JOURNAL OF COMPUTER AND SYSTEM SCIENCES 32, 156–182 (1986)

Reporting and Counting Segment Intersections

BERNARD CHAZELLE*

Department of Computer Science, Brown University, Providence, Rhode Island 02912

Received September 11, 1984; revised April 28, 1985

This paper partly settles the following question: *Is it possible to compute all k intersections between n arbitrary line segments in time linear in k*? We describe an algorithm for this problem whose running time is $O(n(\log^2 n/\log\log n)+k)$. This is the first solution with a time bound linear in the size of the output. To obtain this result we turn away from traditional, sweep-line-based schemes. Instead, we introduce a new hierarchical strategy for dealing with segments without reducing the dimensionality of the problem. This framework is also used to answer related questions. New results include an $O(n^{1.695})$ time algorithm for *counting* intersections (as opposed to reporting each of them explicitly) and an optimal algorithm for computing the intersections of a line arrangement with a query segment. Using duality arguments we also present an improved algorithm for a point enclosure problem.

1. INTRODUCTION

Computing the intersection of arbitrary line segments is one of the most fundamental tasks of computational geometry. For this reason and because of practical motivation, a considerable amount of effort has been devoted to determining the complexity of this problem. Until now, all efficient algorithms known have had a running time of the form $O(f(n)+k\log n)$, where f is a subquadratic function of n. This has left open the basic question: Is it possible to remove the factor $\log n$ and thus make the algorithm linear in the output size? Besides its practical importance, this problem also addresses a key issue, i.e., the relationship between intersecting and sorting. To understand this, one must know that *all* efficient methods previously discovered involve sorting the intersections along a given direction. Consequently, deciding whether *intersecting* arbitrary segments necessarily entails *sorting* the intersections has been one of the outstanding questions in geometric complexity.

We settle this question by presenting the first algorithm to compute all k intersections among n arbitrary segments in time $O(f(n)+k)$. We achieve $f(n)=$

* This research was partly supported by by NSF Grants MCS 83-03925 and the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order 4786.

0022-0000/86 $3.00

$O(n \log^2 n/\log\log n)$, and show that the algorithm is the most efficient to date for each value of k. At the basis of our method, we find a new scheme for computing all k intersections between a fixed arrangement of n lines and a query segment in time $O(k + \log n)$. We also apply the hierarchical approach that underlies the main algorithm to other related problems. In particular we present an $O(n^{1.695})$ time algorithm for *counting* (rather than computing) all intersections between n segments, which constitutes the first subquadratic method known for this problem. We also give an application of our main technique to a point enclosure problem.

The remainder of this paper is organized as follows: in the next section we recall the main results known previously and we outline a preliminary version of the *reporting* algorithm, delaying the final version till the next section for the sake of clarity. In Section 4, we turn our attention to the *counting* problem, and finally give concluding remarks in Section 5.

2. Reporting Intersections: A First Attempt

We consider the following problem: *Given a set S of n line segments in the plane, determine all k intersecting pairs exactly once.*

Previous work on this problem is abundant, so for the sake of brevity we will only mention the most efficient algorithms found so far. A partial solution was first given by Shamos and Hoey [12] with an $O(n)$ space, $O(n \log n)$ time algorithm for determining *whether or not* S contains any intersecting pair. The sweep-line technique used in [12] was later extended by Bentley and Ottmann, who gave an $O(n + k)$ space, $O((n + k) \log n)$ time algorithm for reporting all k intersections in S [1]. The storage requirement of the algorithm was subsequently reduced to $O(n)$ by Brown [3]. An $O(n + k)$ space, $O((n + k) \log n)$ time algorithm by Nievergelt and Preparata also appeared in [11]. In [9] Mairson and Stolfi give an $O(n)$ space, $O(n \log n + k)$ time algorithm for merging two sets of non-intersecting segments ($k = \#$ intersections). Their algorithm can be used as the merge step of a divide-and-conquer algorithm for the general problem, but this leads to the (previously achieved) $O((n + k) \log n)$ time complexity, and thus suggests no solution for breaking the $k \log n$ barrier. For other work dealing with restricted cases, see also [1, 11].

The algorithm which we propose involves breaking up the original problem into two easier sub-problems, both of which deal intimately with the following geometric figure, called a *hammock*: consider the arrangement formed by n arbitrary (infinite) lines in the plane, and clip it along two vertical lines L and R. The planar graph H formed between L and R is called a *hammock* (Fig. 2). It will be essential in the following to be able to

1. construct a hammock in optimal time, and
2. compute the intersections between a hammock and a query segment in optimal time.

I. Weaving the Hammock

Simply computing all k vertices of H can be easily done in $O(n \log n + k)$ time with a procedure similar to insertion-sort. To do so, sort the left endpoints, and for each of them in ascending order, insert the corresponding right endpoint in descending order. Setting up the graph H with all its adjacencies, however, is more difficult. We represent H by means of adjacency lists, i.e., we associate with each vertex a list of its adjacent vertices in, say, clockwise order. Since in general the degree of each vertex is 3 or 4, this representation allows us to traverse any face of the graph in time proportional to its number of vertices in both the clockwise and counterclockwise directions. To handle singularities, we must convert the adjacency-list representation into the *doubly-connected-edge-list* representation [10] or the *quad-edge* structure [7]. More simply, we can use an edge-based representation, whereby each edge e is associated with a 4-field vector containing the names of the four edges sharing an adjacent face with e (Fig. 1). This representation suits our needs for the following reason: from any edge it is possible to start walking clockwise (or counterclockwise, for that matter) along either of its two adjacent faces, and the walk involves a constant number of operations at each step. For consistency we consider the two unbounded upper (resp. lower) edges of H to be adjacent to each other.

Without loss of generality, assume that L lies to the left of R. Let $S = \{s_1, ..., s_n\}$ be the n segments joining L and R and let l_i denote the left endpoint of s_i. Assume that the sequence $l_1, ..., l_n$ is vertically ascending; we proceed to insert into H each segment $s_1, ..., s_n$ in turn. Let s_i be the segment currently considered, and let e denote the edge of H that contains l_i. We traverse the (unbounded) face adjacent to e, *clockwise*, until we reach an edge e' that intersects s_i. If this edge lies in R, s_i does not intersect any segment in $\{s_1, ..., s_{i-1}\}$, so we can update H in constant time. Otherwise, we restart the clockwise traversal from e', now with respect to the other face adjacent to e'. We iterate on this process until we reach an edge that lies in R (Fig. 2).

LEMMA 1. *It is possible to compute the planar graph formed by n segments joining two parallel lines in time $O(n \log n + k)$, where k is the number of intersecting pairs of*

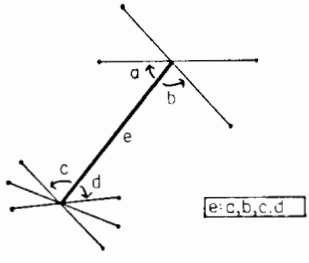

FIGURE 1

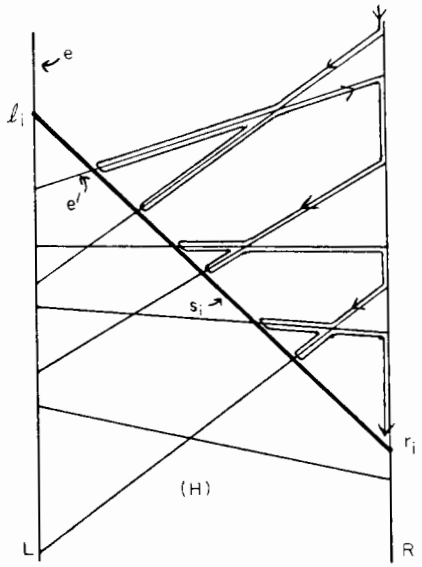

FIGURE 2

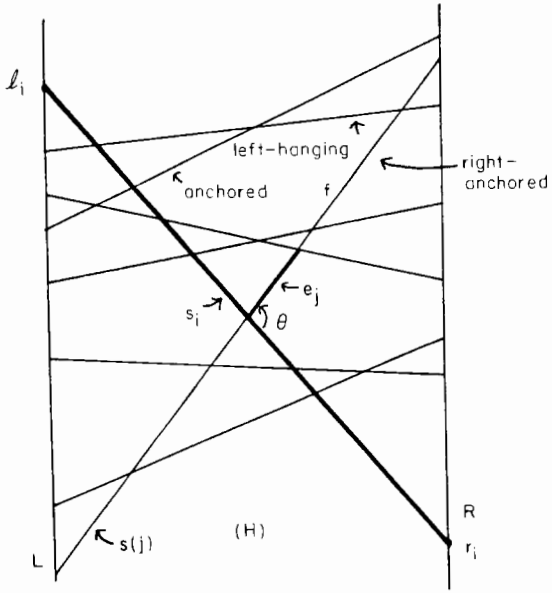

FIGURE 3

segments. If the sorted order of the endpoints on either of the two lines is already available, then the graph can be computed in $O(n+k)$ time.

Proof. It suffices to show that inserting s_i into H requires $O(k_i)$ operations, where k_i is the number of edges in H that intersect s_i. Let $e_1,\ldots, e_p$ be the edges of H (not in L, R, or s_i) that are visited during the computation, and let $s(j)$ be the segment of S supporting e_j. We will show that $p = O(k_i)$, which is sufficient for our purposes (no two edges on R are ever traversed consecutively). Since l_i is the highest point on L so far, all the segments $s(1),\ldots, s(p)$ intersect s_i. Let f be any of the faces in H, above and adjacent to s_i, and let g be the unique edge of f that lies on s_i. Any edge e_j adjacent to f is said to be *anchored* if it is adjacent to s_i, or *left-hanging* (resp. *right-hanging*) if it is not adjacent to s_i but $s(j)$ intersects s_i to the left (resp. right) of g (Fig. 3). Note that the notions of left- and right-hanging are mutually exclusive. We can show that the total number of left-hanging edges is dominated by k_i. Let e_j be the anchored edge that forms the widest interior angle θ with s_i. Since $s(j)$ cannot contribute any left-hanging edge, removing $s(j)$ from H will at most destroy one left-hanging edge (by merging a left-hanging edge with an anchored or left-hanging edge). By induction, we thus prove that the number of left-hanging edges cannot exceed k_i. The same reasoning applied to right-hanging edges completes the proof. ∎

II. Walking through the Hammock

We consider now the problem of computing all t intersections between H and a query segment q that lies within the vertical slab (L, R). The method involves locating the face of H that contains one of the endpoints of q and then walking through H toward the other endpoint. To overcome a number of difficulties soon to be apparent, we augment the representation of H as follows:

1. Preprocess H so as to allow for efficient planar point location. Since it is crucial to achieve linear preprocessing time, we use the methods in [8, or 4]. These methods allow us to locate a query point in optimal $O(\log n)$ time; also given that we can easily triangulate each face of H (since they are convex) and at the same time keep a clockwise-order list of the edges adjacent to each vertex, the preprocessing can be done in $O(n+k)$ time. We assume the existence of *face-lists* giving the edges surrounding each face of H in clockwise order. Conversely, we assume that each edge is associated with two pointers, pointing to the name of each of its adjacent faces. We can dispense with one of these pointers when dealing with edges on L or R.

2. Attach to each face a pointer to its *rightmost* vertex, i.e. the vertex with maximum x-coordinate (resolve ties arbitrarily). Do the same with respect to its *leftmost* vertex.

This preprocessing can be accomplished in $O(|H|)$ time and space, once H has been computed as described earlier—throughout this paper, we will use the

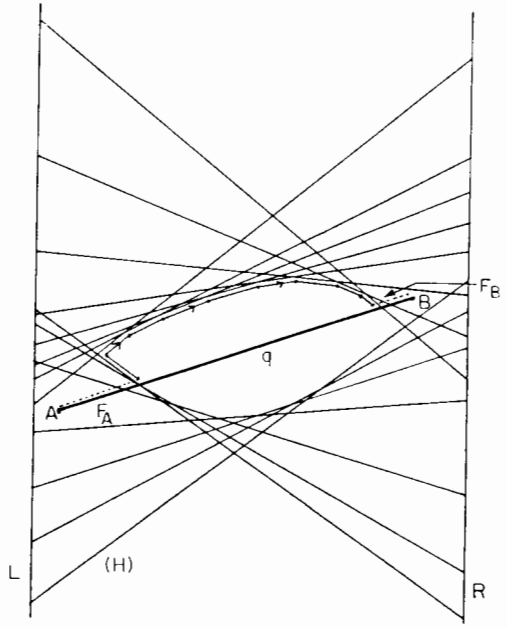

FIGURE 4

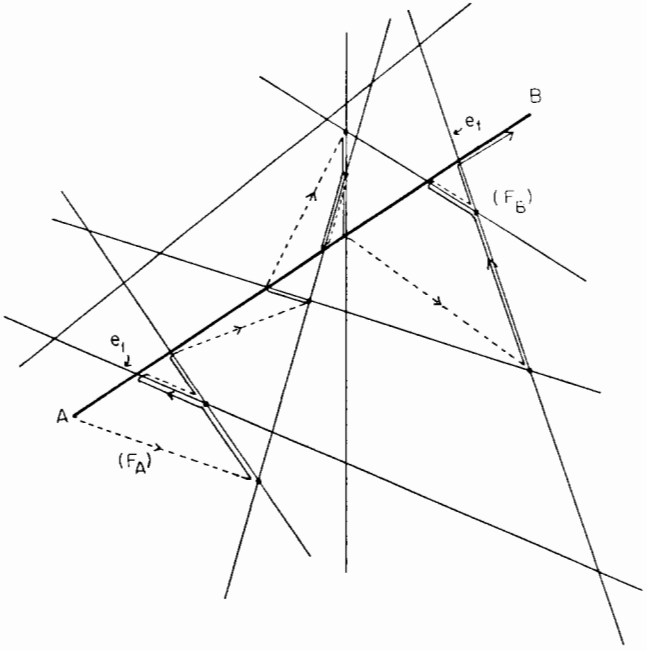

FIGURE 5

notation $|X|$ to designate the size of X, i.e., the size of its representation, whether X is a set or a graph. Let A (resp. B) be the left (resp. right) endpoint of q (if q is vertical, then A is its lower endpoint), and let $e_1,\ldots, e_t$ be the edges intersected by q, in order from A to B. Let f_A (resp. f_B) designate the face containing A (resp. B). It is tempting to retrieve the intersections of q by determining either f_A or f_B, and then walking around each face which q intersects. Unfortunately, this may be highly inefficient when traversing faces with many vertices (Fig. 4). We get around these difficulties by *jumping* directly to the rightmost vertex of each new face encountered (Fig. 5). Let e_i be the last intersecting edge detected so far, and let f be the face adjacent to e_i that we are about to traverse next. Let v be the rightmost vertex of f, and assume without loss of generality that v lies above q (i.e., above the line supporting q). The main point to consider is whether or not the next segment adjacent to v intersects q. If this is the case, we say that the *hanging-condition* is satisfied for f. More precisely, let w be the next vertex after v in clockwise order around f. We say that the *hanging-condition* is satisfied for f (with respect to q) if the segment of S that supports vw intersects q (note that if v lies below q, w is defined as the counterclockwise neighbor of v).

If the *hanging-condition* is satisfied for all faces ($\neq f_B$) intersecting q, then, after locating the faces f_A and f_B, we can start the traversal at the rightmost vertex of f_A and retrieve the name of the face adjacent to e_i that does not contain A. With this information, we can find the rightmost vertex of this face in constant time. Next we traverse the face from there, until we find e_2, at which point we simply iterate on this process. Note that the traversal proceeds clockwise (resp. counterclockwise) if the rightmost vertex just found is above (resp. below) q. This process will terminate when we reach the edge e_t, which is adjacent to f_B. Since f_B has been previously determined, the termination condition can be checked in constant time at every edge e_i newly encountered (Fig. 5).

We next show that when the *hanging-condition* is satisfied for all the faces traversed ($\neq f_B$), the procedure which we just described allows us to determine $e_1,\ldots, e_t$ in $O(\log n + t)$ time. First of all, it is clear that all the edges visited during the computation have their supporting segment intersect q. Let E be the subset of these edges that lie totally or partially above q. We can show that the number of these edges, $|E|$, is $O(t)$, a result which by symmetry will also apply to the edges below. Borrowing terminology from Lemma 1, we see that the edges of E are either *anchored* or *right-hanging*. It directly follows from the previous lemma that $|E| \leqslant 2t$.

It is clear that if B lies on the line R, the *hanging-condition* is always satisfied, so we can apply the technique just described. Furthermore, notice that we actually do not need to perform a planar point location for B; instead, we pursue the traversals through f_B, i.e., until we reach the line R—the same can be said, of course, if the query segment is incident to L instead of R.

OBSERVATION 1. Given a hammock H with $O(n+k)$ vertices and an arbitrary query segment $q = AB$ with B incident to one of the bounding lines, it is possible to report all t intersections between q and the segments of H in time $O(\log n + t)$.

Most important, the computation does not involve any point location with respect to B.

We are now ready to drop the *hanging-condition* and give an algorithm for the general problem. We assume that both f_A and f_B have been already located and that, without loss of generality, $f_A \neq f_B$. Let e_i be the last edge detected ($i < t$) before the *hanging-condition* is found to fail. Let f be the face to the right of e_i and let v be its rightmost vertex, with w the next vertex clockwise (Fig. 6). Without loss of generality, we can assume that v lies above q. Since $i < t$, the segment supporting vw does not intersect q, yet some edge of f further in clockwise order does cross q (from above to below). The algorithm will stop the traversal upon encountering v and will restart counterclockwise, from the rightmost vertex of f_B, detecting e_t, $e_{t-1}, \ldots, e_{i+1}$ in this order, as explained later on. The criterion for termination will be easy to test: simply check whether the last intersecting edge found, e_j, is adjacent to f.

The walk proceeds in quite the same way as before. After determining e_j, we retrieve the rightmost vertex, v_j, of the face f_j adjacent to e_j (to the left), and proceed around f_j in counterclockwise order from v_j, until we cross q. This corresponds precisely to the edge e_{j-1}, at which point we simply iterate on the process. In the course of proving the correctness of this algorithm, we will show that when v lies above q, all the traversing takes place above q, therefore we never have to traverse e_j twice. Note that, of course, this traversing should proceed *always*

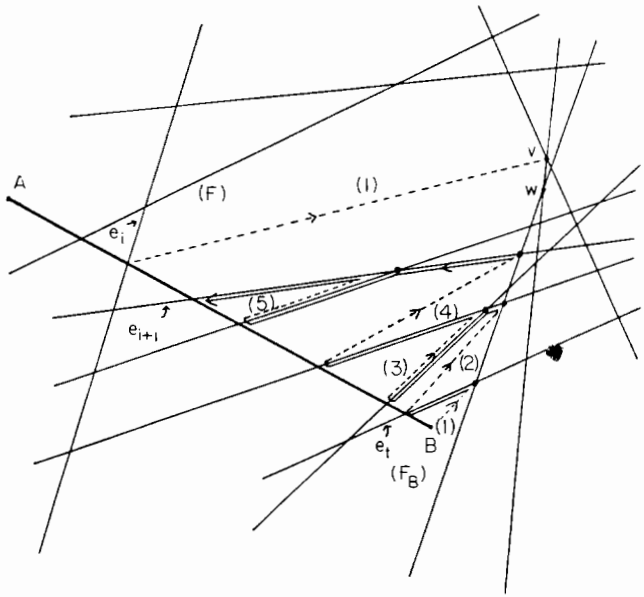

FIGURE 6

clockwise instead of *always counterclockwise*, should v lie below q. This case is mirror-image of the other, so we will not elaborate on it any further.

OBSERVATION 2. The procedure described above reports all the edges $e_t, e_{t-1},\ldots, e_{i+1}$, in $O(t-i)$ steps.

Proof. We assume throughout this proof that v lies above q. The correctness of the method is obvious, so we directly turn to a study of its performance. We will show that all the edges visited belong to segments that intersect q, and that all the traversals take place above q. The latter point will imply that in going from v_j to e_{j-1}, we never have to visit e_j again. For consistency, we define f_{t+1} as f_B. The procedure traverses each face f_j $(i+1<j\leqslant t+1)$ counterclockwise, from the vertex v_j to the edge e_{j-1}. We mimic the traversal of the procedure backwards. Let $w_1,\ldots, w_p$ be the vertices visited during the traversal, in clockwise order, with $w_1w_2=e_{j-1}$ and $w_p=v_j$. For any vertex w_l visited in the course of the traversal, we define R_l as the ray emanating from w_l and passing through $w_{l-1}w_l$. We will show by induction that for $l=2,\ldots, p$:

1. The vertex w_l lies above q.
2. The ray R_l is *left-oriented*, i.e., lies entirely to the left of w_l.
3. The segment $s(l)$ of S supporting $w_{l-1}w_l$ intersects q.

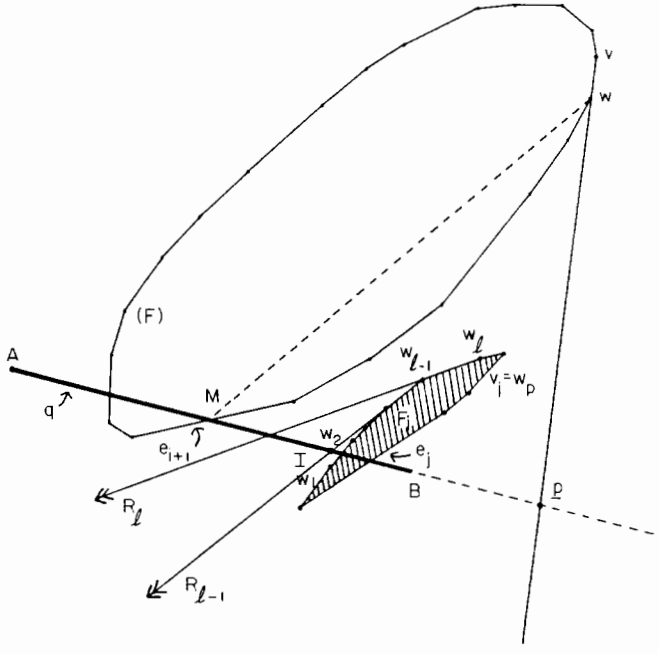

FIGURE 7

Let M (resp. P) be the intersection of the lines passing through q and e_{i+1} (resp. vw) (Fig. 7). The basis ($l=2$) is obviously true, except perhaps for property 2. Assume that R_2 is not left-oriented. This implies that the segment $s(2)$ intersects Mw, hence the interior of f, which leads to a contradiction. Let us now consider the inductive case ($2<l\leqslant p$) and assume that all three properties are true for $l-1$. Because of the convexity of f_j, R_l differs from R_{l-1} by a clockwise turn. Since on the other hand the sequence w_{l-1}, w_l is directed clockwise toward the rightmost vertex of f_j, the ray R_l has to be left-oriented, which establishes property 2. Assume now that w_l lies below q; it then follows from property 2 that the ray R_l, hence $s(l)$, intersect Mw, which for reasons already mentioned, is a contradiction. This proves property 1.

To establish property 3, let's define I as the intersection of $s(l-1)$ and q. Since w_{l-1} lies above q, it must lie in the triangle MwP. We also know that the ray R_l is left-oriented, therefore it must intersect either MP or Mw. The latter case is obviously ruled out since Mw lies totally inside a face of H. It then follows that R_l, and hence $s(l)$, intersect MP. From the convexity of f_j, this intersection lies necessarily to the left of I with respect to q, therefore it lies on the segment MI. This shows that R_l, hence $s(l)$, intersect q, which completes the inductive proof. This also justifies the prescription always to turn in the same direction, i.e., counterclockwise (resp. clockwise) if v lies above (resp. below) q. We have thus proven that all the edges visited lie on segments of S that intersect q. This shows that only left-hanging and anchored edges will ever be traversed on the way from e_l to e_{i+1}. Therefore, the technique of Lemma 1 once again applies directly, which completes the proof. ∎

Putting previous results together, we conclude:

LEMMA 2. *Given a hammock H with $O(n+k)$ vertices and an arbitrary query segment q, it is possible to report all t intersections between q and the segments of H in time $O(\log n+t)$. This assumes that the hammock has been preprocessed appropriately (in $O(n\log n+k)$ time and $O(n+k)$ space).*

Note that the procedure described above returns the intersections with q in sorted order. It is possible to speed up the algorithm by a constant factor if we do not require this feature. This involves taking shortcuts as soon as a segment intersecting q is found. The improvement is not significant enough to be described here, however.

III. The General Reporting Algorithm

Lemmas 1 and 2 can be combined to form the *inner loop* of an algorithm for solving the general *reporting problem*. Let $S=\{s_1,\ldots,s_n\}$ be n arbitrarily oriented segments in the plane. We define a segment tree [2] over the intervals formed by projecting the segments $s_1,\ldots,s_n$ on the x-axis. Let L be the smallest interval that covers all the projections. The set S induces a partition of L into at most $2n-1$ intervals, over which we define a complete binary tree T. For each node v of T, let

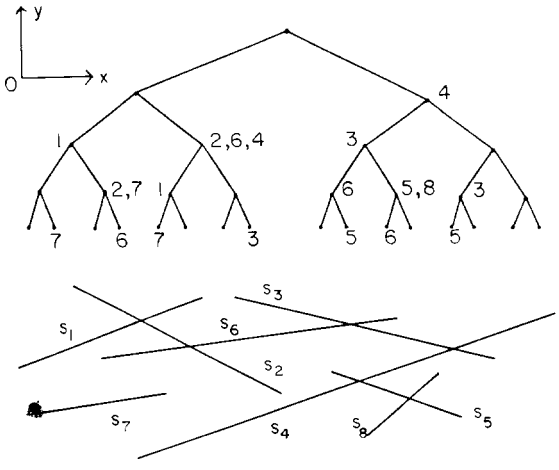

FIGURE 8

$I(v)$ be the interval formed by the union of the intervals associated with the leaves of the subtree rooted at v; let $S(v)$ be the vertical slab with base $I(v)$. We say that an interval J on the x-axis *covers* a node v if $I(v) \subset J$ and $I(z) \not\subset J$, where z is the father of v. We endow each node v of T with a pointer to a list, $L(v)$, that contains the indices of the intervals of S that *cover* v (Fig. 8). The tree T induces a decomposition of each segment in S into $O(\log n)$ *canonical segments*. For simplicity, we will treat the elements of $L(v)$ as canonical segments, i.e., physical subsegments from S, rather than pointers to their respective supporting segments in S. The key observation is that the canonical segments of $L(v)$ form a hammock, denoted $H(v)$. We can show that the only intersections to be reported are

1. the vertices of the hammocks not on the boundary, and
2. the intersections between $H(v)$ and the segments of S which happen to have an endpoint within $S(v)$.

We use Lemmas 1 and 2 to handle the first and second cases, respectively. Unfortunately, there are still a number of difficulties to overcome, the major one being the excessive cost of computing the full-fledged segment tree at once. Instead, we proceed bottom-up (for example) and, most important, on a *level-by-level* basis. Let $v_1,\ldots, v_p$ be the nodes of T (in order from left to right) at level λ. We compute the lists $L(v_1),\ldots, L(v_p)$ by considering each segment s_j in S and examining the search path in T corresponding to its two endpoints. This allows us to determine in $O(\log n)$ time whether s_j contributes any canonical part to level λ. If this is the case, we add the index j into the (one or two) corresponding lists. We make no assumption on the order in which the elements of $L(v_i)$ appear. Next, we compute the hammocks $H(v_1),\ldots, H(v_p)$ by applying the algorithm of Lemma 1. The entire computation requires $O(n \log n + \sum_{1 \leqslant i \leqslant p} |H(v_i)|)$ time, and $O(n + \sum_{1 \leqslant i \leqslant p} |H(v_i)|)$

space. A hammock has two kinds of vertices: the *boundary-vertices* that lie on the *slab-lines*, that is, the vertical boundary lines, and the *intersection-vertices* that are intersections between segments of S. Let $k_h(\lambda)$ denote the total number of intersection-vertices in all the hammocks at level λ. Since any segment of S can contribute at most two canonical segments at any level, we have

$$\sum_{1 \leq i \leq p} |H(v_i)| = O(n + k_h(\lambda)).$$

Next we form each set $W(v_i)$, defined as the subset of S consisting of the segments that have at least one endpoint in the slab $S(v_i)$, yet do not contribute any entry to $L(v_i)$. The idea is to compute the intersections between $H(v_i)$ and the segments of $W(v_i)$. Once again we can compute $W(v_1),..., W(v_p)$ in $O(n \log n)$ time with a simple sort on the x-values of the endpoints. Consider the number of intersections between the (non-boundary) edges of $H(v_i)$ and the segments of $W(v_i)$, and let $k_w(\lambda)$ designate the sum of these numbers for $i = 1,..., p$. Applying the algorithm of Lemma 2, we can compute all these intersections in time $O(k_w(\lambda) + \sum_{1 \leq i \leq p} |W(v_i)| \times \log n)$.

Iterating on the process outlined above for each level λ in T and outputting the $k_h(\lambda) + k_w(\lambda)$ relevant intersections at every stage of the process eventually gives us all k intersections among segments in S exactly once. To see this, let I be an intersection between two segments s and t in S. Let s^* and t^* be the canonical segments of s and t, respectively, that contain I. If s^* and t^* are generated at the same node v of T, i.e., $s^*, t^* \in L(v)$, their intersection will be detected while computing the hammock $H(v)$. Otherwise, assuming that s^* is generated at node v and t^* at node w, it is easy to see that one node, say w, is a descendant of the other. It then follows that t must have an endpoint in $S(v)$, therefore t will appear in $W(v)$, which shows that I will be discovered when computing the intersections between $H(v)$ and $W(v)$. The uniqueness of each intersection report follows directly. Of course, extra care must be given to degenerate cases, arising for example when segments have coinciding endpoints, or more than two segments intersect in one point. These special cases can all be handled in a uniform manner without changing the asymptotic complexity of the algorithm—we omit the details. Summing up all the quantities $k_h(\lambda)$ and $k_w(\lambda)$ for all levels λ in T, we find

OBSERVATION 3. $\sum_{1 \leq \lambda \leq 1 + \lceil \log_2 2n \rceil} (k_h(\lambda) + k_w(\lambda)) = k.$

After an initial $O(n)$ space, $O(n \log n)$ time preprocessing, the algorithm iterates for each level on a two-step procedure: for each node $v_1,..., v_p$ on the current level, call the following functions.

1. HAMCOMP (v_i): Compute $W(v_i)$ and $H(v_i)$, and preprocess the latter for planar point location.

2. INTERCOMP (v_i): Compute the intersections between $H(v_i)$ and $W(v_i)$.

THEOREM 1 (Preliminary result). *Given a set S of n arbitrary segments in the plane, it is possible to report all k intersecting pairs exactly once, in $O(n \log^2 n + k)$ time and $O(n + k)$ space.*

Proof. From the fact that $\sum_{1 \leqslant i \leqslant p} |W(v_i)| \leqslant 2n$. ∎

Note that the space requirement should be in general significantly less than $O(n + k)$. Indeed, at each level, we only need one hammock at a time. The storage needed is therefore $O(n + k^*)$, where k^* is the maximum size of any hammock computed.

3. AN IMPROVED ALGORITHM VIA COST MATCHING

We can show that it is possible to save a factor of "log log n" in the running time of the algorithm by carefully balancing the costs of its various components. The new method is based on a redefinition of segment trees over a representation of segment coordinates in base $\lceil \log_2 n \rceil$. To begin with, we must point out a discrepancy in the performance of the previous algorithm. The method described earlier rests essentially on two main components, HAMCOMP and INTERCOMP, called at each level of the tree. Both parts require $O(n + K)$ space and $O(n \log n + K)$ time per level. In the following, for simplicity, the term K appearing in any expression will be used in a generic sense to designate the number of intersections either reported or simply used by the procedure to whose complexity the expression refers. In this case, for example, we have $K = k_h(\lambda) + k_w(\lambda)$. The apparent evenness of cost between HAMCOMP and INTERCOMP is deceiving. Indeed, we will show that the complexity of HAMCOMP can be made $O(n + K)$, at each level, with relatively little effort. Unfortunately, INTERCOMP is more recalcitrant, and only a fairly heavy treatment will succeed in cutting down its complexity. The development of this section will proceed in four parts. To begin with, we show how to speed up the computation of HAMCOMP. In the second part, we introduce a new type of hammock, and in the third we show how to use it in order to improve the performance of INTERCOMP. In the fourth part we analyze the complexity of the new algorithm; to find out that the space requirement falls slightly short of our expectations. We fix this minor discrepancy by resorting to a more adaptive technique.

3.1. *Speeding Up HAMCOMP*

LEMMA 3. *After $O(n \log n)$ time preprocessing it is possible to compute the sets $L(v_i)$, $W(v_i)$, for all the nodes $v_1, \ldots, v_p$ at level λ, in $O(n)$ space and time.*

Proof. Let $v_1, \ldots, v_p$ designate, as usual, the nodes of T at level λ. We can compute the sets $W(v_1), \ldots, W(v_p)$ in $O(n)$ time and space, by maintaining the search path in T of all the endpoints in S at each level. More precisely, when dealing with level λ, we ensure that we have a pointer from each endpoint in S to the node in T at level λ that is on the search path corresponding to the endpoint. It is easy to

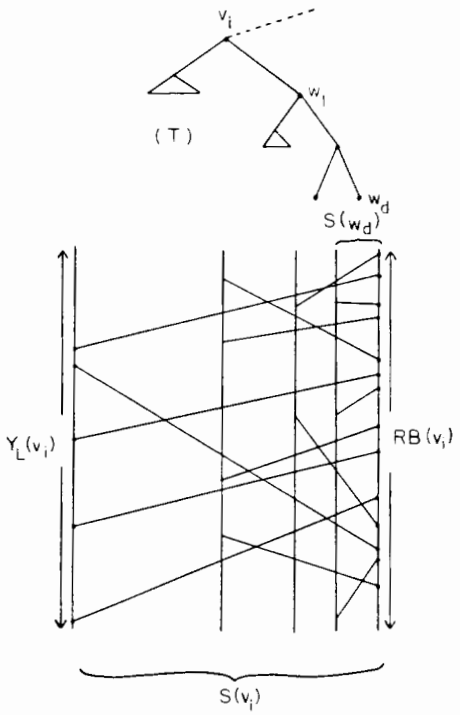

FIGURE 9

maintain these pointers incrementally, at a cost of $O(1)$ per pointer as we proceed to the next level higher. We compute the lists $L(v_1),\ldots, L(v_p)$ in a similar manner. At each level, each segment of S has 0, 1, or 2 canonical segments in T, and it is easy to keep track of the corresponding nodes of the tree by incremental updates. This allows us to maintain, at a cost of $O(n)$ time per level, two arrays $PL[1\cdots n]$ and $PW[1\cdots n]$ defined as follows: $PL[i]$ indicates the 0, 1, or 2 nodes at level λ to which s_i contributes a canonical segment. We also define $PW[i]$ to indicate in which slabs $S(v)$ at level the left and right endpoints of s_i fall. As mentioned before, we do not have to keep a pointer to v in $PW[i]$ if $PL[i]$ already point to it. ∎

Applying Lemma 1 to compute $H(v_i)$ requires the availability of the y-order of the left endpoints of the canonical segments in $L(v_i)$. Since we cannot afford to sort these endpoints, we must proceed incrementally, using information collected at lower levels to form the sorted lists in $O(n)$ time. As usual, v_i denotes a generic node of T at level λ. Let $Y_L(v_i)$ (resp. $Y_R(v_i)$) be the y-sorted list of the left (resp. right) endpoints of the segments in $M(v_i)$ (Fig. 9). Recall that $L(v_i)$ contains the canonical, i.e., clipped parts of the actual segments in S, therefore all the points in $Y_L(v_i)$ or $Y_R(v_i)$ are vertically aligned. Note also that these two lists give exactly the boundary vertices of $H(v_i)$. To compute $Y_L(v_i)$ and $Y_R(v_i)$ efficiently, we use lists

computed at level $\lambda - 1$. Let $w_1,..., w_d$ be the nodes on the *rightmost path* from v_i. We define $RB(v_i)$ to be the y-ordered list of right endpoints of *all* the canonical segments in $L(v_i)$, $L(w_i)$,..., $L(w_d)$. Note that all these endpoints (1) lie on a common vertical line and (2) are, in general, not endpoints in S (Fig. 9). We define $LB(v_i)$ similarly with respect to the left endpoints of the segments in the L-lists on the *leftmost path* from the node v_i. We successively show how to compute $RB(v_i)$ and $LB(v_i)$, and then how to use these lists to compute hammocks. Actually, what we really need for computing the hammocks at level λ are the lists $RB(z)$ and $LB(z)$ for the nodes z at level $\lambda - 1$. Conversely, it will turn out that $H(z)$ is necessary for the computation of $RB(z)$ and $LB(z)$, we will therefore set up the latter lists as the last task performed at level $\lambda - 1$.

LEMMA 4. *Computing the lists $LB(v_i)$ and $RB(v_i)$, for all the nodes $v_1,..., v_p$ at level λ, can be done in $O(n)$ space and time.*

Proof. As usual, let $v_1,..., v_p$ be the nodes of T at level λ, given in order from left to right. To compute $RB(v_i)$ and $LB(v_i)$ at level λ, we assume that $H(v_i)$ is available, as well as $RB(z)$ and $LB(z)$, for all nodes z at level $\lambda - 1$. Because of obvious symmetry, we only describe the computation of $RB(v_i)$. First, we form $Y_R(v_i)$ by retrieving the vertices of the hammock $H(v_i)$ on the right slab-line of $S(v_i)$. Next, we form $RB(v_i)$ by merging $Y_R(v_i)$ and $RB(w_1)$. This step takes time $O(|RB(w_1)| + |Y_R(v_i)|)$. Any segment of S contributes at most two entries in $\{Y_R(v_1),..., Y_R(v_p)\}$, therefore $\sum_{1 \leq i \leq p} |Y_R(v_i)| = O(n)$. Similarly, all the segments of S contributing an entry in $RB(w_1)$ must have their left endpoint strictly within the slab $S(v_i)$, therefore they must be in $W(v_i)$. Since $\sum_{1 \leq i \leq p} |W(v_i)| = O(n)$, the proof is complete. ∎

LEMMA 5. *Computing the hammocks, $H(v_i)$, for all the nodes $v_1,..., v_p$ at level λ can be done in space and time $O(n + k_h(\lambda))$.*

Proof. We show how to use the lists $RB(z)$ and $LB(z)$ at level $\lambda - 1$ in order to compute $Y_L(v_i)$, and hence $H(v_i)$. For the sake of generality, assume that $1 < i < p$; the case $i = 1$ or $i = p$ will follow directly from the general case. Let s be a segment of S that has a canonical part t in $L(v_i)$. This segment falls in one of four categories (Fig. 10): (1, 2) one of the endpoints of s coincides with an endpoint of t, i.e., lies on one of the slab-lines of $S(v_i)$ (this breaks down into two cases); (3) s has a canonical part in $L(z)$, where z is on the *rightmost path* from v_{i-1} ($z \neq v_{i-1}$); (4) s has a canonical part in $L(z)$, where z is on the *leftmost path* from v_{i+1} ($z \neq v_{i+1}$). Note that these four cases are not mutually exclusive:

1. s has a canonical segment in $L(v_i)$ and its left endpoint lies on the left slab-line of $S(v_i)$. The endpoints to fall in this category form a y-ordered list $V_1(v_i)$, which we can set up directly for all nodes v_i at level λ, provided that $s_1,..., s_n$ have been pre-sorted along the y coordinates of their left endpoints. A simple scan through PL (defined in proof of Lemma 3) will provide all these lists in $O(n)$ time.

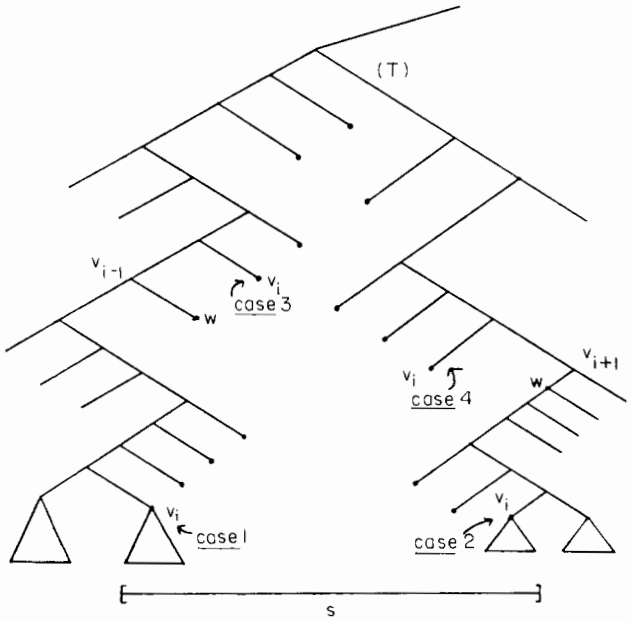

FIGURE 10

2. Similarly, we form the list $V'_2(v_i)$ consisting of the points that lie on the right slab-line of $S(v_i)$ and are the right endpoints of segments of S with a canonical part in $L(v_i)$. Next, we sort the left endpoints of the corresponding canonical segments, using insertion sort, thus ensuring that the time is $O(|V'_2(v_i)| + K)$ time. This provides us with a list of points, $V_2(v_i)$, y-sorted along the left slab-line of $S(v_i)$. Note that, most often, the lists $V_1(v_i)$ and $V_2(v_i)$ will have 0 or 1 elements, so their computation will be in general much simpler than described.

3. Let V_3 be the y-ordered list of the left endpoints of all the canonical segments to fall in case 3. Let w be the right son of v_{i-1}; V_3 is a sublist of $RB(w)$, therefore a linear scan through the latter list will enable us to compute V_3.

4. Let U be the set of canonical segments in $L(v_i)$ that fall in the fourth category. In a first stage, we compute $V'_4(v_i)$, the y-ordered list of $U's$ right endpoints, by applying the same remark as in case 3: $V'_4(v_i)$ is a sublist of $LB(w)$, where w is the left son of v_{i+1}. This allows us to retrieve $V'_4(v_i)$ in a linear pass through $LB(w)$. As in case 2, we use insertion sort (following the order of $V'_4(v_i)$) to set up $V_4(v_i)$, the y-ordered list of the left endpoints of the canonical segments in U. This takes time proportional to the cardinality of U and the number of intersections between the elements of U.

We are now ready to form $Y_L(v_i)$ by merging the four lists $V_1(v_i)$, $V_2(v_i)$, $V_3(v_i)$, $V_4(v_i)$. The result may contain duplicates, which we then discard. As mentioned

earlier, the procedure above also applies to $Y_L(v_1)$ and $Y_L(v_p)$. This allows us to apply the algorithm of Lemma 1 and set up $H(v_1),..., H(v_p)$. To evaluate the time complexity of the procedure just described, it suffices to observe that each list $RB(z)$ and $LB(z)$ at level $\lambda - 1$ is examined at most once, and the insertion sorts of cases 2 and 4 require $O(n + K)$ (where K counts here a subset of the intersection-vertices in $\{H(v_1),..., H(v_p)\}$). Combining this result with Lemma 4 completes the proof. ▮

3.2. *Generalizing Hammocks*

To improve the performance of INTERCOMP, we need to reorganize the computation slightly. Whereas each level in T used to represent a basic stage in the algorithm, we now regroup levels by layers of fixed height. Let α be a positive integer ($1 < \alpha < \log_2 2n$), and let $\lambda_j = \alpha \times j$. We define a layer Λ_j as the set of α consecutive levels $\{\lambda_{j-1} + 1, \lambda_{j-1} + 2,..., \lambda_j\}$. The algorithm will now proceed exactly as usual from the lowest level upwards, but every time we switch to a higher layer, a certain amount of preprocessing on all the nodes in the new layer will be accomplished.

What is the purpose of this preprocessing? The inefficiency of the previous algorithm comes essentially from the repeated application of planar point locations, which alone amounts to $O(n \log n)$ operations per level. We can partly remedy this flaw by balancing out cost factors. The idea is to refine the data structure so that a point need be located in a hammock only once per layer. To do so, we construct a planar graph $K(v)$ for each node v of T at level λ_j, defined as the *superposition* of all the hammocks below v in the layer. We define the *superposition* of two planar subdivisions, A and B, as the new subdivision, C, obtained by merging A and B, thus possibly adding new vertices (Fig. 11). Since the graphs $K(v)$'s will be refinements of

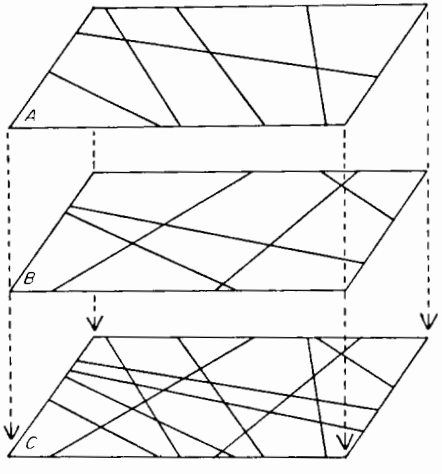

FIGURE 11

every hammock in the layer, we will be able to use an appropriate pointing mechanism to avoid repeated planar point locations.

Let's review the data structures needed at this point to preprocess layer Λ_j. We keep the hammocks $H(w)$ for all $w \in \Lambda_j$, as well as the list PW corresponding to the lowest level in Λ_j, i.e., $\lambda_{j-1}+1$. We also keep all the lists $LB^*(w)$, where $LB^*(w)$ is defined as the sublist of $LB(w)$ consisting of the left endpoints in $\{L(z) \mid z \in \Lambda_j$ on *leftmost path* from $w\}$. The same applies to $RB^*(w)$, defined in a similar way with respect to right endpoints and rightmost paths. We can easily obtain $LB^*(w)$ and $RB^*(w)$ with a single pass through $LB(w)$ (resp. $RB(w)$).

We are now ready to show how to compute $K(v)$. Let T^* be the subtree of T (in the sense of subgraph) consisting of v and all its descendants in Λ_j. We define $K(v)$ as the planar graph formed by superposing all the hammocks $H(w)$, for all nodes w in T^*. Observe that each vertex of $K(v)$ is either a vertex of some hammock $H(w)$, or it is an added vertex; these added vertices will not cost overhead to the computation since, as intersections of segments in S, they are to be reported, anyhow. Unfortunately, storing all of $K(v)$ is prohibitive, so we must devise a method for computing $K(v)$ by pieces, following a *lazy evaluation* scheme.

Let $w_1,\ldots, w_m$ be the leaves of T^* in left-to-right order (note that $m=2^{\alpha-1}$), and let p_i be the path in T^* from v to w_i. Observe that any hammock $H(w)$, for $w \in T^*$, either lies totally outside of the slab $S(w_i)$ or completely overlaps it. The hammocks that fall in the latter category correspond to the nodes of p_i. We define $K_i(v)$ to be the part of $K(v)$ that lies inside $S(w_i)$. More precisely, $K_i(v)$ is the superposition of the hammocks $H(w)$, for all w on p_i, each of them being clipped so as to fit exactly within $S(w_i)$. We also include the vertical lines of $S(w_i)$, so that $K_i(v)$ has exactly the shape of a hammock. We will show how to compute $K_1(v),\ldots, K_m(v)$ in this order.

The vertices on the left boundary of $K_1(v)$ are exactly the points in $LB^*(v)$. We may then apply the algorithm of Lemma 1 to compute $K_1(v)$. Suppose now that we have already computed $K_i(v)$. Let z be the lowest common ancestor of w_i and w_{i+1},

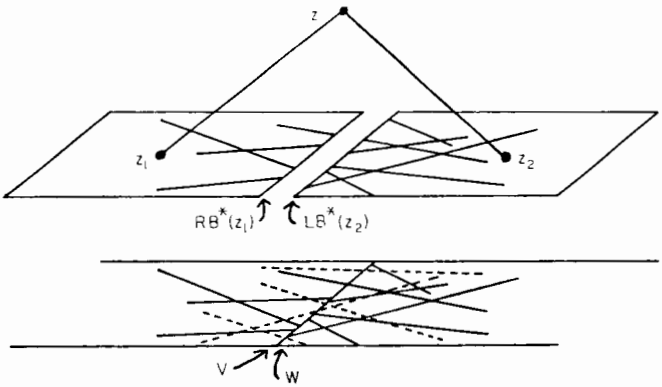

FIGURE 12

and let V (resp. W) be the y-ordered list of the vertices on the right (resp. left) boundary of $K_i(v)$ (resp. $K_{i+1}(v)$). W can be derived from V by deleting from it the points of $RB^*(z_1)$ and adding to it the points of $LB^*(z_2)$, where z_1 and z_2 are respectively the left and right sons of z (Fig. 12). The availability of the lists LB^* and RB^* allows us to construct W from V in two passes through V. Once W is available, we turn to Lemma 1 and compute $K_{i+1}(v)$. As usual, each of the new hammocks will be preprocessed for efficient point location (in linear time).

LEMMA 6. *Let $K=\sum_{1\leqslant i\leqslant \alpha}(k_h(\lambda_j-i+1)+k_w(\lambda_j-i+1))$. The set of hammocks, $H(w)$, for all nodes $w\in\Lambda_j$ can be computed and stored in time and space $O(\alpha n+K)$. The set of new hammocks $K_i(v)$, at layer Λ_j, can be computed in $O(2^\alpha n+K)$ time, with only $O(\alpha n+K)$ space needed, if we keep only one new hammock at any time.*

Proof. The first statement is a direct consequence of Lemma 5. Next we analyze the complexity of computing the new hammocks. The necessary ingredients for these computations are the hammocks $H(w)$ and the lists $LB^*(w)$ and $RB^*(w)$, for all $w\in\Lambda_j$. We already know that these ingredients can be made available in $O(\alpha n+K)$ time and space. Finally, disregarding the $O(\alpha n)$ time spent "using" the lists LB^* and RB^*—each of them is used at most once—the construction of the new hammocks requires time and space proportional to their size. It is therefore sufficient to evaluate the total number of vertices in all the new hammocks at layer Λ_j. As usual, we distinguish between intersection-vertices and boundary-vertices. The cardinality of the first set is clearly $O(K)$, so we can turn our attention to the latter. Let us evaluate the maximum number of times a segment s of S can contribute a boundary-vertex to a new hammock at layer Λ_j. A canonical segment from S at level $\lambda_{j-1}+h$ contributes at most 2^h boundary-vertices. Since, on the other hand, s cannot have more than two canonical segments on the same level, it contributes at most $\sum_{1\leqslant h\leqslant\alpha}2^{h+1}\leqslant 2^{\alpha+2}$ vertices to the new hammocks. This establishes the claimed $O(2^\alpha n+K)$ upper bound on the time complexity of the construction. Finally, since s can contribute only two boundary-vertices to a single new hammock, the storage necessary for any of them is $O(n+K)$, which is dominated by the space requirement of the other structures, i.e., $O(\alpha n+K)$. This completes the proof. ∎

3.3. *Improving INTERCOMP*

INTERCOMP involves computing the intersections between $W(w)$ and $H(w)$ for each node w in T. Following the layer-based strategy, we describe an efficient method for carrying out this computation for all the nodes w at layer Λ_j. As usual, we proceed separately for all the subtrees T^* rooted at level λ_j. As usual, let $w_1,\ldots,w_m$ be the leaves of T^*; obviously,

$$\bigcup_{1\leqslant i\leqslant m} W(w_i)\subseteq \bigcup_{w\in T^*} W(w),$$

but do we actually have a set equality? Not quite, since a segment s in $W(w)$ may have its endpoint in $S(w)$ lie precisely on the slab-line of $S(w_i)$ in such a way that s contributes a segment in $H(w_i)$ and hence does not appear in $W(w_i)$. To remedy this discrepancy, we augment $W(w_i)$ with all such segments. Practically, this involves dropping the prescription in the proof of Lemma 3: "*As mentioned before, we do not have to keep a pointer to v in PW[i] if PL[i] already points to it.*" Of course, the price to pay is that we might occasionally try to apply the algorithm of Lemma 2 with a query segment that is already in the hammock under consideration. This can be detected immediately, however, and will thus not affect the asymptotic complexity of the algorithm.

We compute INTERCOMP by considering each node w_i in turn ($1 \leqslant i \leqslant m$), and performing the following task: compute the intersections of each segment in $W(w_i)$ with the hammocks in $SH = \{H(z) | z \in p_i\}$ (recall that p_i is the path in T^* from v to w_i). To do so, the only data structures needed will be $K_i(v)$ as well as all the hammocks at layer Λ_j. We proceed by considering each segment s in $W(w_i)$ separately. Let $s = AB$, with A the endpoint lying in $S(w_i)$. Note that B may or may not lie in $S(w_i)$. Without loss of generality, assume that A is the left endpoint of AB.

We are now faced with the main bottleneck of our earlier implementation, summed up in the following question: how can we apply the algorithm of Lemma 2 to the pairs $(s, H(z))$ without having to perform a planar point location for every pair? We solve this problem by making use of $K_i(v)$ as a *guide* to the desired hammock-faces; the idea being to perform planar point locations only on the graphs $K_i(v)$. Processing T^* will therefore proceed in two passes. In a first stage, we compute each of the new hammocks $K_1(v), \ldots, K_m(v)$ in turn, storing only *one at a time*. For each $K_i(v)$ we perform a planar point location on each endpoint from $W(w_i)$ that lies in the slab $S(w_i)$. We will actually refine this information so as to be in a position to apply Observation 1 or Lemma 2 at the end of the first pass. After this first, *information-gathering*, pass we will compute INTERCOMP for all nodes in T^*. We describe these two passes successively.

Pass One. If both A and B lie in the slab $S(w_i)$, we can locate the faces of $K_i(v)$ where they lie, which puts us in the conditions of Lemma 2 with respect to the new hammock. We can then compute all the relevant intersections of AB at layer Λ_j in time $O(\log n + K)$. If B does not lie in $S(w_i)$, we must consider whether it lies in $S(v)$ or not. We will assume that it does, since the treatment for this case actually encompasses the treatment for the other case as well.

Let $S(w_h)$ be the slab containing B; $h > i$. Note that AB also appears in $W(w_h)$. To start off, we compute the intersections between AA^* and $K_i(v)$, where A^* is the intersection of AB with the right slab-line of $S(w_i)$. Similarly, when processing $K_h(v)$ later on, we will compute the intersections between BB^* and $K_h(v)$, where B^* is the intersection of AB with the left slab-line of $S(w_h)$ (Fig. 13). This can be done in time $O(\log n + k)$ again, since we are still in the conditions of Lemma 2. Actually, we could even use the faster algorithm of Observation 1, since in both cases one of the endpoints lies on a slab-line. At this point, it is clear that the remaining intersec-

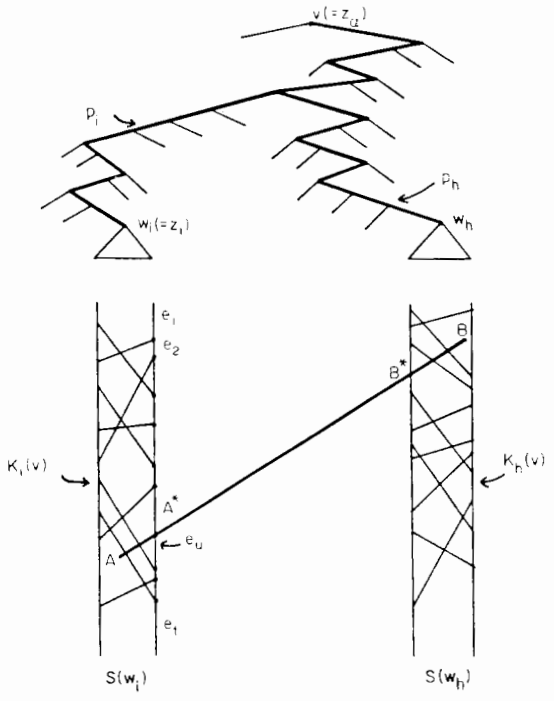

FIGURE 13

tions to report will involve A^*B^*, so we turn our attention to this subsegment exclusively.

Recall that in *Pass* 2, we will have to compute the remaining intersections that involve AB, i.e., the intersections of A^*B^* with $H(z)$, for all $z \in p_l \cup p_h$. If z belongs to $p_l \cap p_h$, both A^* and B^* lie in $S(z)$, so locating both points in $H(z)$ is needed in order to apply Lemma 2. Otherwise, it is easy to see that only one of A^* or B^* lies in $S(z)$, therefore one location only must be performed, since we can then apply the algorithm of Observation 1.

Since A^* and B^* play identical roles, we may without loss of generality concentrate on A^* only. Let $p_l = z_1, \ldots, z_\alpha$; our goal is to determine, for $l = 1, \ldots, \alpha$, which face in $H(z_l)$ contains A^*. We will take advantage of the fact that A^* lies on a slab-line to make these point locations very efficient. Let $e_1, \ldots, e_t$ be the list of edges on the right slab-line of $K_l(v)$, given in y-descending order (Fig. 13). Recall that since Observation 1 has already been applied to the segment AA^* in the hammock $K_l(v)$, we know which edge e_u contains A^*. We can therefore solve our problem by precomputing, for each edge $e_1, \ldots, e_t$, the α faces in SH that contain it. Let's define the array $F_u[1 \ldots \alpha]$ as follows: $F_u[l]$ contains the name of the face in $H(z_l)$ that contains the edge e_u. Note that by construction of $K_l(v)$ this face is well defined and unique.

To save storage, we compute a generic array $F[1 \ldots \alpha]$, which will be successively $F_1, \ldots, F_t$. At the outset, $F[l]$ contains the name of the unique upper (unbounded) face of $H(z_l)$. We will then scan edge $e_1, \ldots, e_t$ in this order, updating the array F accordingly, so that at step v, we have $F = F_v$. Let $H(z_l)$ be the unique hammock from which the upper endpoint of e_v emanates, and let f be the corresponding face. Updating F simply involves setting $F[l] = f$. Of course, before starting this computation, all the segments of the form AA^* will have been processed, so that each point A^* will have been assigned the unique edge e_u where it lies. Now, in the course of updating F, whenever we encounter an edge e_u that contains a point of the form A^*, we look up in F to find the α faces in SH that contain the point. At this stage, we will simply copy these α names in a list $N(A^*)$, and proceed.

Pass Two. Only after all the $K_i(v)$ will have thus been processed and the N-lists computed, we will use these lists in order to apply Observation 1 and Lemma 2 (depending on the position of B). It is easy to see that this two-pass scheme is necessitated by the fact that the algorithm of Lemma 2 requires the locations of both A^* and B^*. Note, however, that storing all the lists N requires only $O(n\alpha)$ storage. With all these lists, we are now in a position to apply either Observation 1 or Lemma 2 depending on the position of B. We can see, as claimed earlier, that the case where B lies outside of $S(v)$ can be treated similarly (applying Observation 1).

LEMMA 7. *Let* $K = \sum_{1 \leq i \leq \alpha} (k_h(\lambda_j - i + 1) + k_w(\lambda_j - i + 1))$. *All the intersections between segments of* $W(w)$ *and hammocks* $H(w)$, *for all nodes* $w \in \Lambda_j$, *can be computed (exactly once) in time* $O((\log n + 2^z)n + k)$, *using* $O(\alpha n + K)$ *space. This accounts for all the preprocessing as well.*

Proof. We review the various components of INTERCOMP, excluding the parts that have already been studied in Lemma 6:

1. Calling on optimal planar point location procedure. From Lemma 3, we know that the total size of the sets W at any level is $O(n)$, therefore $O(n)$ calls are executed, which amounts to $O(n \log n)$ time.

2. Reporting intersections at a cost of $O(1)$ time per report.

3. Setting up the array F: for each w_i, takes time proportional to $|W(w_i)| + \alpha +$ "number of boundary-vertices in $K_i(v)$". From Lemma 3 and the proof of Lemma 6, this will amount to $O(n + \alpha + 2^z n) = O(2^z n)$ time for the entire layer.

4. Checking for intersections in SH: for each w_i, takes $O(\alpha\,|W(w_i)| + K)$ time, hence a total of $O(\alpha n + K)$ time.

The space needed is

$$O\left(\alpha n + \max_{T^*}\left\{\max_{w_i \in T^*} |K_i(v)| + \sum_{w \in T^*} |H(w)|\right\}\right) = O(\alpha n + K).$$

It is routine to fine-tune the implementation so as to report each intersection exactly once. ∎

3.4. *Putting the Pieces Together*

LEMMA 8. *Given a set S of n arbitrary segments in the plane, it is possible to report all k intersecting pairs exactly once, in $O(n(\log^2 n/\log\log n)+k)$ time and $O(n\log\log n+k)$ space.*

Proof. Recall that $K=\sum_{1\leqslant i\leqslant \alpha}(k_h(\lambda_j-i+1)+k_w(\lambda_j-i+1))$. We recap the chronological sequence of the entire algorithm. After an initial $O(n\log n)$ time sort on the coordinates in S, and the $O(n)$ time set-up of the binary tree T, we start the computation of HAMCOMP and INTERCOMP on each layer $\Lambda_1,\ldots,\Lambda_d$, with $d=\lceil(1+\lceil\log_2 2n\rceil)/\alpha\rceil$. This involves updating the arrays PW and PL in $O(n)$ time, as well as computing all the hammocks in the layer. The latter operation requires $O(\alpha n+K)$ time and space, as shown in Lemma 6. We then proceed with the computation of the new hammocks $K_i(v)$ and the execution of INTERCOMP. Lemmas 6 and 7 show that this requires $O((\log n+2^\alpha)\ n+K)$ time and $O(\alpha n+K)$ space. The entire computation will thus require $O(n\log n+(\log n/\alpha)\ n(\log n+2^\alpha)+k)$ time and $O(\alpha n+k)$ space, where k is the total number of intersections between segments of S. Setting $\alpha=\lfloor\log_2\log_2 n\rfloor$ establishes the lemma. ∎

We next show how to reduce the storage requirement to $O(n+k)$. To do so, we switch between the previous algorithm and any of the standard sweep-line algorithms. Recall that these algorithms report all k intersections in time $O((n+k)\log n)$, using $O(n)$ [3] or $O(n+k)$ [1, 11] space. Let $C=\lceil n\log_2^2 n/\log_2\log_2 n\rceil$, and $T(n)$ (resp. $S(n)$) be the time (resp. space) necessary for the computation. The new method we propose consists of starting the computation using the sweep-line algorithm for exactly C steps. If the algorithm terminates within C steps, it will have achieved $T(n)=O(n\log^2 n/\log\log n)$ and $S(n)=O(n+k)$. Otherwise, we will have $k=\Omega(n\log n/\log\log n)$. We finish the computation by using the algorithm of Lemma 8, which gives $T(n)=O((n\log^2 n/\log\log n)+k)$ and $S(n)=O(n\log\log n+k)=O(k)$. We can conclude with the main result of this section.

THEOREM 2. *Given a set S of n arbitrary segments in the plane, it is possible to report all k intersecting pairs exactly once, in $O(n\log^2 n/\log\log n+k)$ time and $O(n+k)$ space.*

Open problem. Is it possible to reduce the time complexity of the algorithm to optimal $O(n\log n+k)$?

Before closing this section, for the sake of completeness, we include an immediate corollary of Lemmas 1, 2.

THEOREM 3. *Given an arrangement of n lines in the plane and an arbitrary query segment q, it is possible to report all t intersections between q and the line arrangement in $O(\log n+t)$ time, using $O(n^2)$ preprocessing.*

4. The Counting Algorithm

In this section we look at the problem of *counting*, rather than reporting the intersecting pairs of segments in S. As before, we represent the segments in a segment tree T which we construct bottom up, level by level. Let $v_1,\ldots, v_p$ be the nodes of T at level λ. We set up the lists $L(v_i)$ and $W(v_i)$ as in the previous section, but we *do not* compute the hammocks $H(v_i)$. Instead, for $i=1,\ldots, p$, we directly count all the intersections between segments of $L(v_i)$ themselves, and between segments of $L(v_i)$ and segments of $W(v_i)$. Consider the dual mapping which puts in one-to-one correspondence the point (a, b) and the line $ax+by+1=0$. The point and the corresponding line are called *dual* of each other. Geometrically, if d is the distance from the origin O to point p, the dual of p is the line perpendicular to Op at distance $1/d$ from O and placed on the other side of O (in order to handle the case of lines passing through the origin, this definition requires that the plane be extended to the *two-sided plane*, as defined in [6]). A segment $s\in S$ is mapped into a *double wedge*, $C(s)$ (Fig. 14), with the property that s intersects a line L if and only if $C(s)$ contains the dual of L.

THEOREM 4. *Given a set S of n arbitrary segments in the plane, it is possible to evaluate the number of intersecting pairs in $O(n^{1.695})$ time, using $O(n)$ space.*

Proof. The algorithm is intimately based on a recent result by Edelsbrunner and Welzl [5]. This result states that it is possible to preprocess a set of n points in $O(n^{1.5}\log^2 n)$ time and $O(n)$ space, so that the number of points lying inside any query triangle can be evaluated in $O(n^{0.695})$ time. This uses a novel data structure, called a *conjugation tree*. As usual, let $v_1,\ldots, v_p$ be the nodes of T at level λ. For each v_i, we will use conjugation trees to compute the number of intersections (1) among the segments of $L(v_i)$ and (2) between segments of $W(v_i)$ and segments of $L(v_i)$.

Let $l(s)$ be the infinite line passing through segment s and $t(s)$ be the dual of $l(s)$; also let $r_i(s)$ be the intersection of s with the slab associated with v_i, that is, s clip-

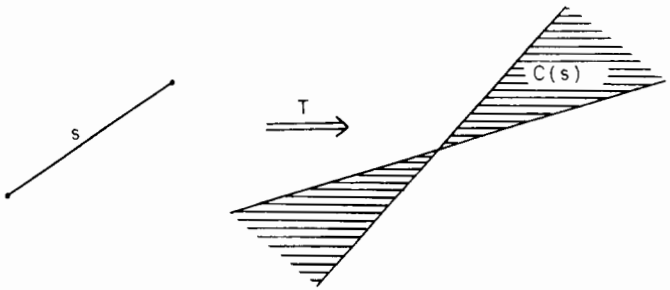

FIGURE 14

ped to fit inside $S(v_i)$. For each level λ, and each node v_i at level λ, perform the following steps:

Step 1. Construct a conjugation tree with respect to the point-set $P_i = \{t(s) | s \in L(v_i)\}$, and set $I = 0$.

Step 2. For each $s \in L(v_i)$, let t_1, t_2 be the two triangles formed by the wedge $C(s)$. Use the conjugation tree to compute $I = I + |t_1 \cap P_i| + |t_2 \cap P_i|$.

Step 3. Set $C = I/2 - |L(v_i)|$.

Step 4. For each $s \in W(v_i)$, let t_1, t_2 be the two triangles formed by the wedge $C(r_i(s))$. Use the conjugation tree to compute $I = I + |t_1 \cap P_i| + |t_2 \cap P_i|$.

The final value of I is precisely the number of intersections between the segments of $L(v_i)$ and those of S. To see this, recall that each segment s of $L(v_i)$ is the intersection of the slab $S(v_i)$ with some segment s_j of S. Since any segment s of $L(v_i)$ intersects $u \in L(v_i)$ if and only if it intersects $l(u)$, we can detect such an intersection by checking whether $t(u)$ lies in $C(s)$. Following this method, Step 2 computes the number of intersections among segments of $L(v_i)$. This actually computes every intersection twice, including the intersection of a segment with itself. The proper value is reset in Step 3. Finally, Step 4 performs a similar sequence of operations with respect to the (clipped) segments of $W(v_i)$.

Applying this procedure to $L(v_i)$ and $W(v_i)$ requires time

$$O(|L(v_i)|^{1.5} \log^2 |L(v_i)| + (|L(v_i)| + |W(v_i)|)\ |L(v_i)|^{0.695})$$

and space $O(|L(v_i)|)$. Since $\sum_{1 \leqslant i \leqslant p} |L(v_i)| \leqslant \sum_{1 \leqslant i \leqslant p} |W(v_i)| \leqslant 2n$, we easily derive that

$$O((n^{1.5} \log^2 n + n^{1.695}) \log n) = O(n^{1.695} \log n)$$

operations and $O(n)$ storage will be required. The proof of correctness follows from Section 2.III. Note that given the conservative approximation of $\log_2(1+\sqrt{5})$ by 1.695, the number of operations can be legitimately written as $O(n^{1.695})$. ∎

It is always interesting to recast a problem in a dual space, if only to appreciate the various disguises it can take on. In particular, the dual form of Theorem 3 states that there exists an $O(n^2)$ data structure, with which it is possible to report the subset of n fixed points lying inside a query double wedge. If k points are to be reported, the algorithm takes $O(\log n + k)$ time.

5. Concluding Remarks

We wish to close this paper on a methodological note. Our main result is an algorithm for intersecting line segments, whose running time is linear as a function of the output size. It is interesting to notice that previous algorithms fail to achieve

this result, partly because of their reliance on what is otherwise one of the most successful methods of computational geometry, i.e., the iterative reduction of the problem's dimensionality (via line-sweeping). Also, straightforward divide-and-conquer seems inadequate because the merging phase seems as difficult as the original problem, when no sorted order on the intersections is available. In order to break the $k \log n$ barrier, we had to base the computation on a hierarchical decomposition of line segments: (1) proceeding bottom-up and (2) dealing only with a subpart of the underlying abstract data structure at any given time. This global approach (which preserves the dimensionality of the problem) may provide a general framework for the study of point sets and their dual, line arrangements.

The main contributions of this paper are: (1) an algorithm for reporting all k intersecting pairs among n arbitrary segments in $O(n(\log^2 n/\log\log n)+k)$ time, using $O(n+k)$ space; (2) an algorithm for computing the number of intersections in $O(n^{1.695})$ time, using $O(n)$ space. Neither of these algorithms has been shown to be optimal, and it appears safe to conjecture that neither is. In particular, it is possible that a (yet) more global treatment of hammocks may reduce the run time of the reporting algorithm to $O(n \log n + k)$, thus making it optimal. It is clear, however, that the very notion of hammocks seems incompatible with any $O(n)$ space algorithm. In this regard, any attempt at reducing the space complexity will have to contemplate a very different approach.

ACKNOWLEDGMENTS

I wish to thank Leo Guibas and Janet Incerpi for helpful comments on the presentation of these results.

REFERENCES

1. J. L. BENTLEY AND T. OTTMANN, Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput* **C–28**, No. 9 (1979), 643–647.
2. J. L. BENTLEY AND D. WOOD, An optimal worst-case algorithm for reporting intersections of rectangles, *IEEE Trans. Comput.* **C–29** (1980), 571–577.
3. K. Q. BROWN, Comments on "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Trans. Comput.* **C–30** (1981), 147–148.
4. H. EDELSBRUNNER, L. GUIBAS, AND J. STOLFI, Optimal point location in a monotone subdivision, to appear.
5. H. EDELSBRUNNER AND E. WELZL, "Halfplanar Range Search in Linear Space and $O(n^{0.695})$ Query Time," Tech. Univ. Graz., IIG Report 111, 1983.
6. L, GUIBAS, L. RAMSHAW, AND J. STOLFI, A kinetic framework for computational geometry, *in* "Proc. 24th Annu. Found. Comput. Sci.," pp. 100–111, 1983.
7. L. J. GUIBAS AND J. STOLFI, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *in* "Proc. 15th Annu. SIGACT Sympos.," pp. 221–234, 1983.
8. D. G. KIRKPATRICK, Optimal search in planar subdivisions, *SIAM J. Comput.* **12**, No. 1 (1983), 28–35.

9. H. G. MAIRSON AND J. STOLFI, Reporting and counting line segment intersections, unpublished.
10. D. E. MULLER AND F. P. PREPARATA, Finding the intersection of two convez polyhedra, *Theoret. Comput. Sci.* **7** (1978), 217–236.
11. J. NIEVERGELT AND F. P. PREPARATA, Plane-sweep algorithms for intersecting geometric figures, *Comm. ACM*, **25**, No. 10 (1982), 739–747.
12. M. I. SHAMOS AND D. J. HOEY, Geometric intersection problems, *in* "Proc. 17th Annu. Found. Comput. Sympos.," pp. 208–215, 1976.