Advanced Algorithms

Class Notes for Thursday, November 15, 2012

Bernard Moret

1 Divide-and-Conquer: Voronoi Diagrams

Voronoi diagrams are widely used in applications in fields as diverse as anthropology, biology and ecology, climatology, resource planning, and finite-element modelling. Recall that, given a collection of distinguished points, usually called *sites*, on the plane, say P_1, P_2, ..., P_n, the Voronoi polygon of one of these points, say P_i, is the geometric locus of points (this time, any points) in the plane that are closer to P_i than to any of the P_j, $j \neq i$. We can easily compute the Voronoi polygon of a point P_i by intersecting $n-1$ halfplanes to produce a convex, possibly unbounded, polygon. We can see that by considering the first few values of n. For $n = 1$, we have only P_1 and, by default, its Voronoi polygon is the entire plane (a convex region). For $n = 2$, the Voronoi polygon of each site is a halfplane, and the boundary between these halfplanes is the perpendicular median of $\overline{P_1P_2}$, the segment joining P_1 and P_2. For $n = 3$, the Voronoi polygon of P_1 is the intersection of two halfplanes, one the halfplane of points closer to P_1 than to P_2, and the other the halfplane of points closer to P_1 than to P_3. Its boundary is thus determined (in general) by two halflines, one a piece of the perpendicular median of $\overline{P_1P_2}$ and the other a piece of the perpendicular median of $\overline{P_1P_3}$. We can compute the intersection of n halfplanes in $\Theta(n \log n)$ time by divide-and-conquer: first pair up the halfplanes and compute all $n/2$ intersections of 2 halfplanes; then pair up the resulting polygons and compute all $n/4$ intersections of 2-sided polygons; then pair up the resulting polygons and computer all $n/8$ intersections of 4-sided polygons; and so forth. At each stage the work involved is the same: the polygons double in size, but their number is halved, so the work is $\Theta(n)$. We have $\log_2 n$ stages, which gives us a total running time of $\Theta(n \log n)$.

Now a Voronoi diagram of a collection of sites is simply the collection of Voronoi polygons for these sites. We can thus compute such a diagram by brute force by computing each polygon separately, in $\Theta(n^2 \log n)$ time. Such an approach, however, is both too slow (not really applicable to instances with tens of millions of sites) and too limited: the entire diagram is more than just a collection of polygons, because it forms a tessellation of the plane—a partition of the plane into faces, here convex polygonal faces, each face being a Voronoi polygon. In computational geometry, a tessellation of the plane into polygonal faces is called a *polygonal subdivision*. We would want to the output of our algorithm to represent such a polygonal subdivision rather than just give us a list of polygons. In a good representation of a polygonal subdivision, neighborhood relationships are easy to determine and exploit: for instance, a fundamental notion is that of a polygon sharing a perimeter edge (or a perimeter vertex) with another polygon.

Computational geometers developed a surprisingly simple and effective data structure for polygonal subdivision, the *doubly connected edge list*, or DCEL. In a DCEL, the basic

units are faces and edges, although the edges are really arcs (also called half-edges in this case), because they are oriented and come in twinned pairs (one half-edge in each direction between the same two vertices). As a polygon is defined by its perimeter, so is the face whose boundary is that polygon. We adopt the convention that perimeters (of convex polygons) are always traversed in counterclockwise order. Notice that an edge shared by two adjacent polygons is traversed in one direction by the perimeter of one polygon, and in the other direction by the perimeter of the other polygon. Now a face is simply a label, with a pointer to one of the half-edges on its perimeter. Each half-edge has two pointers: one to its successor half-edge on the perimeter of the face to which it belongs, and one to its "twin," which is the same edge, but in the opposite direction, and which belongs to the neighboring face across that edge. In addition, each half-edge is also labelled by the label of the face to which it belongs. With just these few characteristics, we can traverse a face (given the face, use its pointer to get a starting half-edge on the perimeter, and memorize it; then move through successor pointers from half-edge to half-edge until we get back to the starting half-edge). We can more from one face to a neighbor by simply using the twin pointer of a half-edge. We can identify all faces that share a vertex at the head of a half-edge by repeatedly taking the successor of the twin of the successor of the current half-edge, which will eventually return us to our starting half-edge having visited every half-edge sharing its head with the head of our starting half-edge. One can devise many more such simple routines that accomplish various operations on the polygonal subdivision. The DCEL is very frugal in its space requirements, yet perfectly suited to algorithms based on neighborhoods. We will use it to represent our 2D Voronoi diagrams.

A Voronoi diagram has all sorts of neat mathematical properties. For our purposes, the most important is that it is a planar graph and thus must obey Euler's formula for planar graphs relating the number of faces, the number of edges, and the number of vertices. If the planar graph is connected, then it must satisfy $F - E + V = 2$, where F is the number of faces, E the number of edges, and V the number of vertices. In our case, we have one face for each input site, so n faces. In addition, a face has at least 3 edges and an edge belongs to at most two faces, so we have $3F \leq 2E$, which, after a bit of algebra, also give us $E \leq 3V - 6$ (for $V \geq 3$). Moreover, Voronoi diagrams are rather special planar graphs, so we can also prove (just for them, not for general planar graphs) $V \leq 2F - 5$. (To see how, note that a polygonal vertex is equidistant from the sites sharing this vertex, which, in non-degenerate cases, will be three sites or more. Using this degree constraint, one can derive the inequality. For degenerate cases, such as all sites on the same line or on the same circle, so that all Voronoi polygons are unbounded, the inequality holds trivially.) The importance of these inequalities is that they tell us that the complexity of a Voronoi diagram is linear in the number of sites; with n sites (or faces), a Voronoi diagram has $\Theta(n)$ edges and $\Theta(n)$ vertices. Thus our DCEL has size $\Theta(n)$ and we not output-bound in the computation.

Next we take a look at a basic question about a Voronoi diagram: which sites have infinite polygons, which have finite ones? The answer is quite simple and perhaps expected.

Theorem 1. *A site has an infinite Voronoi polygon if and only if it lies on the convex hull of the collection of sites.*

Proof. Consider the polygon for site p: that polygon is unbounded if and only if it has at least one unbounded (half-infinite) edge. That edge, being a boundary between two Voronoi polygons, is determined by p and some other site q. Consider the geometric configuration defined by p, q, the line joining the two, the two halfplanes determined by that line, and the perpendicular median of $\overline{pq}$, which subtends the half-infinite edge of interest. Let x be a point on the perpendicular median: by definition, it is equidistant from p and q so that we can pass a circle through p and q that is centered at x. Now, only a part of the perpendicular median forms the boundary between the Voronoi polygons of p and q; x is on that boundary if and only if the circle centered centered at x and passing through p and q contains only points whose nearest neighboring site is one of p or q, i.e, if and only if the circle contains no other site. Now consider moving our point x farther and farther away along the infinite edge; the circle passing through x and y grows larger and larger, but always remains empty of any other site. At the limit, as x goes to infinity, the circle becomes the halfplane determined by the line through p and q. Hence the halfplane to one side of the line determined by p and q does not contain any other site, which means that p and q lie on the convex hull. □

We already know how to compute the convex hull of a set of points in $O(n \log n)$ time and such a computation will give us all of the half-infinite polygonal edges of the Voronoi diagram: they are simply the perpendicular medians of the edges of the perimeter of the convex hull. This result is the key to our design of a geometric divide-and-conquer procedure for computing the Voronoi diagram of a collection of sites in the plane.

Our algorithm will use geometric divide-and-conquer, meaning that we will presort all sites, say in increasing order by abscissa (left to right along the x axis). Our algorithm will make two recursive calls, one passing to the recursive call the leftmost $n/2$ sites, the other passing the rightmost $n/2$ sites. We require that our algorithm return both the convex hull of the set of sites it receives and the Voronoi diagram of that set of sites. On return from the two recursive calls, our algorithm will use the two convex hulls to help it merge the two Voronoi diagrams. (Note that each of the two Voronoi diagrams is a polygonal subdivision of the entire plane, not just of a halfplane.) Our algorithm will merge the two Voronoi diagrams in linear time (no mean feat!), so that the recurrence relation describing the running time will the same as for mergesort, with the same solution: $O(n \log n)$.

Thus the algorithm of interest is the merging algorithm. We begin by merging the two convex hulls—we have seen how to do this in linear time when we studied a divide-and-conquer algorithm for convex hulls, using the fact that convex polygons given by their perimeter are, for all practical purposed, already sorted along any direction. In the present case, the two hulls are guaranteed to be disjoint—they are in different halfplane thanks to our geometric divide-and-conquer. Merging two hulls separated by a vertical line can be viewed as finding an upper tangent to the two polygons and a lower tangent; each tangent determines a new segment of the overall hull and the two tangency points in one hull partition the hull perimeter into an inside and an outside polygonal lines. The inside polygonal lines are eliminated, while the outside polygonal lines are joined to each other using the two edges from the tangents. This can be done indirectly by using the Graham scan to con-

struct the merged hull, then searching the result for the points of transition between the left hull and the right hull, or it can be done directly by walking the perimeter of each hull. The two joining segments are the key components of the merging procedure: they determine, as just saw, two of the half-infinite polygonal boundaries of the final Voronoi diagram, the two new ones—the others are determined by segments of left hull and segments of the right hull and so are already part of the two Voronoi diagrams returned by the recursive calls. The two new half-infinite edges are the two ends of an infinite polygonal line that sorts out which of the two Voronoi diagrams is to be used where in the plane: to the left of the polygonal line we use the diagram produced by the first ("left") recursive call and ignore the one produced by the second recursive call, whereas to the right of the polygonal line we use the diagram produced by the second recursive call and ignore the one produced by the first recursive call.

Denote by VD_1 the Voronoi diagram on the first $n/2$ points (produced by the first recursive call) and by VD_2 the Voronoi diagram on the last $n/2$ points (produced by the second recursive call). Let the top segment joining the two hulls be $\overline{AB}$, with A on the left hull and B on the right hull. Finally, let P_A be the Voronoi polygon of site A in VD_1 and P_B be the Voronoi polygon of site B in VD_2. As we start from infinity (far away) on the perpendicular bisector of $\overline{AB}$, we are on the boundary between the Voronoi diagram of A and that of B in the final Voronoi diagram, but we are also in P_A and in P_B. As we move from infinity along that bisector toward $\overline{AB}$ and possibly beyond it, we will eventually intersect the boundary of one of P_A or P_B. We can find out the intersection by looking for the intersection of the bisector with P_A and separately looking for the intersection with P_B. In each case, we use the DCEL structure to walk the perimeter of the polygon, testing each edge for intersection with the bisector and stopping as soon as an intersection is found. We can then compare the two intersection points and find which one occurs first along the bisector as we move in from (plus) infinity. Note that this takes time proportional to the number of edges in the polygon. Say that the intersection is with P_B: thus, as we continue moving, we will leave polygon P_B and enter its neighbor across the intersected edge (something we can immediately get with the twin pointer in the DCEL), call it polygon C. As we cross this boundary, however, the two nearest sites will change from being the pair (A,B) to the pair (A,C), so that the perpendicular bisector that defines our line of travel will turn, in this case turning to the right in the direction of motion. We are now in polygon A from VD_1 and in polygon C from VD_2 and must again find out which of the two polygons our new bisector will intersect. For C, this is the same procedure we used for A and B; for A, however, there is no need to test again what we already tested, since the direction of motion can only cause an intersection farther down the perimeter, so we resume the search from where it left off in the previous step. This, of course, means that each polygon will be swept at most once in its entirety during the course of the algorithm. If we now find that the next intersection is with C, then we will enter neighboring polygon D from VD_2, and so the new bisector will now be determined by the site pairs (A,C) and so our direction of travel will make a further turn to the right in the direction of travel. (Changing the nearest site in VD_1 causes a turn to the left in the direction of travel.) As we continue tracing the polygonal line, the boundary between the part of the plane where the Voronoi diagram is determined by the

first $n/2$ points and that where it is determined by the last $n/2$ points, we will eventually end up traversing the perpendicular median to the bottom edge joining left and right hulls, at which point we are done. Along the way, at every intersection, we have to update the DCEL structure of the new Voronoi diagram, which we do by slowing weaving together the two DCEL structures returned by the two recursive calls, adding connections between them (the polygonal edges from the polygonal line, which are linked on the left to the DCEL of VD_1 and on the right to the DCEL of VD_2) As we continue tracing the polygonal line, the boundary between the part of the plane where the Voronoi diagram is determined by the first $n/2$ points and that where it is determined by the last $n/2$ points, we will eventually end up traversing the perpendicular median to the bottom edge joining left and right hulls, at which point we are done. Along the way, at every intersection, we have to update the DCEL structure of the new Voronoi diagram, which we do by slowing weaving together the two DCEL structures returned by the two recursive calls, disconnecting parts that lie on the "wrong" side of the polygonal line and replacing them by connections between the two DCELS mediated by the polygonal edges from the polygonal line, which are linked on the left to the DCEL of VD_1 and on the right to the DCEL of VD_2.

The DCEL maintenance is straightforward, of perhaps rather technical and detailed; it takes linear time overall. As we have seen, detecting the intersection will at most require sweeping each polygon once, resulting in a running time proportional to the total number of edges in VD_1 and VD_2, which we have seen is $\Theta(n)$. Hence the merging procedure takes linear time, as desired, so that the entire algorithm builds a Voronoi diagram for a collection of n sites in $\Theta(n \log n)$ time. The proof of correctness relies on the invariant that, at any time, we follow the boundary between two Voronoi polygons, say those of sites X and y, of the final diagram, while also being located in the intersection of Voronoi polygons P_X from VD_1 and P_Y from VD_2.

This divide-and-conquer algorithm shows that geometric divide-and-conquer can be a very powerful tool in computational geometry, but that the power comes at the expense of added complexity: the merging process is rather intricate. As it turns out, this is not the fastest algorithm to compute Voronoi diagrams, although the asymptotic time is optimal. An alternative approach uses a sweep of the plane; it is elegant and more streamlined, but also much more involved—for instance, its sweepline must maintain a "wavefront" of multiple, possibly nested and interwoven, parabolas. The Computational Geometry textbook recommended in the Reading Materials has an excellent section on this approach, usually called Fortune's algorithm.