25. Plane-sweep: a general-purpose algorithm for two-dimensional problems illustrated using line segment intersection

Learning objectives:

- line segment intersection test
- turning space dimensions into time dimensions
- updating a y table and detecting intersections
- sweeping across and intersection

Plane-sweep is an algorithm schema for two-dimensional geometry of great generality and effectiveness, and algorithm designers are well advised to try it first. It works for a surprisingly large set of problems, and when it works, tends to be very efficient. Plane-sweep is easiest to understand under the assumption of nondegenerate configurations. After explaining plane-sweep under this assumption, we remark on how degenerate cases can be handled with plane-sweep.

The line segment intersection test

We present a plane-sweep algorithm [SH 76] for the *line segment intersection test*:

```
   Given n line segments in the plane, determine whether any two
intersect;
   and if so, compute a witness (i.e. a pair of segments that
intersect).
```

Bounds on the complexity of this problem are easily obtained. The literature on computational geometry (e.g. [PS 85]) proves a lower bound $\Omega(n \cdot \log n)$. The obvious brute force approach of testing all $n \cdot (n - 1) / 2$ pairs of line segments requires $\Theta(n^2)$ time. This wide gap between $n \cdot \log n$ and n^2 is a challenge to the algorithm designer, who strives for an optimal algorithm whose asymptotic running time $O(n \cdot \log n)$ matches the lower bound.

Divide-and-conquer is often the first attempt to design an algorithm, and it comes in two variants illustrated in Fig. 25.1: (1) Divide the data, in this case the set of line segments, into two subsets of approximately equal size (i.e. n / 2 line segments), or (2) divide the embedding space, which is easily cut in exact halves.

25. Plane-sweep: a general-purpose algorithm for two-dimensional problems illustrated using line segment intersection

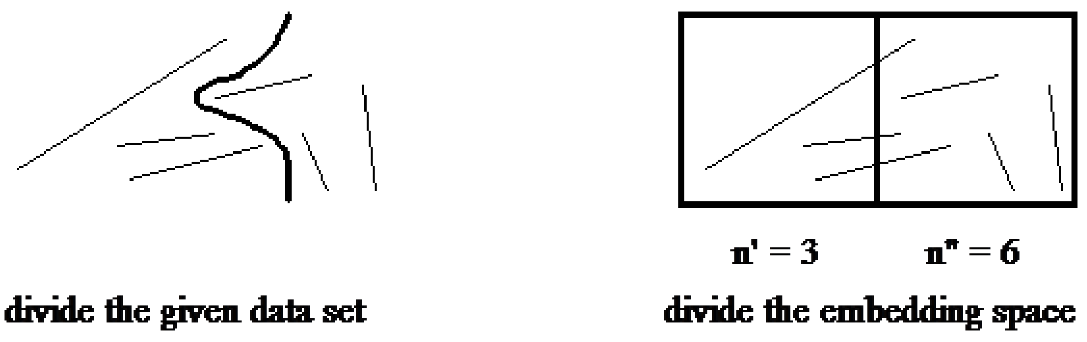

Exhibit 25.1: Two ways of applying divide-and-conquer to a set of objects embedded in the plane.

In the first case, we hope for a separation into subsets S_1 and S_2 that permits an efficient test whether any line segment in S_1 intersects some line segment in S_2. Exhibit 25.1 shows the ideal case where S_1 and S_2 do not interact, but of course this cannot always be achieved in a nontrivial way; and even if S can be separated as the figure suggests, finding such a separating line looks like a more formidable problem than the original intersection problem. Thus, in general, we have to test each line segment in S_1 against every line segment in S_2, a test that may take $\Theta(n^2)$ time.

The second approach of dividing the embedding space has the unfortunate consequence of effectively increasing our data set. Every segment that straddles the dividing line gets "cut" (i.e. processed twice, once for each half space). The two resulting subproblems will be of size n' and n", respectively, with n' + n" > n, in the worst case n' + n" = 2 · n. At recursion depth d we may have $2^d \cdot n$ subsegments to process. No optimal algorithm is known that uses this technique.

The key idea in designing an optimal algorithm is the observation that those line segments that intersect a vertical line L at abscissa x are totally ordered: A segment s lies below segment t, written $s <_L t$, if both intersect L at the current position x and the intersection of s with L lies below the intersection of t with L. With respect to this order a line segment may have an upper and a lower neighbor, and Exhibit 25.2 shows that s and t are neighbors at x.

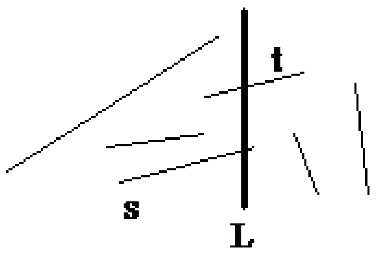

Exhibit 25.2: The sweep line L totally orders the segments that intersect L.

We describe the intersection test algorithm under the assumption that the configuration is nondegenerate (i.e. no three segments intersect in the same point). For simplicity's sake we also assume that no segment is vertical, so every segment has a left endpoint and a right endpoint. The latter assumption entails no loss of generality: For a vertical segment, we can arbitrarily define the lower endpoint to be the "left endpoint", thus imposing a lexicographic (x, y)-order to refine the x-order. With the important assumption of non-degeneracy, two line segments s and t can intersect at x_0 only if there exists an abscissa $x < x_0$ where s and t are neighbors. Thus it suffices to test all segment pairs that become neighbors at some time during a left-to-right sweep of L - a number that is usually significantly smaller than n · (n – 1) / 2.

As the sweep line L moves from left to right across the configuration, the order $<_L$ among the line segments intersecting L changes only at endpoints of a segment or at intersections of segments. As we intend to stop the sweep as soon as we discover an intersection, we need to perform the intersection test only at the left and right endpoints of segments. A segment t is tested at its left endpoint for intersection with its lower and upper neighbors. At the right endpoint of t we test its lower and upper neighbor for intersection (Exhibit 25.3).

The algorithm terminates as soon as we discover an intersecting pair of segments. Given n segments, each of

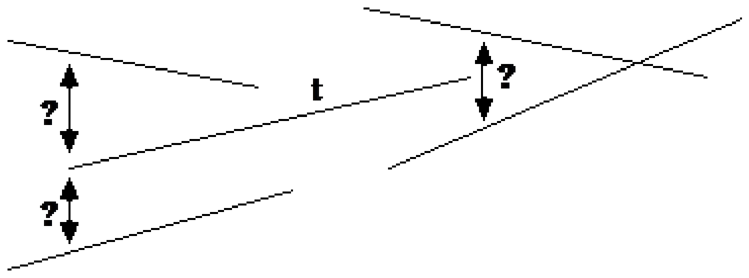

Exhibit 25.3: Three pairwise intersection tests charged to segment t.

which may generate three intersection tests as shown in Exhibit 25.3 (two at its left, one at its right endpoint), we perform the O(1) pairwise segment intersection test at most 3 · n times. This linear bound on the number of pairs tested for intersection might raise the hope of finding a linear-time algorithm, but so far we have counted only the geometric primitive: "Does a pair of segments intersect - yes or no?" Hiding in the background we find bookkeeping operations such as "Find the upper and lower neighbor of a given segment", and these turn out to be costlier than the geometric ones. We will find neighbors efficiently by maintaining the order $<_L$ in a data structure called a y-table during the entire sweep.

The skeleton: Turning a space dimension into a time dimension

The name *plane-sweep* is derived from the image of sweeping the plane from left to right with a vertical line (front, or cross section), stopping at every transition point (event) of a geometric configuration to update the cross section. All processing is done at this moving front, without any backtracking, with a look-ahead of only one point. The events are stored in the x-queue, and the current cross section is maintained by the y-table. The skeleton of a plane-sweep algorithm is as follows:

```
initX;  initY;
while  not emptyX  do  { e := nextX;  transition(e) }
```

The procedures 'initX' and 'initY' initialize the x-queue and the y-table. 'nextX' returns the next event in the x-queue, 'emptyX' tells us whether the x-queue is empty. The procedure 'transition', the advancing mechanism of the sweep, embodies all the work to be done when a new event is encountered; it moves the front from the slice to the left of an event e to the slice immediately to the right of e.

Data structures

For the line segment intersection test, the x-queue stores the left and right endpoints of the given line segments, ordered by their x-coordinate, as events to be processed when updating the vertical cross section. Each endpoint stores a reference to the corresponding line segment. We compare points by their x-coordinates when building the x-queue. For simplicity of presentation we assume that no two endpoints of line segments have equal x- or y-coordinates. The only operation to be performed on the x-queue is 'nextX': it returns the next event (i.e. the next left or right endpoint of a line segment to be processed). The cost for initializing the x-queue is O(n · log n), the cost for performing the 'nextX' operation is O(1).

The y-table contains those line segments that are currently intersected by the sweep line, ordered according to $<_L$. In the slice between two events, this order does not change, and the y-table needs no updating (Exhibit 25.4). The y-table is a dictionary that supports the operations 'insertY', 'deleteY', 'succY', and 'predY'. When entering the left endpoint of a line segment s we find the place where s is to be inserted in the ordering of the y-table by comparing s to other line segments t already stored in the y-table. We can determine whether $s <_L t$ or $t <_L s$ by determining on which side of t the left endpoint of s lies. As we have seen in chapter 14 in the section "Intersection", this tends to be more efficient than computing and comparing the intersection points of s and t with the sweep line. If we implement the dictionary as a balanced tree (e.g. an AVL tree), the operations 'insertY' and 'deleteY' are performed in O(log n) time, and 'succY' and 'predY' are performed in O(1) time if additional pointers in each node of the tree point to the successor and predecessor of the line segment stored in this node. Since there are 2 · n events in the x-queue and at most n line segments in the y-table the space complexity of this plane-sweep algorithm is O(n).

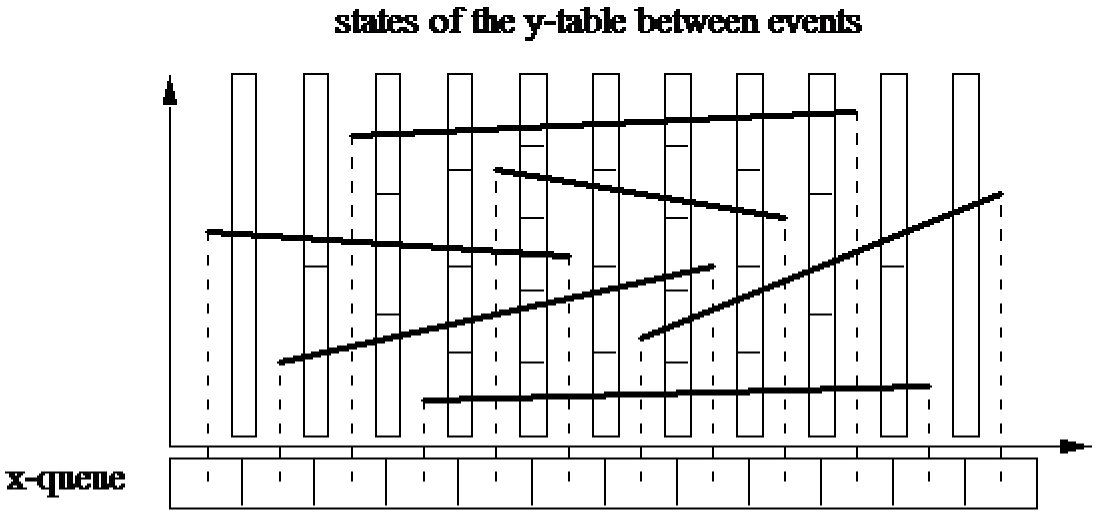

Exhibit 25.4: The y-table records the varying state of the sweep line L.

Updating the y-table and detecting an intersection

The procedure 'transition' maintains the order $<_L$ of the line segments intersecting the sweep line and performs intersection tests. At a left endpoint of a segment t, t is inserted into the y-table and tested for intersection with its lower and upper neighbors. At the right endpoint of t, t is deleted from the y-table and its two former neighbors are tested. The algorithm terminates when an intersection has been found or all events in the x-queue have been processed without finding an intersection:

```
procedure transition(e: event);
begin
  s := segment(e);
  if  leftPoint(e)  then  begin
    insertY(s);
```

```
        if  intersect(predY(s), s) or intersect (s, succY(s))  then
          terminate('intersection found')
      end
      else  { e is right endpoint of s }  begin
        if  intersect(predY(s), succY(s))  then
          terminate('intersection found');
        deleteY(s)
      end
    end;
```

With at most 2 · n events, and a call of 'transition' costing time O(log n), this plane-sweep algorithm needs O(n · log n) time to perform the line segment intersection test.

Sweeping across intersections

The plane-sweep algorithm for the line segment intersection test is easily adapted to the following more general problem [BO 79]:

Given n line segments, report all intersections.

In addition to the left and right endpoints, the x-queue now stores intersection points as events—any intersection detected is inserted into the x-queue as an event to be processed. When the sweep line reaches an intersection event the two participating line segments are swapped in the y-table (Exhibit 25.5). The major increase in complexity as compared to the segment intersection test is that now we must process not only 2 · n events, but 2 · n + k events, where k is the number of intersections discovered as we sweep the plane. A configuration with n / 2 segments vertical and n / 2 horizontal shows that, in the worst case, $k \in \Theta(n^2)$, which leads to an $O(n^2 \cdot \log n)$ algorithm, certainly no improvement over the brute-force comparison of all pairs. In most realistic configurations, say engineering drawings, the number of intersections is much less than $O(n^2)$, and thus it is informative to introduce the parameter k in order to get an output-sensitive bound on the complexity of this algorithm (i.e. a bound that adapts to the amount of data needed to report the result of the computation).

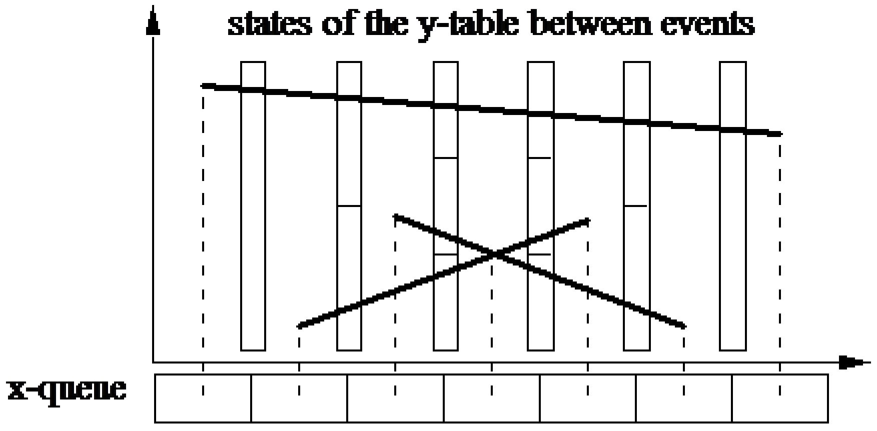

Exhibit 25.5: Sweeping across an intersection.

Other changes are comparatively minor. The x-queue must be a priority queue that supports the operation 'insertX'; it can be implemented as a heap. The cost for initializing the x-queue remains O(n · log n). Without further analysis one might presume that the storage requirement of the x-queue is O(n + k), which implies that the cost for calling 'insertX' and 'nextX' remains O(log n), since $k \in O(n^2)$. A more detailed analysis [PS 91], however, shows that the size of the x-queue never exceeds $O(n \cdot (\log n)^2)$. With a slight modification of the algorithm [Bro 81] it can even be guaranteed that the size of the x-queue never exceeds O(n). The cost for exchanging two intersecting line segments in the y-table is O(log n), the costs for the other operations on the y-table remain the same. Since there are 2 · n left and right endpoints and k intersection events, the total cost for this algorithm is O((n + k) · log n). As most realistic applications are characterized by k ∊ O(n), reporting all intersections often remains an O(n · log n) algorithm in practice. A time-optimal algorithm that finds all intersecting pairs of line segments in O(n · log n + k) time using O(n + k) storage space is described in [CE 92].

Degenerate configurations, numerical errors, robustness

The discussion above is based on several assumptions of nondegeneracy, some of minor and some of major importance. Let us examine one of each type.

Whenever we access the x-queue ('nextX'), we used an implicit assumption that no two events (endpoints or intersections) have equal x-coordinates. The order of processing events of equal x-coordinate is irrelevant. Assuming that no two events coincide at the same point in the plane, lexicographic (x, y)-ordering is a convenient systematic way to define 'nextX'.

More serious forms of degeneracy arise when events coincide in the plane, such as more than two segments intersecting in the same point. This type of degeneracy is particularly difficult to handle in the presence of numerical errors, such as rounding errors. In the configuration shown in Exhibit 25.6 an endpoint of u lies exactly or nearly on segment s. We may not care whether the intersection routine answers 'yes' or 'no' to the question "Do s and u intersect?" but we certainly expect a 'yes' when asking "Do t and u intersect?" This example shows that the slightest numerical inaccuracy can cause a serious error: The algorithm may fail to report the intersection of t and u, which it would clearly see if it bothered to look - but the algorithm looks the other way and never asks the question "Do t and u intersect?"

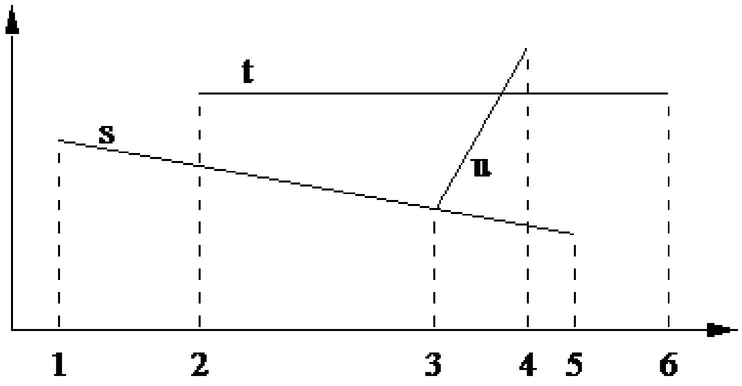

Exhibit 25.6: A degenerate configuration may lead to inconsistent results.

The trace of the plane-sweep for reporting intersections may look as follows:

1. s is inserted into the y-table
2. t is inserted above s into the y-table, and s and t are tested for intersection: No intersection is found
3. u is inserted below s in the y-table (since the evaluation of the function s(x) may conclude that the left endpoint of u lies below s); s and u are tested for intersection, but the intersection routine may conclude that s and u do not intersect: u remains below s

4. Delete u from the y-table
5. Delete s from the y-table
6. Delete t from the y-table

Notice the calamity that struck at the critical step 3. The evaluation of a linear expression s(x) and the intersection routine for two segments both arrived at a result that, in isolation, is reasonable within the tolerance of the underlying arithmetic. The two results together are inconsistent! If the evaluation of s(x) concludes that the left endpoint of u lies below s, the intersection routine *must* conclude that s and u intersect! If these two geometric primitives fail to coordinate their answers, catastrophe may strike. In our example, u and t never become neighbors in the y-table, so their intersection gets lost.

Exercises

1. Show that there may be $\Theta(n^2)$ intersections in a set of n line segments.
2. Design a plane-sweep algorithm that determines in $O(n \cdot \log n)$ time whether two simple polygons with a total of n vertices intersect.
3. Design a plane-sweep algorithm that determines in $O(n \cdot \log n)$ time whether any two disks in a set of n disks intersect.
4. Design a plane-sweep algorithm that solves the line visibility problem discussed in chapter 24 in the section "Visibility in the plane: a simple algorithm whose analysis is not" in time $O((n + k) \cdot \log n)$, where $k \in O(n^2)$ is the number of intersections of the line segments.
5. Give a configuration with the smallest possible number of line segments for which the first intersection point reported by the plane-sweep algorithm in chapter 25 in the section "Sweeping across intersections" is not the leftmost intersection point.
6. Adapt the plane-sweep algorithm presented in chapter 25 in the section "Sweeping across intersections" to detect all intersections among a given set of n horizontal or vertical line segments. You may assume that the line segments do not overlap. What is the time complexity of this algorithm if the horizontal and vertical line segments intersect in k points?
7. Design a plane-sweep algorithm that finds all intersections among a given set of n rectangles all of whose sides are parallel to the coordinate axes. What is the time complexity of your algo rithm?