Advanced Algorithms

Class Notes for Thursday, November 8, 2012

Bernard Moret

1 Sweeping and Data Structures: Segment Intersection

The problem we are going to study today is simple to formulate, but leads to a number of interesting discoveries. We are given a collection of segments in the plane and want to know if any two of them intersect. Once we have solved this problem, we shall take up the somewhat more demanding version in which we want to know the number of intersecting pairs. Obviously, both problems can be solved in $O(n^2)$ time by the brute-force method of testing every one of the $\binom{n}{2}$ pairs of segments for possible intersection; therefore our goal is to develop a subquadratic method that does not have to test all pairs.

1.1 The decision problem

In order to minimize the number of pairs that must be considered, a rather obvious first step is to consider only segments that share a range of abscissae. We implement this idea by ordering the vertices by their abscissae and traversing the sorted list of vertices, all the while maintaining a list of active segments, i.e., segments for which we have seen the left endpoint, but not yet seen the right endpoint. Viewed abstractly, this process is one of *sweeping* along the x-axis from left to right. When a segment becomes active, we consider potential intersections with the other active segments; when it becomes inactive, we remove it from all further consideration. Unfortunately, if implemented in an obvious manner, this approach does no better than the brute-force algorithm, since all segments may share a range of abscissae.

In order to limit the number of tests for intersection, we need to analyze the spatial relationships between active segments as the sweep line moves to the right. Consider the set of points formed by the intersection of the vertical sweep line with the active segments. (For now, we ignore vertical segments.) The ordering of these points by their ordinates induces an ordering on the active segments and, except at a segment endpoint (which adds or removes an active segment), this ordering does not change as the sweep line moves right, unless an intersection is encountered. It follows that, at any time during the sweep, the only candidates for intersection are adjacent segments in the induced ordering. Hence an algorithm based on such a sweep can maintain the invariant that no two adjacent active segments intersect, until either all segments are processed or an intersection is found. While the invariant holds, the only events that can invalidate it are the insertion or deletion of an active segment, since these operations modify the induced ordering. Thus we need to test any two segments newly made adjacent in that ordering. When a segment becomes active, we must test it for intersection with the segments that lie immediately above and below it. When a segment becomes inactive, we must check the segments immediately above and below it, as they now become adjacent and thus candidates for intersection. A maximum of $3n$ tests for segment intersection are thus performed, with $2n$ of these tests occurring at left endpoints and n at right endpoints.

We need a data structure for maintaining the induced ordering on the active segments. This data structure must offer efficient support for insertion and deletion of segments and for search for inorder predecessor (for the segment below) and inorder successor (for the segment above). As each of the four operations can be executed a linear number of times, none can take more than amortized logarithmic time. A balanced search tree, such as a red-black tree, with all of its leaves held in a doubly-linked list, guarantees logarithmic worst-case behavior for insertion and deletion and constant-time behavior for predecessor and successor operations, thereby meeting our requirements. A splay tree guarantees amortized logarithmic time for all four operations and can also be used. Note that this structure does not store keys, but segments; the values used to guide the search during insertion and deletion are the ordinates of the segments at the *current* sweep position and so must be computed on the fly from the value of x and the description of the segments.

Let us then examine one step of our algorithm in more detail. At a given step, the algorithm processes a new endpoint. If the endpoint is a left endpoint (the beginning of a segment), the segment is inserted into the tree and is tested for intersection against its newly acquired inorder predecessor and successor. If the endpoint is the end of a segment, that segment is removed from the tree and its old predecessor and successor are checked against each other for intersection. (Note that, if several intersections exist in the configuration, the algorithm does not necessarily find the leftmost such—but for our purposes it is enough to find any one of these intersections.) Vertical segments present a small problem, since both their endpoints appear at once, but are correctly handled if the endpoints are sorted using their ordinates as tiebreakers. Coincident endpoints present another small problem, easily resolved by considering left endpoints to come before right endpoints in the order, which ensures that a segment is not deleted before its potential intersection with a new segment is detected. The running time of a step is logarithmic and so the entire algorithm runs in $\Theta(n \log n)$ time in the worst case. This is optimal in a comparison-based model of computation: we know that, in such a model, it takes $\Omega(n \log n)$ comparisons to decide whether each number in a collection of n numbers is unique (the so-called *Element Uniqueness* problem) and it is easy to see that we can decide that question by setting up a specialized instance of segment intersection, where each number x_i from the original problem becomes a short vertical segment with endpoints $(x_i, 0)$ and $(x_i, 1)$—no two segments will intersect if and only if all x_is are unique.

Finally, note that, instead of sorting all endpoints in advance, we could simply place all of them in a priority queue and retrieve the next endpoint using a DELETEMIN operation. This change does not alter the asymptotic worst-case running time, since it takes linear time to set up the priority queue and logarithmic time for each DELETEMIN operation, but it may speed up the program if an intersection exists. If an intersection exists and is detected at the kth endpoint, then the version using the priority queue will run in $\Theta(k \log n)$, which could be significantly better than $\Theta(n \log n)$. Incidentally, it is also useful, as always in coding, to add sentinels, in our case two long horizontal segments, each beginning before and extending past, any of the input segments, one safely above all input segments and the other safely below all input segments. This addition only increases n by 2 and removes any need to test for boundary cases—we will always find an inorder predecessor and an inorder successor for any input segment.

1.2 The counting problem and testing for simplicity

If we want to count intersecting pairs, we cannot stop at the first intersection we detect. Since intersections themselves alter the ordering along the sweep line (the two intersecting segments exchange places), we must now treat intersections as events in the sweep. In turn, this requires us to use the priority queue approach rather than a first sorting pass, since the order in which events occur is now determined dynamically. By maintaining bidirectional pointers between the leaves and their corresponding internal nodes, the search tree data structure can be updated in constant time to reflect the discovery of an intersection. However, the extra events may considerably increase the overall running time: if there are m intersecting pairs, our algorithm will now run in $\Theta(n \log n + m \log m)$ worst-case time. On the good side, this maintenance strategy allows us to report the intersections in left-to-right order. On the bad side, when m approaches n^2 (which is possible—the largest value is $n^2/4$, our algorithm is outperformed by the brute-force $\Theta(n^2)$ algorithm, although this latter does not report the intersections in left-to-right order. It is in fact possible to match the brute-force algorithm when m is in $\Theta(n^2)$, using a rather complicated variation of our algorithm in which intersections are not treated as events in left-to-right order; such an algorithm runs in $\Theta(n \log n + m)$ worst-case time, but it is very complex as well as not particularly instructive, so we shall not present it.

As often in computational geometry, we have a lot of details to worry about for our new algorithm. Some are due to ties in the ordering: what if more than two segments intersect in the same point? what if, at a given position of the sweep line, we have left endpoints, right endpoints, and intersections? what if some left endpoints are also intersections? what about vertical segments, including multiple vertical segments on the same vertical? and so forth. These problems require a systematic way to break ties in the sweep. We already saw with the decision version that we needed to handle left endpoints before right endpoints; and it makes good intuitive sense to handle intersections (our third type of events) in-between the two—after left endpoints, so that we have all segments in hand when testing the consequences of a switch in the ordering, but before right endpoints, so that we have not yet lost any of the segments that may be involved in an intersection with a newly discovered neighbor. Now, if k segments, for $k > 2$, intersect at the same point, we must report $\binom{k}{2}$ intersections, since we must report all pairs of intersecting segments; so how does this work? When we handle an intersection, we flip the two segments involved and test each for a possible intersection with its new neighbor; here both segments have $k-2$ possible neighbors (ties on the y axis) with which they intersect, so that the tests will uncover more of the $\binom{k}{2}$ intersections, which will be inserted in the priority queue at the very same sweep position on which the algorithm is currently working, and so processed immediately, before any right endpoints. A bit of thought shows that this order for handling events, supplemented by a third-level tiebreaker based on the y coordinate events in the same category, suffices to resolve all issues raised through ties.

Another category of problems is the multiple discovery of the same intersection. Consider two long segments with shallow slopes that intersect very far to the right; and now assume that, lodged in the wedge formed by these two segments are many short horizontal segments, none of which intersects with either of the two long segments and no two of which overlap along the x axis. Every time we come to the end of one of the short segments, the two long ones become immediate neighbors again and so their intersection is “rediscovered.” Thus we need to ensure that we do not insert the same future intersection event multiple times in the priority queue. The same problem arises with multiple segments intersecting in the same point: the successive switches among them them caused by handling successive intersections can cause the same pair to be tested twice and we must again avoid placing the same intersection twice in the priority queue—indeed, in the worst case, we could create an infinite loop if we keep placing the same intersection in the queue, then handling it by switching segments. So we need to identify that an event has already been scheduled; we can do this by creating a unique identifier for each event and storing these identifiers in a search tree—the extra cost is just $O(\log n)$ per identifier for inserting or looking for one in the tree and so does not affect the asymptotic running time.

Now we can easily use our counting algorithm to test a polygon for simplicity in $\Theta(n \log n)$ time. We simply turn the perimeter list of edges into a collection of segments and run our counting algorithm. If the algorithm stops with a count of n, the polygon is simple: because it is a polygon, each of its vertices is an intersection (between two consecutive edges). Thus it must have at least n intersections; if it has no more than n, then the only pairs of intersecting edges are adjacent edges and they intersect only in their common endpoint. If the algorithm gets to a count of $n+1$, we immediately stop and report that the polygon is not simple. Thus the algorithm runs through at most $n+1$ intersections, as well as through at most all $2n$ endpoints, and consequently takes $\Theta(n \log n)$ time in the worst case. (It is in fact possible to devise a linear-time algorithm for simplicity testing, but the solution is extremely complex and far beyond the scope of this course.)

1.3 Point location in a map using persistent data structures

What we did with segments is in fact easily generalizable. For instance, if we want to test whether two simple polygons, one with n_1 vertices and the other with n_2 vertices, intersect, we can simply take the $n_+ 1 + n_2$ segments and use our intersection counting routine. If the count of intersections is $n_1 + n_2$, then the two polygons do not intersect (although one could be contained within the other); otherwise they do. More applications with something other than segments, such as circles, are also possible.

Beyond these generalizations lie even more uses for the two-step sorting approach. Consider point location in a map, where a map is understood to be a partition of the plane into disjoint simple polygons. The partition implies that no two of the polygons share anything other than a fraction of their polygonal boundary (the area of their intersection is always 0), that every point in the plane belongs to at least one polygon, and that a point belongs to more than one polygon exactly when it sits on a polygonal boundary. Locating a point in such a map is solving the following query: given the coordinates of the point, return the ID of the polygon(s) to which the point belongs.

If you think about the process used for segment intersection, you realize that we have already solved that problem in an indirect manner. We can view the map just as a collection of all of the segments forming the polygonal boundaries. If we run our segment intersection algorithm on this collection of segments, it proceeds from event to event maintaining a binary search tree for the vertical ordering of segments within a vertical stripe of the plane that extends from minus to plus infinity along the y axis and from the current event to the next event along the x axis. These vertical stripes are completely determined by the polygonal vertices of the map, so that we can sort them and thus also locate a given point within the proper stripe in logarithmic time. Then, if we can use the version of the binary search tree that matches this stripe, we can locate our point along the vertical dimension between two segments, thereby determining to which polygon our point belongs and thus solving the point location problem.

Preparing a searchable array or binary tree for locating the correct stripe is just a matter of sorting the vertices; the array or tree uses linear storage and searching it takes logarithmic time. The problem arises with the binary search trees used for maintaining the vertical ordering: in our segment intersection algorithm, there is only one such tree, which gets modified by each event, but in order to carry out point location, we need to have available for searching every tree produced during the course of the algorithm. If we simply make a copy at each step, the space requirements of our data structure will be huge—$\Theta(n^2)$, where n is typically very large (in the millions for a good-sized map). The location process would be very fast (logarithmic), but the storage might easily exceed capacity. It turns out that there is a solution to this problem, albeit one whose details are beyond the scope of our course. The solution is to use a so-called *persistent* data structure—as opposed to the kind of data structures we normally used, which are termed *ephemeral*. In an ephemeral data structure, the state of the data structure always reflects the latest events; queries against such a data structure of necessity return up-to-date information. In contrast, in a persistent data structure, queries are made using both a key (in the usual way) and a time stamp—the query being made against the state of the data structure at the time indicated by the time stamp. In order to build such a data structure, new pointers and, if necessary, new nodes, are introduced to reflect changes, while keeping the old pointers (and old nodes, if any), distinguishing new from old via a time stamp. When querying a persistent search tree, for instance, we first use the normal key to find out whether we should move left or right down the tree, then we use the time stamp of the query to decide which of the several left (or right, if we have to move right) child pointers to use. The number of levels of search remains the same, but we run two tests at each level, one to choose left or right and the other to choose the proper time stamp. In the case of our polygonal map, incremental changes from one event to the other are very small (one more or one less entry in the tree), so that the total amount of additional information stored in the tree cannot exceed one extra pointer per visited node for each event. Since the total number of nodes visited in the binary search tree during the segment intersection algorithm is $O(n \log n)$, the total storage requirements are in $O(n \log n)$—in fact, they are linear, but we cannot verify this claim without more details on the data structure. Thus the principles used in building a persistent search tree are quite elementary; the difficulty is in the analysis.