industries and so can afford to pay large license fees. Because LPs are just linear algebra at work, there is a huge amount of interesting mathematics around LPs; from a computational standpoint, however, most interesting questions remain unsolved.

We are interested here in a trivial version of LPs, where we have $n = 2$; obviously, this is not a practical restriction, since we are not going to be able to model a lot of real-world problems with just two variables. Computationally, however, this is an interesting application of randomized incremental construction. Since what we have in abundance is constraints (m is not bounded), the incremental part will come from adding one constraint at a time. However, We are *not* going to build the intersection of the halfplanes corresponding to the constraints; instead, we will just maintain the optimal solution corresponding to the collection of constraints treated so far. To understand the algorithm, we first turn, as often in computational geometry, to the one-dimensional version of the problem.

In one dimension, each constraint is of the form $ax \leq b$ and so has solution $x \leq \frac{b}{a}$ if a is positive, $x \geq \frac{b}{-a}$ otherwise. We solve such a problem simply by examining each constraint and, for all those with solution $x \leq \frac{b}{a}$, retaining the smallest $\frac{b}{a}$, while, for all those with solution $x \geq \frac{b}{-a}$, retaining the largest $\frac{b}{-a}$—the solution, being an extreme point, is one of these two values. Thus the problem can be solved in deterministic linear time.

Now consider the two-dimensional version. We shall assume that it is bounded, that $(0,0)$ is a feasible solution, and that it is a maximization problem—this is typical of the vast majority of LP problems. We do not assume that it is feasible: our algorithm will uncover infeasibility if it is present. The optimal solution (if one exists) is a vertex of the convex polygon determined by the intersection of the halfplanes corresponding to the constraints. Thus we start with just two constraints—two halfplanes; the intersection of the two is a polygon with two edges and one vertex, and that vertex is (for now, and taking into account the boundedness) the optimal solution. Now we have an initial optimal vertex, v_{opt}, and $m-2$ unused constraints, so we start processing the remaining constraints, one at a time, in random order, maintaining v_{opt} as we go. At step i, we consider the ith constraint; if its halfplane HP_i contains v_{opt}, then v_{opt} remains the optimal solution and we do nothing. On the other hand, if HP_i does not contain v_{opt}, then this point cannot be the solution as it fails to meet the ith constraint, so we must find the new optimal solution. But this new solution *must* lie on the boundary of HP_i, since the new polygon will include an edge determined by HP_i , but otherwise be composed of vertices that already existed (and were not optimal). In other words, our new optimal solution must lie on a line—we have a one-dimensional LP problem to solve! So we simply use each HP_j, for $1 \leq j < i$, to constrain the interval on the boundary of HP_i, and then we test the two extremities of this interval to determine the new optimal solution. The running time of this part is $\Theta(i)$, since we must test each of the previously used $i-1$ halfplanes. Overall, then, our randomized algorithm takes time

$$\tilde{O}(\sum_{i=3}^{n}(p_i \cdot \alpha \cdot i + (1-p_i) \cdot \beta))$$

where α and β are positive constant factors, and p_i is the probability that HP_i does not contain the optimal solution obtained after processing the first $i-1$ constraints. To derive this probability, we again use backwards analysis. Suppose we have processed the first i constraints and have some optimal vertex v_{opt}; now remove the last constraint applied: what is the probability that v_{opt} will change? It changes exactly when the last constraint was one defining of the two two halfplanes whose boundaries intersect at v_{opt} itself. That is, there are 2 choices, out of i, that would cause a change in the optimal solution; in other words, we can write $p_i = \frac{2}{i}$. Substituting in the $\tilde{O}$ expression above, we get

$$\tilde{O}(\sum_{i=3}^{n}(\frac{2}{i} \cdot \alpha \cdot i + \frac{i-2}{i} \cdot \beta))$$

and so the solution is simply

$$\tilde{O}(n)$$

Our 2D algorithm runs in linear expected time—and note how very simple it is!