Advanced Algorithms

Class Notes for Monday, November 5, 2012

Bernard Moret

1 Sorting, Sweeping, and Divide-and-Conquer

Our next template for algorithm design is familiar to all of us from mergesort and quicksort, two different uses of the same general principle: break up a problem into smaller pieces, solve the pieces, then assemble the solutions obtained on the pieces into a solution for the whole.

The main area of application of this class of methods is computational geometry. That such is the case is due to one simple characteristic of computational geometry: almost any question we can ask about geometry is computationally trivial in one dimension (along a line), but much harder in two or more dimensions. (Moreover, the difficulty often grows as the dimensionality grows.) As a result, the main approach to algorithm design in computational geometry is to reduce the given problem to a collection of subproblems of *lower dimensionality*. Sorting and sweeping can easily do that: consider, for instance, sweeping the plane with a line (e.g., moving a vertical line from minus infinity to plus infinity along the x axis): between two successive positions of the sweep line, the work takes place in an infinitely tall, but very narrow strip—an analog of the 1-dimensional line. Divide-and-conquer can do that as well, sometimes to exactly the same effect (what are called "oblivious" divide-and-conquer algorithms, algorithms that run through exactly the same steps on any input data of the same size), sometimes with a bit more flexibility and effectiveness.

Computational geometry, as we shall study it for a few weeks, deals with geometric objects amenable to a finite, discrete description and attempts to provide data structures and algorithms to manipulate these objects and answer questions about them. It is a crucial component of CAD/CAM (computer-aided design/computer-aided manufacturing) systems

The most elementary of these objects are points, lines and line segments, polygonal lines and polygons, planes, polyhedra, and simple nonlinear curves and surfaces such as circles and spheres. These objects can then be combined to form more complex structures, such as a partition of the plane into a collection of polygonal regions (an abstraction of a map). Operations we may want to perform on these objects include intersection, union, inclusion, and various optimization tasks (smallest enclosing disk, shortest path in the presence of obstacles, etc.); questions we may want to ask about these objects include location (where does a point fall on a map?), trajectories (will two mobile objects collide?), vicinity (find the nearest neighbor), etc. Finally, and perhaps most importantly, much work in computational geometry is devoted to building dynamic data structures, that is data structures that can be updated as well as queried—in effect, geometric databases.

Doing geometry with a computer has two main pitfalls. One is that almost every geometric structure can degenerate into something much simpler and perhaps unexpected. For instance, the formal way to define a line is to give a point and a vector—in accordance with Euclid's axiom that exactly one line exists that is parallel to a given direction and passes through a given point. The more common way, however, is to give two points—again, exactly one line exists that passes through both points, *provided, however, that these two points are distinct*. In an algorithm, if we assume that the input is two points defining a line, we must first test whether these two points are distinct and take appropriate action in case they are not. A more extreme example is a polygon; here the obvious way to specify one is to list its vertices on a tour of the perimeter, returning to the starting point. For instance, a triangle can be given by its three vertices. However, what if two of the points listed are the same? What if all three are the same? Much of the time we will be sorting geometric objects based on a coordinate and our algorithm will treat the general case where no two objects have the same coordinate; but the code for the algorithm must also handle cases where two or more objects have the same coordinate. Thus special cases are a constant affliction on the programmer in computational geometry.

Another problem is precision. If we use only integers, it is immediately clear that we cannot address questions of intersection. But even if we use floating point values, we are limited in precision: we cannot distinguish two values that differ by less than the machine ε. In other words, we cannot represent the plane, only a grid of points. But then we are back to the integer situation: it is easy to define two lines (two pairs of points on the grid) such that their intersection is not a point of the grid—and then this intersection cannot be accurately computed. Our floating-point routine will, of necessity, return a grid point, and at best this is a close neighbor of the correct answer. Further queries about relative locations of lines and points may then fail. There is a simple answer to this problem: use arbitrary-precision arithmetic—there are good library packages for this purpose, such as the GNU Multiple Precision (GMP) arithmetic package. However, of course, such arithmetic is always much more expensive than fixed-precision arithmetic. The standard library for computational geometry, CGAL (Computational Geometry Algorithms Library) uses the GMP package, but keeps multiple representations of any given geometric objects so as to be able to use whichever representation (in particular, fixed vs. multiple precision) is best for each algorithm—each algorithm must specify its precision requirements.

In our quick tour of computational geometry, we will ignore both problems, as neither one directly affects the design of algorithms; but any implementer must take both into account, either by tailoring each routine or by designing some general-purpose solution.

1.1 Polygons: simplicity and convexity

A polygon is defined by its perimeter, which is a closed polygonal line. In turn, a polygonal line is simply a succession of line segments, each sharing an endpoint with the previous segment. Defined in this way, a polygon can be something a bit unexpected: for instance, we can have a triangle where all three vertices are collinear (lie on the same line). If we insist (as we shall) that all vertices be distinct and that consecutive segments share *exactly* *one* point, a common endpoint, then a lot of strange cases disappear, but we can still have intersecting nonconsecutive segments—for instance, we could have a quadrilateral formed by twisting one edge of a rectangle, in which two edges intersect (forming a shape similar to an hourglass). These are legitimate polygons, but of little interest to us, as they do not define a region of the plane. Our interest lies in so-called *simple* polygons, polygons that have a well defined interior and so partition the plane into two regions—inside and outside. In such a polygon, a circuit of the perimeter will always "see" the outside on the same side of the edge currently being crossed, and the inside on the other side—that is, if one takes a point that is infinitesimally distant from the current edge on one side, that point is always outside the polygon and, if on the other side, it is always inside the polygon. Simple polygons may of course have a very complex shape, but from any point in the plane, one can always trace a half-line (from that point to infinity) that does not intersect any vertex of the polygon and count the number of edges intersected by that half-line to decide whether the point is inside the polygon (the count is odd) or outside the polygon (the count is even).

Among simple polygons, one type is of particular interest: the convex polygons. Intuitively, a polygon is convex if it does not have any "dimple," i.e., if none of its interior angles (an interior angle is an angle between two consecutive perimeter segments, measured on the inside of the polygon) exceeds π radians. A more formal definition of convexity, however, is both useful mathematically and applicable to objects other than polygons. Let U be a ground set of objects, with a binary operation $\oplus$ and a scalar operation $\cdot$ (where the scalars can be real or rational numbers). Let $s_1, s_2, \ldots, s_k$ be elements of U and $a_1, a_2, \ldots, a_k$ some scalars (elements of $\mathcal{R}$). A *linear combination* of these elements is $a_1 \cdot s_1 \oplus a_2 \cdot s_2 \oplus \ldots \oplus a_k \cdot s_k$, which we just write $\sum_{i=1}^{k} a_i \cdot s_i$. (Note that, by definition of the binary operation and the scalar operation, a linear combination of elements of U is an element of U.) If we also have $\sum_{i=1}^{k} a_i = 1$ and $\forall i,\, a_i \geq 0$, then we call such a combination a *convex combination*. In particular, the convex combination of two elements under addition can be written in the more familiar form $\alpha x + (1-\alpha)y$, with $0 \leq \alpha \leq 1$.

A set $S \subseteq U$ is said to be convex if and only if every convex combination of elements of S is itself an element of S—that is, S is convex if and only if it is closed under convex combinations. For instance, a line segment is a convex object if viewed as a collection of points: any convex combination of points along the segment is just a point on the segment. A simple polygon is said to be convex if the set of points forming its interior and its boundary is a convex set. In view of the definition of convexity, we can also view a convex polygon to be the smallest convex subset of the plane that contains all of the given vertices; or as the set of points in the plane that can be generated by forming convex combinations of the given vertices. The connection with the intuitive idea of "no dimple" is then clear: if a dimple existed, we could select a vertex inside the polygon on each side of the dimple and trace the segment joining these two vertices. In a convex polygon the entire segment (any convex combination of the two endpoints) would lie inside the polygon, but with a dimple, some of the segment would lie outside, meaning that some of the convex combinations of the two endpoints do not belong to the polygon.

Convexity is a useful property for many reasons—convex shapes are particularly easy to manipulate, they have minimum surface area, etc. (Note that a disk or a ball is convex.)

We shall return to convexity for several questions, but for now let us just see how we can manipulate convex polygons for a simple geometric problem: given two convex polygons, compute their intersection.

1.2 Intersection of two convex polygons

As a first example of the usefulness of sweep methods (based on sorting), let us look at the problem of computing the intersection of two convex polygons. We begin by observing that the intersection of two convex sets is itself a convex set.

Theorem 1. *The intersection of two convex sets is a convex set.*

Proof. Let S_1 and S_2 be two convex sets and denote the operations on these sets by $+$ (the binary operation) and $\cdot$ (the scalar product). Consider the intersection, $S = S_1 \cap S_2$. Let x and y be any two elements of S; we need to prove that, for any value of α between 0 and 1, $\alpha \cdot x + (1-\alpha) \cdot y$ is an element of S. Since x and y belong to S, they also belong to S_1, which is known to be convex, so that for any value of α between 0 and 1, $\alpha \cdot x + (1-\alpha) \cdot y$ is an element of S_1. Similarly, since x and y belong to S, they also belong to S_2, which is known to be convex, so that for any value of α between 0 and 1, $\alpha \cdot x + (1-\alpha) \cdot y$ is an element of S_2. Hence, for any value of α between 0 and 1, $\alpha \cdot x + (1-\alpha) \cdot y$ is an element of $S_1 \cap S_2 = S$, as desired. ☐

Thus the intersection of two convex polygons is itself a convex polygon (perhaps the empty one).

Next note that we can view any convex polygon to be formed of two chains of vertices, each starting at the leftmost vertex and each ending at the rightmost vertex (where leftmost and rightmost are defined with respect to some axis, such as the x axis): one chain moves up from the starting vertex and is composed of the upper part of the perimeter of the polygon—it is an upward convex chain—and the other moves down from the starting vertex and is composed of the lower part of the perimeter of the polygon—it is a downward convex chain. we can compute the intersection of two convex polygons by separately computing the intersection of their upper chains, producing the upper chain of the result, and the intersection of their lower chains, producing the lower chain of the result. Note also that, given the perimeter of a convex polygon, we can produce the upper chain and the lower chain in linear time: we scan the perimeter to identify the leftmost vertex and the rightmost vertex and set up new pointers to them, then we cut the circular linked list at these two points. Since each chain is a sorted list of vertices (sorted along the chosen axis), we can sort all vertices of the two upper chains in linear time simply by merging the two chains (as in the merging step of mergesort).

Once all vertices of the two upper chains are in sorted order, we proceed along this sorted order until we encounter one or more vertices from one polygon and the first vertex from the other polygon: it is only at this point that the intersection of the two polygons may start. We need to test whether the first vertex from the second polygon is inside the first; if so, then the upper chain of the intersection starts at this first vertex. Otherwise, we follow the upper chain of the lower polygon and the lower chain of the upper polygon, step by step along the sorted order, until the two intersect (if they do—otherwise the intersection is empty); the intersection point is then the start of the chains for the intersection. At any following step, we are in a vertical strip delimited by the x coordinates of two consecutive vertices in the sorted order. In that strip, the two upper chains show as two segments (or partial segments) and the lower of the two segments is on the intersection chain, unless these two segments intersect within the vertical strip, in which case the intersection chain within the strip is made of the beginning of the segment of one chain, the intersection point, and the end of the segment of the other chain. Processing a strip takes $O(1)$ time; the number of strips is $O(n)$; and so the entire sweep takes linear time.