SIAM J. COMPUT.
Vol. 14, No. 3, August 1985

AN $O(E \log E + I)$ EXPECTED TIME ALGORITHM FOR THE PLANAR SEGMENT INTERSECTION PROBLEM*

EUGENE W. MYERS†

Abstract. It is an open question in computational geometry as to whether there exists an $O(E \log E + I)$ algorithm to determine the I intersections of a collection of E line segments in the plane. An approach utilizing a work list bubble sort and a distribution-based search is presented. The resulting algorithm has $O(E \log E + I)$ expected time complexity. In the worst case the algorithm has the same complexity as the algorithm of Bentley and Ottmann [IEEE Trans. Comput., 28 (1979), pp. 643-647]: $O(E \log E + I \log E)$. The algorithm requires only $O(E)$ space and in contrast to prior work, no restrictions are placed upon the nature of the intersections.

Key words. concrete complexity, computational geometry, scan-line algorithm, work list bubble sort, distribution-based search

1. Introduction. A recent trend in computer science has been the study of geometric problems in terms of their algorithmic complexity [3], [12], [13], [17]. In particular, geometric intersection problems have been studied extensively [2], [5], [11], [18]. The results are applicable to many practical problems including computer graphics, automated printed circuit layout, and computer-assisted architectural design. Efficient algorithms for the intersection problem treated here provide the basis for faster hidden-line elimination in the "object-space" framework [8], [16], [19].

The problem to be examined is as follows. Given E line segments in the plane, list (count) all intersecting pairs. Shamos and Hoey [18] presented an $O(E \log E)$ algorithm to detect whether intersections exist. By extending their technique, Bentley and Ottmann [2] demonstrated an $O(E \log E + I \log E)$ time, $O(I)$ space algorithm for listing all intersecting pairs of segments where I is the number of intersections. A subsequent refinement by Brown [4] reduced the space requirement to $O(E)$. Although this appears to be an improvement over the naive $O(E^2)$ algorithm, it is not the case when I approaches E^2. The problem of designing a definitively superior $O(E \log E + I)$ time, $O(E)$ space algorithm was first conjectured in [18] and is currently open.

An $O(E \log E + I)$ *expected time* algorithm is presented here. The necessary statistical hypothesis is that the $2E$ endpoints of the segments are uniformly distributed. The algorithm's worst case performance is again $O(E \log E + I \log E)$. The algorithm requires only $O(E)$ space. By a direct refinement of the method of Ottmann and Bentley, A. Schmitt [15] has simultaneously designed an algorithm with the same time complexity as the one given here. However, his algorithm uses $O(I)$ space and requires the more stringent statistical hypothesis that both the segment endpoints and the I points of intersection be uniformly distributed. The algorithm here achieves the less restrictive assumption through the novel use of a work list bubble sort to detect intersections between segment endpoints.

In the next section, the problem statement is formalized and preliminary constructions and definitions are made. Section 3 illustrates the central role of a work list bubble sort and distribution-based searching in the algorithm presented in § 4. The discussion in § 5 highlights a number of subproblems that can be solved in $O(E \log E + I)$ worst-case time.

* Received by the editors July 19, 1982, and in final revised form May 3, 1984. This work was supported in part by the National Science Foundation under Grant MCS-8210096.

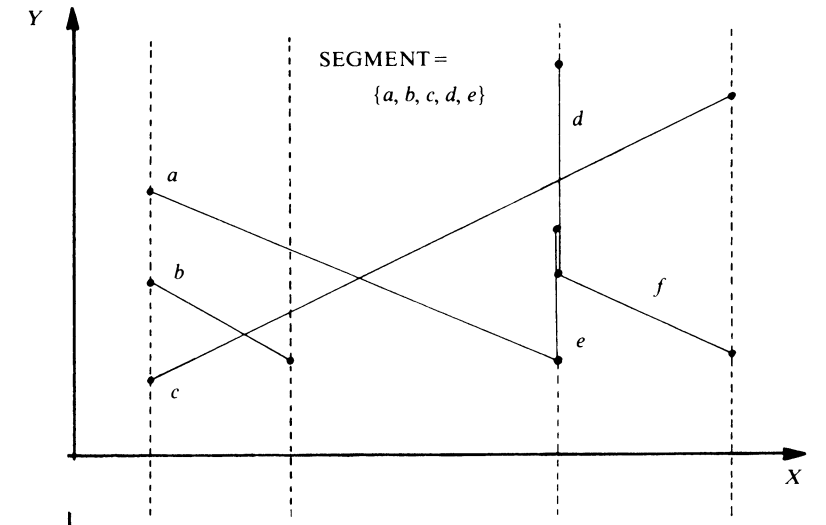
† Department of Computer Science, University of Arizona, Tucson, Arizona, 85721.

2. Preliminaries. The problem is to find all the intersections of a collection of E planar line segments, $SEGMENT = \{e_1, e_2, \cdots, e_E\}$. Each line segment is specified by the x- and y-coordinates of its endpoints: $e = \langle\langle \underline{x}_e, \underline{y}_e\rangle, \langle \bar{x}_e, \bar{y}_e\rangle\rangle$. It is stipulated that in an $O(E)$ prepass the segment endpoints have been arranged so that the start-point $\langle \underline{x}_e, \underline{y}_e\rangle$ is to the "left" of the final-point $\langle \bar{x}_e, \bar{y}_e\rangle$, i.e. $\underline{x}_e < \bar{x}_e$ or $\underline{x}_e = \bar{x}_e$ and $\underline{y}_e \leqq \bar{y}_e$.

As in the algorithms of Shamos and Hoey [18] and Bentley and Ottmann [2], the key conceptualization is to imagine a vertical *scan line* sweeping from left to right along the x-axis. The critical events are those moments at which this scan line reaches the abscissa of a line segment endpoint. Let $EVENT = \langle x_1, x_2, \cdots, x_{EV}\rangle$ be the list obtained by sorting the set of abscissas of segment endpoints into ascending order. Observe that $EV \leqq 2E$ as each segment has two endpoints.

For each event, x_i, there may be more than one segment having an endpoint at x_i. Such segments will be distinguished on the basis of whether x_i is the abscissa of their start-point, final-point, or both (i.e. vertical segments). Formally, let $BEG(i) = \langle e_1, e_2, \cdots, e_{B(i)}\rangle$ be the list of segments for which $\underline{x}_e = x_i \neq \bar{x}_e$; let $END(i) = \langle e_1, e_2, \cdots, e_{E(i)}\rangle$ be the list of segments for which $\underline{x}_e \neq x_i = \bar{x}_e$; and let $VERT(i) = \langle e_1, e_2, \cdots, e_{V(i)}\rangle$ be the list of segments for which $\underline{x}_e = x_i = \bar{x}_e$. It is further stipulated that the segments in each of these lists occur in descending order of their start-point ordinates $\underline{y}_e$. Figure 1 illustrates these lists.

Algorithmically, all the lists above can be constructed with a single sort. First form an auxiliary list consisting of the following 4-tuples. For each nonvertical segment e introduce the 4-tuples $\langle e, \underline{x}_e, 0, -\underline{y}_e\rangle$ and $\langle e, \bar{x}_e, 2, -\underline{y}_e\rangle$. For each vertical segment e introduce the 4-tuple $\langle e, \underline{x}_e, 1, -\underline{y}_e\rangle$. In $O(E \log E)$ time, heapsort this auxiliary list according to the lexicographical order of the second, third, and fourth components.



EVENT	x_1	x_2	x_3	x_4
BEG	$\langle a, b, c\rangle$	ϕ	$\langle f\rangle$	ϕ
VERT	ϕ	ϕ	$\langle d, e\rangle$	ϕ
END	ϕ	$\langle b\rangle$	$\langle a\rangle$	$\langle c, f\rangle$

FIG. 1. *Fundamental structures.*

Then note that *EVENT* is the list of second components with duplicate entries removed. Further note that the list of first components is the concatenation of the lists $BEG(1)$, $VERT(1)$, $END(1)$, $BEG(2), \cdots, END(EV)$ in the order given. The boundaries of this partition are easily identified by examining the second and third components. Thus all the desired lists can be extracted in a final $O(E)$ sweep over the sorted auxiliary list.

The elements of *EVENT* divide the range $[x_1, x_{EV}]$ into the *event intervals* $[x_i, x_{i+1}]$ for i between 1 and EV-1. By construction, the abscissa of a segment endpoint cannot lie strictly within any of the event intervals. Thus if a segment has a point whose abscissa is interior to an event interval then the segment is guaranteed to span the entire interval (i.e., have a point at abscissa x for every x in the interval). The set of segments spanning the ith event interval is formally defined with the recursive definition:

$$SPAN(i) = \quad \text{If } i = 0 \text{ Then } \varnothing \text{ Else } SPAN(i-1) \cup (BEG(i) - END(i)).$$

Note that for all i, $SPAN(i)$ does not contain any vertical segments.

The algorithm presented here centers on computing the *x-order*, $<_x$, of the segments for every event x. Intuitively, $<_x$ ranks segments according to the order in which they intersect a scan line at x. For nonvertical segments let m_e be the slope of segment e and let $y_e(x)$ be the ordinate value of segment e at abscissa x. For vertical segments let $m_e = \infty$ and let $y_e(x) = y_e$. Formally, the rank of a segment on a scan line at x is the lexicographical rank of the ordinate value/slope pair $\langle y_e(x), m_e \rangle$:

$$e <_x f \text{ iff } y_e(x) < y_f(x) \text{ or } y_e(x) = y_f(x) \text{ and } m_e < m_f.$$

Figure 2 gives an example of the x-order of a collection of segments. Clearly one can make analogous definitions for the other x-relations: $=_x$, $\leqq_x$, $>_x$, $\geqq_x$, and $\neq_x$.

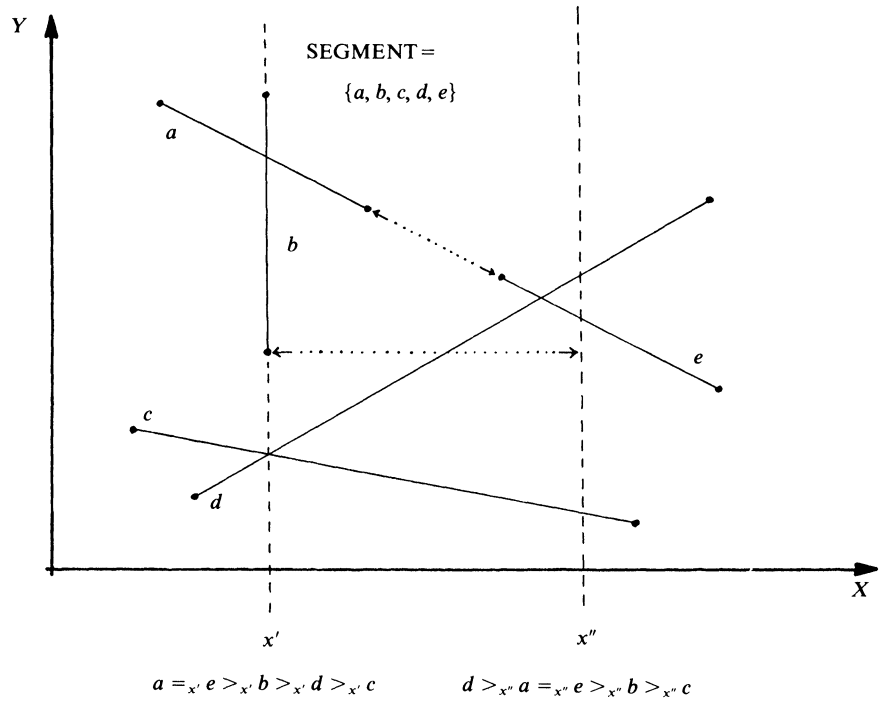


FIG. 2. *X-order example.*

In prior scan line algorithms [2], [15], [18] for the planar intersection problem, the following degenerate situations were explicitly excluded from the problem domain.

(a) Vertical lines.

(b) Multi-segment intersections (three or more segments intersecting at a point).

(c) Infinite intersections (collinear segments that intersect).

In this paper the problem domain is unrestricted. The preceding definitions and all subsequent results have been designed with these anomalous cases in mind. For example, the inclusion of slope in the definition of $<_x$ differs from prior work. This additional refinement insures that multi-segment intersections will be properly treated as collinear and only collinear segments are equal in any (and every) x-order.

A characterization of the conditions under which segments intersect in terms of the above constructions is now presented. Lemma 1 asserts that segments e and f intersect if there is an event interval in which e and f satisfy one of three mutually disjoint conditions. Condition 1.1 characterizes intersections involving a vertical segment. Condition 1.2 covers the case in which two nonvertical segments intersect at the start point of one (or both) of the segments. The main thrust of Lemma 1 is embodied in Condition 1.3, the *exchange predicate*, which reflects an observation first made in [18]. It asserts that any other intersection between nonvertical segments e and f is characterized by a reversal in the x-order of e and f at the endpoints of an event interval.

LEMMA 1. *Without loss of generality if e and f are not vertical then assume* $\underline{x}_e \geqq \underline{x}_f$, *otherwise assume that e is vertical and if f is also vertical then further assume* $\underline{y}_e \leqq \underline{y}_f$. *Segments e and f intersect if and only if there exists i such that*

(1.1) $e \in VERT(i)$ *and* $f \in SPAN(i-1) \cup BEG(i) \cup VERT(i)$ *and* $y_f(x_i) \in [\underline{y}_e, \bar{y}_e]$,

or

(1.2) $e \in BEG(i)$ *and* $f \in SPAN(i-1) \cup BEG(i)$ *and* $y_f(x_i) = \underline{y}_e$,

or

(1.3) $e, f \in SPAN(i)$ *and* $(e <_{x_i} f$ *and* $e >_{x_{i+1}} f$ *or* $f <_{x_i} e$ *and* $f >_{x_{i+1}} e)$.

Proof. ($\Rightarrow$) Suppose e and f intersect at $\langle x, y\rangle$ and if they are collinear that this is the point of intersection with smallest abscissa (ordinate for vertical segments). First consider the case where e and f are not vertical and $\underline{x}_e \geqq \underline{x}_f$. If $x = \underline{x}_e$ then let i be the integer for which $e \in BEG(i)$. Observe that $y_f(x_i) = y_f(x) = y_e(x) = \underline{y}_e$ and that $x_i = x \in [\underline{x}_f, \bar{x}_f]$ implies $f \in SPAN(i-1) \cup BEG(i)$, i.e. Condition 1.2 holds. If $x \neq \underline{x}_e$ then let i be the integer for which $x \in (x_i, x_{i+1}]$. Observe that e and f cannot be collinear and that $x \in (\underline{x}_e, \bar{x}_e] \cap (\underline{x}_f, \bar{x}_f]$ implies $e, f \in SPAN(i)$. If $e <_{x_i} f$ then $y_e(x_i) < y_f(x_i)$. Moreover, e and f intersect at $x \in (x_i, x_{i+1}]$ implies $y_e(x_{i+1}) \geqq y_f(x_{i+1})$ and $m_e > m_f$ implies $e >_{x_{i+1}} f$. Similarly $e >_{x_i} f$ implies $e <_{x_{i+1}} f$. Thus Condition 1.3 holds.

Now suppose that e is vertical. Let i be the integer for which $e \in VERT(i)$ and note that $x = x_i$ and $y \in [\underline{y}_e, \bar{y}_e]$. If f is not vertical then $y_f(x_i) = y$ and $x \in [\underline{x}_f, \bar{x}_f]$ implies $f \in SPAN(i-1) \cup BEG(i)$, i.e. Condition 1.1 holds. If f is also vertical then e and f must be collinear, $f \in VERT(i)$, and $\underline{y}_f \geqq \underline{y}_e$ implies $y_f(x_i) = \underline{y}_f = y$ as y is the smallest ordinate of an intersection point. Thus Condition 1.1 holds.

($\Leftarrow$) Conditions 1.1 and 1.2 imply e and f intersect at $\langle x_i, y_f(x_i)\rangle$. Now suppose Condition 1.3 is true. If $y_e(x_i) = y_f(x_i)$ or $y_e(x_{i+1}) = y_f(x_{i+1})$ then immediately e and f intersect. Otherwise $e <_{x_i} f$ and $e >_{x_{i+1}} f$ implies $y_e(x_i) < y_f(x_i)$ and $y_e(x_{i+1}) > y_f(x_{i+1})$. That is, e is below f at x_i, above f at x_{i+1}, and both segments span the interval. They must intersect in the interval $[x_i, x_{i+1}]$. Similarly e and f intersect when $f <_{x_i} e$ and $f >_{x_{i+1}} e$. □

3. The key methods—work list bubble sort and distribution-based search. Consider the following simplification of the intersection problem. Suppose that $EVENT = \langle x_1, x_2\rangle$ and that $BEG(1) = END(2) = SEGMENT$. There is only one event interval and no

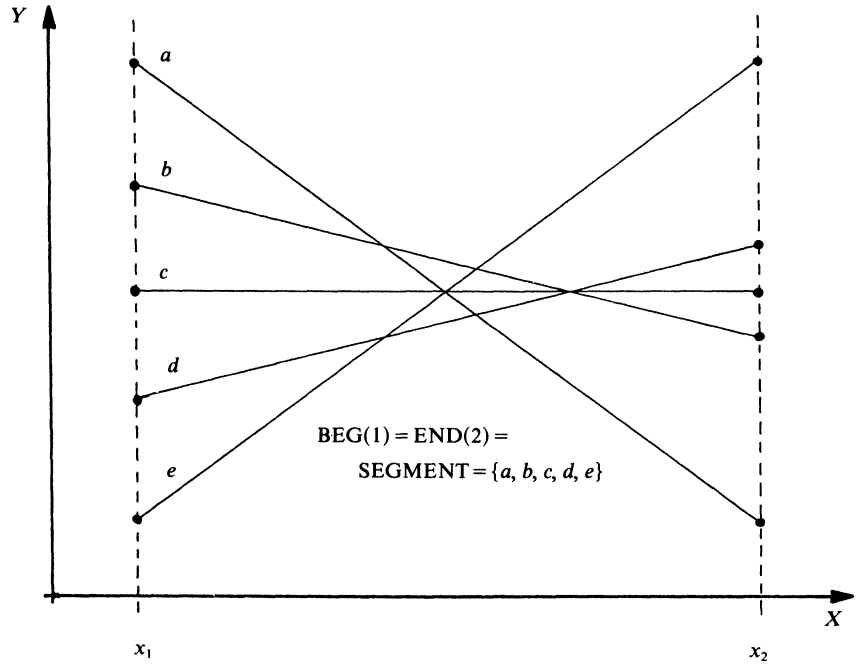


FIG. 3. *A single interval intersection problem.*

vertical segments. An instance of a "Single Interval Intersection Problem" is depicted in Fig. 3. By Lemma 1 it suffices to find the intersections satisfying Conditions 1.2 and 1.3 for the interval $[x_1, x_2]$. An algorithm is sketched that detects these intersections in $O(E \log E + I)$ time and $O(E)$ space.

As a first step, sort the segments into x_1-order in $O(E \log E)$ time. Observe that segments with equal ordinate values, $y_e(x_1)$, occur in contiguous sub-lists of this ordering. Every pair of segments from such a sub-list and only these pairs satisfy Condition 1.2 of Lemma 1. The sub-lists can be detected in $O(E)$ time and the set of segment pairs from each sub-list can then be listed in $O(I)$ time where I is the number of intersections.

It remains to find all the intersections satisfying the exchange predicate (1.3). Consider the application of a bubble sort to the x_1-ordered segments to obtain the x_2-ordering of the segments. The key observation is that the set of segment pairs exchanged in performing the sort is exactly the set of segment pairs satisfying the exchange predicate.

The naive version of bubble sort requires $O(E^2)$ time. What is needed is an $O(E+1)$ time algorithm where I is the number of intersections or equivalently the number of exchanged pairs. This can be accomplished with the following *work list* variant of bubble sort. Let the current oder of segments be the x_1-order computed n the first step. Initialize the work list to be the set of all segments e for which e and its successor are *not* in x_2-order. While the work list is not empty, perform the following steps.

(3A) Pop a segment e from the work list. Let f be its successor in the current order.
(3B) Remove f and the predecessor of e from the work list if present.
(3C) Exchange e and f in the current order and report that they intersect.
(3D) Push the current predecessor of f onto the work list if it and f are not in x_2-order. Push e onto the work list if it and its current successor are not in x_2-order.

The validity of the algorithm follows from the fact that at the start of each iteration the work list consists of those segments for which it and its current successor are not in x_2-order. The algorithm terminates as each iteration strictly reduces the "exchange distance" to the x_2-ordered arrangement of segments.

Steps 3A through 3D can be performed in constant time with the following structures. Model both the current order and work list as doubly-linked lists in which each cell contains a pointer to the record modeling the appropriate segment. Let each segment record contain a pointer to its cell in the current order and a pointer to its cell in the work list if present, the nil pointer otherwise. In constant time, cells can be popped and pushed from the work list and an arbitrary cell within the list can be deleted. Given a cell in the work list (current order) one can reach the corresponding cell in the current order (work list) via the segment record. The links of the current order list give current successors and predecessors. Segments are exchanged in this order by interchanging just the segment record pointers of the relevant adjacent cells.

With these models the work list bubble sort requires $O(E+I)$ time and $O(E)$ space. The initial work list is formed in an $O(E)$ sweep over the x_1-order of the segments. Steps 3A through 3D are repeated exactly I times as an intersection is reported in each iteration. Thus the sort proper takes $O(I)$ time. Only $O(E)$ space is required at any given moment as each segment occurs in the work list at most once. Note that intersections are not necessarily reported in order of increasing abscissa. For example, when applied to Fig. 3, the algorithm reports intersections out of order regardless of the initial state of the work list and its implementation as a stack of queue.

The general intersection problem can be viewed as $EV-1$ separate single interval problems. The algorithm presented here solves these individual problems in increasing event order. One can imagine the scan-line as jumping from event to event while the work list bubble sort detects the intersections between jumps. The $O(E \log E)$ cost of determining the initial x-order for each interval problem readily distributes across the computation: the bubble sort for the ith interval delivers the initial order for the $(i+1)$st interval. Distributing the $O(E)$ cost of computing the initial work list for each interval problem is more difficult. If during the processing of some interval, a segment comes to have a successor in the current x-order with which it satisfies the exchange predicate then the segment must immedately be placed in the work list of the interval in which the exchange predicate is satisfied. This requires a search for the interval containing the abscissa of the point of intersection. An $O(\log E)$ bisecting search on the ordered *EVENT* list could be used. However, a distribution-based search reduces the cost to $O(1)$ expected time under the assumption that the elements of *EVENT* are uniformly distributed in the interval $[x_1, x_{EV}]$.

Recently, much attention has been given to distribution-based sorting methods [1], [6], [7], [10], [20]. The technique sketched here is the search analogue of the sorting-by-partition method in [10]. Consider evenly dividing the interval $[x_1, x_{EV}]$ into EV subintervals (buckets) of size $\Delta = (x_{EV} - x_1)/EV$. Observe that an arbitrary x is in the bucket $[x_1 + \Delta H(x), x_1 + \Delta(H(x)+1))$ where $H(x) = \lfloor (x - x_1)/\Delta \rfloor$. For each i in $[0, EV]$, let $MIN(i)$ be the ordinal position of the smallest event in the interval $[x_1 + \Delta i, x_{EV}]$.

$$MIN(i) = \min \{ j \mid x_j \in EVENT \text{ and } H(x_j) \geqq i \}.$$

For the limiting case, let $MIN(EV+1) = MIN(EV)$. It is an elementary exercise to design an algorithm that constructs the vector MIN in an $O(E)$ sweep of the ordered *EVENT* list. Under the hypothesis that the events in $EVENT - \{x_1, x_{EV}\}$ are uniformly distributed over the interval $[x_1, x_{EV}]$, the expected number of events in a bucket is

$(EV-2)/EV$ as there are EV buckets and the $EV-2$ events are found in a given bucket with equal probability. Thus the expected value of $MIN(i)-MIN(i-1)$ is also $(EV-2)/EV$ for all i.

Suppose segments e and f are known to intersect by the exchange predicate (1.3) in some event interval. This interval can be found as follows. First determine the abscissa of the segments' point of intersection, $XINTER(e,f)$, by analytic means. Let $\delta = H(XINTER(e,f))$. It follows from the construction of MIN that $x_{MIN(\delta)-1} < XINTER(e,f) \leqq X_{MIN(\delta+1)}$. Thus the event interval, $[x_{i-1}, x_i]$, in which the segments satisfy the exchange predicate must have i in the range $[MIN(\delta), MIN(\delta+1)]$. The desired event interval can then be found by searching this limited subrange of event intervals. By the uniform distribution hypothesis the expected number of intervals in the subrange is $(MIN(\delta+1)-MIN(\delta))+1<2$. Thus by employing a bisecting search over this subrange, the expected search time is $O(1)$ and the worst case is guaranteed to be $O(\log E)$.

On a floating-point computer the calculation of real quantities is not exact due to the introduction of rounding errors. When realized on such a machine the search algorithm above is correct only if the computation of H is monotonic. That is, $x' \geqq x \Rightarrow \tilde{H}(x') \geqq \tilde{H}(x)$ where $\tilde{H}$ is the computed value of H (as opposed to its true value). If this condition is not met by the host hardware then the computed value of δ may differ from its true value. In such instances the index of the desired event interval will not be in the range $[MIN(\delta), MIN(\delta+1)]$ and the search will fail. This weakness can be removed by employing a finger-based seach [9] for resolving bucket collisions. In this context the search would start at the $MIN(\delta)$th event interval (the finger) and then search right or left for the correct interval in a bisecting fashion. The expected and worst-case times for the search remain unchanged.

The uniform distribution hypothesis was chosen for simplicity. More generally, the expected time for the search is $O(1)+E(\max(\log N_i))$ where E denotes "expected value" and N_i is the number of events in the ith bucket. From [1] it follows that the search is still $O(1)$ if the underlying density function is bounded, Riemann-integrable, and has compact support. The N-tree method of Ehrlich [7] could also be used here. It has the advantage of performing well over a wider class of density functions [20] (including those with exponentially vanishing tails). Its primary disadvantage is that it requires $O(E^2)$ space in the worst case.

4. The algorithm. The work list variation of bubble sort leads to an efficient algorithm for the single interval problem. The distribution-based search gives an $O(1)$ expected time method for determining the event interval in which a pair of segments intersect. With these methods the planar segment intersection problem can be solved efficiently. The simple data structures employed in the single interval problem must be enhanced to meet the more dynamic requirements of the general algorithm. The extension of these structures and the primitive operations that will be assumed are described in the paragraphs below.

In the general algorithm, the current x-order contains just those segments spanning the current event interval. This implies that one must be able to add and delete segments from this order efficiently. The doubly-linked list model of the current x-order is extended by superimposing a height-balanced tree (assume an AVL tree) upon it. Conversely, one may view the current x-order to be modeled as an AVL tree XOT (X-Ordered Tree) whose symmetric order is explicitly threaded with a doubly-linked list. As before, each cell in the tree points to its corresponding segment record and vice versa. The following primitives are assumed.

Add(e, i)—Add segment e to XOT under the x_i-order.
Delete(e)—Delete segment e from XOT.
Exchange(e)—Exchange e and Above(e) in the current ordering of XOT.
Above(e) (Below(e))—The segment immediately above (below) e in the current ordering of XOT if it exists, Λ otherwise.

The maintenance of the doubly-linked threads in XOT is an elementary exercise. With this model Add and Delete are $O(\log E)$ operations and Exchange, Above, and Below are $O(1)$ as before. Note that Exchange destroys any particular x-order. However, the operation Add(e, i) is only applied when XOT is x_i-ordered.

In the general algorithm there are $EV-1$ distinct work lists, $WORK(i)$, one for each event interval $[x_i, x_{i+1}]$. The collection of work lists is modeled as an array of pointers to the first cell of a doubly-linked work list of the form described in § 3. The following primitives are assumed.

Push(e, i)—Add segment e to $WORK(i)$.
Pop(e, i)—Delete a segment from $WORK(i)$ and return it in e.
Remove(e)—Delete segment e from the work list containing it (if any).

In the previous section it was shown that the primitives in this repertoire can all be done in $O(1)$ time.

The final primitive is assumed to implement the distribution-based search algorithm sketched in § 3.

Hash(e, f)—A function returning the index of the interval in which e and f satisfy the exchange predicate (1.3).

The primitive Hash is only invoked with segments e and f that are known to satisfy the exchange predicate. It follows from the treatment at the end of § 3 that Hash requires $O(1)$ time in the expected case and $O(\log E)$ time in the worst case.

The complete algorithm is presented below as Algorithm 1. This algorithm begins by initializing XOT and every work list to be empty. It then proceeds by processing the event intervals in left to right order. This iteration constitutes the major loop of Algorithm 1 and at the start of its ith iteration it is claimed that

(4.1) XOT contains the segments in $SPAN(i-1)$ in x_i-order.

In each iteration, all the intersections in the current interval are detected and reported. This task is performed in four steps.

(4A) The segments in $BEG(i)$ are added to XOT. Intersections satisfying Condition 1.2 of Lemma 1 are reported. After this step XOT contains the segments in $SPAN(i-1) \cup BEG(i)$ in x_i-order.
(4B) The segments in $VERT(i)$ are checked for intersections satisfying Condition 1.1 of Lemma 1. This step has no net effect on XOT.
(4C) The segments in $END(i)$ are deleted from XOT. After this step XOT contains the segments in $SPAN(i)$ in x_i-order.
(4D) The intersections satisfying the event exchange predicate for the current interval are detected as XOT is bubble sorted into x_{i+1}-order. Note that after this step XOT satisfies Assertion 4.1 for the next iteration.

In Algorithm 1, each of Steps 4A through 4D appear as minor loops. It is claimed that before an iteration of any minor loop, the collection of work lists satisfies the condition:

(4.2) $e \in WORK(j)$ iff
e and Above(e) satisfy the event exchange predicate in the jth event interval and $j \geqq i$ where i is the index of the current interval.

The invariance of this minor loop predicate is maintained throughout the algorithm by pushing and removing work lists elements to correctly reflect the effect of every Add, Delete, and Exchange operation on *XOT*. The critical feature of Assertion 4.2 is that it implies $WORK(i) = \{e \in SPAN(i) \mid e$ and $Above(e)$ are not in x_{i+1}-order$\}$ just before Step 4D is about to be performed for the ith event interval. Thus the work list bubble sort performed in Step 4D is correct as $WORK(i)$ is correctly initialized at its outset.

The specification of Algorithm 1 contains several repeated code fragments that have been collected into *macro* definitions. References to these macros are underlined; their definition follows the algorithm. Keep in mind that macro parameters are passed by substitution.

ALGORITHM 1. The planar segment intersection algorithm

```
/* Initialize work lists and AVL tree */
For i ← 1 to EV Do
  WORK(i) ← ∅
XOT ← ∅

/* For each event x_i in increasing order do */
For i ← 1 to EV Do

(4A) /* Add segments in BEG(i) and list their start point intersections */
     For e ∈ BEG(i) in order Do
       Add(e, i)
       f ← Below(e)
       If f ≠ Λ Then
         Remove(f)
         Enter(f)
       Enter(e)
       Report(e, y_g(x_i) = y_e)

(4B) /* Find all intersections with segments in VERT(i) */
     For e ∈ VERT(i) in order Do
       Add(e, i)
       Report(e, y_g(x_i) ∈ [y_e, ȳ_e])
     For e ∈ VERT(i) in order Do
       Delete(e)

(4C) /* Delete segments in END(i) */
     For e ∈ END(i) in order Do
       f ← Below(e)
       Delete(e)
       Remove(e)
       If f ≠ Λ Then
         Remove(f)
         Enter(f)

(4D) /* Find all "event exchange" intersections in [x_i, x_{i+1}] */
     While WORK(i) ≠ ∅ Do
       Pop(e, i)
       "e and Above(e) intersect"
       f ← Below(e)
```

```
        Exchange(e)
        Remove(Below(e))
        If f ≠ Λ Then
          Remove(f)
          Enter(f)
        Enter(e)

Macro Report(e, cond)
  g ← Below(e)
  While g ≠ Λ and cond Do
    "e and g intersect"
    g ← Below(g)
  g ← Above(e)
  While g ≠ Λ and cond Do
    "e and g intersect"
    g ← Above(g)

Macro Enter(e)
  g ← Above(e)
  If g ≠ Λ and e > min(x̄e, x̄g) g Then
    Push(e, Hash(e, g))
```

Algorithm 1 is correct regardless of the class of intersections present. The correctness and generality of the algorithm follow directly from Lemma 1 and the discussion above. One subtle point: the macro *Report* in Step 4A (4B) correctly reports those segments that intersect with e by Condition 1.2 (1.1) as the segments are contiguous in *XOT*'s order and the intersecting pairs are reported only once as e is being entered into *XOT* for the first and only time. In Theorem 1, the algorithm is shown to have the time and space performance claimed at the outset of the paper.

THEOREM 1. *Algorithm* 1 *requires* $O(E \log E + I)$ *expected time when EVENT is uniformly distributed. Algorithm* 1 *requires* $O(E \log E + I \log E)$ *time in the worst case. Algorithm* 1 *requires* $O(E)$ *space.*

Proof. The initialization of *XOT* and the work lists at the outset of the algorithm require $O(E)$ time. As was shown at the time of their introduction, the construction of all the other auxiliary structures requires $O(E \log E)$ time.

The body of Step 4A is repeated once for each segment in a *BEG*-list. Thus it is repeated at most E times. The macro *Report* contains a **while** loop for which an intersection is reported in each iteration. The body of the while loop takes $O(1)$ time. The other primitives in Step 4A take either $O(1)$ or $O(\log E)$time. Thus the total time taken by Step 4A is $O(E \log E + I_1)$ time in the worst case where I_1 is the number of intersection reported in the step. Similarly, the total times taken by Steps 4B and 4C are $O(E \log E + I_2)$ and $O(E \log E)$ respectively.

The body of Step 4D is repeated once for each intersecting pair satisfying the exchange predicate. All operators are $O(1)$ with the exception of Hash which is $O(1)$ in the expected case and $O(\log E)$ in the worst case. Thus the total cost of performing Step 4D is $O(I_4)$ in the expected case and $O(I_4 \log E)$ in the worst case. The total number of intersections, I, is the sum of I_1, I_2, and I_4. Thus the overall performance of Algorithm 1 is $O(E \log E + I)$ in the expected case and $O(E \log E + I \log E)$ in the worst case.

With the exception of the work lists, it is clear that all the data structures involved require $O(E)$ space. Assertion 4.2 implies that there are at most $E - 1$ segments in all of the work lists at any time. This follows as there are at most $E - 1$ adjacent segments in any ordering and a given segment pair satisfies the exchange predicate in at most one event interval. Thus Algorithm 1 required only $O(E)$ space. □

5. Discussion. The use of the work list bubble sort implies that Algorithm 1 does not report intersections in increasing order of abscissa. However, this is true only within each event interval; the intervals themselves are processed in increasing order. Informally, one may say that the intersections are sorted "with respect to the event intervals". Thus the intersections can be totally ordered by simply sorting each collection of intersections reported in each execution of Step 4D. Although this requires $O(I \log E)$ time in the worst case, in practice it should result in some additional efficiency . Moreover, if appropriate a distribution-based method [6], [7], [10] could be used to solve each sorting subproblem in a total of $O(I)$ expected time.

If all segments are either vertical or horizontal, then Algorithm 1 performs in $O(E \log E + I)$ worst-case time. Simply observe that no pair of segments satisfies the exchange predicate. Consequently, no segment will ever be entered into a work list; the body of Step 4D will never be executed ; and Hash will never be invoked. But the $O(\log E)$ worst-case performance of Hash is solely responsible for the $I \log E$ term in Algorithm 1's performance. This result was first shown in Bentley and Ottmann's paper [2] but was posed as a distinct algorithm. In this paper it is simply a direct consequence of the *general* algorithm.

A slight variation of Algorithm 1 gives an $O(E \log E + I)$ worst-case algorithm for yet another restricted intersection problem. Suppose that all segments are constrained to have their left endpoints at x_1. Formally, assume $BEG(1) \cup VERT(1) = SEGMENT$; the *END* lists are unrestricted. To solve the "single start intersection problem" modify Algorithm 1 as follows. Replace all references to $WORK(i)$ with references to a *single* work list and modify all Pop and Push primitives to operate exclusively on this one work list. The second parameter of these primitives becomes superfluous and consequently the one and only call to Hash is removed. Since the primitive Hash is no longer employed, this modified algorithm must run in $O(E \log E + I)$ time in the worst case. Observe that while *XOT* is no longer reasonably ordered in later iterations of the major loop, the algorithm is correct as Add is not invoked after the first iteration.

In problem instances where the intersection density $\alpha = I/E^2$ is high, the following situation will frequently arise in the course of an event interval bubble sort. An intersecting pair of segments momentarily become adjacent in *XOT* and are entered into some work list only to be removed when another exchange separates them. The effort expended by Hash to find the appropriate work list was wasted. Such redundant searches can be eliminated by introducing a temporary work list, $WORK_T$, which is empty at the beginning of each bubble sort. During a sort, newly adjacent segments that intersect in an event interval other than the current one are placed in $WORK_T$. Only when the given sort is complete are the segments that remain in $WORK_T$ transferred via Hash to their respective work lists. This variation is not asymptotically superior to Algorithm 1 but does significantly reduce the number of searches for high density problems.

6. Conclusion. An $O(E)$ space, $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem has been presented. The key techniques are the use of a work list bubble sort for solving individual event interval problems and the use of a distribution-based search to seed the work lists for these intervals. The expected time result still leaves open the question of whether or not a comparison based method must take $O(E \log E + I \log E)$ worst-case time. However, several restricted problems were observed to have $O(E \log E + I)$ worst-case algorithms.

The problem was treated in full generality. Vertical segments, multi-segment and infinite intersections were all permitted. As Sutherland et al. [19] have observed, these singularities must be carefully treated in order for the algorithm to be useful in graphics applications.

It was noted earlier that a sorted intersection list could be produced in $O(E \log E + I)$ expected time under the assumption that the abscissas of the intersection points are uniformly distributed in each event interval. Such a list readily provides the basis for a hidden-line computation. The method of Sechrest and Greenberg [16] suggests that with the use of coherence all computations can be done in $O(I)$ time except for the embeddings of locally minimum points.

The algorithm described in this paper performs a one-time analysis on a set of planar line segments. In many contexts it would be useful to incrementally obtain a solution. For example if several new edges are added to the problem or if the locations of some of the existing segments are perturbed, the new solution could be computed by simply detecting how it differs from the previous solution. This can obviously be done in $O(E)$ time per input modification by a direct extension of the naive $O(E^2)$ algorithm. The problem of arranging a more efficient incremental algorithm appears very difficult. If I_e is the number of intersection involving segment e, are $O(I_e)$, $O(I_e + \log E)$, or even $O(I_e \log E)$ incremental methods possible? Rosen [14] has noted in the context of data flow analysis that highly efficient one-time algorithms do not necessarily lead to efficient incremental algorithms.

Acknowledgments. The author would like to thank the referees for their careful and detailed reviews which lead to a greatly improved presentation. The author would also like to thank Peter J. Downey, Webb Miller, and Tim A. Budd for their many helpful suggestions.

REFERENCES

[1] S. G. AKL AND H. MEIJER, *On the average-case complexity of "bucketing" algorithms*, J. Algorithms, 3 (1982), pp. 9-13.

[2] J. L. BENTLEY AND T. A. OTTMANN, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Computers, 28 (1979), pp. 643-647.

[3] J. L. BENTLEY AND M. I. SHAMOS, *Divide-and-computer in multidimensional space*, Proc. 8th ACM Symposium on Theory of Computing, 1976, pp. 220-230.

[4] K. Q. BROWN, *Comments on "algorithms for reporting and counting geometric intersections"*, IEEE Trans. Computers, 30 (1981), pp. 147-148.

[5] D. P. DOBKIN AND B. CHAZELLE, *Detection is easier than computation*, Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 146-153.

[6] W. DOBOSIEWICZ, *Sorting by distributive partitioning*, Inform. Proc. Lett., 7 (1978), pp. 1-6.

[7] G. EHRLICH, *Searching and sorting real numbers*, J. Algorithms, 2 (1981), pp. 1-12.

[8] W. R. FRANKLIN, *A linear exact hidden surface algorithm*, ACM Computer Graphics, 14 (1980), pp. 117-123.

[9] L. J. GUIBAS, E. M. MCCREIGHT, M. F. PLASS AND J. R. ROBERTS, *A new representation for linear lists*, Proc. 9th ACM Symposium on Theory of Computing, 1977, pp. 49-60.

[10] H. MEIJER AND S. G. AKL, *The design and analysis of a new hybrid sorting algorithm*, Inform. Proc. Lett., 10 (1980), pp. 213-218.

[11] M. H. OVERMARS, *General methods for 'all elements' and 'all pairs' problems*, Inform. Proc. Lett., 12 (1981), pp. 99-102.

[12] M. H. OVERMARS AND J. VAN LEEUWAN, *Dynamically maintaining configurations in the plane*, Proc. 12th ACM Symposium on theory of Computing, 1980, pp. 135-145.

[13] F. P. PREPARATA, *A new approach to planar point location*, this Journal, 10 (1981), pp. 473-492.

[14] B. K. ROSEN, *Linear cost is sometimes quadratic*, Proc. 8th ACM Conference on Principles of Programming Languages, 1981, pp. 117-124.

[15] A. SCHMITT, *Reporting intersections of line segments: An improvement of the Ottman-Bentley algorithm*, Proc. 8th Conference on Graph Theoretic Concepts in Computer Science, H. J. Schneider and H. Gotter, Eds., Carl Hanser Verlag, Munchen, 1982, pp. 257-266.

[16] S. SECHREST AND D. P. GREENBERG, *A visible polygon reconstruction algorithm*, ACM Trans. Graphics, 1 (1982), pp. 25-42.

[17] M. I. SHAMOS, *Geometric complexity*, Proc. 7th ACM Symposium on Theory of Computing, 1975, pp. 224-233.

[18] M. I. SHAMOS AND D. HOEY, *Geometric intersection problems*, Proc. 17th ACM Symposium on Foundations of Computer Science, 1976, pp. 208-215.

[19] I. E. SUTHERLAND, R. F. SPROULL AND R. S. SCHUMACKER, *A characterization of ten hidden surface algorithms*, Comput. Surveys, 6 (1974), pp. 1-55.

[20] M. TAMMINEN, *Analysis of N-trees*, Inform. Proc. Lett., 16 (1983), pp. 131-137.