AN OPTIMAL ALGORITHM FOR INTERSECTING LINE SEGMENTS IN THE PLANE

Bernard Chazelle
Herbert Edelsbrunner

CS-TR-148-88

April 1988

An Optimal Algorithm for Intersecting Line Segments in the Plane

BERNARD CHAZELLE
Dept. of Computer Science
Princeton University
Princeton, NJ 08544

HERBERT EDELSBRUNNER
Dept. of Computer Science
University of Illinois
Urbana, IL 61801

Abstract: The main contribution of this work is an $O(n \log n + k)$ time algorithm for computing all k intersections among n line segments in the plane. This time complexity is easily shown to be optimal. Within the same asymptotic cost our algorithm can also construct the subdivision of the plane defined by the segments and compute which segment (if any) lies right above (or below) each intersection and each endpoint. The algorithm has been implemented and performs very well. The storage requirement is on the order of $n + k$ in the worst case, but it is considerably lower in practice. To analyze the complexity of the algorithm we use an amortization argument based on a new combinatorial theorem on line arrangements.

Bernard Chazelle wishes to acknowledge the National Science Foundation for supporting this research in part under Grant CCR-8700917. Herbert Edelsbrunner is pleased to acknowledge the support of Amoco Fnd. Fac. Dev. Comput. Sci. 1-6-44862 and the National Science Foundation under Grant CCR-8714565.

1. Introduction

Intersecting a collection of line segments is one of the most fundamental tasks in computer graphics. It arises in windowing and clipping operations as well as in hidden-surface removal [19]. For scenes consisting of up to a few dozen vectors the naive method which consists of checking all pairs of segments for intersection is often as good a solution as any. Of course, the approach quickly breaks down as the number of segments becomes large. A popular method consists of having a vertical line step through each endpoint from left to right. A left (resp. right) endpoint gives rise to an insertion (resp. deletion) of the corresponding segment into (resp. from) the set of segments currently *active*. At each insertion the new segment is tested for intersection against all the other active segments. Although it does not take much imagination to see that this method can sometimes perform as poorly as the naive one, it remains a leading candidate because of its simplicity and its good behavior in practice.

Nearly a decade ago now, Bentley and Ottmann [3] proposed an ingenious sweep-line algorithm which is quite efficient across the board. Unfortunately the method is suboptimal; it is also considerably more difficult to implement than the solutions above. It is a fundamental algorithm, nevertheless. For this reason it will be the starting point of our investigation. If the input consists of n segments and the output is a collection of k intersecting segment pairs, the algorithm requires $O(n \log n + k \log n)$ time and $O(n + k)$ space. A clever observation by Brown [5] succeeded in trimming down the space requirement to $O(n)$.

In the wake of this result, researchers began to look at particular cases. Nievergelt and Preparata [20] considered the problem of merging two planar convex subdivisions: they showed that the resulting figure could be computed in $O(n \log n + k)$, where n and k are respectively the input and output sizes. Mairson and Stolfi [18] extended this result by showing that the regions in the original subdivisions need not be convex. More generally, a collection of n nonintersecting red segments can be merged together with n nonintersecting blue segments in $O(n \log n + k)$ time. Also, Guibas and Seidel [15] showed that $O(n + k)$ time is sufficient to merge two convex subdivisions. Progress on the general intersection problem was achieved in (Chazelle [6]), where an algorithm with a running time of $O(n \log^2 n / \log \log n + k)$ was described. Unlike most intersection algorithms this one had the particularity of *not* following a sweep-line approach. Instead, it used a hierarchical subdivision of segments and the key notion of a *hammock* (more on this exotic terminology below).

The main contribution of the present paper is an $O(n \log n + k)$ time algorithm for the general segment intersection problem; recall that n denotes the number of segments and k the number of pairwise intersections. (We insist on the word pairwise, as for example 4 segments intersecting in one point give 6 pairwise intersections.) The running time of the algorithm is easily shown to be optimal. The storage requirement is $O(n+k)$. It remains open whether the intersections can all be computed in optimal time and $O(n)$ space. One last piece of advertising about our algorithm: if so desired, it can be made to compute within the same time and space bounds the so-called *vertical map* of the set of segments. This is the planar subdivision obtained by drawing a rectangular frame around the segments and connecting by a vertical line segment every endpoint to the edges immediately above and below (Fig.1). It is a simple exercise to show that the number of vertices and edges of the vertical map is linearly related to n and k.†

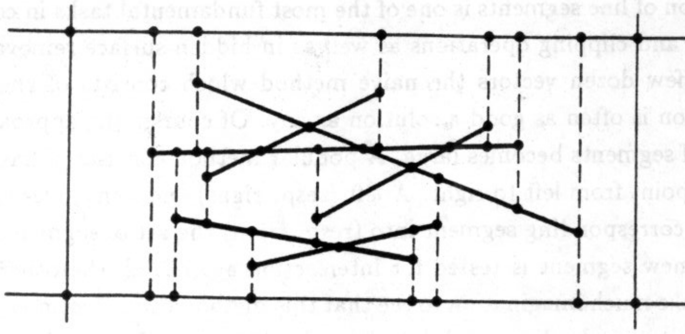


Figure 1. The vertical map of a set of n line segments enclosed in a rectangular frame.

The algorithm is not nearly as simple as one might wish (but after all we do not know of any *simple* algorithm, no matter how slow, for computing the vertical map of a set of segments). The algorithm is only slightly more complicated than Bentley and Ottmann's algorithm [3]. Our implementation consists of about 1500 lines of C-code (discounting driver and I/O routines). The program performs very well in practice. One of its advantages over simpler methods is that for given values of n and k all inputs perform within the same order magnitude. This means that no matter how contrived the input the algorithm will never fail us by slowing down dramatically. We do not know of any simple method for which this most desirable property is true.

The intersection algorithm is not based on any one or two ideas. It makes use of a large number of concepts which have gained recent prominence in the design of geometric algorithms. Even though not all of the following notions enter the final composition of the algorithm, all are crucial to its development in one way or the other: *sweep-line, k-set, segment tree, finger tree, dovetailing, amortization, functional data structuring, topological sweep.* For example, the algorithm does not use segment trees, finger trees, or topological sweep. Yet, it is a big help to know these concepts in order to understand the algorithm. The reader will appreciate these fine points better as he plows his way through the paper. The amortization argument used to prove the claimed time bound is based on a new combinatorial theorem for line arrangements (see Capsule 10).

† First, note that if our goal is to derive upper bounds we can assume that the segments are in general position. Every intersection is a vertex and so is every endpoint. Furthermore, every endpoint gives rise to two more vertices when it is connected to the edges immediately above and below. We also get four vertices from the covering frame. This adds up to $k + 6n + 4$ vertices. Every intersection has degree 4, the four corners of the frame have degree 2, and the other vertices have degree 3. This gives us $2k + 9n + 4$ edges. Using Euler's relation, we now derive that the number of regions (inside the frame) is $(2k + 9n + 4) - (k + 6n + 4) + 1 = k + 3n + 1$. Note that the intersection points can be connected to the edges above and below by vertical segments without asymptotically increasing the time or space requirements of the intersection algorithm.

2. An Intersection Algorithm in Twenty-Two Easy Pieces

All in all, the algorithm is quite simple but, as the keywords given in the previous paragraph suggest, it is made of many pieces. To lay it all out in one block may give the reader indigestion. More seriously, it may convey an unpleasant impression of black magic. We believe that understanding the whys and wherefores and not just the whats of each piece will make the reading of this paper much easier. For this reason we will describe the algorithm as the end result of successive transformations applied to a simple starting solution. Our presentation may seem to proceed in roundabout ways, presenting ideas here that are abandoned later, digressing on why such and such approach would not work, etc. But in the end we believe that although this strategy makes for a longer paper it will actually shorten the reading time. The intersection algorithm is really quite simple when presented in a progressive didactical fashion. This is what we have tried to do in this paper.

To give the reader something to hold on to while plowing through our discussion, let us just say that the main idea of the algorithm is to cut off the segments in preprocessing to satisfy a certain normalization criterion. Then the sweep-line proper goes into action. Rather than maintaining the cross-section and nothing else, we also include the entire scene formed on the right of the sweep-line by the segments currently active in the cross-section. This is to help us predict a bit ahead of time what is coming up. This section consists of twenty-two short capsules, every one of which introduces a few simple ideas.

Capsule 1. A Classical Intersection Algorithm

"Thursday's child has far to go"

Bentley and Ottmann's solution to segment intersection is a model illustration of the sweep-line paradigm. We will make it the starting point of our investigation. While a quick review of the algorithm is in order, we will still assume that the reader is familiar with this classical algorithm. Let $S = \{ s_i \mid 1 \le i \le n \}$ be a collection of n line segments in the plane. Inherent to the notion of sweep-line is the existence of a Cartesian system of reference (O, xy).

For the ease of exposition we shall assume that no two endpoints have the same x or y coordinates. This, in particular, applies to the two endpoints of the same segment, and thus rules out the presence of vertical or horizontal segments. Similarly, we shall assume that the intersection of two segments $s_i \cap s_j$ $(i < j)$, if nonempty, consists of a single point. Finally, we wish to exclude situations where three or more segments run concurrently through the same point. Note that in practice these assumptions are grossly unrealistic. Your "average" graphics scene (whatever that means) is likely to contain a vertex adjacent to many edges, not to mention horizontal or vertical edges. Our rationale is that the key ideas of the algorithm are best explained without having to worry about *special* cases every step of the way. Relaxing the assumptions is very easy (no new ideas are required) but tedious. That's for the theory. Implementing the algorithm so that the program works in all cases, however, is a daunting task. There are also numerical problems which alone would justify writing another paper. Following a venerable tradition, however, we shall try not to worry too much about it. We feel only marginally guilty doing so because we can always fall back on the techniques exposed in (Edelsbrunner and Mücke [12]) to remove singularities in geometric computation.

Imagine sweeping the plane by moving a vertical line L (called the ***sweep-line***) from left to right starting at the leftmost endpoint and stopping at the rightmost one. At any time during the sweep L provides a *cross-section* of the set of segments. As a geometric entity the cross-section changes all the time while L sweeps the plane. Combinatorially, however, it consists of a well ordered sequence of segments which changes only at discrete intervals: the cross-section may gain or lose an element, or two consecutive elements may swap places. Let us define the *schedule* of the sweep as the set of positions of L where the (combinatorial) cross-section undergoes some changes. There are two types of *events* in the schedule: $2n$ events are contributed by endpoints, while k events originate from intersections between segments.

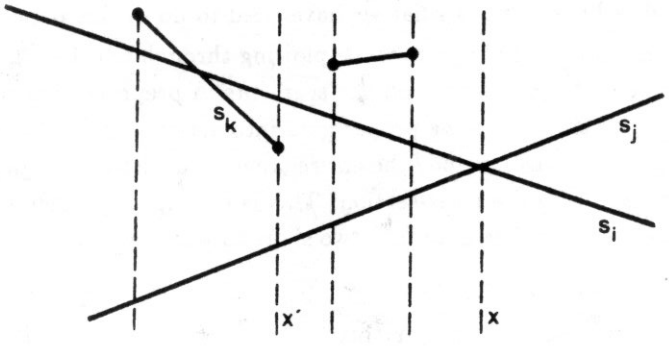


Figure 2. We delete segment s_k from the cross-section when the vertical sweep-line process passes x', the x-coordinate of its right endpoint. As we do this, segments s_i and s_j become adjacent and we discover that they are about to intersect.

Bentley and Ottmann's algorithm is a direct simulation of the sweep. One difficulty is that the schedule is not available at the outset. (If it were, then of course, the problem would be solved and no simulation would be needed.) All we can do is to start with a partial schedule containing all the events of type *endpoint*. Then the idea is to build the schedule on the fly, making sure that enough of it has been computed whenever the next event is being requested. To make things concrete, let us represent an event of type ***intersection*** as a triplet (x, i, j), where x is the abscissa of the intersection $s_i \cap s_j$. A simple observation ensures the feasibility of this scheme: any event (x, i, j) of type *intersection* must be preceded by a ***witness***, that is, an earlier event (of abscissa $x' < x$) right after which the cross-section contains the segments s_i and s_j in adjacent positions (Fig.2). The idea is then to keep checking all new adjacencies in the cross-section and insert the newly discovered entries in the current schedule. Bentley and Ottmann show that a proper choice of data structures leads to an $O(n \log n + k \log n)$ running time for conducting the entire simulation. Let us not worry about the underlying data structures at this point.

Capsule 2. A Simpler Schedule

"This is the way we sweep the house"

An inherent feature of the Bentley-Ottmann algorithm is to report the intersections in sorted order from left to right. (That is after all what the idea of a schedule is all about.) Although there might be reason to believe that this sorting can be performed in $O(n \log n + k)$ time, it is a difficult open problem and choosing that route for improvement might be ill-fated. Indeed, the bad news is that the (open) problem of sorting $X+Y$ in quadratic time (Fredman [13]) is a special case of our sorting problem. Rather, we give up once and for all the idea of sorting the intersections and redefine a *schedule* as the list of $2n$ events of type *endpoint*. In this way, it will be easy to compute the schedule ahead of time; moreover it will never change. So, why didn't we think of it earlier, one might ask?

Well, of course, there are a few snags in our way now. Whereas formerly a schedule interval could witness only a modest change in the current cross-section (an insertion, a deletion, or a swap), hell may break loose now (so to speak). Indeed, most of the action might take place during two consecutive events on the schedule. Figure 3 illustrates this point. The scene between L and L' is called a *hammock* in [6]: it is a subdivision of the closed vertical strip between L and L' created by various segments joining the two lines in question. A hammock is a little like a line arrangement [10]. For example, with our simplifying assumptions, all the vertices which are not on the boundary of the hammock have degree 4, that is, are adjacent to four edges. Unlike a line arrangement, however, the size of a hammock may be fairly unrelated to the number of segments used for its construction. If m is the number of segments then the number λ of degree-4 vertices may lie anywhere between 0 and $m(m-1)/2$.

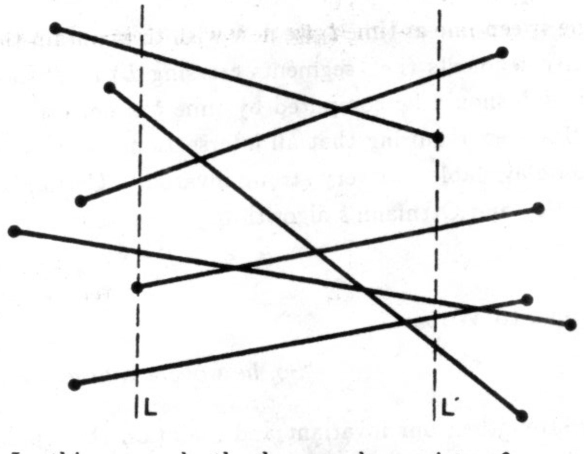


Figure 3. In this example the hammock consists of $m = 6$ segments (clipped outside the strip between L and L') and the number of intersections in the hammock is $\lambda = 6$.

As is shown in (Chazelle [6]) it is possible to construct the hammock between L and L' in $O(m + \lambda)$ time. By constructing it, we mean computing the vertices with all adjacency information.

Using this approach to handle the events of the schedule we may have to spend on the order of $n+k_i$ time between two consecutive schedule events, where $\sum_i k_i = k$. This leads to a nontrivial quadratic algorithm —a big step forward, indeed! Figure 4 tells us what went wrong: many segments might end up being looked at numerous times with no intersections on which to amortize the costs. This suggests a crucial modification of our strategy. Instead of building hammocks one at a time from left to right, we should build *as much as we can* while looking at a given segment so as to avoid having to go back to it later.

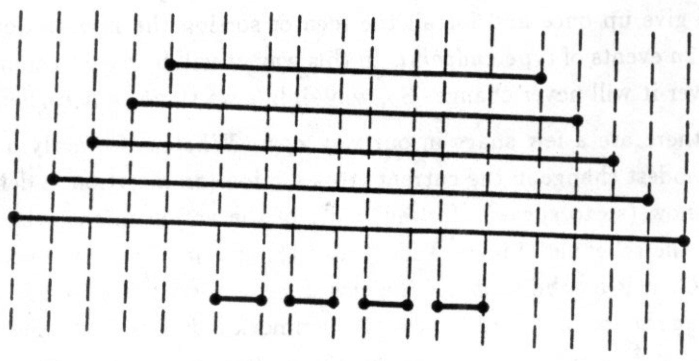


Figure 4. Too many segments extending over too many strips defined by two adjacent events.

In other words, if L is the sweep-line at time t, we now wish to maintain the following invariant: all intersections between active segments (i.e., segments crossing L) must have been computed at time t. That everything left of L should be computed by time t is nothing surprising: that is the whole idea of a sweep-line. However, requiring that all intersections between active segments right of the sweep-line should also be available is a very strong invariant. Certainly much stronger than anything we have seen in Bentley and Ottmann's algorithm.

Capsule 3. From Hammocks to Webs

"So, he wove a subtle web in a little corner sly"

While we are at it, why not strengthen our invariant and maintain the whole *web* formed by the active segments? Figure 5 (minus the dashed segment) depicts a web. This generalizes the notion of a hammock: if A is the set of active segments (the segments of S that intersect the current sweep-line L) and L^+ is the closed halfplane lying to the right of L, then the web of L is the subdivision of L^+ formed by the set of active segments $\{ s \cap L^+ \mid s \in A \}$. To ensure that the web is a convex subdivision we add *stops* to dangling edges. To do so, we draw a vertical segment passing through each right endpoint: the segment should be of maximum length but it should not create any vertices of degree 4.

Assume that the web of the sweep-line L has been computed. How difficult is it to add a new segment into it? Figure 5 illustrates the problem ahead. To add the dashed line into the web, we can follow it from left to right and integrate it into the web-structure as we move along. To begin with, we locate the edge of L that contains the left endpoint of the new segment (let us assume that we know how to do that). This gives us the first region of the web to be crossed by the segment. We compute its exit point (if any) by walking around the region in, say, clockwise order, starting from the entrance point. Using a standard adjacency-list representation of the web, a data structure which we shall call the *web-structure* throughout, there should be no major problem doing so. The web-structure can be thought as a collection of records with data and pointer fields to support run-of-the-mill operations such as walking around a region of the web in both directions, finding edges adjacent to a given one, etc. Many data structures are available for this purpose, e.g., *winged-edge* (Baumgart [1]), *DCEL* (Preparata and Shamos [21]), *quad-edge* (Guibas and Stolfi [16]). We will remain purposely vague about the web-structure until the 20th capsule of this section, where implementations issues are discussed. All we need to know at this point is that the web-structure is *edge-based*. This means that its basic records are associated with edges. Regions and vertices are not represented explicitly in the web-structure; they are defined implicitly. The reason for doing so is that it is simpler that way.

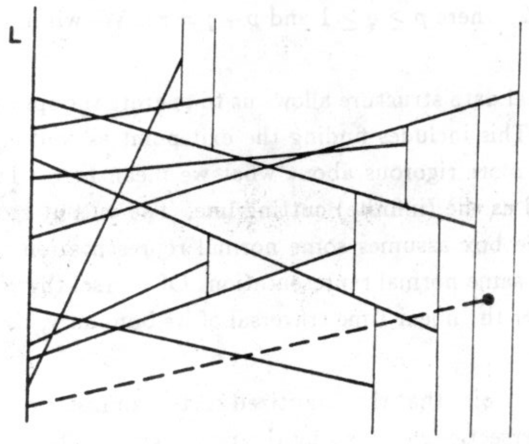


Figure 5. The solid lines define the web. When we add the dashed line segment we cut three vertical stops.

Anyway, let us return to our discussion. Once the exit point has been reached, we iterate on this process across the next region, and so forth. Either the segment extends completely to the right of the web (the easy case), or it stops somewhere in the middle of a region (as in Figure 5). We can detect this case by keeping track of x-coordinates during the traversal of the regions. We have omitted a few details, but none which require a great deal of creativity to fill in.

It is unclear at this point how efficient this method will turn out to be. Certainly, traversing boundaries might be costly. More worrisome are those vertical *stops*, which we might keep on cutting through, as for example in Figure 5. These intersections are *not* part of the output and a little thinking shows that their number could extend far beyond $O(n \log n + k)$. Obviously, there is no point pursing the present line of reasoning until we show how to avoid this pitfall. How about assuming that it never happens? Here is a hypothetical situation where the difficulty is easily overcome. Assume that all the segments in S have their left endpoints right on the line L (forget for a moment that this contradicts our assumptions). We call this the ***halfsegment case***. We avoid the problem of cutting vertical stops if we make sure that the segments are inserted in an order corresponding to the right-to-left order of their right endpoints.

Capsule 4. The Complexity of the Halfsegment Case

"I spent a penny of it, I spent another"

Let us examine in detail the complexity of inserting a new segment. If we keep the left endpoints on the line L in some balanced dynamic search tree (say, a red-black tree [14], [22], [23]), it is easy to get started in $O(\log n)$ time. To simplify our task, let us ignore for a while the thorny case of the last region traversed. Recall that it might be cut into three parts, instead of only two. Each of the other regions traversed starts with a certain number of edges, say, $m \geq 3$, and is split into two regions of sizes $p+2$ and $q+2$, where $p \geq q \geq 1$ and $p+q = m$. We will now request an act of faith on the part of the reader.

Suppose that some magical data structure allows us to operate the split operation in $O(1+\log q)$ time (as opposed to $O(m)$). This includes finding the exit point as well as partitioning the region accordingly. Le us try to be more rigorous about what we mean here. The input to the magical box is the region itself as well as the (infinite) cutting line. The output consists of the two regions resulting from the cut. If the box assumes some normal representation for the regions then the output should be given in the same normal representation. Of course, the representation of a region by a black box should allow for the linear time traversal of its boundary, the cutting edge should be available in constant time, etc.

Under these conditions we claim that the amortized cost of an insertion is equal to $O(\log n + k_i)$, where k_i is the number of intersections created by the new segment. Note that this would allow us to contruct a web of size k in optimal $O(n \log n + k)$ time. But instead of rejoicing prematurely, let us try to justify our claim. We use an amortization argument using *credits* (any nonfloating currency would work just as well). Recall that the cost incurred while traversing and splitting a region of size $p+q$ $(q \leq p)$ is $O(1+\log q)$. Let us agree that $1+\log q$ credits have enough worth to pay for this cost.† We assume that credits are real numbers, so one should not worry about transcendental numbers of credits or things like that.

As a credit invariant we require that each region of m edges should hold exactly $m - \log m$ credits. A neat side-effect of the insertion order is that any region traversed gives rise to at least one new (genuine) intersection. We can then safely grant ourselves 6 credits for each region traversed.

† All logarithms are taken to the base 2, unless specified otherwise.

The accounting for a region goes as follows. We start out with a total of $m - \log m + 6$ credits to spend. To maintain the credit invariant we need $p + 2 - \log(p+2)$ credits for one region and $q + 2 - \log(q+2)$ for the other. Furthermore, we must pay for the split itself, which amounts to another $1 + \log q$. The feasibility of the scheme depends on the sign of the quantity

$$\Delta = m - \log m + 6 - (p + 2 - \log(p+2)) - (q + 2 - \log(q+2)) - (1 + \log q).$$

Because $q \le p$, we have

$$2^{\Delta} = \frac{2(p+2)(q+2)}{(p+q)q} \ge \frac{(p+2)(q+2) + (q+2)^2}{pq + q^2} > 1,$$

which shows that $\Delta > 0$. This proves that the construction of the web can be completed in $O(n \log n + k)$ operations. Of course, to be honest, there remain two stones unturned:

(i) the treatment of terminating regions;

(ii) the *magical* algorithm for splitting regions in logarithmic time.

Capsule 5. Completing an Insertion

"With a hop, step, and a jump"

This is how we deal with terminating regions. When we enter a new region we will just pretend that the segment to be inserted extends completely past the region (for example, by stretching the right endpoint to infinity). Then, in $O(1 + \log q)$ time we can split the region into two subregions of size $p + 2$ and $q + 2$, which is consistent with what we did before (Fig.6). Once the exit point has been found it is immediate to determine whether the region is terminating or not. If it is, then we start two traversals of the boundary, one clockwise and the other counterclockwise, both initiating from the exit point. The purpose of these walks is to find the edge right above (resp. below) the right endpoint of the segment. Inserting the vertical stops is now child's play.

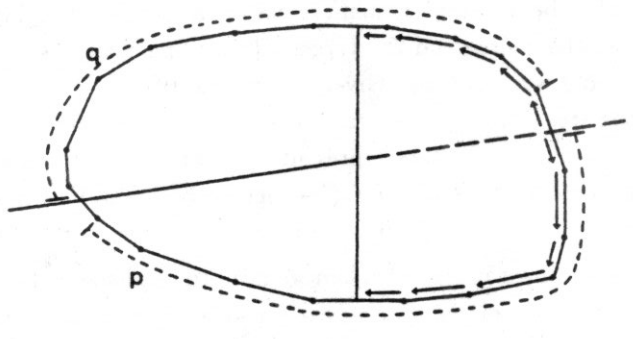


Figure 6. A terminating region is cut into four pieces, rather than three. This causes no harm because the two right regions will never again be traversed. It is important that we do the second step (finding the endpoints of the stop) from right to left.

How does our previous accounting argument fare with this new method? Quite well, indeed. We now have a 4-way split of an m-edge region into two left regions of size $a+3$ and $b+3$ and two right regions of size $c+3$ and $d+3$, where $a+b+c+d=m$. Because of the order of insertions the two right regions will never be used any more, so we can leave them alone and not worry about their credit invariants. Also for this reason we can give ourselves an extra $O(1+c+d)$ credits. Observing that the sizes of the left regions can exceed by at most one edge what they would have been without the vertical cuts, the previous accounting will leave us just short of a constant number of credits as far as the first split is concerned. These missing credits can be taken from our pool of extra credits. Now what about the last two splits and the walks? We can generously estimate the combined costs to be $O(1+\log(1+c)+\log(1+d)+c+d)$. The extra credits will also take care of this easily. This concludes our discussion of terminating regions.

Capsule 6. How to Cut Up a Region Fast

"With rings on her fingers and bells on her toes"

The time has now come to discuss the magical method by which an exit point can be found so rapidly. Two key ideas are: (i) looking for the exit point by binary search; and (ii) using finger trees to take advantage of uneven splits. To simplify the discussion we will break the problem into simpler components.

Let us say a few words about a special type of finger trees developed in (Hoffman et al [17]). A *homogeneous finger search tree* will be, for our purposes, a red-black tree augmented with special pointers. Each leaf stores a distinct key, and as is usual with search trees, a left-to-right scan through the nodes will show keys appearing in nondecreasing order. What makes this red-black tree special is that it supports fast access in the vicinity of any item. If you are holding a pointer to a leaf and you are searching for a key stored in a leaf which is at a distance d away from the leaf you know (i.e., stored d leaves away), then the search will take only $O(1+\log(\min\{d,n-d\}))$ time. Notice that if the list of leaves is thought of as being circular then the cost of a search is roughly the logarithm of the shortest distance between the source and the target. These finger trees support an even more drastic operation: the so-called 3-way splitting. Given pointers to two leaves, the operation involves removing all the items of the tree between the two leaves (inclusive) and form two new trees, one representing the removed sublist and the other representing the remaining items. There again the time to perform this operation can be kept within $O(1+\log(\min\{d,n-d\}))$, where d is the number of leaves removed.

To bring these finger trees to bear on our problem we need to implement the search for the exit point as a binary search. Given a region M of m edges, let us artificially partition it into its lower and upper parts. These two parts share a common edge joining the leftmost and rightmost vertices. Because of symmetry we can restrict our attention to, say, the upper part. Let $v_1,\ldots,v_u$ be the vertices of this upper part in order from left to right. Suppose that we have built a finger tree whose leaves store the v_i's in the same order. Let us now turn to the segment to be inserted. If the point of entrance into the upper part lies on v_1v_u then the exit point will be found on some edge v_iv_{i+1} $(1\le i\le u-1)$. In constant time we can determine whether any v_j lies on the segment or, if not, on which side. Therefore, just as intended, the tree lends itself to binary search. The exit point will be found in $O(1+\log(\min\{i, u-i\}))$ time, and the ensuing split will be completed within the same asymptotic complexity.

What happens now if the entrance point lies on some edge $v_i v_{i+1}$ $(1 \le i \le u-1)$? Since an exit point on $v_1 v_u$ can be detected and computed in constant time, let us suppose that it lies on $v_j v_{j+1}$ $(1 \le j \le u-1)$. Note that because the exit point always lies to the right of the entrance point we must have $i < j$. The binary search alluded to in the previous paragraph will (almost) do the job here. The only difference comes from the equivalence between a vertex v_ℓ being above or below the segment and branching left or right in the tree. We say that the vertex v_ℓ associated with the current node in the tree is *greater* than the exit point (i.e., branch left) if and only if v_ℓ is below the segment *and* $\ell > i$. The time complexity will be $O(1+\log(\min\{j-i, u-j+i\}))$ time, which also includes the split of the upper part.

Putting the pieces together involves totalling up the costs of splitting both lower and upper parts. Of course, all the work may take place in only one of the two parts, in which case our claim on the performance of the magical search is easily vindicated. If both parts must be worked on, the complexity will be of the form $O(\alpha(p_1, p_2, q_1, q_2))$, where

$$\alpha(p_1, p_2, q_1, q_2) = 1 + \log(\min\{p_1, q_1\}) + \log(\min\{p_2, q_2\})$$

and $p_1, q_1, p_2, q_2 \ge 1$, $p = p_1 + p_2$ and $q = q_1 + q_2$. Without loss of generality, let us assume that $q \le p$. We have

$$\alpha(p_1, p_2, q_1, q_2) \le 1 + \log q_1 + \log q_2 = 1 + \log(q_1 q_2) < 1 + \log((q_1+q_2)^2) < 2(1+\log q).$$

This proves our claim that the complexity of a split is $O(1+\log(\min\{p, q\}))$. Thus, all is well that ends well. We have not exactly achieved the goal we were pursuing, but we do have a nice result. This deserves a lemma.

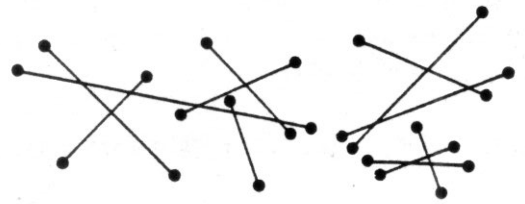
Lemma 1. [The Halfsegment Case] – *Given n line segments in the plane passing through a (known) common line, all k pairwise intersections among them can be computed in $O(n \log n + k)$ time and $O(n+k)$ space.*

Capsule 7. A Sweep Tree and a Little Help from an Oracle

"Eleven, twelve, dig and delve"

We surely have cleared a big hurdle, but we must face reality. The halfsegment case might be somewhat restrictive. Collections of segments often *do not* pass through a common line. As we remarked earlier, however, our scheme will work as long any new segment the sweep-line encounters has its right endpoint to the left of all the active right endpoints. We call this desirable feature the *clearance property*: technically, the property ensures that given any two segments in S either their projections on the x-axis do not intersect at all —except possibly at the endpoints— or one contains the other. Figure 7(a) suggests that the clearance property is not nearly as strong as one might fear. Figure 7(b) illustrates a typical violation of that property. Whatever its restrictive nature, a redeeming aspect of the clearance property is that any set of segments can always be made to satisfy it. Simply cut the segments along the vertical lines passing through all the endpoints. Of course, the fact that this may produce a quadratic number of subsegments should temper our enthusiasm. However, there always remains the prospect of a better cutting strategy, so let us just pursue this idea to see how far we can go. In the following, until specified otherwise, we shall assume that S satisfies the clearance property.

Imagine the sweep-line in action. For concreteness, we assume that at all times the sorted sequence of active segments is represented by a red-black tree.† Also, the web is supposed to be fully available as a web-structure. Obviously, the tree of active segments, which we call the *sweep tree*, will serve as an entrance door into the web, so we must provide its nodes with pointers to the corresponding edges of the web.


(a) Twelve segments which satisfy the clearance property.

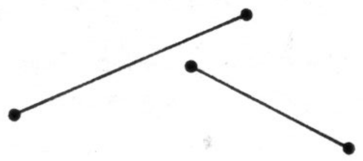

(b) It takes only two segments to violate the clearance property.

Figure 7

As it turns out it is convenient to let the web-structure grow beyond what is strictly necessary to represent the web. What do we mean by this? Figure 8 depicts a situation where upon reaching L the next move should be to delete segment s from the web. This can be done by a local transformation of the web. We wish to argue that this work is wasted. Why bother with it? Can't we just agree that everything in the web-structure left of the sweep-line is just garbage left behind? This may not help storage but it will save us work. In other words, while the web grows and shrinks all the time, the web-structure only grows. The invariant will be that clipping the current web-structure

† There are two kinds of red-black trees: one where keys are stored at the leaves, and another where keys are stored at all the nodes. We choose the latter. Not that we have much to justify this choice at this point. Later, however, we will find a very important reason. So, patience!

by removing everything that lies left of the sweep-line should give us exactly the current web. By extension (and only when ambiguity could not cause trouble) we shall refer to the web as the planar subdivision represented by the web-structure. For example, we will speak about edges of the web to the left of the sweep-line as well defined entities.

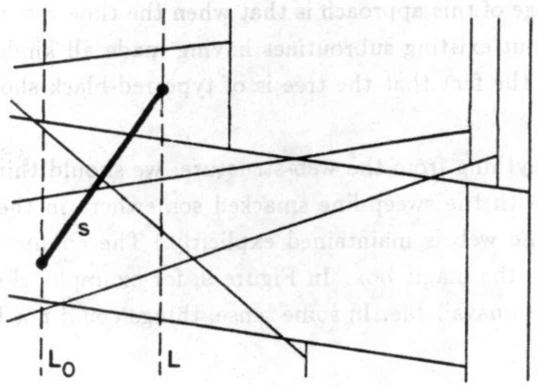


Figure 8. When the sweep-line reaches the right endpoint of s, we abstain from deleting s from the web-structure.

Of course, although we do not modify the web-structure when the sweep-line reaches the right endpoint of a segment, we must still update the sweep tree accordingly. For one thing, the node associated with the segment s must go. Worse than that, the vertical order among active segments may get all upset by the move from L_0 to L. Reconstructing the proper sweep tree seems as hopeless as our earlier attempts at maintaining hammocks through a sweep. So, was the idea of the sweep tree totally unproductive? Not really, we will see later. But for the time being, let us use the magical trick again, and assume that some benevolent oracle will do the work for us at no charge. Here is what our *magic box*, as we shall call this oracle, will do for us. As the sweep-line moves from one endpoint to the next (as directed by the schedule) the magic box will update the entire sweep tree, including the pointers into the web-structure. Furthermore, if we present the magic box with a point on the sweep-line it will tell us (presumably by consulting the sweep tree) which edges of the web (if any) lie immediately above and below the point. Most important, *all* the work performed by the magic box is done *free of charge*. With this wonderful device in our hands, let us look at the insertion of a new segment. There are a few important points to make right now, so let us take a deep breath and open a special section on insertion.

Capsule 8. A Closer Look at the Insertion Process

"Put your right foot in and shake it all about"

The magic box is the guardian of the sweep tree. It is also its exclusive operator. As long as the magic box is around we will never be allowed to use the sweep tree ourselves. The reason is that the magic box may decide to implement the sweep tree in such a bizarre way that we would not be able to tell what is what if we were allowed to peek at it. More formally, the magic box is an abstract data type whose functional specifications have been laid out, and nothing should be assumed about its implementation. The advantage of this approach is that when the time comes to implement the box we will not have to worry about existing subroutines having made all kinds of assumptions about the sweep tree. For example, the fact that the tree is of type red-black should now be completely irrelevant.

Since we do not delete anything from the web-structure, we should think of it as representing a planar convex subdivision with the sweep-line smacked somewhere in the middle of it. No link between the sweep-line and the web is maintained explicitly. The connection between these two entities is entirely provided by the magic box. In Figure 9, for example, the intersections between L and the edges of the web are unavailable. In some sense, things could not be simpler, could they?

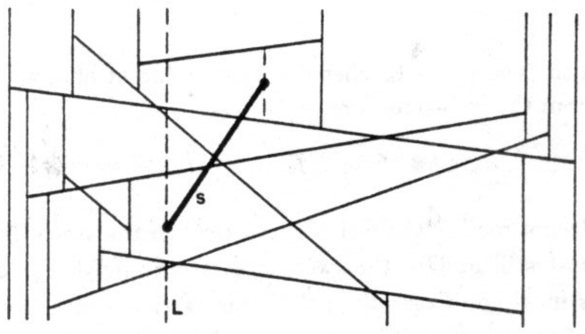


Figure 9. Inserting a new segment.

Let us see about inserting a new segment. Its left endpoint lies on the sweep-line, while by virtue of the clearance property, its right endpoint falls to the left of all the right endpoints in the web (Fig.9). Recall that these right endpoints are those vertices adjacent to vertical stops. Our first step will be to call on the magic box to find out which edges lie above and below the left endpoint of the segment. We are then ready to follow the instructions which we wrote out for the halfsegment case. Are things really that easy? There are two minor details to which we should perhaps pay attention.

1. Since the sweep-line does not coincide with any boundary of the web, the left endpoint of the new segment will be left dangling if we do not provide additional vertical stops. The symmetry between left and right endpoints provides an easy fix: just handle the traversal of the first and

the last regions in the same way. Given the edges of the web above and below the left endpoint, we begin by connecting them via a vertical stop (Fig.10). This puts us at the starting stage of the halfsegment case. We have already studied this in detail, and the reader would surely feel insulted if we told him what to do next.

2. The second point to make is not a real problem. It is just an observation to make sure that the reader appreciates all the subtle differences between what faces us now and the much easier halfsegment case. Many regions visited during the insertion may lie partly to the left of the sweep-line (Fig.9). Since the connection between the sweep-line and the web is not enforced explicitly this should not surprise us. However, it changes our perspective a little. Before, if you recall, the actual web was only what lay to the right of the sweep-line. The fact that more was actually stored in the web-structure was viewed as a mere product of our laziness. Now, on the contrary, we are saying that what lies to the left of the sweep-line might be of vital importance to the algorithm itself and should, by virtue of the extension mentioned earlier, be made part of the web itself.

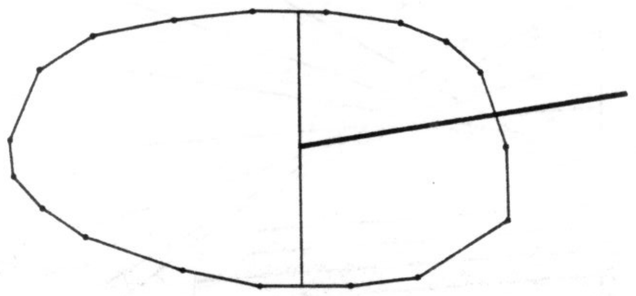


Figure 10. The left endpoint is connected by a vertical segment to the edges immediately above and below. Then the segment is traced from left to right and integrated into the web just as in the case of halfsegments.

The intersection algorithm is complete. The schedule consists of $2n$ events, half of which (the right endpoints) involve absolutely no work on our part. It does not take great insight to show that the complexity analysis of the halfsegment case carries over to this algorithm almost verbatim. To summarize, we have in our possession a full-fledged algorithm for intersecting n line segments in $O(n \log n + k)$ time. Well, of course, modulo a couple of provisos: the clearance property and the magic box. Before attacking these two problems, let us straighten out one minor item. This will give the reader a well deserved break.

Capsule 9. Fingers Are Just Too Complicated

"Says Simple Simon to the pieman"

Fingers are nice and cute but (ironically) impractical. What programmer would seriously consider implementing an algorithm which calls for the concurrent maintenance of finger trees when these could easily number in the thousands? To add insult to injury, most of these finger trees would likely be very small in practice, and price would be paid for overhead with little or no gain at all. Of course, from a theoretical perspective, we could stick to our fingers and the remainder of the algorithm would work just as well. From a practitioner's viewpoint, however, the only alternative seems to be going back to the sequential traversal of the regions. Suppose that we choose to walk clockwise from the entrance to the exit. Figure 11 clearly shows that this is not exactly a well inspired move. Inserting segments labeled 1 through 5 would lead to the repeated traversal of the same sequence of consecutive edges —something too horrid to be even mentioned.

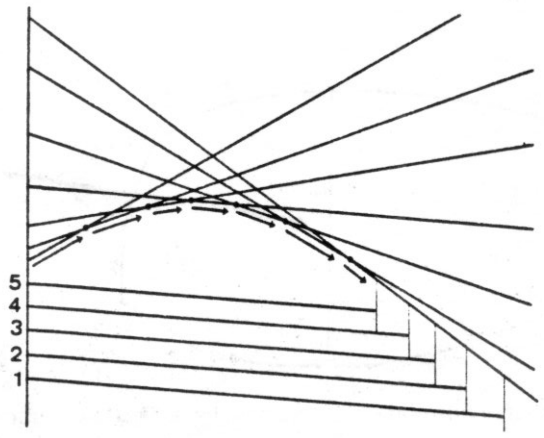


Figure 11. As we insert segments 1, 2, 3, 4, and 5, in this order, we repeatedly traverse a long chain of edges. Even in the halfsegment case this may lead to an $\Omega(n\sqrt{n})$ time algorithm, even though the number of intersections is $O(n)$: take $n - \lfloor\sqrt{n}\rfloor$ horizontal segments and a convex chain of length roughly $\sqrt{n}$.

Going counterclockwise instead of clockwise is unlikely to fare much better. However, going in both directions might do the trick. It certainly would take care of the problem depicted in Figure 11. What we have in mind here is a ***dovetailing*** mechanism by which two walks start concurrently from the entrance point, one running clockwise and the other running counterclockwise. Both processes end as soon as one of the walks discovers the exit point. Note that the discovery will always precede the eventual crossing of the two roving pointers, so at worst we will have traversed the entire boundary of the region. Figure 12 illustrates this new way of searching for the exit point.

So, let us just discard finger trees once and for all. What is left is simply the web-structure and nothing else. Well, we do not mean to forget about the magic box, do we? As before, nothing need be done when the sweep-line reaches a right endpoint. The case of a left endpoint gives rise, as usual, to the insertion of a vertical segment into the web. We have just seen how to traverse intermediate regions. Are the first and last regions handled much differently? Not really. As far as the starting region is concerned, the magic box helps us to insert the vertical stop in constant time. Then the dovetailing proceeds just as described. Regarding the last region, we begin by a simple observation. If we keep an eye on the x-coordinates of the vertices visited at all times, we will naturally detect the occurrence of the last region. This detection will take place during the discovery of either the lower or upper endpoint of the vertical stop. The idea is just to continue the dovetailing as though nothing happened until the other endpoint is found. Splitting the region in three is now a trivial operation. This concludes the insertion. For illustration, the labeling in Figure 13 indicates at which steps the endpoints of the vertical stop are discovered. Note that with a different distribution of vertices we could have the same walk discover both endpoints of the stop.

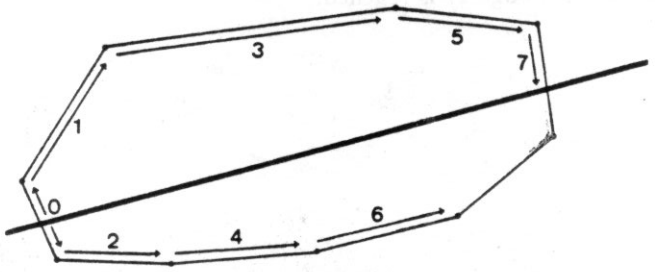


Figure 12. We enter the region via edge 0. Then we alternate between a clockwise and counterclockwise step around the boundary. The sequence of edges traversed is $1, 2, 3, 4, 5, 6, 7$. We exit the region at edge labeled 7.

We could now just go ahead and try to prove that the running time of the algorithm has not increased dramatically. This turns out to be a nasty business. For this reason we will commit a venial sin. We will add some instructions to the algorithm to make its analysis easier. Before we describe how to do that, let us take a look at the problems which we are facing.

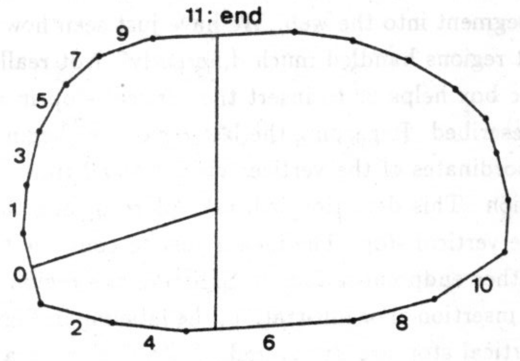


Figure 13. The region is entered at the edge labeled 0 and the boundary is then traversed in clockwise and counterclockwise directions in a dovetailing fashion. When the counterclockwise path reaches edge 6 we can already suspect that we might be in the terminating region of the new segment. This is verified when edge 11 is reached.

Capsule 10. A Combinatorial Excursion

"The cow jumped over the moon"

Before describing our corrigendum, we would like to give the reader a sense of the difficulty. To begin with, our previous amortization argument can be thrown away. The rules of the game have now changed in a profound way. Let us see why on a simple example (Fig.14). The dashed boundary represents a region of the web consisting of $p = 2^k + 1$ edges, into which we insert p segments in the order indicated by the labels. Segment 1 cuts through the middle of the region (i.e., has about $p/2$ edges on either side). Segment 2 and 3 cut through the middle of the left and right subregions, respectively, and so forth. With finger trees the time required for the insertion of all p segments would be on the order of

$$\log p + 2\log(p/2) + \cdots + 2^{k-1}\log(p/2^{k-1}),$$

which is $\Theta(p)$. With our dovetailing strategy, the time would be on the order of

$$p + 2(p/2) + \cdots + 2^{k-1}(p/2^{k-1}),$$

which is $\Theta(p \log p)$.

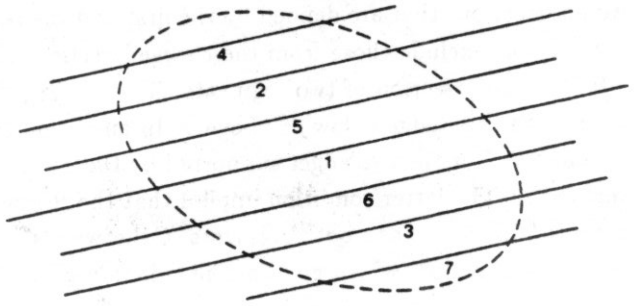


Figure 14. We add segments 1 through 7 in this order. If there were p such segments (rather than 7) and the polygon had p edges, we could end up spending $\Theta(p \log p)$ time on these insertions alone.

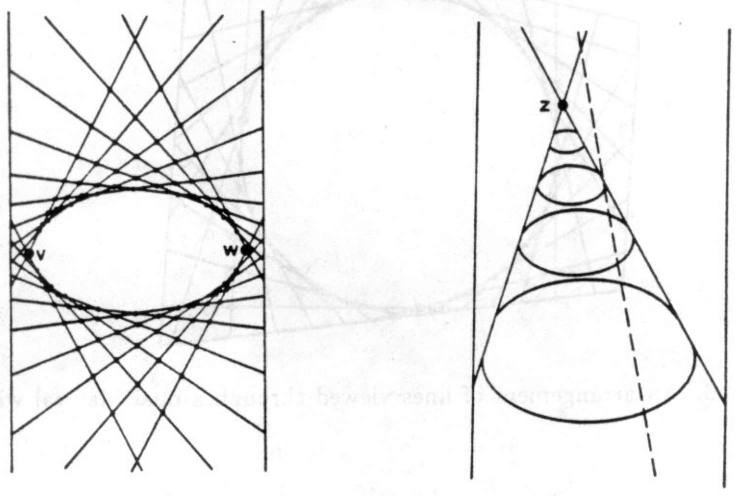
So, was foresaking finger trees such a good idea after all? One may wonder. Clearly, if the answer is yes, we will need a more *global* argument: global in the sense that the region to be sliced is itself the witness of a large number of intersections and this will have to be used somehow. Indeed, the clearance property tells us that because we are inserting new segments all across the region the segments which contribute the edges of the region must extend completely to the left and the right of the region in question. This forces these segments to be in grid-like position, as depicted in Figure 15(a). Thus the region alone is witness to $\Theta(p^2)$ intersections —enough to make us forget about the $\Theta(p \log p)$ cost of insertion. Of course, this does not really prove anything, but at least gives us cause fo a little optimism. The key fact is that in the charging scheme alluded to above, a given vertex will be charged at most a constant number of times.

Lemma 2. *Consider a hammock defined by n segments connecting two vertical lines; let $deg(r)$ be the number of edges bounding region r in this hammock. Then*

$$\sum_r deg^2(r) = O(n+k),$$

where k is the number of intersecting segment pairs and the sum is taken over all regions r in the hammock.

Proof: If v and w are respectively the leftmost and rightmost vertices of a region r in the hammock (assuming that these vertices are unique) we define the *upper chain* of r as the polygonal line running clockwise from v to w. Similarly, the *lower chain* is the polygonal line running clockwise from w to v. Let u and ℓ be the number of upper and lower edges, respectively (i.e., edges in the upper and lower chains); the total number of edges of r is equal to $p = u + \ell$. The number of intersections among the segments supporting the edges of r is at least $\binom{u}{2} + \binom{\ell}{2} > (p-2)^2/8$. In case v or w (or both of them) are not unique (and thus r has one or two edges on the vertical lines defining the hammock) we have a similar result. Now, $p \leq u + \ell + 2$ and therefore $\binom{u}{2} + \binom{\ell}{2} > (p-4)^2/8$. In other words, r can claim credit for (that is, charge) at least $(p-4)^2/8$ intersections. The proof will be done if we can show that each vertex is charged by at most two regions. Indeed, we will succeed to do this except for the two intersections that are defined by the first and the last edge of the upper chain and the lower chain. So, let us exclude those from the charged vertices of r. Consider now a vertex z in the hammock. It is the intersection of two segments. To be charged by a region r, it is necessary that r should lie either in the wedge below or above z. In any event, r has to be incident to both segments passing through z. But the two edges supported by these segments cannot be first and last on the upper (lower) chain. The latter condition implies that the region is either the lowest below z or the highest above z that is incident to both segments. Otherwise, there is a segment that cuts all regions below (or above) this region, which gives a contradiction (Fig.15(b)). ∎



(a) Any two segments supporting edges of the upper (lower) chain intersect.

(b) Vertex z is charged only by the lowest region below z that is incident to both lines passing through the vertex.

Figure 15

Although Lemma 2 does not quite solve our problem, it tells us something very important about the distribution of regions and vertices. As such, it is an interesting combinatorial result about arrangements of lines. For the sake of argument, let us pursue this fascinating study one step further. The proof of Lemma 2 still holds if we replace the two vertical lines of the hammock by two arbitrary distinct lines. Of course, the shape of the window through which we look at an arrangement of lines influences the definition of chains. A chain of a region will now start and end at a vertex that allows a tangent line parallel to a side of the window. The observation that the sum of the chain lengths squared is at least proportional to the square of the total number of edges of the region is true as long as the window is bounded by a constant number of sides (Fig.16). A vertex will be charged by at most four regions, one in each wedge. This allows us to state the following generalization of Lemma 2.

Theorem 3. *Let R be the set of regions obtained as the intersections of a convex polygon P with the regions of an arrangement of n lines. If P has a constant number of vertices, then*

$$\sum_r deg^2(r) = O(n+k),$$

where k is the number of line pairs intersecting inside P and the sum is taken over all regions r in P.

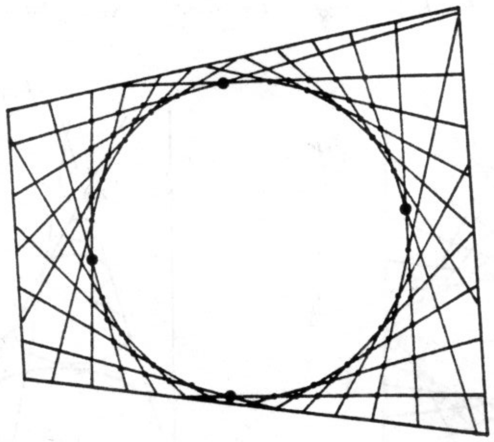


Figure 16. An arrangement of lines viewed through a quadrilateral window.

To see how important the restriction to a window of constant size actually is, consider a circular window in Fig.16. Let this window contain the vertices of the n-gon (24-gon in the figure) but no other vertices. Then $\sum_r deg^2(r) = \Omega(n^2)$, although there are only $O(n)$ intersections inside the window.

What are the implications of Lemma 2 to our analysis? A region with p edges might be entitled about p^2 credits since it can be identified with roughly that many intersections (up to within a constant factor). This credit invariant is both good and bad. If we split a large region in the middle then we release many credits. If, however, a new segment is inserted in such a way that it increases the number of edges from p to $p+1$ (think of a segment cutting off only one vertex of the region), we will have to feed $(p+1)^2 - p^2 = 2p+1$ new credits into the system. To earn these credits, we must argue that the new segment contributes on the order of p new intersections. But is that true?

Suppose that we are in a hammock: all segments, including the new one, join two vertical lines between which the region lies. If the new segment cuts through two consecutive upper (resp. lower) edges, it creates about u (resp. ℓ) new intersections. This is not exactly what we wanted, but there an easy fix: redefine the credit invariant to be $u^2 + \ell^2$. The new intersections can be converted into either $\Theta(u)$ or $\Theta(\ell)$ new credits, which is sufficient to maintain the invariant. Once again, this discussion is not leading to an answer but to a problem. Indeed, let us return to the general case where a segment is inserted into a web.

Capsule 11. Adding Extra Vertical Edges

"That was against the rule,
It made the children laugh and play"

Our previous argument falls through for the following reason. Let L be the current sweep-line (passing through the left endpoint of the segment to be inserted) and let L' denote the vertical line passing through the right endpoint. Again, let us concentrate on a region being cut through and through by the new segment. The clearance property ensures that each segment which contributes an edge to the region either stretches all across the strip (L, L') or falls completely on one side. Figure 17 gives a dramatic illustration of what could happen. The region has upper and lower edges in large quantities, yet the new segment generates only two additional intersections. To maintain the credit invariant we need on the order of $(\ell+1)^2 - \ell^2$ extra credits, which might be very hard, not to say impossible, to get.

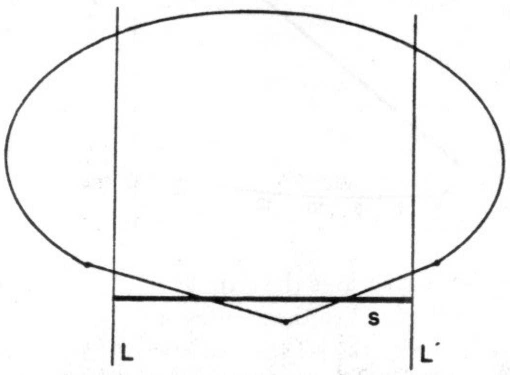


Figure 17. The new segment s cuts a region whose lower chain consists of many edges. However, it does not cut more than two segments supporting those edges.

Another source of worry is the exposition of the edges outside the strip. Suppose that many segments joining L and L' are to be inserted later on: what will happen if these segments cut the region through both its lower and upper edges? This will force our dovetailing to go all the way around the region to find the exit point. This excursion outside the strip could turn out to be very costly. All the more costly as we may have very few intersections to show for. We certainly do not need more roadblocks at this point. Fortunately, a minor addition to the dovetailing mechanism will take care of it.

The dovetailing can be modeled as two scouts walking around the boundary of the region, one clockwise and the other couterclockwise. We will require our scouts to keep track at all times of whether they are inside or outside the strip (L, L'). A scout always begins his walk inside the strip. He may then leave it and re-enter later on. Further down the road the same scout might leave again and finally re-enter. Of course, we should recall that scouts can never cross, which certainly puts a limit on how many times they can go in and out.

Now comes the addendum. The scouts work in dovetailing fashion until the exit point has been found. If the region extends to the left (resp. right) of the strip, then the dovetailing must continue until this has been discovered and a left (resp. right) vertical edge has been added to the web. (Of course, there is no need to add a vertical edge if there is already one there and the addition would create a multiple edge). These vertical edges will serve as *shortcuts* and ensure that straying outside the strips can occur at most twice per region. This is illustrated in Figure 18.

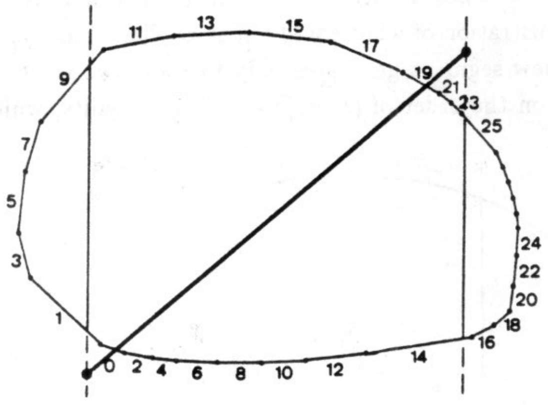


Figure 18. The clockwise scout leaves the strip at step 1, re-enters at step 9; the left vertical edge is added. Then the scouts continue their walks. The exit point is discovered by the clockwise scout at step 19. By that time the counterclockwise scout has left the strip through its right boundary and is walking on with the goal of re-entering and finding the upper endpoint of the right vertical edge. As it turns out, this endpoint will be discovered first by the clockwise scout. At that very step (#25) the dovetailing stops. Note that if the counterclockwise scout had not left the strip by the time the exit point was discovered the dovetailing would still have had to go on to check whether the region might extend past the right boundary of the strip. Many different scenarios are possible. As is the case in our example, one vertical edge (the left one) is computed entirely by one scout, while the other one is handled jointly by the two scouts.

Here is another way to look at the dovetailing. Let ℓ a variable equal to 1 if a left vertical edge is needed or 0 if it is not. Note that $\ell = 1$ if and only if the left endpoint of the segment to be inserted lies to the right of the leftmost vertex of the region. We define r similarly with respect to the right vertical edge. If at any time during the dovetailing we happen to leave the strip to the left we can immediately set ℓ to 1. If on the contrary we reach the leftmost vertex of the region we can set ℓ to 0. A similar remark applies to r. Instead of stopping the dovetailing as soon as the exit point has been found we pursue it until each of the following is true: (i) the exit point has been found; (ii) both ℓ and r have been set to some value; (iii) if ℓ (resp. r) has been set to 1 then the corresponding vertical edge has been added to the web. Note that all of these conditions can be made true by a single one of the two concurrent walks or by a combination of the two.

Is the algorithm different when we are dealing with the starting or terminating region? Not at all. Actually, a minute's reflection shows that we have added consistency into the whole algorithm. The vertical stops introduced in the first and last regions are just particular cases of vertical edges. Of course, the way in which they are discovered is a bit different: with the help of the magic box no extra work is really needed for the starting region. Two final observations to ensure that everything is clear:

(i) Each region traversed could end up with two new vertical edges: this includes the first and last regions.

(ii) Because of our assumption of distinct coordinates, a vertical stop consists of at most two collinear edges; other vertical edges (of the *shortcut* type) are never split during later insertions.

We could go back to our pennies and dimes now and try to concoct a new amortization argument to prove that we are on the right track. Money ain't everything, however. We have pulled the rug on two tricky items (the clearance property and the magic box) and we begin to feel guilty about it. But fear not, we shall return to our credit cards very shortly.

Capsule 12. Relaxing the Clearance Condition

"Who cut off their tails with a carving knife"

What can we do if the segments do not satisfy the clearance property? Cut them in small pieces. Of course, we would like as few pieces as possible, or at least not too many. Here is a reasonable strategy based on the binary representation of the endpoints' ranks. To begin with, we should observe that the y-coordinates of the segments are completely irrelevant to the problem. We might as well simplify the problem. Given two closed intervals I and J on the real line, we say that I and J are ***astride*** if one endpoint of I lies in the interior of J and the other one lies outside J (note that this implies that one endpoint of J lies in the interior of I and one lies outside I). A set of intervals is called ***clear*** if no pairs of intervals are astride.

Let V be a set of n intervals of the form $[x_i, y_i]$ $(0 \le i < n)$. We define $C = (z_0, \ldots, z_m)$ as the sequence of ***distinct*** endpoints formed by V: we have $m + 1 \le 2n$, with equality occurring when all endpoints are distinct. Consider the following sequence of special intervals; they are so special to us, indeed, that we shall call them ***canonical*** intervals. We take all intervals one endpoint apart:

$$[z_0, z_1], [z_1, z_2], [z_2, z_3], \ldots, [z_{j2^0}, z_{(j+1)2^0}], \ldots, [z_{m-1}, z_m].$$

Then we take intervals two endpoints apart:

$$[z_0, z_2], [z_2, z_4], [z_4, z_6], \ldots, [z_{j2^1}, z_{(j+1)2^1}], \ldots$$

In general, we take intervals 2^k endpoints apart, which gives

$$[z_0, z_{2^k}], [z_{2^k}, z_{2^{k+1}}], \ldots, [z_{j2^k}, z_{(j+1)2^k}], \ldots$$

In each case, j runs as high the corresponding interval fits entirely into $[z_0, z_m]$; similarly, k runs as high as the corresponding sequence is not empty.

An elementary geometric progression shows that the total number of canonical intervals can never exceed $2m - 1$. Most important, the set of canonical intervals is always clear. It thus gives us a good kit for cutting up the intervals V. We would like to be able to take any interval $[x_k, y_k]$ of V and partition it into a short sequence of canonical intervals. By short, we mean of size at most $2 \log m$. Let us try a small example. The *ranks* of the coordinates as opposed to their *values* will determine the partitioning, so we can make our life simpler and choose $z_i = i$, and say, $[x_k, y_k] = [2, 7]$.

The idea is to find the highest value of k such that 2 is a multiple of 2^k and $2 + 2^k$ does not exceed 7. The values $k = 0, 1$ pass the test, but $k = 2$ fails. This gives us the first canonical interval, namely, $[2, 2 + 2^k] = [2, 4]$. The initial interval $[2, 7]$ is now reduced to $[4, 7]$. We pursue this game keeping k at its current value ($k = 1$). The value $k = 1$ passes the test, but $k = 2$ fails. So, we add the canonical interval $[4, 6]$ to the sequence, and iterate with respect to the interval $[6, 7]$ and the value $k = 1$. We now find that the current value of k fails. So, we restart the whole process from right to left, resetting k to the value 0. Starting at 7, we find that $k = 0$ is the only value of k for which 7 is a multiple of 2^k. Since $7 - 2^0$ is no less than 6, we get another canonical interval, $[6, 7]$. The two ends have met, so we may stop.

Note that we do not obtain the intervals in order from left to right. As it turns out, this is of no consequence as far as our application is concerned. There are hundreds of other ways to carry out the partitioning, and one should feel free to add all the bells and whistles he might wish. We include a formalized version of the algorithm in pseudo C-code. An interval of the form $[z_i, z_{i+\lambda}]$ is partitioned in at most two intervals of length 2^k, for any fixed k. It follows that at most $2k$ intervals will be necessary, where $2(2^0 + 2^1 + \cdots + 2^{k-1}) \geq \lambda$. This gives us an upper bound of $\lceil 2 \log(\lambda + 2) \rceil - 2$ on the size of the partition. The time required for the partitioning is of the same order of magnitude.

It is now time to return to the clearance property. The previous scheme allows us to enforce this property by trading the original set of segments for a slightly larger set. This is called the *augmentation process.* If we start out with n segments, the analysis above indicates that the augmented set will have its size within $O(n \log n)$. We are basically done with the clearance property. Well, actually, there is still one little hitch. Recall the property sought: whenever a new segment is to be inserted into the web, the interior of the vertical strip defined by its endpoints should be free of any endpoint. The clearance property makes this condition realizable. Of course, this assumes that the segments are inserted in the proper order. If all the endpoints have distinct x-coordinates (which has been our original assumption) then the left-to-right order of the left endpoints induces the proper sweeping order among the segments. This is all very well, but in spite of all our assumptions, the augmented set of $O(n \log n)$ segments is very unlikely to have distinct coordinates. There are just too few coordinates to share among too many segments.

How should the sweep-line break ties? When it is moved to its next destination it may now encounter several endpoints vying for attention. Recall that right endpoints are just to be ignored. As far as left endpoints go, the sweep-line must treat the tied segments in the *right-to-left* order of their *right* endpoints. Any other order would obviously violate the condition mentioned One remarkable property of the augmentation process is that this order coincides with the order of the

```
left $= x_k$; right $= y_k$;
step $= 1$;
while (left+step $\le$ right)
  { step2 $= 2\times$step;
    if ((left % step2)||(left + step2 > right))
      { add $[z_{\text{left}}, z_{\text{left+step}}]$ to sequence;
        left += step; step = step2;
      }
  }
step $= 1$;
while (left+step $\le$ right)
  { step2 $= 2\times$step;
    if ((right % step2)||(left + step2 > right))
      { add $[z_{\text{right-step}}, z_{\text{right}}]$ to sequence;
        right -= step; step = step2;
      }
  }
```

right endpoints of the original segments. What we mean to say is that if two augmented segments have the same left x-coordinate then the relative x-order of their right endpoints (if there is no tie) is the same as the relative x-order of the right endpoints of the *original* segments. This simple remark will give us a very convenient method for breaking ties and doing other useful things as well.

The alert reader will perhaps have recognized glimpses of the ubiquitous segment tree [4] in the partitioning scheme described above. We can identify the canonical intervals with the nodes of a complete binary tree on m leaves. Each interval $[x_k, y_k]$ is partitioned into a collection of canonical intervals, at most two of which are identified with nodes on the same level of the tree. In a total of $O(n \log n)$ operations, we can compute node-lists indicating, for each node of the tree, which segments use the canonical interval at that node for their partition. The correct processing order is then obtained by traversing the tree in *preorder*. Within a given node-list the processing order is immaterial as far as the insertion is concerned.

For reasons which it is too early to mention, however, we wish to prescribe a specific order among elements of the same node-list. We will assume a normalization condition. A schedule is said to be in *normal form* if the following is true for any node v of the tree: given any two segments s_i and s_j of S which contribute entries to the node-list at v and are such that the right endpoint of s_i precedes the right endpoint of s_j in x-order, the segment σ contributed by s_j must be inserted before the segment σ' contributed by s_i. In other words, ties at node v are broken by looking at the x-coordinates of the right endpoints of the original segments.

Figure 19 illustrates the augmentation process on 4 segments. Since deletions are ignored let us give the schedule of insertion for the sweep-line. The entry $i : (jd, kd')$ means "insert the portion of segment i running between the x-coordinates of the d endpoint of segment j and the d' endpoint of segment k (d and d' are placeholders for ℓ = left or r = right)". The processing order will be

$1 : (1\ell, 2r)$
$2 : (2\ell, 3\ell)$
$3 : (3\ell, 2r)$
$2 : (3\ell, 2r)$
$4 : (4\ell, 2r)$
$3 : (2r, 4r)$
$4 : (2r, 4r)$
$1 : (2r, 1r)$
$3 : (4r, 3r)$.

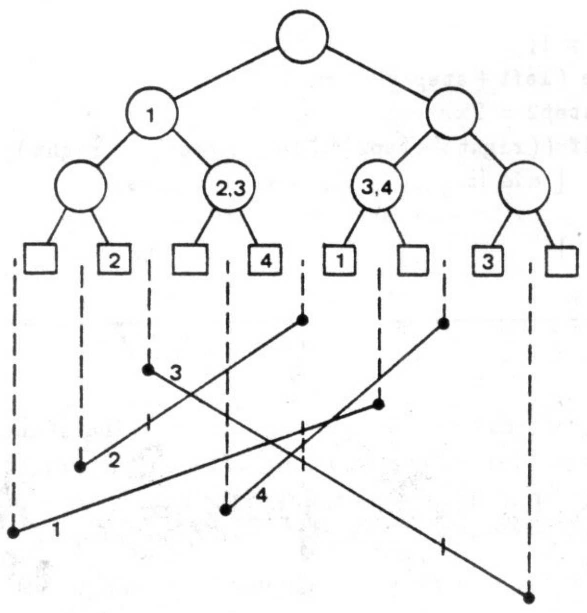


Figure 19. Using the canonical intervals defined by the segment tree, we decompose the 4 segments into 9 segments that now satisfy the clearance property. The schedule takes the 9 segments by traversing the tree in preorder.

This completes our discussion of the clearance property. Note that the augmented set can be treated just like an ordinary set of segments. The only difference is in the fact that x-coordinates might be shared among endpoints. This will not create (much) difficulty. In one way it will turn out to be a bonanza when we get to discuss the magic box. If a segment is partitioned into $s > 1$ subsegments only the leftmost endpoint of the segment should be hard to insert. The $s-1$ other "left" endpoints should be trivial to handle since they coincide with the right endpoints of subsegments already inserted. In fact, there is another interpretation of the augmentation process. It is a way to schedule the insertion of a segment into the web. At certain points it might be wise to suspend the insertion process for a segment just to resume it later when more pieces of other segments are in the web. This is exactly what we do when we "cut" each segment into smaller pieces. To distinguish endpoints from the original set of segments from the ones created during the augmentation process, we refer to the former ones as the ***original*** endpoints; the others are called ***augmented*** endpoints. We use the same terminology to distinguish between the segments of S and the segments resulting from the augmentation process.

There is another important difference brought about by the sharing of coordinates. Let vw be a vertical stop of the ***right*** type, that is, created during the cutting of a terminating region (Fig.13). Let v (resp. w) be the upper (resp. lower) endpoint of the segment. The segment vw consists of two collinear edges. Later, this vertical segment may be further split by augmented endpoints in the process of insertion. These splits occur from the left; they may cause the boundary of the region immediately to the right of the segment vw to contain an arbitrarily large number of consecutive vertical edges (Fig.20). This is called a ***vertical chain***. Our goal is to be able to treat an entire vertical chain as a single vertex. The reason is that repeated traversals of vertical chains could have a devastating effect on the running time of the algorithm.

Let us place ourselves at the point in time when the sweep-line has just reached the position of the vertical chain and no further action has yet been taken. One solution would be to require that in the web-structure any record referring to an edge e lying on vw should have handy a pointer to the unique web edge vv' whose left endpoint is v as well as the edge ww' whose left endpoint is w. The edges vv' and ww' are respectively called the *ceiling* and the *floor* of e (Fig.20). This provision would make it possible to circumvent vertical chains entirely. If we encountered one of them when dovetailing around a region we would be able to skip it entirely: an operation referred to as a *vertical skip* in the following. Also, if we started the insertion of a segment whose left endpoint was a vertex of such a chain we could again skip to the leftmost vertex of the lower or upper chain of the starting region in constant time. Another subtle point of terminology. Vertical chains are kept separate from upper and lower chains. For illustration, let us walk around a generic region in clockwise order, starting from the bottom of the vertical chain. We will successively traverse (1) the vertical chain, (2) the upper chain, (3) a vertical edge, and (4) the lower chain. Parts (1) and (3) may not be defined; although we might occasionally refer to (3) as a vertical chain as well, notice that it cannot consist of more than one vertical edge. The distinction between vertical and upper/lower chains is crucial. When we later assign credit invariants to lower and upper chains this will specifically exclude vertical chains.

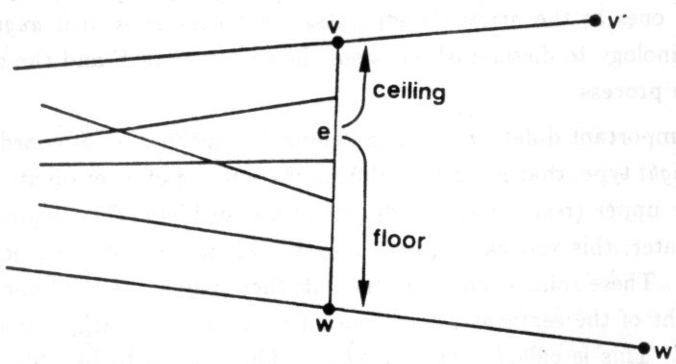


Figure 20. Many augmented endpoints can have the same x-coordinates and thus create long vertical chains.

The provision which we have just described is handy because it illustrates exactly the effect we want to achieve in the simplest terms. Each edge e of a chain should have ready access to its floor and its ceiling. Unfortunately our solution is plagued with a major flaw and must be abandoned. The problem is that we do not really care about e's floor and ceiling until a segment adjacent to e comes to be inserted. But when this happens, segments adjacent to the chain may have already been inserted and in the process assigned edge e a new floor or a new ceiling. As it turns out, all but at most one vertex of the chain are augmented endpoints, so insertions of the *continuing* type (i.e., of segments whose left endpoints are augmented) are likely to pop out of the vertical chain in random manner. How do we maintain floor and ceiling assignments under these conditions? One could use finger trees for that purpose. But wait! Wasn't dovetailing used to avoid fingers in the first place? No comment...

Fortunately there is a very simple solution. Upon encountering a vertical chain we will indulge in a bit of preprocessing. This encounter will occur the first time we try to insert a segment whose left endpoint lies on the vertical chain in question. Since floors and ceilings come only later into the picture we replace them by special edges called *companions*. Rather than jumping to the floor or ceiling directly, we jump to companion edges. At the end of the preprocessing each edge of the chain will have a pointer to its unique *companion* edge on the chain. To define the notion of a companion, let us go through the history of insertions emanating from the chain under consideration. At the beginning, the highest and lowest edges of the chain have each other for companion. No other edge has a companion. Let p be a vertex of the chain (distinct from v and w) and let p_h (resp. p_ℓ) be the vertex right above (resp. below) p. Assume that the first segment adjacent to the chain which gets to be inserted is of the form pq. Then if pp_h (resp. pp_ℓ) does not already have a companion it is given the highest (resp. lowest) edge of the chain for companion. The insertion of pq makes two vertical chains out of one, and the definition proceeds recursively in the obvious manner. Note the importance of not giving anyone a new companion if it already has one. Without this rule everybody would end up with itself as companion (the ultimate society). It is clear that if whenever we insert a segment adjacent to the chain, we know the companions of the chain edges adjacent to it, vertical skipping becomes trivial. When an insertion cuts a vertical chain into two pieces, all new floors and ceiling can be computed (implicitly) in constant time.

How difficult is it to compute companions? Let $v_1, v_2, \ldots, v_m$ be the vertices of the chain in descending order, with $v_1 = v$ and $v_m = w$. Assume for the time being that each of these vertices is an augmented endpoint, and for each i $(1 < i < m)$ let x_i be the x-coordinate of the right endpoint of the original segment associated with v_i. From our previous remark concerning the equivalence of orderings between the x_i's and the right endpoints of the augmented segments, the schedule of events ensures that the segments are inserted in an order such that $x_{i_2} \geq x_{i_3} \geq \cdots \geq x_{i_{m-1}}$.

It is clear that the particular orientation of the segments emanating from the chain has nothing to do with companionship: all that matters is how far these segments extend to the right, since this determines their order of insertion. There exists a simple linear time algorithm for computing companions. See pseudo-code. The symbols **push** and **pop** denote the standard stack operations. Each entry in the stack is an integer between 1 and m. The top of the stack is denoted `t`. To ensure the consistency of the while loop we set both x_1 and x_m to $+\infty$, but we make the convention that $x_1 > x_m$. Figure 21 depicts the algorithm in action as well as its relation to the computation of left-to-right maxima. The curved arrows represent the companion assignment; they are computed from left to right by the algorithm. To summarize, this preprocessing should be applied the first time we attempt to insert a segment emanating from the chain. Then the entire chain is scanned and the routine above executed.

```
companion($v_1v_2$) = $v_{m-1}v_m$;
push(1);
for (i = 2; i $\leq$ m; i++)
  { while ($x_t \leq x_i$)
      { if (i > t+1) companion($v_tv_{t+1}$) = $v_{i-1}v_i$;
        pop;
      }
    if (i > t+1) companion($v_{i-1}v_i$) = $v_tv_{t+1}$;
    push(i);
  }
```

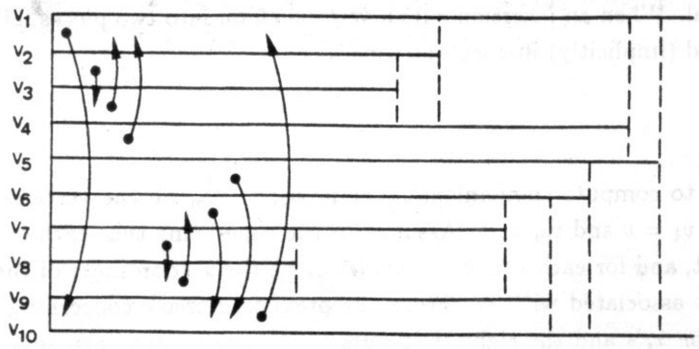


Figure 21. The arrows indicate the companionships among line segments. We have drawn all segments horizontal because their actual slopes are irrelevant.

If one of the v_i's is not an augmented endpoint (only one may not be so) then it must be the right endpoint of an original segment. Although no segment will be inserted from it later, this case does not differ from the general one in any significant way. More interesting is the case of an original left endpoint lying on the chain. Granted, the magic box will figure out on what edge of the chain that endpoint lies. But what about vertical skips? When the insertion of the segment takes place we will conveniently forget the fact that we do not know its floor and ceiling and boldly start the dovetailing from the entrance point. The scouts will be at liberty to scan through the vertical chain. This exception to the rule is acceptable because it will happen only once per chain. The costs incurred can be accounted for by the chains themselves. For the sake of consistency we can therefore pretend that in all cases vertical skips can be implemented with overhead amounting at worst to an extra multiplicative factor.

Yes, gentle reader, this was a long capsule, indeed. We hope it has convinced you that life without the clearance property is not all that bad. The price to pay is an augmented set of segments, a more intricate algorithm for determining the schedule, the notion of a normal schedule, the necessity of vertical skips and its consequences in terms of preprocessing vertical chains.

Capsule 13. Implementing the Magic Box

"For nobody's toeses are posies of roses"

The only problem with magic boxes is that they are magic. Of course, one can always leave their construction as an exercise to the reader... Let us return to the point where the magic box was introduced. We had just agreed to keep the sorted sequence of active segments in a so-called *sweep tree.* This is a red-black tree whose symmetric traversal gives the name of active segments in ascending order. In our case, the collection of segments is the augmented set emanating from the cutting described above. Recall that a segment is said to be active if it intersects the sweep-line. As the set of segments has been augmented, so has the schedule: its events are now determined by the endpoints of all the segments in the augmented set. The previous capsule has made this plain, so let us move on.

As long as the sweep-line stays put, the sweep tree can be updated quite easily. Consider the insertion of a new segment. If the left endpoint is original then the edges of the web right above and below the point in question are retrieved by binary search in the sweep tree. The tree is updated accordingly (insertion of a new node and the whole bit). If the left endpoint is of the augmented type things are even simpler. The sweep tree can be completely circumvented. We assume that an ***appearance table*** is maintained throughout the sweep to tell us if (and where) a given segment has already contributed to the web. If s_i (the ith segment in the ***original*** set) has already contributed at least one edge to the web, the ith entry in the table will be a pointer to the ***rightmost*** such edge. Note that this edge always has a vertical stop on the right. If on the contrary s_i has yet to become active, then the ith entry is equal to 0.

Going back to the insertion of the new segment, we first find out the name s_i of the original segment (before being cut up) —a constant time operation with a proper representation of the augmented set. Next, we look up entry i in the appearance table. If it is equal to 0 we search the sweep tree for the two web edges respectively above and below the left endpoint of the new segment. Then we insert a new node pointing to the new web edge. The dovetailing can get started from there. If the ith entry of the appearance table is not null then we jump directly to the edge to which it points. The dovetailing will begin in the vicinity of that edge. Updating the sweep tree is particularly easy. Consider Figure 22. The appearance table takes us to edge e. The two vertical edges adjacent to the right endpoint of e give us the starting point of the dovetailing. (In the particular case of Figure 22 a vertical skip is in order.) At this point we assume that just as each node of the sweep tree points to some edge of the web, there exists a reverse link to go from the edge to the corresponding node of the tree. This will allow us to update the node which points to the edge e to point to e' instead. Note that this reverse link could be put in the appearance table directly. The reason we choose not to do so is simple: later we will revamp the sweep tree, so by disconnecting it completely from the appearance table, the definition we just gave of the table will not have to be changed.

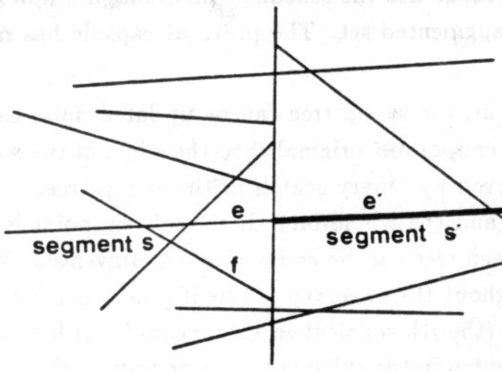


Figure 22. In order to insert segment s' we use the appearance table to jump to the right endpoint of its predecessor, segment s. This is also the right endpoint of edge e, the rightmost edge of s in the web.

What we have done in this capsule is to show how to implement the magic box while the sweep-line is stationary. Put differently, we have described how to *make use* of the sweep tree. We must now deal with the major problem yet unsolved: how to update the sweep tree, especially when a large number of intersections take place between two schedule events and the order along the sweep-line gets all mixed up? Since we have long forsaken the possibility of treating intersections in order from left to right, we are not going to sweep over intersections, pause for a moment, do some work, and go on. The sweep-line algorithm has been entirely laid out in the previous capsules, and we shall not touch it any more. The magic box will have to be implemented as a separate subroutine. It might use the web as a tool, but it will not affect its construction, nor any aspect of the sweeping process for that matter.

These fine words having been spoken, we now need a new insight. Here is one: since we are proclaiming east and west that the sweep-line and the magic box should be separate and independent, why tie the definition of the sweep tree so closely to that of the sweep-line? Why should *all* the nodes of the sweep tree point to edges crossing the sweep-line for example. After all, when an insertion occurs only a logarithmic number of nodes are visited, so our current invariant might be too strong.

Capsule 14. A Functional Look at the Sweep Tree

"But lies in bed till eight or nine,
Lazy Elsie Marley"

The answer will come from a lazier approach to the updating of the sweep tree. Instead of maintaining a global invariant on the structure of the tree, we will allow the tree to decay somewhat (not in balance but in the information that it stores): the main requirement will be that whenever a binary search is to be activated the nodes visited can be "reconfigured" at little cost, or at least at a cost which can be charged to new intersections. To understand what we are really trying to do here, we must go back to the notion of a ***functional data structure.*** This is an abstraction developed in (Chazelle [7]), where a data structure is not regarded merely as a particular arrangement of data but rather as a particular arrangement of functions. The main idea is to relieve elementary components of a data structure from the necessity of storing data. Sometimes, the benefit of this approach is to bring upon storage the same effect as *lazy evaluation* and *call by need* are known to have on execution time. In other cases, it provides a much more adaptive data structure which runs faster. This will be the case here. The reader who is not too familiar with [7] might be a little befuddled by this discussion. The smoke should dissipate soon, though, as we put these ideas into practice.

Regarded as a data structure, the sweep tree is a particular way of *arranging* data in a structured manner: the arrangement is that of a balanced binary tree and the data consists of pointers to edges in the web. Note that the child pointers are part of the arrangement, and thus for our purposes are *not* considered as data. Interpreted as a functional data structure, the sweep tree becomes an arrangement of functions. What does that mean? The functional version of the sweep tree (or *functional sweep tree*, as we shall call it) is obtained by replacing the data inside each node by the function whose evaluation is associated with that node. To understand this last sentence it may help to back up a little.

Let v be a node of the sweep tree and let s be the augmented segment from which the web edge pointed to by v originates. The raison d'être of v is to allow a client —in this case, the search algorithm— to answer questions of the type: *"is query point p above, on, or below segment s?"* If one thinks about it, however, this misses the essential point of a binary search tree: if $\sigma(v)$ denotes the symmetric rank of node v, then s happens to be the $\sigma(v)$th active segment in ascending order. So, we can reformulate the question entirely in terms of v and p. It comes down to evaluating the function $f_v(p)$, where $f_v(p)$ is equal to 1 if, say, *"point p lies above the $\sigma(v)$th active segment* and -1 (resp. 0) if p lies below (resp. on) the segment in question.

The only difference between the sweep tree and its functional counterpart is that the data at v is replaced by the function f_v. Everything else remains the same; in particular, the bijection between nodes and active segments, where symmetric order among nodes correspond to ascending order among active segments. The reader may now wonder how we go about implementing a functional sweep tree. In particular, what do we store at node v, now that data has been purged out? Recall that a functional data structure is just an abstraction and that it has to be converted back into a real data structure in order to be implemented. However, we have gained complete freedom in the way we choose to implement the functions f_v's.

How will we take advantage of this freedom? To evaluate f_v at any given time one must know which is the $\sigma(v)$th currently active segment in ascending order. The naive implementation of f_v consists of maintaining this information at all times, whether it is needed or not. As we know, this will have the effect of essentially sorting all the intersections by x-coordinates, which is bad news, to say the least. Consider the history of the $\sigma(v)$th active segment. Let $t_1, t_2, t_3, \ldots$ be the times at which this segment, denoted α_i, changes. Every time the sweep tree is used by the algorithm only a logarithmic number of nodes are visited, so it is reasonable that two consecutive visits of v might occur respectively at time θ ($t_3 < \theta < t_4$), when α_3 is the relevant segment, and at time θ' ($t_9 < \theta < t_{10}$), when the active segment is now α_9. The naive implementation of f_v requires that at time t_i the new segment α_i should be computed. But we do not have to do that. Instead, it is all right to require that only when v is visited the relevant α_i should be computed. One way of doing this is to compute α_3 at time θ and then do nothing until time θ', at which point we enumerate $\alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8, \alpha_9$ in this order and decide that α_9 is the relevant segment. The latter implementation of f_v is likely to be easier since it does not depend on some global clock but instead follows its own local timetable.

Capsule 15. The Levels of a Set of Segments

"And then she may walk in two"

We begin with a short but necessary digression. A fundamental notion in the study of line arrangements is that of a *level* (Edelsbrunner [10]). For $k = 1, \ldots, n$, we define the k-level of an arrangement of n nonvertical lines as the collection of edges with exactly $n - k$ lines above. (More rigorously, an edge belongs to the k-level if the vertical ray emanating upwards from any point of the edge intersects precisely $n - k + 1$ lines; note that an edge is a relatively open set and therefore does not contain its endpoints.) The collection of k-levels partitions the set of edges into polygonal lines running in a monotone fashion from $x = -\infty$ to $x = +\infty$. A sweep-line cuts through a subset of edges (the active edges) in bijection with the set of levels. The bijection between the sweep-line and the active edges can thus be also regarded as a bijection between the sweep-line and the levels of the arrangement.

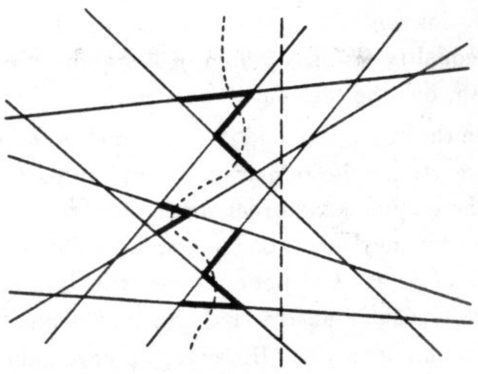


Figure 23. The "topological line" intersects each level of the arrangement exactly once; the edges containing those intersections are shown in boldface.

What if we replace each active edge by (either itself or) some other edge along the same level? (Fig.23). This will likely have the effect of ***bending*** the sweep-line. The idea that a sweep-line need not be straight to be useful is not new. It goes back to (Edelsbrunner and Guibas [11]), who developed the notion of a ***topological sweep***. The active edges in a topological sweep must submit to the following ***adjacency requirement***: given any two consecutive edges, there exists a region of the arrangement for which one of the edges is an ***upper edge***, while the other is a ***lower edge***. (The notion of an upper or a lower edge should be familiar to the reader: we mentioned it in our capsule on finger trees.)

Topological sweep is a step in the right direction for what we need. But we can go further yet. We shall relax the adjacency requirement and keep only the bijection between active edges and levels of the arrangement. Before the reader gets utterly confused, we must straighten out our terminology. Recall our earlier pledge that the sweep-line would remain straight. So what are we talking here of bending the sweep-line? What we meant to say is this:

> *The sweep-line is and will always remain a straight line. The web edges pointed to by the nodes of the sweep tree should be in bijection with the levels of the arrangement. Furthermore, no such edge should lie completely to the right of the sweep-line.*

Figure 23 illustrates this statement. The sweep-line is the dashed vertical line. The nodes of the sweep tree are in one-to-one correspondence with the edges shown in bold. Things could get much wilder than in Figure 23: indeed, any bold edge could be replaced by any edge to its left on the same level. An extreme (yet valid) example is shown in Figure 24. The 12 nodes of the sweep tree point to 12 distinct edges in the arrangement, all collinear to a total of only 2 lines! We have strayed quite far from topological sweeping, where no two edges along the sweep-line, or rather sweep-curve, can come from the same line.

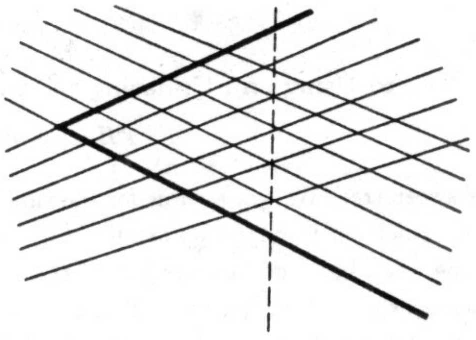


Figure 24. The bold edges indicate the current state of the sweep tree. It contains one edge of each level.

Of course there is only one problem: everything we have said so far applies to line arrangements, but are we not dealing with line segments? As it turns out, we have lucked out. Generalizing our previous discussion to line segments is quite simple. First of all, the notion of a level is perfectly well defined among the n segments of our set S (Fig.25). The definition is somewhat different. The simplest one is procedural. Starting from each left endpoint in S, in turn:

(i) Walk to the right until you reach either an intersection or an endpoint. In the latter case, stop.

(ii) Follow the edge to your right or your left, whichever takes you in ascending x-order. Then go back to step (i).

We shall refer to the traversal in steps (i,ii) as an *x-walk*. An x-walk can be started from any edge in the arrangement of segments: it always proceeds in ascending x-order (hence the name), and if allowed to run its full course, always terminates at the right endpoint of some segment of S. We just alluded here to the possibility of stopping the walk at any convenient time, say, when crossing the sweep-line. Did you guess that this would be one of our favorite moves down the road?

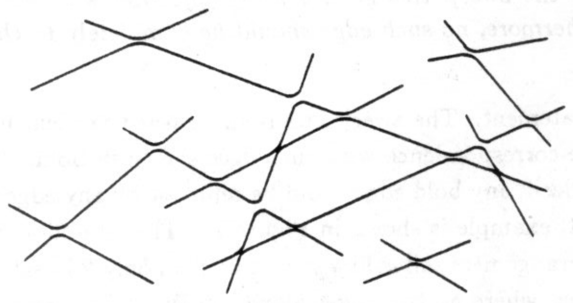


Figure 25. Levels in a set of line segments.

Capsule 16. X-walking as the Key to Using and Updating the Sweep Tree

"Hush-a-bye, baby, on the tree top"

Let us relate all of this back to the sweep tree. We will assume for the time being that as the sweep-line advances nothing ever gets to be deleted from the web-structure. That is, no clever garbage collector takes the initiative to wipe out edges and vertices left of the sweep-line on the grounds that perhaps they have become useless or something like that. This assumption was already implicit earlier, as we observed that the dovetailing itself may have to venture left of the sweep-line on occasions. Here, it becomes all the more important as the sweep tree may relate to edges way to the left of the sweep-line. For this reason, the term web will now refer to the entire planar subdivision constructed so far, and not just the part of it lying to the right-hand side of the sweep-line. We are now ready for the invariant attached to our generic sweep tree node v. Consider the sweep tree in its previous version and traverse its nodes in symmetric order. The web edges, $u_1, u_2, \ldots, u_\ell$, obtained by following the appropriate pointers from these nodes, correspond precisely to the active segments in ascending vertical order. Now, consider the sweep tree "new-look" and perform a similar symmetric traversal. As before each node will store a pointer to some edge of the web, so let $e_1, e_2, \ldots, e_m$ be the sequence of edges obtained during the traversal.

No edge e_i $(1 \leq i \leq m)$ lies completely to the right of the sweep-line. Also, there exists a sequence of indices $1 \leq i_1 < \cdots < i_\mu \leq m$ with the following property: for each edge u_j $(1 \leq j \leq \ell)$ an x-walk in the current web starting at the edge e_{i_j} will lead to the edge u_j.

In plain English this is what happens. Ideally, we would like a complete bijection between the nodes of the old sweep tree and those of the new one. Ideally again, while a node of the old tree would point to some edge u of the web crossing the sweep-line, the same node in the new tree would point either to u itself or, if not, at least to an edge from which u can be reached via an x-walk. This would be ideal. As it turns out, we will be considerably better off in the end if we allow a few extra (dummy) nodes in the tree. An x-walk starting from one of these extra nodes will never get to the sweep-line: it will reach the right endpoint of a segment before crossing the sweep-line.

The alert reader may be thinking that if we did not blithely ignore right endpoints, and instead deleted the relevant nodes from the sweep tree, we would not wind up with spurious nodes in our tree. As a matter of fact, things are not so simple. The problem with deleting from the tree is that when the sweep-line reaches a right endpoint, there may be no ***corresponding*** node in the tree. To be sure, there is some node in the tree from whose web edge an x-walk will indeed take us to the right endpoint in question. But finding it involves performing an x-walk ***backwards*** as well as many other things which we are just too tired to think about at this point. We concede, however, that such an approach is feasible.

Anyhow, we hope our new sweep tree is well understood by now. Although the functional interpretation of the data structure is quite simple, there is so much freedom of action that one might be surprised by certain configurations. Once again, a few rambling remarks should help clear up whatever smoke may remain in the reader's eye.

1. As opposed to the old sweep tree, which is completely specified by the edges that intersect the sweep-line, the new tree depends on the previous history of the sweep-line. Figure 26 depicts a fairly realistic configuration (except for the clearance property which we have conveniently forgotten). The symmetric traversal of the old tree would lead to the edges labeled $1, 3, 4, 5, 6$ in this order. (Edge 2 is to be inserted later, so let us just ignore it now.) Traversing the new sweep tree in symmetric order will give us $a, b, c, d, e, f, g, h, i$ (Fig.27). Note that not only h and i (like c and d) originate from the same segment, but h precedes i in symmetric order even though the edge labeled h lies higher than the one labeled i. The bijection maps $1, 3, 4, 5, 6$ to b, c, d, f, h, respectively. Edges a, e, g, i are spurious.

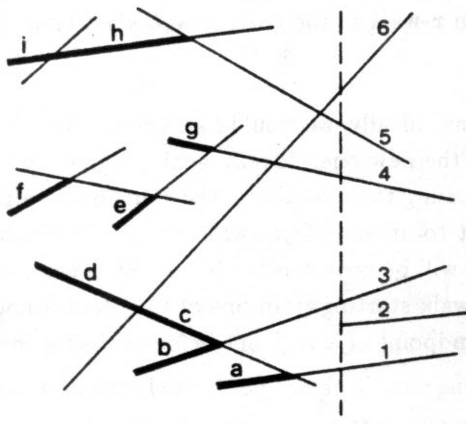


Figure 26. Edges $a, b, c, d, e, f, g, h, i$ are stored in the new sweep tree in this order, while $1, 3, 4, 5, 6$ would be the order of the edges in the old sweep tree.

2. Following the philosophy of a functional data structure, we use the new sweep tree just as we would use the old one. Only the implementation of the functions f_v's differs. The binary search proceeds as usual. To determine the outcome of a *comparison* at a given node v we begin by checking whether the associated web edge intersects the sweep-line. If not, then it lies totally to the left of it. So, we start an x-walk until we reach the sweep-line. When this is done the comparison can be performed in constant time. Of course, we must update the information stored at node v: the pointer to the web will now point to the last edge traversed during the x-walk. For example, the visit, during a search, of the node associated with the edge d in Figure 26 would conclude with 4 dislodging d as the edge pointed to by the node in question. If for some reason the x-walk has to be aborted (as it would from edges a, e, g, i in Figure 26) then we delete the node v from the sweep tree (an operation which is structural to the tree and totally independent of the functions associated with the nodes). Once the deletion has been completed we simply restart the binary search from the root of the tree, as though nothing had happened. Of course, we might get stuck in the same fashion the next time around, but clearly this cannot happen for ever.

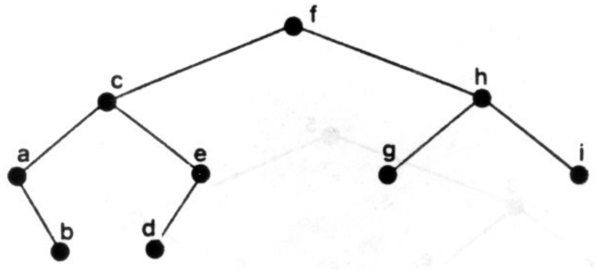


Figure 27. The new sweep tree is a sorted binary tree whose nodes correspond to certain edges to the left of the sweep line. Figure 26 shows these edges in the web.

3. How about an insertion? Dealing with a segment whose left endpoint is augmented does not involve the sweep tree at all (only the appearance table and the web-structure), so let us assume that the left endpoint is original. Look at Figure 26 and imagine that the left endpoint of 2 is the current event in the schedule. To insert the corresponding segment the first thing on the agenda is to perform a binary search in the sweep tree. Here is a likely scenario (Fig.27). The root points to f: an x-walk from f takes us to the edge labeled 5, so 5 dislodges f as the edge pointed to by the root. The outcome of the comparison is "below". Therefore, we pursue the search by visiting the left child of the root, which leads us to c. After the required x-walk, we replace the pointer to c by a pointer to 3: the outcome of the comparison is still "below". The left child of the current node takes us to the edge a. The ensuing x-walk leads nowhere, that is, to a deletion, so we must restart from scratch. The sweep tree is as depicted in Figure 28. The second binary search will proceed more smoothly. The pointer to b will be made to point to 1, and this will be the final change to the tree. Note that when a deletion occurs there is no need to restart from the root (although it does not hurt to do so). Perhaps going back just a few steps prior to the time of deletion would do just as well. We leave this finetuning as an exercise to the reader.†

† The reader did not think he would get away without a homework, did he?

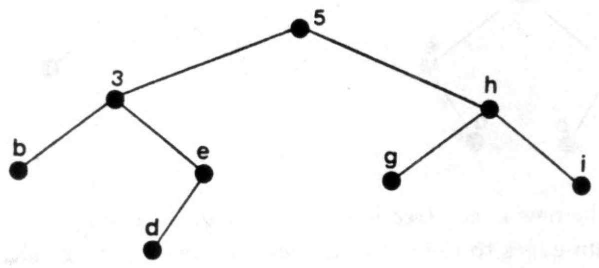


Figure 28. The sweep tree obtained from the one in Figure 27 after x-walking f and c and deleting the node which pointed to a.

4. A few more words concerning x-walks. We have seen how easy it is to perform an x-walk in constant time per edge traversed. But we were dealing with arrangements of segments and not webs. The difference is that we may now have vertical edges. The rule is simple: *ignore* these edges altogether. Whenever the x-walk encounters a vertex adjacent to a vertical edge, it should try to proceed along the edge collinear to the current edge. If there is no such edge, then we have reached a right endpoint, and a deletion will follow (unless, of course, we have reached the sweep-line). There are 3 different cases (up to mirror-image configurations): Figure 29 summarizes what to do. Note that case (3) is not exactly a special case. It may happen all the time. The two bold edges lie on two segments which used to be one in the original set S. When the one on the left was inserted, the two vertical edges were added to the web (as stops). Later, when the edge on the right was inserted, no use was made of the sweep tree because the appearance table contained all the information needed. For this reason it could well be the case that the node of the sweep tree which leads to these edges (via an x-walk) was not visited in a long time and consequently kept way back to the left of the sweep-line (Fig.29).

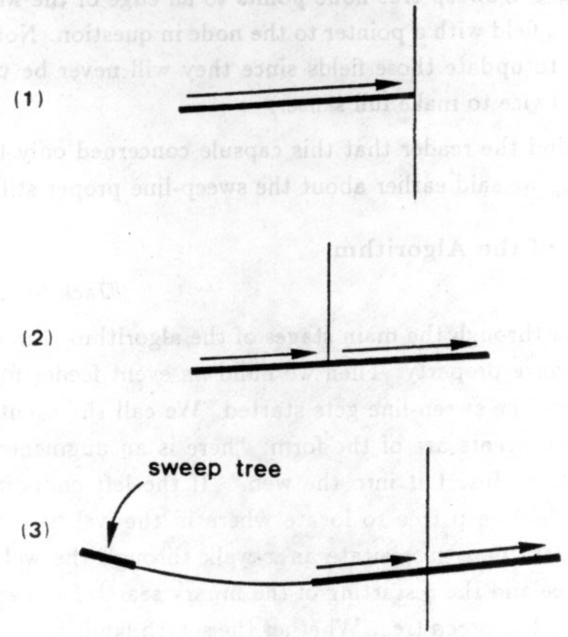


Figure 29. The three possible cases in x-walking: in case (1) we encounter a right end and we stop; in cases (2,3) we ignore the vertical edges and continue walking straight.

5. Note that the sweep tree is updated only when inserting the left endpoint of an original segment. In the schedule this represents usually only 1 out of $\Theta(\log n)$ events. Right endpoints are ignored and left augmented endpoints are handled via the appearance table, so one should expect the sweep tree to be often lagging far behind the sweep-line.
6. Caution should be used when updating the web-structure because of side-effects. For example, adding an edge to the web often causes the splitting of one or two other edges. Let e be such an edge (a nonvertical edge) and let e_ℓ and e_r be the two edges resulting from the split: e_ℓ (resp. e_r) is the piece of e to the left (resp. right) of the splitting point. Let s_i be the segment of S from which e originates. Suppose now that the appearance table contains a pointer to e (obviously, in its entry i). To comply with the definition of the appearance table, the ith entry will now have to point to e_r. Note that no pointers from the web to the appearance table are necessary for this update (which is good because we do not have such pointers in the web-structure). All we need to know is the name (i) of the segment. The second type of side-effects concerns the sweep tree. If a node of the tree happens to have a pointer to the edge e then it should be replaced by a pointer to e_ℓ. The main difference with the previous case is that we need to find out which node of the tree needs updating. First of all, note that at most one node of the sweep tree can point to the same edge e. This is because each node of the tree is in correspondence with a distinct level in the web. A simple solution is to augment the web-structure with the

appropriate information. If a sweep tree node points to an edge of the web then the record for this edge should contain a field with a pointer to the node in question. Note that during x-walks it is not even necessary to update those fields since they will never be used again. (This last sentence should be read twice to make full sense.)

Once again, we want to remind the reader that this capsule concerned only the implementation of the magic box and everything we said earlier about the sweep-line proper still applies verbatim.

Capsule 17. A Summary of the Algorithm

"Jack be nimble, Jack be quick"

Let us have a quick run-down through the main stages of the algorithm. First, we cut up the input segments to ensure the clearance property. Then we build an event feeder for on-line enumeration of the events of the schedule. The sweep-line gets started. We call the event feeder to know what our next move should be. All events are of the form: "here is an augmented segment whose left endpoint lies on the sweep-line. Insert it into the web." If the left endpoint is original then we perform a binary search in the sweep tree to locate where in the web the insertion should start. Each visit of a node in the tree may necessitate an x-walk through the web. An aborted x-walk causes a deletion from the tree and the restarting of the binary search from scratch; this is the only possible cause of deletion from the sweep tree. Whether the search lands us on a vertical chain or not makes no difference since we shall not use vertical skips (at least for the starting region). If the left endpoint of the segment to be inserted is augmented then we look up the appearance table to locate it in the web. If we land in the midst of a vertical chain then we preprocess it to enable vertical skipping (unless the preprocessing has already been performed). The insertion proper proceeds by way of a dovetailing walk around the boundary of the region to be cut up (using vertical skips if necessary). The dovetailing stops as soon as the exit point has been found and vertical edges have been computed (or found not to be necessary). The insertion goes from one region to the next until the right endpoint of the new segment has been reached. Special cases include the starting and terminating regions. Caution must be taken in cutting up edges of the web because of side-effects.

This concludes the algorithmic part of our discussion. It now remains to analyze the complexity of the algorithm.

Capsule 18. In the Final Analysis

"Oh, my little twopence, pretty little twopence"

The data structures as well as the algorithm have been simplified to the extreme. It now remains to prove that the time performance has not been jeopardized in the process. Although it is hard to see why the algorithm should have become much worse, proving that it is not so is a different matter. Let us review the potential rough spots in the analysis. All preprocessing, including the set-up of the schedule and the cutting of segments, can be done in $O(n \log n)$ time. As for the sweep proper, we only have to consider insertions of new segments. The cost of using the magic box can now be assessed. If the segment is not the first of its kind in the web then the appearance table gets the insertion started in constant time. Otherwise, the sweep tree is searched. Let us look at this in some detail.

If we are lucky only a single path of the sweep tree is visited and the search time amounts to $O(\log n + W)$, where W is the number of edges traversed during x-walks. Since these edges will never be visited again by x-walks (remember that x-walks never back up) we can charge them the cost of the x-walks. The term $O(\log n)$ is charged to the (original) left endpoint of the segment being inserted. Sometimes, recall, the search is terminated before completion and resumed from the root after deletion of a node. Using the same charging scheme to take care of x-walks, these aborted searches end up requiring each $O(\log n)$ time. This cost is charged to the right endpoint of the edge pointed to by the node being deleted. This endpoint is original so there will be at most n deletions. As described earlier, all side-effects are handled in constant time. To conclude to an $O(n \log n + k)$ time algorithm we must therefore prove that cutting up regions during insertion takes $O(k)$ amortized time. This would be easy if we used finger trees, but recall that Capsule 9 has ruled that out.

Instead, we cut up a region by dovetailing from the entrance point until the exit point is found. Also, we add all relevant vertical edges whenever necessary (the reader is invited to go back to Capsule 11 if this sounds mysterious). Our analysis will proceed from the general to the particular. We shall begin with the situation where the region to be cut up lies completely within the vertical strip bounding the current segment to be inserted. When this is done we will look at the special cases, i.e., starting or terminating regions, vertical edges, etc.

Let s be the segment to be inserted and R the region to be cut up. We assume that R satisfies what we call the ***spanning property***: this means that the region lies inside the vertical strip defined by the endpoints of s. We shall examine later what happens if we relax this property.

A. Assuming the spanning property: Our first observation is that no edge of R is vertical. Indeed, the schedule ensures that the strip does not contain any segment endpoint (original as well as augmented) that is already in the web. We can then define without ambiguity the leftmost (resp. rightmost) vertex v (resp. w) of the polygon R. Let p be the number of edges in the lower chain of R, that is, in the polygonal line running clockwise from w to v. Similarly, q denotes the number of edges in the upper chain of R. Trivially, we have $p \geq 1$ and $q \geq 1$, and $p + q$ is the number of edges bounding R. We require that the region R should hold exactly $p^2 + q^2 + \min\{p, q\}$ credits.

$$\textit{credit invariant for } R: \; p^2 + q^2 + \min\{p, q\}$$

By distinguishing among several cases we shall see how this invariant can be maintained under the insertion of s. We assume that 21 additional credits are available upon entering the region R. Each credit can be used to pay for a constant amount of computation. It will be obvious as we go along what exactly we mean by this, so there is not much point in trying to describe here what work can be accomplished with a single credit. As usual, any credit that is available but not used is thrown away. Regarding the 21 initial credits, we charge their cost to the points of intersection between s and the boundary of R.

Before we plunge into the case-analysis, it might help to shed a little light on the rather peculiar credit function chosen. Informally, the quadratic terms (p^2 and q^2) cover cuts which are neither too close to v or w or nearly tangent to R. Near-tangency requires little work but possibly many credits just to maintain the invariant: these are generated from scratch by the intersections newly created. When the cut is close to v and w, the quadratic terms become poor credit providers, hence the term $\min\{p, q\}$. The ever-alert reader, once again ahead of the action, will have undoubtedly recognized the kinship between dovetailing and the min operator. But this is enough hand-waving: time to roll up our sleeves and get into the nitty-gritty of the argument. There are two fundamental cases.

First Case: *Cutting a single chain.* Both entrance and exit points lie on the lower chain (the case of the upper chain is similar and can be obviously omitted). Note that our assumptions on the input set S rules out special cases, e.g., cutting through a vertex of R. Figure 30 illustrates the situation. Let p_1 be the number of edges below s and let $p_2 = p - p_1$. Note that $p_1 \geq 0$ and $p_2 \geq 2$. (The reader should ponder why we cannot have $p_2 = 1$; hint: remember our initial assumptions.) For convenience let us say that the region R is of type (p, q), meaning that R has p edges in its lower chain and q edges in its upper chain. Then a region of type (p, q) gives way to two regions, one of type $(p_1 + 2, 1)$ and the other of type $(p_2 + 1, q)$. We have two sources of credits available: those already present within the region R and the 21 extra credits given to help us make it through the cutting of R. We need more. How can we claim additional credits?

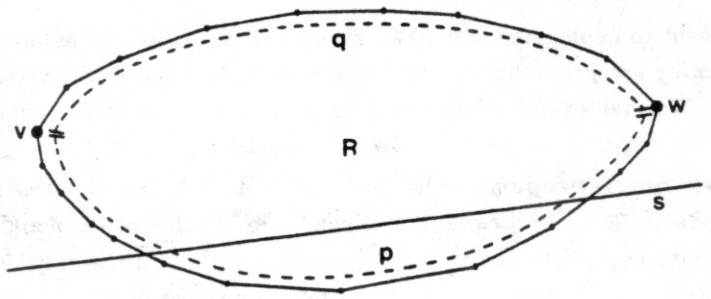


Figure 30. Segment s intersects the lower chain of region R. We assume here that R lies in the strip formed by the vertical lines passing through the two endpoints of s.

As illustrated in Figure 31, the fact that s extends both to the left and to the right of R (the spanning property) implies that each of the p_2 edges of the lower chain above or through s comes from a segment that intersects s (the points z_i in Figure 31). So it seems that asking each intersection to generate, say, 4 credits for our use is legitimate. As long as we do not make more than one, two, or say, a constant number of requests to the *same* intersection z_i later in the computation we will still be in the running for an optimal (amortized) insertion time.

Unfortunately, a given intersection may be charged too many times by this process. As it turns out, a closer look at these multiple requests reveals an intriguing pattern which points to a quick fix. To make our discussion more concrete, let $z_1, \ldots, z_{p_2}$ denote the intersections between s and the segments of S contributing edges (above or through s) to the lower chain of R: the order is taken to follow a counterclockwise traversal of the lower chain from v to w (Fig.31). We modify the charging scheme as follows: *only* the intersections $z_2, z_3, \ldots, z_{p_2-1}$ are charged 4 credits. In other words, nothing has changed but for the fact that we now leave the points z_1 and z_{p_2} alone. This gives us a total of $4p_2 - 8$ credits.

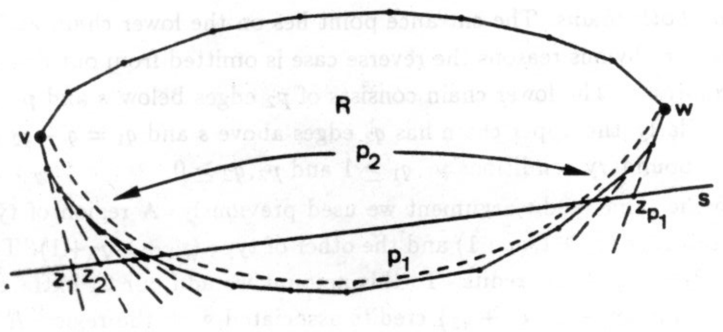


Figure 31. Because of the spanning property, segment s intersects the p_2 segments that support the edges of the lower chain lying above (or passing through) s.

This sounds familiar, doesn't it? Indeed, recall that the proof of Lemma 2 in Capsule 10 shows that each vertex is now asked to contribute to the wealth of at most two regions: one above and one below the two segments that meet at the vertex. We refer to this method of producing credits as the *intersection charge*. Let us now assess our wealth. The credits at the outset amount to

$$(p_1 + p_2)^2 + q^2 + \min\{p_1 + p_2, q\} + 21 + 4p_2 - 8,$$

while after the cut the two new regions combined must hold a total of

$$(p_1 + 2)^2 + 1 + \min\{p_1 + 2, 1\} + (p_2 + 1)^2 + q^2 + \min\{p_2 + 1, q\}$$

credits. The running time of the dovetailing can be covered by $\min\{p_1 + 1, p_2 + q\}$ credits. If the clockwise walk discovers the exit point then the cost of checking the spanning property is subsumed. Otherwise, the entire lower chain will have to be visited, which will take $O(p_1 + p_2)$ time. We combine all possible cases by charging $\min\{p_1 + p_2, p_2 + q\}$ credits for the dovetailing.

Note that we could multiply this number by any constant if it were more convenient. We must now estimate the balance, denoted Δ, which is the number of old credits (of region R), plus 21, plus the intersection charge, minus the number of credits needed for the two new regions and the work it takes to create them. We have

$$\Delta = 2p_1p_2 - 4p_1 + 2p_2 + 6 + \min\{p_1 + p_2, q\} - \min\{p_2 + 1, q\} - \min\{p_1 + p_2, p_2 + q\},$$

therefore

$$\Delta = 2p_1p_2 - 4p_1 + p_2 + 6 + \min\{p_1 + p_2, q\} - \min\{p_2 + 1, q\} - \min\{p_1, q\}.$$

Because $p_2 - \min\{p_2 + 1, q\} \geq -1$ and $\min\{p_1 + p_2, q\} - \min\{p_1, q\} \geq 0$, we easily derive $\Delta \geq 2p_1(p_2 - 2) + 5 > 0$. This concludes the analysis of the first case.

Second Case: *Cutting both chains.* The entrance point lies on the lower chain and the exit point lies on the upper chain (for obvious reasons the reverse case is omitted from our discussion). Figure 32 illustrates our assumptions. The lower chain consists of p_2 edges below s and $p_1 = p - p_2$ edges above or through s. Similarly, the upper chain has q_2 edges above s and $q_1 = q - q_2$ edges below or through s. We have the boundary conditions $p_1, q_1 \geq 1$ and $p_2, q_2 \geq 0$.

Let us go through the same credit argument we used previously. A region of type (p, q) gives way to two regions, one of type $(p_2 + 1, q_1 + 1)$ and the other of type $(p_1 + 1, q_2 + 1)$. The intersection charge provides $4(p_1 - 2) + 4(q_1 - 2)$ credits. To this amount we add our 21 extra credits and the $(p_1 + p_2)^2 + (q_1 + q_2)^2 + \min\{p_1 + p_2, q_1 + q_2\}$ credits associated with the region R. On the debit side the two new regions require a total of

$$(p_2 + 1)^2 + (q_1 + 1)^2 + 1 + \min\{p_2, q_1\} + (p_1 + 1)^2 + (q_2 + 1)^2 + 1 + \min\{p_1, q_2\}$$

credits. The cutting cost amounts to $\min\{p_1 + q_2, p_2 + q_1\}$. If the clockwise walk discovers the exit point then checking the spanning property requires another $O(q_1)$ time. A similar argument made with respect to the counterclockwise walk shows that $p_1 + q_1$ credits are sufficient to cover all this extra work. This makes $p_1 + q_1 + \min\{p_2, q_2\}$ an adequate amount of credits to pay for the entire dovetailing. The total balance comes to

$$\Delta = 2p_2(p_1 - 1) + 2q_2(q_1 - 1) + p_1 + q_1 - 1 + \min\{p_1 + p_2, q_1 + q_2\}$$
$$- \min\{p_2, q_1\} - \min\{p_1, q_2\} - \min\{p_2, q_2\}.$$

Using the fact that $p_1 + q_1 - \min\{p_2, q_1\} - \min\{p_1, q_2\} \geq 0$ and $\min\{p_1 + p_2, q_1 + q_2\} - \min\{p_2, q_2\} \geq 1$ we derive that $\Delta \geq 0$.

This concludes our discussion of the *easy* case where the spanning property is satisfied. We now turn to a much more delicate task: assessing the full implications of vertical edges.

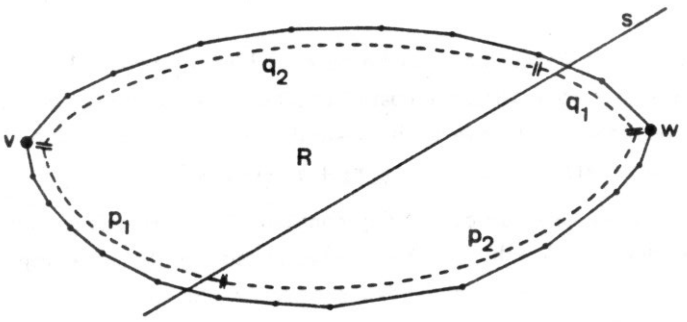


Figure 32. Segment s intersects both chains of R. Because of the spanning property, s intersects the segments of at least $p_1 + q_1$ edges bounding R.

B. Relaxing the spanning property: We shall begin with a general discussion of vertical edges. Assume that R already has two vertical edges on each side (vv_0 and ww_0), each of which extends beyond the vertical strip defined by the segment s (Fig.33(a)). Either edge may have been added during an earlier dovetailing traversal or it may be more simply a vertical stop. In the latter case, the edge vv_0 may contain the right endpoint v_1 of a segment (Fig.33(a)). Of course, this means that vv_0 is not really an edge but a vertical chain. It is interesting to observe that because of the scheduling of insertions this situation cannot happen with ww_0. Suppose now that v and the left endpoint of s share the same x-coordinate. Then vv_0 may contain an arbitrary number of right endpoints $v_1, v_2, \ldots$ (Fig.33(b)). Why is that so? If v is strictly to the left of s then the sweep-line has advanced too far ahead for any augmented right endpoint to be left stranded on vv_0. If, however, v and the left endpoint of s are vertically aligned, things are different. Any number of augmented right endpoints can lie on vv_0. Fortunately, we have made provision to enable vertical skips over vertical chains.

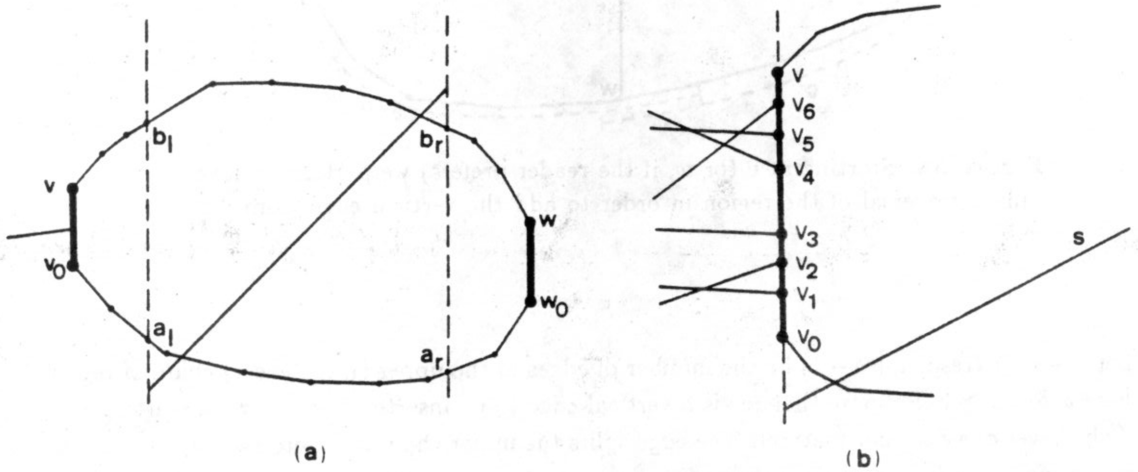


Figure 33. The vertical edges of a region may come from an earlier dovetailing process or they may be vertical stops. In the second case they are likely to consist of several edges (as illustrated in (b)).

For complete generality we shall assume a situation which paradoxically can never happen. That is, having v lie strictly to the left of the left endpoint of s and at the same time having an arbitrary number of left vertical edges $v_0v_1, v_1v_2, \ldots$ The combination of two factors allows us to ignore the presence of vertical chains in the complexity analysis. For one thing the dovetailing is not affected by it because of vertical skipping. The only case where we ignore vertical skips and walk right through vertical chains is when we insert a segment whose left endpoint is original. This can happen only once per vertical chain, however, so we can essentially ignore the effect on the complexity. Also, credit invariants are defined with respect to lower and upper chains only, which specifically excludes vertical chains. For these reasons, we can (mentally) replace vv_0 and ww_0 by single vertices and everything we say will still be valid.

Let us now turn to the cutting up of R caused by the insertion of s. For the sake of generality we shall continue to assume that R stretches over both sides of s. (The other cases will follow readily after that.) As a warm-up exercise, let us find out the cost of adding a vertical edge. Suppose we want to insert one of the vertical edges along the dashed lines of Figure 33(a). If we start from one of the endpoints, how much time will that take us using dovetailing? Figure 34 makes this question more concrete.

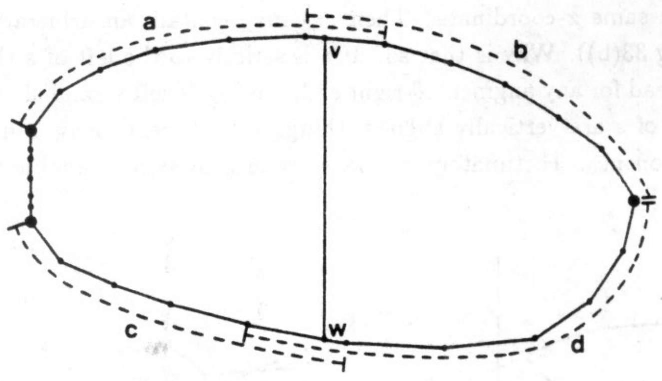


Figure 34. Starting at v (or w, if the reader prefers) we perform a dovetailing traversal of the region in order to add the vertical edge from v to w.

Let $a+b-1$ (resp. $c+d-1$) be the number of edges in the upper (resp. lower) chain of region R, where $a, b, c, d \geq 1$. Suppose that vw is a vertical edge to be inserted into the web, starting at v or w (whichever one does not matter). The edge splits the upper chain of R into two pieces: the left one consisting of a edges, and the other one comprising b edges. Similarly, the lower chain is split into two polygonal lines of respective size c and d. The credit invariant ensures us the disposal of $(a+b-1)^2 + (c+d-1)^2 + \min\{a+b-1, c+d-1\}$ credits. We will request 6 extra credits for the creation of the edge vw. (The new vertical edge will be delighted to pay for these extra credits.) The two regions created require a total of $a^2 + c^2 + \min\{a, c\} + b^2 + d^2 + \min\{b, d\}$ credits, to which we must add $\min\{a+c, b+d\}$ to pay for the dovetailing. This gives us a balance of

$$\Delta = 2(a-1)(b-1) + 2(c-1)(d-1) + 3 - \min\{a+c, b+d\} + \delta,$$

where $\delta = \min\{a+b, c+d\} - \min\{a, c\} - \min\{b, d\}$. To prove that Δ is nonnegative we distinguish among four cases which, by symmetry, cover all possible cases. In the first three cases we use the fact that $\delta \geq 0$.

(i) $a, b, c, d \geq 2$: $\Delta \geq 2(a-1) + 2(c-1) + 3 - (a+c) = a + c - 1 > 0$.

(ii) $a, b \geq 2$ and c or d is equal to 1: $\Delta \geq (a-1)(b-1)+(a-1)(b-1)+3-\min\{a+b+c, a+b+d\} \geq a-1+b-1+3-(a+b)-\min\{c,d\} \geq 0$.

(iii) $a = 1$ and $c = 1$: $\Delta \geq 3 - \min\{2, b+d\} > 0$.

(iv) $a = 1$ and $d = 1$: $\Delta \geq 3 - \min\{1+c, b+1\} + \min\{1+b, c+1\} - 1 - 1 > 0$.

What this little digression shows is quite remarkable. Any vertical edge can be added to the web at an amortized cost of 6 extra credits, provided that we know the starting point of the dovetailing. Returning now to the two vertical edges $a_\ell b_\ell$ and $a_r b_r$ (Fig.33(a)), let c_ℓ (resp. c_r) be the minimum number of credits necessary to cover a walk around the boundary of R from a_ℓ to b_ℓ (resp. a_r to b_r). The value of c_ℓ (resp. c_r) is equal to the number of nonvertical edges on the piece of boundary running from a_ℓ (resp. a_r) to b_ℓ (resp. b_r) clockwise or counterclockwise, whichever is shorter. We define c as the maximum of c_ℓ and c_r.

Suppose that some benevolent oracle inserts the two vertical edges *at no cost* for us. We claim that once all the credit invariants have been restored following these two insertions we will have in our hands at least c extra credits to dispose of. Without loss of generality assume that $c = c_\ell \geq c_r$. We shall request the oracle to insert $a_\ell b_\ell$ first. Then we restore the credit invariant. Since the cost of insertion is null, our little digression above shows that the number of credits released in the restoration can pay for a dovetailing walk from a_ℓ to b_ℓ. This gives us the extra c credits desired. Since the next insertion will not consume any credit (rather, it will release some more) we can keep these c credits for ourselves. Note that if $c = c_r$ then the order of insertion for the oracle must be reversed. This establishes our claim.

By inserting the two vertical edges the oracle creates a configuration which satisfies the spanning property. The cost of inserting the new segment s is therefore under control, as was shown earlier. Let us examine the difference in credit accounting between the fictitious, oracle-aided insertion and the real one. As far as the restoration of credit invariant is concerned there is no real difference. The splits may occur in a different order but the end result is just the same. Thus we only have to worry about the cost of dovetailing. More precisely we have to show that the credits used to pay for the fictitious dovetailing added to the c extra credits suffice the cover the cost of the actual dovetailing. Is that clear?

To avoid rather tedious combinatorial derivations we will use a trick. It may sound a little suspicious at first, but a second reading should chase away any lingering doubt. We need the useful notion of *delayed* dovetailing. Imagine that in the course of dovetailing one of the two (concurrent) walks gets stopped, say, at a red light, and misses its turn to move t consecutive times, while the other walk keeps going as though nothing was happening. It is fairly obvious that the red light will not delay completion by more than $2t$ steps. To see this, imagine that animated by a spirit of solidarity the other walk decides to stop while the light is red. Then the dovetailing will go through exactly the same sequence as before, only $2t$ steps behind.

The relevance of this argument is obvious. Since none of the termination conditions are tested outside of the vertical strip, we may mentally replace the polygonal line $a_\ell, \ldots, v, \ldots, b_\ell$ by the edge $a_\ell b_\ell$ and place a red light at b_ℓ for a duration equal to the length λ of the polygonal line. Let λ'

be the length of the polygonal line $a_\ell, \ldots, w, \ldots, b_\ell$. Recall that $c_\ell = \min\{\lambda, \lambda'\}$. Since the two concurrent walks in the dovetailing never cross each other, no more than c_ℓ edges of the polygonal line $a_\ell, \ldots, v, \ldots, b_\ell$ can ever be visited. This means that it is safe to set the duration of the red light at b_ℓ to the value c_ℓ instead of λ.

We play a similar trick with respect to the polygonal line $a_r, \ldots, w, \ldots, b_r$, placing a red light at b_r of duration c_r. With these modifications the real dovetailing is nothing but a delayed version of the fictitious dovetailing. Its cost will be covered by the credits used to pay for the nondelayed dovetailing (which we have) plus a number of credits proportional to the duration of the red lights, i.e., $c_\ell + c_r$. We have c credits for that purpose, which is therefore sufficient. We are applying here a general rule concerning credits. If credits are to be consumed directly then we can always do with any number proportional to what is strictly needed. If, however, they are to be recycled —for example, to maintain credit invariants— then we must use what is available and not a multiple thereof.

This concludes our analysis of the cutting operation for the case where the region under consideration is neither the first nor the last one to be cut up during the insertion. We have omitted the case where the region stretches over on one side of the vertical strip but not on the other. This is really a subcase of the one discussed above, and we trust that the diligent reader will eagerly and easily work it out for himself.

C. At the beginning and at the end: Our final word on the analysis of the running time concerns the starting and terminating regions cut up during the insertion of segment s. Let us begin with the starting case.

If s is a brand-new segment, its (original) left endpoint is located in the web via the magic box and the insertion of the vertical stop comes for free. (Recall that, although original, the left endpoint could very well fall right in the midst of a vertical chain. In that case, of course, no vertical stop is needed.) As we have seen earlier the credit invariant is maintained with no difficulty. The remainder of the insertion proceeds as usual. If the insertion is of the continuing type then locating the (augmented) left endpoint comes for free, courtesy of the appearance table. Everything is as usual, except for one minor point. Remember the vertical chains. Well, the segment could come out of a vertex somewhere in the middle of such a chain. As we said earlier vertical chains can be treated as single vertices using floors and ceilings, so it is possible to avoid traversing these chains. End of the problem.

We now turn to the case of a terminating region. Its treatment is really no different from that of any intermediate region. Here is why. We will modify the position of the segment s in such a way that the algorithm will not realize it, yet everything will conform to the treatment of an intermediate region. As shown in Figure 35 the trick is to bend and stretch the segment so as to create an exit point. Nothing in the insertion process, including the addition of the vertical edge, will be altered by this modification. This proves that our previous analysis applies just the same.

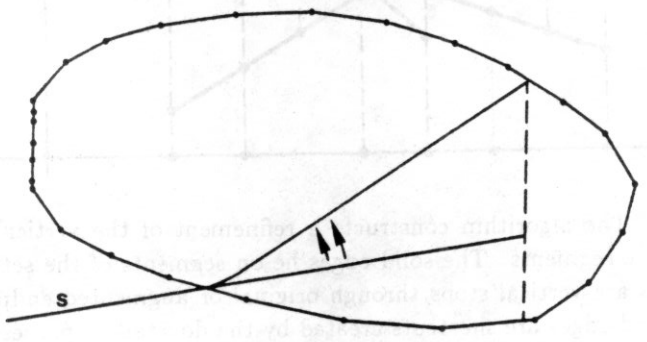


Figure 35. By bending segment s at its entrance into the region so that it intersects the boundary at the vertical projection of its right endpoint, we can reduce the case of a vertical stop to the general case of vertical edge addition. This device is used only in the counting argument, and not in the algorithm.

This brings to an end the analysis of the running time of the intersection algorithm. We have proven that n segments in the plane can be intersected in $O(n \log n + k)$ time, where k is the number of intersections. What is the space complexity of the algorithm? Obviously, no more than $O(n \log n + k)$. Can we somehow reduce this to $O(n + k)$? This fascinating question will be the subject of our next capsule.

Capsule 19. Lowering the Storage Requirement

"A halfpenny loaf will serve us all"

Figure 36 depicts the output produced by the algorithm on the input set shown in Figure 19. We have the entire vertical map (after another pass through the web) and more. Indeed, the augmented endpoints are all there, in addition to the numerous vertical edges which accompany them. It is easy to clean up the map by getting rid of all these extra edges and vertices. A depth-first search through the map allows us to do the mop-up in time linear in its size. (Implementing this operation is actually not nearly as simple as it sounds. It is nevertheless a standard exercise in graph manipulation which we will leave as is.)

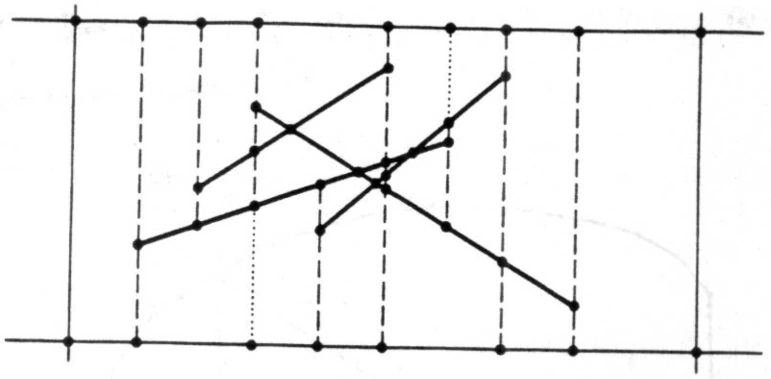


Figure 36. The algorithm constructs a refinement of the vertical map defined by the segments. The solid edges lie on segments of the set. The dashed edges are vertical stops through original or augmented endpoints, and the dotted edges are shortcuts created by the dovetailing process.

The obvious question remains, however: can the storage requirement be kept in $O(n+k)$ during the course of the algorithm itself? In practice, no one should really care too much about this. Storage limitations are less and less of a problem, especially when it is a question of $O(n \log n + k)$ versus $O(n + k)$. Furthermore, one should not forget that k will often be the dominating term, anyway. But for the sake of a nice theoretical result, let us show how to achieve $O(n + k)$ storage. We will give a quick-and-dirty method.

Intuitively, one would like to argue that, although a segment of S may be split into a logarithmic number of pieces in the augmentation process, at most a constant number of the cuts need be present in the web-structure at any given time. Consider the edges produced by the algorithm which will eventually go after the mop-up. These fall into three categories: (i) the vertical stops created at augmented endpoints (and augmented endpoints only); (ii) the vertical edges created to provide shortcuts; (iii) the edges of the web adjacent to any of the above. Edges of type (i) or (ii) are called *trouble-makers*.

Suppose that we have a *regular* mop-up algorithm to get rid of trouble-makers (not necessarily all of them): such an algorithm is called regular if it removes edges one at a time in such a way that at all times the current web-structure forms a convex planar subdivision. Getting rid of edges (iii) can be done on the fly by adding a few instructions to any regular mop-up algorithm. This involves checking locally for vertices of degree 2 and replacing pairs of collinear edges by their unions. We omit the details. Of course, we should not expect to have all edges of type (iii) removed by this method if it fails to get rid of all trouble-makers.

We need a piece of terminology to characterize special pairs of vertices in the map. Consider the insertion of segment $s = pq$. If its left endpoint p is original we call on the magic box to determine the point v (resp. w) on the edge immediately above (resp. below) p. The pair (v, w) is called a *trigger*. If p is an augmented endpoint, the insertion begins by looking up the appearance table. This leads us to an edge of the web whose right endpoint is precisely p. The vertices v and w immediately above and below p form two pairs (v,p) and (p,w), both of which are triggers. Note in passing that with the requirement of using vertical skips, the vertices v and w are not necessarily visited during the dovetailing (they are visited only if they also happen to be at the very top or bottom of the corresponding vertical chain). There are other possibilities for a pair of vertices to be a trigger. If vw is a vertical edge (a shortcut) vertically aligned with p and created while inserting s, then the pair (v,w) is also called a trigger. This holds regardless of whether p is original or augmented. Now what about the other endpoint q? Every vertical stop vw vertically aligned with q and created while inserting s gives rise to two triggers (v,q) and (w,q), following the usual maneuver. Finally, every vertical edge (shortcut) vertically aligned with q and created while inserting s also gives a trigger. Triggers are kept in an array of queues. Each entry in the array corresponds to a distinct x-coordinate. When a trigger is created it is stored in the queue corresponding to the x-coordinate of its associated endpoint (p or q). With a proper pointing mechanism we can easily ensure that inserting a new trigger is done in constant time.

Our regular mop-up algorithm will work as follows. Right before L moves to its next location (or at the very end of the sweep), we examine the queue corresponding to the current x-coordinate and remove all duplicates. Then we remove each trigger (v,w) from the queue and we initiate a clockwise traversal from w to v (assuming without loss of generality that v lies above w). If we encounter a trouble-maker during the traversal, we remove it and stop. Otherwise we just go on until v is reached.

This is our mop-up algorithm. It is obviously regular, so we can complete it with the removal of edges of type (iii) as mentioned above. It remains for us to show that the mop-up removes enough edges to keep the storage requirement of the overall algorithm down to $O(n+k)$. Then we will show that the mop-up procedure itself runs in time $O(n\log n+k)$. Finally, we will prove that the mop-up does not get in the way of the main intersection algorithm, that is, does not destroy information needed by the algorithm. Finally, we will show how the event feeder itself can be confined to $O(n)$ storage. So, we have our work cut out.

A. The storage requirement: It suffices to prove that the total number of trouble-makers remains in $O(n+k)$ at all times. Let us look at a snapshot of the web in the course of the algorithm. Since trouble-makers are associated with augmented edges, any one of them can be removed without violating the convexity of the region that contains it. Let us conceptually remove all of them at the completion of the whole intersection algorithm. This gives us a convex planar subdivision $\mathcal{S}$. Each trouble-maker has associated with it a unique region of $\mathcal{S}$ that contains it. We will show that during the course of the algorithm no region of $\mathcal{S}$ ever contains more than one trouble-maker to the left of L at any given time. Since $\mathcal{S}$ has $O(n+k)$ regions and the number of trouble-makers lying to the right of L is $O(n+k)$ (from the previous complexity analysis), our point will thus be made.

Suppose that our claim is not true and that we have two trouble-makers in the same region of $\mathcal{S}$ which lie left of L. If there are more than two such trouble-makers, choose the two rightmost of that type. The rightmost trouble-maker was inserted in the trigger-queue before L swept across it, therefore it must have been subsequently removed from the queue and caused the leftmost trouble-maker to be mopped up. Note that this argument only works because we wait for L to reach the rightmost trouble-maker. In other words, our claim might be wrong as far as what lies right of the sweep-line is concerned.

B. The mop-up time: Once again consider a region of $\mathcal{S}$. None of its edges are visited more than twice while mopping (once for each of its two adjacent regions). A trigger can be regarded as a barrier which effectively blocks off any further attempt at repeating the same traversal. How about the removal of duplicates in the trigger-queues? To ensure linear time we must label the triggers in a canonical fashion so that bucket sort can be used. This is an easy exercise.

C. How safe is mopping? Are we not by any chance sawing off the branch on which we are sitting by blindly destroying the web-structure? Since triggers, as mentioned above, act as barriers the cutting process itself is fairly safe. The dovetailing will never have to complain that an edge it should have been visiting is no longer there. How about the magic box, however? Remember that the web edges pointed to by the sweep tree can lag arbitrarily far behind the sweep-line. So, it could certainly happen that an x-walk traverses an edge of type (iii) which the mop-up will have deleted. What will save the show, however, is the fact that the mop-up does not compromise the integrity of the web as the representation of a planar convex subdivision. Furthermore, since only vertical edges are effectively removed (nonvertical edges are only merged) and vertical edges are essentially ignored by x-walks, nothing is in jeopardy if, of course, one is to use caution in the implementation of the mop-up. For example, it must be clearly understood that the pointers between the elements of the trigger-queues and the edges of the web must go both ways. The reason is that a modification of the web may affect a vertical edge pointed to by a trigger-queue.

D. The event feeder: This refers to the computation of the schedule of events. The method which we gave earlier really requires $\Theta(n \log n)$ storage in the worst case. Although the number of events could be of that order of magnitude, it would be nice to have a co-routine `GetNextEvent` which feeds the next event to any one that calls it, and requires only linear storage.

The event feeder can be thought of as a preorder traversal of the segment tree associated with the segments. A segment tree for 4 segments is depicted in Figure 19. Our goal is to perform the preorder traversal without keeping the entire segment tree in storage at any point in time. As explained in Capsule 12, only the x-coordinates of the endpoints are important. Let $C = (z_0, z_1, \ldots, z_{2n-1})$ be the sorted sequence of distinct x-coordinates that we get from the segments. For each pair of integers $j \geq 0$ and $k \geq 0$, there is a node of the segment tree that corresponds to the interval $[z_{j2^k}, z_{(j+1)2^k}]$, provided that $(j+1)2^k \leq 2n-1$. Note that the segment tree is a complete binary tree whose leaves are $\lceil \log n \rceil + 1$ nodes away from the root. The ***node list*** of a node v is the set of segments whose x-intervals contain the interval of v but not the interval of v's parent. We define the ***extended node list*** of v as the set of segments that belong to the node list of v or any descendent of v. Clearly, the extended node lists are supersets of the node lists. Still, a single segment can be in the extended node lists of at most 4 nodes on a single level of the tree. This is because a segment is in the extended node list of a node v but not its regular node list only if an endpoint of the segment falls within the interval of v. Thus, extended node lists need $O(n \log n)$ storage altogether. We will see that extended lists provide enough information to allow for the preorder traversal of the tree without constructing the entire tree.

Consider a path from the root to a leaf in the segment tree. We claim that the sum of the lengths of all extended node lists on this path is $O(n)$. For one thing, the sum of the lengths of the node lists themselves is $O(n)$. The reason is that a segment can belong to at most one node list on the path. Also, the sum of the lengths of the extended node lists, not counting the regular node lists, is in $O(n)$. Recall that a segment is in the extended but not in the regular node list of a node v only if one of its endpoints (or rather the x-coordinate of one endpoint) falls within the interval of v. Because we assume that no two endpoints share the same x-coordinates, the number of endpoints in the intervals of the nodes on the path decreases by a factor of two each time we go down one level. This gives us a geometric series which adds up to $O(n)$.

The event feeder that we have in mind is a recursive procedure where each call is represented by a node of the segment tree. Consider a (recursive) call of this procedure represented by node v. The arguments for this call are the extended node list of v as well as two integers j and k, where $[z_{j2^k}, z_{(j+1)2^k}]$ is the interval of v. First, we compute the node list of v (the segments in the extended node list that cover $[z_{j2^k}, z_{(j+1)2^k}]$) and feed those for insertion into the web structure in order of decreasing right extent. This ordering does not require sorting if the segments in the incoming extended node list are already ordered in this fashion. Next, we compute the extended node list of v's left child (all segments that are not in v's node list and whose x-intervals intersect $[z_{2j2^{k-1}}, z_{(2j+1)2^{k-1}}]$) and we call the procedure recursively for this child. Finally, we compute the extended node list of v's right child (all segments that are not in v's node list and whose x-intervals intersect $[z_{(2j+1)2^{k-1}}, z_{(2j+2)2^{k-1}}]$) and we call the procedure recursively for this child of v. It should be clear that this procedure only stores the extended node list of a single path of the segment tree at any point in time, so that the required storage is $O(n)$.

Capsule 20. Discussion of an Implementation

"Boys and girls, come out to play"

We shall not give a detailed account of the program. Rather, we shall describe the main data structures and highlight the tricky spots of the implementation. We begin with something very simple, but which one seldom gets right the first time around: a primitive for intersecting two segments.

A. A useful geometric primitive: The function `SegInter` computes the intersection of two segments `seg1` and `seg2`, whose endpoints are specified by the suffixes `_left` and `_right`. The function returns 1 if the segments intersect and 0 if not. (A segment AB is understood here as the locus of points $\alpha A + (1-\alpha)B$, where $0 \le \alpha \le 1$.) The function is easily enhanced to compute the intersection explicitly whenever appropriate.

The (very useful) function `ccw` (A, B, C) characterizes all possible relative orientations of the three points A, B, C. It works as follows. If A and B are distinct then the function returns 1 if A, B, C form a right turn and -1 if they form a left turn. It gives 0 if C lies in AB, 2 if B lies in AC, and -2 if A lies in BC. Finally, if $A = B$ then the function returns 0 if $A = B = C$ and 2 if C is distinct from A.

Many other geometric primitives are needed in the algorithm, but we shall restrict ourselves to these two which are by far the most important.

```
typedef struct {
        double x, y;
        } point;

SegInter(seg1_left, seg1_right, seg2_left, seg2_right)
point seg1_left, seg1_right, seg2_left, seg2_right;
{
    int a, b, c, d;
    a = ccw(seg2_left, seg2_right, seg1_right);
    b = ccw(seg2_left, seg2_right, seg1_left);
    c = ccw(seg1_left, seg1_right, seg2_right);
    d = ccw(seg1_left, seg1_right, seg2_left);
    return (a×b≤ 0) && (c×d≤ 0);
}
```

```
ccw(A, B, C)
point A, B, C;
{
    double cax, cay, cbx, cby, det, abx, aby;
    cax = A.x−C.x; cay = A.y−C.y;
    cbx = B.x−C.x; cby = B.y−C.y;
    if ((det=(cax×cby)−(cay×cbx))< 0) return 1;
    if (det> 0) return −1;
    if ((cax×cbx≤ 0)||(cay×cby≤ 0)) return 0;
    abx = B.x−A.x; aby = B.y−A.y;
    if ((cax×abx> 0)||(cay×aby> 0)) return −2;
    return 2;
}
```

B. Driving the program: Let us now turn to the top level of the program. The structure **subsegment** is used to feed the next event to the sweep. We need to know the name of the next segment to be inserted (**segindex**) as well as its range on the x-axis. Remember that the segment will in general be a subsegment of one of the original segments. We specify this range by two x-coordinates **Xcoord[Xleft]** and **Xcoord[Xright]**. The array **double *Xcoord** contains the $2n$ original endpoints in nondecreasing order. The function **InitSweep** initializes all the global structures used during the sweep. The function **GetNextEvent** is the event feeder: it returns the status of the current event. An extra event (**status = ALLOVER**) is added at the very end of the sweep to signal termination. By side-effect the function **GetNextEvent** returns all the information needed for processing the next insertion (**HandleEvent(event)**). There is little to say about the event feeder, **GetNextEvent**, which we have not already discussed, so let us move right along. Next on the agenda is, of course, the meat of the meal: **HandleEvent**. But before that, we must introduce the global variables of the program.

```
typedef struct {
      int segindex;
      int Xleft, Xright;
      } subsegment;

Sweep()
{
    int status;
    subsegment event;
    InitSweep ();
    while ((status = GetNextEvent(&event)) != ALLOVER)
      HandleEvent(event);
}
```

C. The global variables: The sweep tree is a collection of records of type **csnode**. Access to its root is done via a handle **csHead** which points to the root **csRoot** of the tree. For convenience, we make **csHead** a node in its own right: it corresponds to a dummy infinite horizontal edge below all the segments. Speaking of dummies, each leaf points to a dummy node **z**. As expected, a node of the sweep tree consists of three fields: (i) a pointer to the web-structure; (ii) an array of two pointers (to the children of the node); (iii) a *color* (recall that the sweep tree is a red-black tree). We also define a few obvious macros for working on the sweep tree.

```
typedef struct csNODE {
      struct webNODE *toweb;
      struct csNODE *link[2];
      int red;
      } csnode;
csnode *csHead, *z;
#define lchild(p) ((p)→link[LEFT])
#define rchild(p) ((p)→link[RIGHT])
#define csRoot rchild(csHead)
```

The web-structure consists of two types of records: the vertical edges and the others. The need to distinguish between vertical and nonvertical edges comes from the use of vertical skips. In practice, one will probably want to forget about them altogether. This is what we have done here. It is much hassle for little benefit, as the size of vertical chains tends to be quite small in practice. Of course, in theory, things are a tad different.

```
typedef struct webNODE {
        int segindex, edgetype;
        point leftV, rightV;
        struct webNODE *Lcw, *Lccw, *Rcw, *Rccw;
        csnode *tocs;
        } webnode;
```

A webnode must tell us (i) the name of its supporting segment (**segindex**); (ii) whether it is vertical or not (**edgetype**); (iii) its two endpoints, **leftV** and **rightV**; (iv) pointers to the four adjacent edges: for example, the edge reached clockwise from the leftmost endpoint (**Lcw**); (v) a pointer (***tocs**) to the node of the sweep tree which might be pointing to the edge. Two minor comments about these specifications.

If the edge is vertical then **segindex** is used differently. Instead of being the name of a segment of S it is used as an index into **Xcoord**. More precisely, the value of **Xcoord[segindex]** is equal to the x-coordinate of the edge. Technically, the fields **leftV** and **rightV** are superfluous. The endpoints of the edge are defined implicitly by the adjacent edges. Moreover, their representations as **double** is bound to introduce numerical errors into the computation. If all computations were to be carried out exactly then these fields would be of no use whatsoever. If this is not the case, however, omitting them is a mistake. Indeed, it makes any manipulation of the web-structure very time-consuming. The fields (**leftV** and **rightV**) should not be considered as part of the web-structure per se but as local caches used to speed up the computation.

The following may not look like much, but it is what saves us from an $\Theta(n \log^2 n + k)$ algorithm: the all-powerful appearance table.

webnode **SeginWeb

SeginWeb[i] is NULL if the ith segment of S has not yet contributed any edge to the web. Otherwise, it points to the rightmost webnode that is supported by the segment in question.

What about initialization? The only thing to mention is the initial state of the web. It consists of 12 edges, as shown in Figure 37. The illustration is self-explanatory. In this way, the only edges ever to necessitate special care in the course of the algorithm are the 8 edges sticking out from the corners. Of course, these edges are really never used further, so everything is all right.

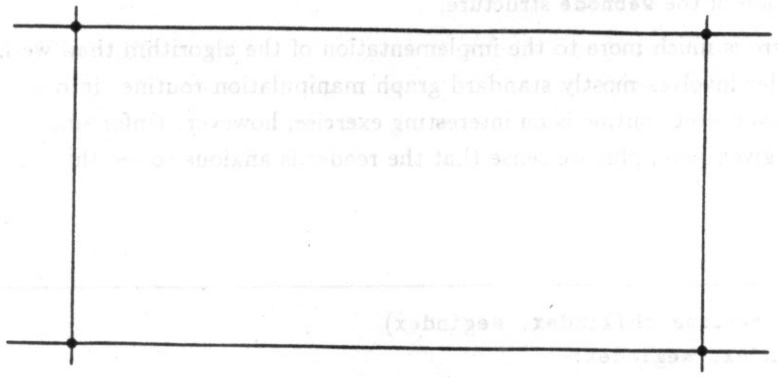


Figure 37. The initialization of the web structure.

D. Using the sweep tree: The function `HandleEvent` distinguishes among three distinct starting cases. Once the first region has been traversed all cases mix into one. The three cases are: (i) here comes a brand-new segment whose left endpoint falls stranded in the middle of a region; (ii) here comes a brand-new segment whose left endpoint falls right on a vertical edge (this is hardly an unlikely case, by the way); (iii) an old segment is being extended. The last case is checked by looking up the appearance table `SeginWeb`.

In case of (i,ii) the sweep tree is used, along with x-walks and the whole bit. The function `cs_search(Xindex, segindex)` returns the csnode pointing to the webedge at or above the segment of name `segindex` along the vertical line $x =$ `Xcoord[Xindex]`. An x-walk always covers the least distance possible: if two consecutive webedges have their common endpoint right at $x =$ `Xcoord[Xindex]`, the x-walk stops right upon encountering the *left* edge. The presence of two undefined subroutines requires some explanations. The call to the function `web_NextOnLevel` updates the first (dummy) node of the sweep tree in symmetric order, `csHead`. Let us say a few words about `web_NextOnLevel`. The function implements the elementary move of an x-walk. It returns the next edge right of and on the same level as the web edge passed as its argument. Whenever a vertical edge is encountered the function tries to ignore it unless it cannot. In that case the x-walk must abort: the function returns `NULL`. Further use of the function is made during the implementation of what we called f_v earlier: the function `compare(p, Xindex, segindex)`. It returns `RIGHT` (resp. `LEFT`) if x-walking from `p`$\rightarrow$`toweb` leads below the segment labelled `segindex` along the vertical line $x =$ `Xcoord[Xindex]`. If the x-walk cannot proceed all the way to the line the function returns `CANCEL`. Finally, the function `SegBelow( Xind, SegBelow, SegAbove)` returns 1 if the line passing through the segment labelled `SegBelow` lies strictly below the line passing through `SegAbove` along the vertical line $x =$ `Xcoord[Xindex]`. Else it returns 0.

As we mentioned earlier we must be particularly cautious when modifying the web-structure. Whenever an edge e is split into two edges it is important to check whether the sweep tree and the appearance table need updating as a result. We already discussed this problem in detail and there is little use adding to it. We want to mention, however, that these side-effects motivate the field **tocs** in the definition of the **webnode** structure.

Of course, there is much more to the implementation of the algorithm than we have discussed here. The remainder involves mostly standard graph manipulation routines into which there is no need to go. The dovetailing routine is an interesting exercise, however. Unfortunately, the code is a bit too long to be given here, plus we sense that the reader is anxious to see this paper come to an end.

```
csnode *cs_search(Xindex, segindex)
int Xindex, segindex;
{
    webnode *web_NextOnLevel();
    csnode *p, *fp, *succ;
    webnode *q;
    int status;
    p = csHead; q = p→toweb;
    q→tocs = NULL;
    while (q→rightV.x < Xcoord[Xindex] )
      q = web_NextOnLevel(q);
    p→toweb = q; q→tocs = p;
    do
      { p = csRoot; fp = csHead;
        do
          { status = compare(p, Xindex, segindex);
            switch(status)
              { case CANCEL: cs_delete(p, fp); break;
                case LEFT: fp = succ = p; p = lchild(p); break;
                case RIGHT: fp = p; p = rchild(p); break;
              }
          }
        while ((status != CANCEL)&&(p != z));
      }
    while (status == CANCEL);
    return succ;
}
```

```
webnode *web_NextOnLevel(WebEdge)
webnode *WebEdge;
{
    webnode *p;
    p = WebEdge→Rcw;
    if (p→edgetype == VERTICAL)
      { if (p→Lcw→edgetype == VERTICAL) return NULL;
        else return p→Lcw;
      }
    p = WebEdge→Rccw;
    if (p→edgetype == VERTICAL)
      { if (p→Rccw→edgetype == VERTICAL) return NULL;
        else return p→Rccw;
      }
    if (p→Lcw == WebEdge) return WebEdge→Rccw;
    else return WebEdge→Rcw;
}
```

```
compare(p, Xindex, segindex)
csnode *p;
int Xindex, segindex;
{
    webnode *web_NextOnLevel();
    webnode *q;
    q = p→toweb;
    while (q→rightV.x < Xcoord[Xindex])
      { q = web_NextOnLevel(q);
        if (q == NULL)
          { p→toweb→tocs = NULL;
            return CANCEL;
          }
      }
    p→toweb→tocs = NULL; p→toweb = q;
    q→tocs = p;
    if (SegBelow(Xindex, p→toweb→segindex, segindex))
      return RIGHT;
    return LEFT;
}
```

Capsule 21. Could We Have Done It Better?

"Bake a cake as fast you can"

We could claim without proof that $\Theta(n \log n + k)$ is as fast as any intersection algorithm can get and probably get away with it. It is, indeed, not too difficult to see. Following (Dobkin and Lipton [9] and Ben-Or [2]) we model the complexity of an algorithm by an algebraic decision tree. Each step of the computation corresponds to the visit of a node in the tree. With each node is associated a polynomial of fixed degree: the sign (and/or nullity) of its value determines the branching of the computation. A node can produce the output of at most one intersecting pair. To a given collection of n segments corresponds a certain path in the tree whose traversal will produce the output of all k intersecting pairs. Usually, the minimum height of such a tree determines a lower bound on the time complexity. Here, of course, we already know that the height will have to be $\Theta(n^2)$. A more refined complexity measure should incorporate both input and output sizes as parameters. We define the function $T(n, k)$ as the minimum, over all valid computation trees, of the maximum length of any path associated with a collection of n segments intersecting in exactly k points. Keep in mind, however, that to be valid a computation tree must be able to handle ***any*** input of n segments.

We will now prove the existence of a constant c such that $T(n, k) \geq c(n \log n + k)$, for all *feasible* pairs (n, k).† Our claim is obviously true if k is the dominating term, so we can safely assume that $k < n \log n$. As we already said, although $T(n, k)$ concerns only a subtree of the computation tree, it is crucial that the whole tree should be valid and not just the subtree in question. This will rule out arguments of the sort: $T(n, 0)$ is constant since the algorithm does not even have to get started! The point is that the algorithm does not know that fact ahead of time.

For any $n > n_0$ and any $k < n \log n$ there exists a set of n segments that intersect in exactly k points and follow the schema of Figure 38. Half the segments are points on a horizontal line while the remaining segments lie in a grid fashion. Obviously, the horizontal segments in the grid can always be arranged so as to intersect exactly k vertical segments.

† Note that k and n are not completely independent; for example, we must have $0 \leq k \leq \binom{n}{2}$. It is easy to see, however, that this is the only condition. Here is a simple way to generate a set of n segments with any desired number of intersections. Start with n (unbounded) lines in general position. Intersect the lines with a large disk containing all the $\binom{n}{2}$ intersection points. Now take each segment in turn and shrink it to a point. We will thus go from $\binom{n}{2}$ intersections to 0 while losing only one intersection at a time.

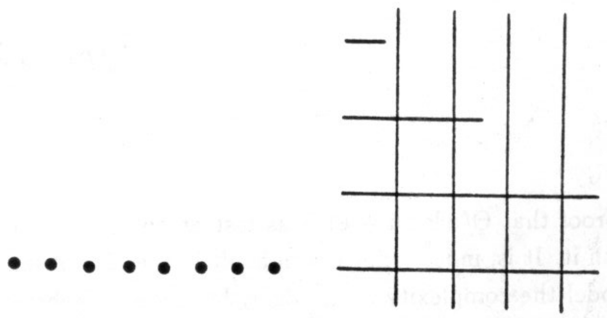


Figure 38. Half the segments are single points and lie on a horizontal line. The other half consists of horizontal and vertical segments which intersect in $k < n \log n$ points.

Consider the grid segments as fixed once and for all. This means that their coordinates appear not as variables in the polynomials of the computation tree but as coefficients. Let $p = \lfloor n/2 \rfloor$ and $x_1, \ldots, x_p$ be the positions of the $\lfloor n/2 \rfloor$ points on the horizontal line. Let us define the *scope* of a computation path to be the sequence of points $(x_1, \ldots, x_p)$ whose corresponding sets of segments are accepted by the path in question. The sequence $(x_1, \ldots, x_p)$ can also be interpreted as a point in p-space. It is clear that the scopes must partition the set

$$K = \{ (x_1, \ldots, x_p) \mid \prod_{1 \le i < j \le p} (x_i - x_j) \ne 0 \}.$$

A typical scope will consist of a few connected regions of $\Re^p$, every one of which must fall completely within a connected component of K. The reason is that if a scope had a connected component astride two regions of K then it would contain a piece of some hyperplane $x_i = x_j$. This would invalidate the computation tree since we could thus add an extra intersection and keep the algorithm unaware of it.

An immediate consequence is that the number of paths accounted in $T(n,k)$ is at least equal to the number of connected components in K (which is equal to $p!$) divided by the maximum number of connected components of a scope. Adapting a classical result on the Betti numbers of algebraic varieties, Ben-Or [2] shows that the number of connected components is dominated by b^{p+h}, for some constant $b > 0$, where h is the maximum number of polynomials associated with a path of the tree. We have $h \le T(n,k)$, therefore the number of paths is at least equal to $p!/b^{p+T(n,k)}$. Since the tree is ternary this number is at most equal to $3^{T(n,k)}$. Putting things together, we easily derive that $T(n,k) = \Omega(n \log n)$, if $k < n \log n$. We can now state our lower bound theorem with assurance.

Lemma 4. [A Lower Bound] – *Any algorithm which correctly computes segment intersections among any set of n segments will take on the order of $n \log n + k$ time in the worst case, for any value of the input size n and any feasible value of the output size k.*

Capsule 22. The End

"The nightingale sings when we are at rest"

We have reached the end of our journey. The main open problem remains to work out some way of cutting down the storage requirement to $O(n)$. Now, it may not be possible to do so while keeping a running time of $O(n \log n + k)$. An interesting lower bound might be lurking there. Given the importance of the problem, both on theoretical and practical grounds, this deserves serious investigation. We conclude this section with a summary of our main result.

Theorem 5. [General Segment Intersection] – ***All k pairwise intersections among n segments in the plane can be computed in $O(n \log n + k)$ time. The running time is optimal. The storage requirement is $O(n + k)$. If so desired, the algorithm will compute the vertical map of the set of segments within the same time and space bounds.***

3. Open Problems

Two natural questions have been left unanswered. Is it possible to reduce the storage requirement to $O(n)$? Clarkson [8] has recently provided a positive answer to this question if we are ready to use a probabilistic algorithm that takes $O(n \log n + k)$ expected time. The deterministic case remains open. Another challenge is to extend the ideas of this paper to compute intersections of analytic curves.

Acknowledgments: We would like to thank Bob Sedgewick and Chris Van Wyk: our C-code for red-black trees is mostly based on their own implementations. We also thank Andrew Appel and David Dobkin for helpful discussions about fragments of the implementation.

REFERENCES

1. Baumgart, B.G., *A polyhedron representation for computer vision*, 1975 National Computer Conference, AFIPS Conf. Proceedings, 44, AFIPS Press (1975), 589–596.
2. Ben-Or, M. *Lower bounds for algebraic computation trees*, Proc. 15th Ann. ACM Symp. Theory Comput. (1983), 80–86.
3. Bentley, J.L., Ottmann, T. *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput. C–28 (1979), 643–647.
4. Bentley, J.L., Wood, D. *An optimal worst-case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput. C–29 (1980), 571–577.
5. Brown, K.Q. *Comments on "Algorithms for reporting and counting geometric intersections,"* IEEE Trans. Comput. C–30 (1981), 147–148.
6. Chazelle, B. *Reporting and counting segment intersections*, J. Comput. Sys. Sci. 32 (1986), 156–182.
7. Chazelle, B. *A funçtional approach to data structures and its use in multidimensional searching*, *SIAM J. Comput.* (1988), in press. Prelim. version in Proc. 26th Ann. IEEE Sympos. Found. Comput. Sci. (1985), 165–174.
8. Clarkson, K.L. *Applications of random sampling to computational geometry, II*, Proc. 4th Ann. ACM Sympos. Comput. Geom. (1988), to appear.
9. Dobkin, D.P, Lipton, R. *Multidimensional searching problems*, SIAM J. Comput. 5 (1976), 181–186.
10. Edelsbrunner, H. *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, Germany, 1987.
11. Edelsbrunner, H., Guibas, L.J. *Topologically sweeping an arrangement*, Proc. 18th Ann. ACM Sympos. Theory Comput. (1986), 389–403.
12. Edelsbrunner, H., Mücke, E.P. *Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms*, Proc. 4th Ann. ACM Sympos. Comput. Geom. (1988), to appear.
13. Fredman, M. *On the information theoretic lower bound*, Theoret. Comput. Sci. 1 (1976), 355–361.
14. Guibas, L.J., Sedgewick, R. *A dichromatic framework for balanced trees*, Proc. 19th Ann. IEEE Sympos. Found. Comput. Sci. (1978), 8–21.
15. Guibas, L.J., Seidel, R. *Computing convolutions using reciprocal search*, Proc. 2nd Ann. ACM Sympos. Comput. Geom. (1986), 90–99.
16. Guibas, L.J., Stolfi, J. *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Trans. Graphics 4 (1985), 75–123.
17. Hoffman, K., Mehlhorn, K., Rosenstiehl, P., Tarjan, R. *Sorting Jordan sequences in linear time using level-linked search trees*, Inform. Control, to appear.

18. Mairson, H.G., Stolfi, J. *Reporting and counting intersections between two sets of line segments*, Proc. NATO Advanced Study Inst. Theoret. Found. Comput. Graphics and CAD, Il Ciocco, Castelvecchio Pascoli, Italy, Springer-Verlag, 1987.

19. Newman, W.M., Sproull, R.F. *Principles of interactive computer graphics*, McGraw-Hill, New York, 1973.

20. Nievergelt, J., Preparata, F.P. *Plane-sweep algorithms for intersecting geometric figures*, Comm. ACM, 25 (1982), 739–747.

21. Preparata, F.P., Shamos, M.I. *Computational geometry – an introduction*, Springer-Verlag, New York, 1985.

22. Sedgewick, R. *Algorithms*, Addison-Wesley, 1983.

23. Tarjan, R.E. *Data structures and network algorithms*, SIAM, Philadelphia, PA, 1983.