Algorithmica (1996) 15: 626–660

Algorithmica

Connected Component and Simple Polygon Intersection Searching[1]

P. K. Agarwal[2] and M. van Kreveld[3]

Abstract. Efficient data structures are given for the following two query problems: (i) preprocess a set $\mathcal{P}$ of simple polygons with a total of n edges, so that all polygons of $\mathcal{P}$ intersected by a query segment can be reported efficiently, and (ii) preprocess a set $\mathcal{S}$ of n segments, so that the connected components of the arrangement of $\mathcal{S}$ intersected by a query segment can be reported quickly. In these problems we do not want to return the polygons or connected components explicitly (i.e., we do not wish to report the segments defining the polygon, or the segments lying in the connected components). Instead, we assume that the polygons (or connected components) are labeled and we just want to report their labels. We present data structures of size $O(n^{1+\varepsilon})$ that can answer a query in time $O(n^{1/2+\varepsilon}+k)$, where k is the output size. If the edges of $\mathcal{P}$ (or the segments in $\mathcal{S}$) are orthogonal, the query time can be improved to $O(\log n+k)$ using $O(n\log n)$ space. We also present data structures that can maintain the connected components as we insert new segments. For arbitrary segments the amortized update and query time are $O(n^{1/2+\varepsilon})$ and $O(n^{1/2+\varepsilon}+k)$, respectively, and the space used by the data structure is $O(n^{1+\varepsilon})$. If we allow $O(n^{4/3+\varepsilon})$ space, the amortized update and query time can be improved to $O(n^{1/3+\varepsilon})$ and $O(n^{1/3+\varepsilon}+k)$, respectively. For orthogonal segments the amortized update and query time are $O(\log^2 n)$ and $O(\log^2 n+k\log n)$, and the space used by the data structure is $O(n\log n)$. Some other related results are also mentioned.

Key Words. Arrangements, Data structures, Intersection searching, Partition trees.

1. Introduction.

The general *intersection-searching* problem involves preprocessing a set of objects into a data structure so that the objects intersected by a query object can be reported efficiently. This problem is quite general, and numerous geometric query-type problems can be formulated in this setting. For example, the widely studied *range searching* problem requires preprocessing a set of points into a data structure so that the points intersecting a query region (rectangle, simplex, etc.) can be reported efficiently. In most of the work the researchers have assumed the input objects to be of simple shape, i.e., with constant description complexity (e.g., points, lines, segments, circles), while in most of the applications they are more complicated and are defined in terms of simple objects, e.g., polygons, defined as a sequence of noncrossing segments. Typically the definition of an object is stored in some data structure. The goal is to return the pointers to the data structures storing the definitions of the objects (and not the definitions themselves) that intersect a query object. For this purpose we can assume that the objects are labeled and one wishes to return the labels of the objects that intersect a query object. In most of

[1] Part of this work was done while the second author was visiting the first author on a grant by the Dutch Organization for Scientific Research (N.W.O.). The research of the second author was also supported by the ESPRIT Basic Research Action No. 3075 (project ALCOM). The research of the first author was supported by National Science Foundation Grant CCR-91-06514.

[2] Computer Science Department, Duke University, P.O. Box 90129, Durham, NC 27708-0129, USA.

[3] Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.

Received August 5, 1994; revised December 15, 1994. Communicated by B. Chazelle.

the applications, the set of simple objects that define the input objects for intersection searching can be processed, but the query time will no longer be output-sensitive; see below for more discussion on this topic.

In this paper we consider some of the problems in which the input objects are not of simple shape. We assume that the objects are defined by segments, and the query object is also a segment. We study the following "abstract" problem, which we call the *colored segment intersection searching* problem:

> Given a collection $\mathcal{S}$ of n segments and an m-coloring $\chi\colon \mathcal{S} \to \{1, 2, \ldots, m\}$ of $\mathcal{S}$, preprocess $\mathcal{S}$ into a data structure, so that the set of colors of those segments in $\mathcal{S}$ that intersect a query segment can be reported efficiently.

By coloring the segments of each object with the same color and the segments of different objects with different colors, we can reduce the original intersection-searching problem to the colored segment intersection searching problem. The following examples illustrate this idea:

1. *Polygon intersection searching.* Let $P_1, \ldots, P_m$ be a set of simple polygons, and let $\mathcal{S}$ be the set of n segments defining these polygons. We want to preprocess the polygons, so that the polygons intersecting a query segment can be reported quickly. We do not want to report all the edges of polygons, just the indices of these polygons. By assigning the color i to the edges of P_i, this problem can be reduced to colored segment intersection searching.
2. *Connected component intersection searching.* The *connected components* in the arrangement, $\mathcal{A}(\mathcal{S})$, of a set $\mathcal{S}$ of segments are the connected components of the planar graph formed by $\mathcal{A}(\mathcal{S})$.[4] More formally, two segments of $\mathcal{S}$ are in the same connected component if there is a path between them along the edges of $\mathcal{A}(\mathcal{S})$. These connected components form a partition $\mathcal{S}^1, \ldots, \mathcal{S}^m$ of $\mathcal{S}$. For the sake of convenience, we refer to $\mathcal{S}^1, \ldots, \mathcal{S}^m$ as the connected components of $\mathcal{A}(\mathcal{S})$. We wish to preprocess $\mathcal{S}$ so that the connected components of $\mathcal{A}(\mathcal{S})$ intersecting a query segment can be reported quickly. Again, for each $\mathcal{S}^i$, we do not want to report all the segments in $\mathcal{S}^i$; we just want to return the index i. By coloring the segments of $\mathcal{S}^i$ with the color i, we can reduce it to colored segment intersection searching. Notice that, unlike the previous problem, determining the color of each segment in $\mathcal{S}$ is not trivial, because it involves finding for each segment $e \in \mathcal{S}$ the connected component of $\mathcal{A}(\mathcal{S})$ in which e lies.

The connected components and their labeling are an important concept in image processing, geographic information systems, etc.; see, e.g., [10]. The *segment intersection-searching problem*, in which one wants to report all segments of $\mathcal{S}$ intersecting a query segment, is a special case of the colored segment intersection searching because, by coloring each segment of $\mathcal{S}$ with a distinct color, we can reduce it to the colored segment intersection-searching problem. Other colored intersection-searching problems have been studied independently by Janardan and Lopez [19], Gupta *et al.* [15]–[17], and Nievergelt and Widmayer [22].

---

[4] The *arrangement* of $\mathcal{S}$ is the planar graph induced by $\mathcal{S}$. The *vertices* of $\mathcal{A}(\mathcal{S})$ are the intersection points of $\mathcal{S}$, the *edges* are maximal portions of segments of $\mathcal{S}$ that do not contain any vertex, and the *faces* are the connected components of $\mathbb{R}^2 - \bigcup \mathcal{S}$.

In the last few years, much work has been done on the segment intersection-searching problem [2], [4], [6], [8], [11], [14], [24]. Agarwal and Sharir [4] showed that S can be preprocessed using $O(n^{1+\varepsilon})$ space and time, so that all k segments of S intersecting a query segment can be reported in time $O(n^{1/2+\varepsilon}+k)$.[5] (The n^{ε} factor in the size and query time can be reduced to the $\log^{O(1)} n$ factor using a more sophisticated partition tree due to Matoušek [20].) Roughly speaking, their data structure stores a family of subsets of S, called *canonical subsets*, so that the segments of S intersecting a query segment can be represented as the union of $O(n^{1/2+\varepsilon})$ pairwise disjoint canonical subsets, and they can be computed in $O(n^{1/2+\varepsilon})$ time. This algorithm can easily be extended to our problem by storing the set of colors for each canonical subset and reporting the colors of the output canonical subsets. The sets of colors in the canonical subsets of the query output are no longer pairwise disjoint, so in the worst case the same color may be reported by all canonical subsets, thereby implying that the query time is $O((1+k)n^{1/2+\varepsilon})$, which is much worse than the desired bound of $O(n^{1/2+\varepsilon}+k)$.

We are not aware of any data structure that gives a faster algorithm for colored segment intersection searching than the one sketched above. In fact, we do not know any algorithm even for connected-component or polygon intersection searching except for a few special cases [18]. The algorithm of Agarwal [1] for computing many faces in an arrangement of segments can be modified to compute its connected components in $O(n^{4/3}\log^2 n)$ time. If the segments are orthogonal, Imai and Asano have presented an $O(n\log n)$-time algorithm for computing the connected components [18]. None of these algorithms works for reporting the connected components that intersect a query segment. If the query object is a line, a connected-component intersection query can be answered as follows (see also [19]): For each connected component of $\mathcal{A}(S)$, find its convex hull and preprocess the convex hull edges for segment intersection searching. Since a connected component of $\mathcal{A}(S)$ intersects a line ℓ if and only if its convex hull intersects ℓ, and at most two edges of a convex hull intersect ℓ, all k connected components of S intersecting a query line can be reported in time $O(n^{1/2+\varepsilon}+k)$. Observe that the above approach works for colored segment intersection searching also (assuming that the query is again a line) as long as the segments of the same color form a connected graph. For example, the polygon intersection queries for lines can be answered using the same approach. This approach does not work if the query object is a segment.

This paper begins with a relatively simple algorithm for colored segment intersection searching for the case in which the segments are orthogonal and the query segment is also orthogonal (Section 2). We show that we can preprocess a set of n orthogonal segments in $O(n\log^2 n)$ time into an $O(n\log n)$ size data structure so that a query can be answered in time $O(\log n+k)$. A similar bound has been attained by Janardan and Lopez [19].

In Section 3 we present a dynamic solution for colored segment intersection searching for the case in which the segments of S and the query segments are orthogonal. The query time is $O((k+1)\log^2 n)$, where k is the number of colors reported. The insertion or deletion of a segment takes $O(\log^2 n)$ amortized time. If only insertions are performed, the amortized query time can be improved to $O(\log^2 n + k\log n)$. An algorithm for dynamic colored segment intersection searching does not immediately yield an

[5] Throughout this paper, ε stands for a positive constant that can be chosen arbitrarily small with an appropriate choice of other constants in the algorithms.

algorithm for connected-component intersection searching because an update may affect several connected components of $\mathcal{S}$. We present an $O(n \log^2 n)$-time on-line algorithm to construct the connected components of n orthogonal segments (Section 4).

Section 5 solves the static version of the connected-component searching problem for arbitrary segments. For any fixed $\varepsilon > 0$, the preprocessing time is $O(n^{4/3+\varepsilon})$, the space is $O(n^{1+\varepsilon})$, and a query takes $O(n^{1/2+\varepsilon} + k)$ time, where k is the output size. The query time can be improved by increasing the size of the data structure, as with the standard segment intersection searching. In particular, for a parameter $n \leq N \leq n^2$, a query can be answered in time $O(n^{1+\varepsilon}/\sqrt{N} + k)$ using $O(N^{1+\varepsilon})$ space. A variant of this algorithm yields an efficient data structure for polygon intersection searching (Section 6).

In Section 7 we describe how the data structure described in Section 5 can be modified so that new segments can be inserted in $O(n^{1/2+\varepsilon})$ amortized time without increasing the asymptotic query time. The insertion of a segment is fairly expensive because, before inserting, we need to determine the connected components that intersect. If we allow the size of the data structure to be $O(n^{4/3+\varepsilon})$, the update and query time can be improved to $O(n^{1/3+\varepsilon})$ and $O(n^{1/3+\varepsilon} + k)$. This yields an $O(n^{4/3+\varepsilon})$-time on-line incremental algorithm for computing the connected components in arrangements of n segments.

The basic idea behind our algorithms is to store families of canonical subsets of $\mathcal{S}$ so that any query selects only a small number of them. Within each canonical subset, we use a data structure that finds any color only a few, a constant number, times. This approach ensures that no color is reported often, and thus the query time will be low. An interesting feature of our dynamic algorithms is the *lazy* update of data structures. Part of the update work is performed later by the subsequent query algorithms; however, a query is answered correctly at all times. The idea is reminiscent of the union-find structure with path compression [9]. A UNION operation performs its task correctly, but makes the structure less efficient. A FIND operation adjusts the structure so that subsequent FIND operations can be performed more efficiently. This is exactly what happens in our solution for the maintenance of connected components.

2. The Orthogonal Colored Segment Intersection Searching. In this section we consider the colored segment intersection-searching problem for orthogonal segments; i.e., given a set $\mathcal{S}$ of orthogonal segments and a color assignment $\chi\colon \mathcal{S} \to \{1, \ldots, m\}$, we preprocess $\mathcal{S}$ into a data structure so that the colors of the segments that intersect a query (orthogonal) segment γ can be reported efficiently. We preprocess horizontal and vertical segments separately. Reporting the colors of horizontal segments that intersect a query horizontal segment is essentially the following one-dimensional problem: preprocess a set of colored intervals into a data structure so that the colors of intervals that intersect a query interval can be reported efficiently. This problem can be solved optimally either by modifying the data structure described below in an obvious manner or by using the algorithm described in [15]. Hence, it suffices to describe how to preprocess a set of horizontal segments so that the colors of segments that intersect a query vertical segment can be reported quickly. The symmetric case in which vertical segments are stored can be solved in the same way and is not discussed.

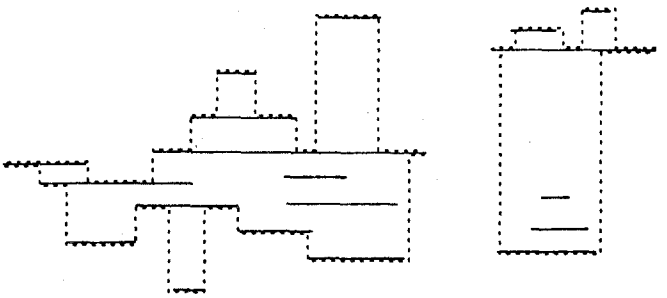

Fig. 1. Upper and lower envelopes of a set of horizontal segments.

For simplicity, we assume that all horizontal segments of $\mathcal{S}$ have different y-coordinates, and that they are sorted in an increasing order of their heights. We construct a balanced binary tree T on the segments of $\mathcal{S}$. The ith leftmost leaf of T is associated with the ith segment of $\mathcal{S}$. For each node v of T, let $\mathcal{S}_v \subseteq \mathcal{S}$ denote the set of segments associated with the leaves of the subtree rooted at v, and let $n_v = |\mathcal{S}_v|$. For an internal node v, let y_v denote the y-coordinate of the highest segment associated with its left child (i.e., the segment associated with the rightmost leaf of the subtree rooted at the left child of v). We associate the horizontal line ℓ_v: $y = y_v$ with v. At each node other than the root of T, we store the secondary structure described below.

DEFINITION 2.1. The *upper envelope* $U = U(\mathcal{S})$ of a set $\mathcal{S}$ of segments is the pointwise maximum when each segment $e \in \mathcal{S}$ is viewed as a partially defined linear function. U is a piecewise-linear (not necessarily continuous) function whose graph consists of portions of the segments of $\mathcal{S}$. If all the segments of $\mathcal{S}$ are horizontal, then U is a *histogram* or *Manhattan skyline* (see Figure 1),[6] and every *breakpoint* of U is an endpoint of some segment of $\mathcal{S}$. The *lower envelope* $L = L(\mathcal{S})$ of $\mathcal{S}$ is defined similarly.

Let $\mathcal{S}_v^i \subseteq \mathcal{S}_v$ be the set of segments of color i, and assume without loss of generality that $\mathcal{S}_v^1, \ldots, \mathcal{S}_v^{m_v}$ are the nonempty subsets of $\mathcal{S}_v$. Clearly, $m_v \leq n_v$. Assume that v is a left child of its parent. For each i, let U_v^i be the upper envelope of $\mathcal{S}_v^i$. The set $U_v = \{U_v^1, \ldots, U_v^{m_v}\}$ is a set of m_v histograms, each of a different color. Let $x_1 < \cdots < x_r$ be the x-coordinates of all breakpoints of the histograms in U_v. We have $r \leq 2n_v$. The vertical ordering of the histograms in U_v changes only at the breakpoints, so it remains the same in every interval (x_j, x_{j+1}). We store the vertical ordering of U_v for all intervals (x_j, x_{j+1}), using the persistent data structure of Sarnak and Tarjan [27]. The preprocessing time and space required are $O((m_v + r)\log(m_v + r))$ and $O(m_v + r)$, respectively. For any vertical query segment e whose topmost endpoint lies above all histograms, we can report all k histograms that intersect e in time $O(\log(m_v + r) + k)$. In

[6] Usually, a histogram or Manhattan skyline includes the vertical segments as well, but for the sake of convenience we omit them.

more detail, we compute the persistent data structure by sweeping a vertical line from left to right. At each breakpoint we stop and update the structure. Assume that the histogram U_v^i has a breakpoint at some x_j. Then we perform the following two operations:

(i) If U_v^i is defined in the interval (x_{j-1}, x_j), we delete it from the structure.
(ii) If U_v^i is defined in the interval (x_j, x_{j+1}), we insert it into the structure using the y-coordinate of U_v^i at (x_j, x_{j+1}) as the key.

Since r and m are at most $2n_v$, the persistent data structure requires $O(n_v)$ space and $O(n_v \log n_v)$ preprocessing time. If v is the right child of its parent, we construct a similar secondary structure on the lower envelopes L_v^i of $\mathcal{S}_v^i$.

Let γ be a vertical query segment. We follow a single path of T starting from the root until we find a node z such that the line ℓ_z intersects γ, as follows. Suppose we are at a node v of T. If v is a leaf and γ intersects the segment associated with v, we report its color. If v is an internal node with w and u being its left and right children, respectively, we do the following: If γ lies above the line ℓ_v, it cannot intersect the segments in $\mathcal{S}_w$, so we descend to u and repeat the same step. Similarly, if γ lies below ℓ_v, we descend to w and repeat the same step. Finally, if γ intersects ℓ_v, we visit the secondary structures of the children w and u. Since γ intersects ℓ_v, γ intersects a segment of $\mathcal{S}_w^i$ (or $\mathcal{S}_u^i$) if and only if γ intersects U_w^i (resp. L_u^i). Therefore, we search the persistent data structures stored at w and u with γ, and report the upper and lower envelopes intersected by γ; this in turn gives the colors of the segments in $\mathcal{S}_v = \mathcal{S}_w \cup \mathcal{S}_u$ intersected by γ.

As for the query time, we spend $O(\log n)$ time in finding the highest node v for which ℓ_v intersects γ and $O(\log n + t)$ time in searching through the persistent data structures, where t is the number of envelopes in these structures that intersect γ. There are at most two envelopes of the same color, so the overall query time is $O(\log n + k)$, where k is the number of colors of segments in $\mathcal{S}$ that intersect γ.

A similar data structure can report the colors of vertical segments intersected by a horizontal segment. We therefore conclude

THEOREM 2.2. *Given a set $\mathcal{S}$ of n orthogonal line segments in the plane along with a color assignment $\chi\colon \mathcal{S} \to \{1, \ldots, m\}$, $\mathcal{S}$ can be preprocessed in time $O(n \log^2 n)$ into a data structure of size $O(n \log n)$, so that all k colors of segments of $\mathcal{S}$ intersecting a given orthogonal query segment can be reported in $O(\log n + k)$ time.*

The above theorem implies that, by first computing the connected components of $\mathcal{S}$ in time $O(n \log n)$ [18] and then coloring the segments of each connected component with a distinct color, we can preprocess $\mathcal{S}$ for connected component intersection searching. We thus have

COROLLARY 2.3. *The connected components of a set $\mathcal{S}$ of n orthogonal segments in the plane can be preprocessed in time $O(n \log^2 n)$ into a data structure of size $O(n \log n)$, so that all k connected components of $\mathcal{S}$ intersecting an orthogonal query segment can be reported in $O(\log n + k)$ time.*

Similarly, we also obtain

COROLLARY 2.4. *A set of simple rectilinear polygons, consisting of n line segments in total, can be preprocessed in time $O(n \log n)$ into a data structure of size $O(n \log^2 n)$ so that all k polygons intersecting an orthogonal query segment can be found in $O(\log n + k)$ time.*

3. Dynamic Orthogonal Colored Segment Intersection Searching. In this section we describe another data structure that maintains a collection $\mathcal{S}$ of orthogonal segments dynamically and supports the following operations:

INSERT $(\mathcal{S}, e, i)$: Insert the segment e of color i to the set $\mathcal{S}$.
DELETE $(\mathcal{S}, e, i)$: Delete the segment e of color i from $\mathcal{S}$.
REPORT $(\mathcal{S}, \gamma)$: Report the colors of segments of $\mathcal{S}$ that intersect a query (orthogonal) segment γ.

As mentioned before, the data structure of Theorem 2.2 does not allow efficient updates, so we have to use a different approach. As in the static case, we consider the situation in which $\mathcal{S}$ consists of only horizontal segments (i.e., the segment to be inserted or deleted is horizontal), and the query segment is vertical. The other case is completely symmetric, and we use a separate structure for it.

We store $\mathcal{S}$ in a two-level data structure—the primary structure is a balanced binary tree and the secondary structure is an interval tree. The secondary structure answers the one-dimensional version of the colored segment intersection queries. That is, it maintains a collection of colored intervals so that the intervals containing a query point can be reported efficiently.

We first describe the data structure for maintaining intervals and then present the overall data structure.

3.1. *Dynamic Colored Interval Intersection.* We wish to store a collection $\mathcal{B}$ of intervals into a data structure so that an interval can be inserted into or deleted from the structure, and the colors of intervals in $\mathcal{B}$ containing a query point $x \in \mathbb{R}$ can be reported quickly.

We maintain a dynamic interval tree T that supports insert and delete operations, see, e.g., pp. 192–199 of [21]. A real number x_v is associated with each internal node of T. An interval $b \in \mathcal{B}$ is stored at the highest node of T for which $x_v \in b$. Let $\mathcal{B}_v \subseteq \mathcal{B}$ denote the set of intervals stored at v, and let C_v be the set of colors of intervals in $\mathcal{B}_v$. We maintain three secondary structures on $\mathcal{B}_v$:

(i) TCL_v: It stores C_v as a balanced binary search tree (e.g., red–black tree). Suppose a node v of TCL_v stores color i. Let L_v^i and R_v^i be the list of left and right endpoints, respectively, of the intervals in $\mathcal{B}_v$ of color i. We store L_v^i, R_v^i at v as balanced binary search trees. Let l_i be the smallest element in L_v^i, and r_i be the largest element in R_v^i.
(ii) TL_v: It stores the set $\{l_i \mid i \in C_v\}$ as a balanced binary search tree.
(iii) TR_v: It stores the set $\{r_i \mid i \in C_v\}$ as a balanced binary search tree.

We perform three operations on the interval tree:

(i) INSERT (T, b, i),
(ii) DELETE (T, b, i), and
(iii) REPORT (T, x).

The third operation reports the colors of intervals that contain a given point x. The structures TL_v and TR_v are used for the REPORT-operation, and the structure TCL_v is needed for the DELETE-operation. Intuitively, to answer a query we are only interested in determining whether an interval of color i contains x. Therefore, the structures TL_v and TR_v store the intervals of each color that are "most likely" to be an answer to the query.

INSERTING AN INTERVAL. To insert a new interval $b = [l, r]$ of color i into T, we first add b to T using the standard procedure, see [21] for details. Let $v \in T$ be the node at which b is stored. First, we insert its color i into TCL_v, using the standard insertion procedure for balanced binary search trees. Then we insert the endpoints of b into the lists L_v^i and R_v^i. Finally, if l is the leftmost endpoint of the intervals in $\mathcal{B}_v$ of color i, we delete the current l_i from TL_v and insert l into TL_v. We do the same for r if r becomes the new r_i.

Since the secondary data structures of two nodes can be merged in linear time and an interval can be inserted into a secondary structure in $O(\log n)$ time, the total amortized time required for inserting b is $O(\log n)$ (see [21]).

DELETING AN INTERVAL. To delete an interval $b = [l, r]$ of color i, stored at a node v of T, we delete the endpoints of b from L_v^i, R_v^i and delete b from T. If $l_i = l$, we delete l from TL_v and add the new l_i (the smallest element of L_v^i after deleting l) into TL_v. We do the same for r if $r_i = r$. The total time spent is $O(\log n)$.

ANSWERING A QUERY. To report the colors of intervals intersected by a query point x, we follow a path in T starting at the root. At each node v we do the following. Suppose $x > x_v$. Then an interval $b = [l, r] \in \mathcal{B}_v$ intersects x if and only if $r \geq x$. So, we traverse TR_v from right to left until we encounter an endpoint $p < x$. We report the colors of intervals corresponding to the endpoints traversed. We then descend to the right child of v. The case $x \leq x_v$ is analogous.

We visit $O(\log n)$ nodes of T, and a color is reported at most once at each node. Since a binary search tree supports the max-operation in constant time, hence, the overall query time to report k colors of intervals intersecting a query point x is $O((k+1)\log n)$. Hence, we have

LEMMA 3.1. *We can maintain a set of colored intervals in a data structure of linear size so that an interval can be inserted into or deleted from the structure in $O(\log n)$ amortized time, and so that all k colors of intervals containing a query point can be reported in time $O((k+1)\log n)$.*

SEMIDYNAMIC CASE. If we perform only insert operations, the secondary data structure stored at each node v can be simplified, and the amortized query time can be improved to $O(\log n + k)$ (though a specific query may take much longer). In this case we do not need the secondary structure TCL_v, since it was needed only to find a new leftmost or rightmost interval when an interval was deleted from L_v or R_v. A second change is the following. For each color i, let U^i denote the union of the intervals in $\mathcal{B}$ of color i. Instead of storing the intervals of $\mathcal{B}$, we now store another collection $\mathcal{E}$ of (colored) intervals so that for each color i, the union of intervals in $\mathcal{E}$ of color i is exactly U^i. This ensures that an interval in $\mathcal{B}$ of color i intersects a point x if and only if an interval in $\mathcal{E}$ of color i intersects that point. Moreover, we no longer require that all endpoints in L_v, R_v have distinct colors. (We point out that we neither attempt to store a minimum number of intervals, nor require the intervals of the same color to be disjoint.) As usual, an interval $b \in \mathcal{E}$ is associated with the highest node v such that $x_v \in b$. Let $\mathcal{E}_v$ be the set of intervals associated with v. At each node v, we maintain two binary search trees L_v and R_v storing the left and right endpoints, respectively, of intervals in $\mathcal{E}_v$.

Inserting an interval b is straightforward. We first insert b into the primary tree T as earlier. If b is stored at a node v, we add the left and right endpoints of b to L_v and R_v. The total time spent is obviously still $O(\log n)$.

The colors of intervals containing a query point x are retrieved in the same way as earlier except that we perform an additional step. If we find $t_i > 1$ intervals $b_1, \ldots, b_{t_i}$ of the same color, say i, we delete them from the corresponding secondary structures. Let $b = \bigcup_{j=1}^{t_i} b_j$; b is a single connected interval because $x \in b_j$ for all $j \leq t_i$. We insert b into T. The total time spent in processing $b_1, \ldots, b_{t_i}$ is $O((t_i + 1)\log n)$. If the above procedure returns $k' = \sum t_i$ intervals of k distinct colors, i.e., $k' = k + \sum(t_i - 1)$, then the actual running time for reporting and updating is

$$\begin{aligned} &O(k' + \log n) + \sum_{i:\, t_i > 1} O((t_i + 1)\log n) \\ &\quad = O\left(\log n + k + \sum_{i:\, t_i \geq 1} (t_i - 1)\right) + \sum_{i\, t_i > 1} O(t_i \log n) \\ &\quad = O(\log n + k) + \sum_{i:\, t_i \geq 1} O((t_i - 1)\log n). \end{aligned}$$

$\sum_{i\, t_i \geq 1}(t_i - 1)$ elements are deleted, and any interval can be deleted at most once. We charge the $O(\log n)$ time to each such interval to account for the second term in the time complexity. Hence, the amortized insert and query time are $O(\log n)$ and $O(\log n + k)$, respectively.

LEMMA 3.2. *We can maintain a collection of colored intervals in a data structure of linear size so that a new interval can be inserted in $O(\log n)$ amortized time, and so that all k colors of intervals containing a query point can be reported in $O(\log n + k)$ amortized time.*

3.2. *Two-Dimensional Structure.* To obtain a dynamic data structure for the two-dimensional colored segment intersection problem, we apply a so-called *range restriction*; see, e.g., [23], [28], and [31]. We maintain a balanced binary tree on the

y-coordinates of the segments, and every node of the tree stores a data structure as described above. The effect of the performance is a multiplicative factor of $\log n$ in the update and query time and the space requirements. Hence, we obtain

THEOREM 3.3. *We can maintain a set $\mathcal{S}$ of colored orthogonal segments in a data structure of size $O(n \log n)$ so that a segment can be deleted from or inserted into the structure in $O(\log^2 n)$ amortized time, and so that all k colors of segments intersecting an orthogonal query segment can be reported in time $O((k+1)\log^2 n)$. If we allow only insertions, the amortized query time can be improved to $O(\log^2 n + k \log n)$.*

4. Maintaining Connected Components of Orthogonal Segments. The data structure as described above cannot be used directly for maintaining the connected components of $\mathcal{A}(\mathcal{S})$ because insertion of a segment can merge several connected components into one. Therefore, if we label the segments in the ith connected component by color i, then the insertion of a segment can change the colors of many segments. It is very expensive to update the colors of all these segments explicitly. The data structure for maintaining the connected components should support the following operations:

INSERT $(\mathcal{S}, e)$: Insert a new segment e to the set $\mathcal{S}$.
REPORT-COMPONENT $(\mathcal{S}, \gamma)$: Report the connected components of $\mathcal{S}$ that intersect the query segment γ.
SAME-COMPONENT (e_1, e_2): Determine whether two segments $e_1, e_2 \in \mathcal{S}$ lie in the same connected component of the arrangement of $\mathcal{S}$.

We assume that the color of a segment is i if it is in the ith connected component of the arrangement. Let $\mathcal{S}^i \subseteq \mathcal{S}$ denote the set of segments in the ith connected component of $\mathcal{S}$ (or the segments of color i). The connected components now change dynamically as we insert segments. Since it is very expensive to change the color of every segment explicitly, we do it implicitly. In particular, we maintain a union-find data structure $UF(\mathcal{S})$ to update the colors of segments. It can merge two sets in $O(\log n)$ amortized time, and can report the connected component containing a given segment (or the color of a given segment) in $O(1)$ time, see [9]. Throughout this section we use the structure $UF(\mathcal{S})$ to find the color of a segment, and by the phrase "find the color of a segment e" we mean "perform FIND (e) using $UF(\mathcal{S})$." Using the structure $UF(\mathcal{S})$, the SAME-COMPONENT query can easily be answered in $O(1)$ time.

Apart from $UF(\mathcal{S})$, we preprocess $\mathcal{S}$ into a data structure described for the semidynamic case in Sections 3.1 and 3.2. A query is answered in the same way as earlier except that we use $UF(\mathcal{S})$ to find the color of an interval. To insert a new segment e, we find the colors of all segments in $\mathcal{S}$ that intersect e, merge these connected components using $UF(\mathcal{S})$, and then insert e as in Section 3.1. The same analysis as above shows that the amortized query time is $O(\log^2 n + k \log n)$, where k is the number of distinct colors reported. As for the insertion time, if e intersects t components of $\mathcal{A}(\mathcal{S})$, then we spend $O(\log^2 n + t \log n)$ amortized time to find these components and another $O(\log^2 n)$ time to insert e. Notice that after inserting e, the number of connected component reduces by $t - 1$, so the amortized insert time can be shown to be $O(\log^2 n)$. Hence, we have

THEOREM 4.1. *We can store a collection $\mathcal{S}$ of n orthogonal segments into a data structure so that a new segment can be inserted in $O(\log^2 n)$ amortized time and so that the set of connected components intersecting a query segment can be reported in $O(\log^2 n + k \log n)$ amortized time. Given two segments in $\mathcal{S}$, we can determine in $O(1)$ time whether they are in the same connected component of the arrangement of $\mathcal{S}$.*

The above theorem immediately gives an on-line algorithm to compute the connected components of a set of n orthogonal segments. An optimal $O(n \log n)$-time solution to the off-line problem was given by Imai and Asano [18].

COROLLARY 4.2. *An on-line algorithm exists that computes the connected components of n orthogonal segments in $O(n \log^2 n)$ time.*

5. Reporting Connected Components in Segment Arrangements. In this section we consider the problem of preprocessing a set $\mathcal{S} = \{e_1, \ldots, e_n\}$ of n segments with arbitrary orientations, so that the connected components of $\mathcal{A}(\mathcal{S})$ intersected by a query segment can be reported quickly. Recall that each connected component of $\mathcal{A}(\mathcal{S})$ is labeled, and the goal is to report these labels (not the segments in those connected components). In other words, let $\mathcal{S}^1, \ldots, \mathcal{S}^m$ be the partition of $\mathcal{S}$ induced by the connected components of $\mathcal{A}(\mathcal{S})$. Set $\chi(\mathcal{S}^i) = i$. We wish to preprocess $\mathcal{S}$ for answering colored segment intersection queries. However, unlike the colored segment intersection searching problem, in which the color of each segment is given, we have to compute the colors of segments in $\mathcal{S}$.

Let $G(\mathcal{S})$ be a graph with n vertices $\{1, \ldots, n\}$ with (i, j) being connected by an edge if there is a nonconvex face f of $\mathcal{A}(\mathcal{S})$ (a face that contains an endpoint of some segment in $\mathcal{S}$) such that both e_i, e_j appear on the same connected component of ∂f. It can be shown that two segments $e_i, e_j \in \mathcal{S}$ are in the same connected component of $\mathcal{A}(\mathcal{S})$ if and only if i and j are in the same connected component of $G(\mathcal{S})$ [13]. We compute, in time $O(n^{4/3} \log^2 n)$, the faces of $\mathcal{A}(\mathcal{S})$ that contain an endpoint of some segment in $\mathcal{S}$ [1]. By a result of Aronov *et al.* [5], the total number of edges in these faces is $O(n^{4/3})$. We index the connected components of $\mathcal{A}(\mathcal{S})$ arbitrarily. Next, we construct $G(\mathcal{S})$ in additional $O(n^{4/3})$ time and compute the colors of all segments in $\mathcal{S}$. After having computed the colors of segments, we preprocess $\mathcal{S}$ as follows.

We construct a segment tree T on $\mathcal{S}$; see [26] for details on segment trees. Each node u of T is associated with an x-interval b_u and a vertical strip $I_u = b_u \times [-\infty, +\infty]$. The left bounding line of I_u is denoted by L_u, and the right bounding line by R_u. A segment $e \in \mathcal{S}$ is associated with a node u if $b_u \subseteq \bar{e}$ and $b_{\text{parent}(u)} \not\subseteq \bar{e}$, where $\bar{e}$ is the x-projection of e. For a node u, let $\mathcal{S}_u$ denote the set of segments associated with u, and let $\mathcal{E}_u$ denote the set of segments associated with the proper descendants of u; set $n_u = |\mathcal{S}_u|$ and $m_u = |\mathcal{E}_u|$. The segments of $\mathcal{S}_u, \mathcal{E}_u$ are referred to as *long* and *short* segments, respectively. We have

$$(1) \qquad \sum_{u \in T} n_u = O(n \log n), \qquad \sum_{u \in T} m_u = O(n \log n).$$

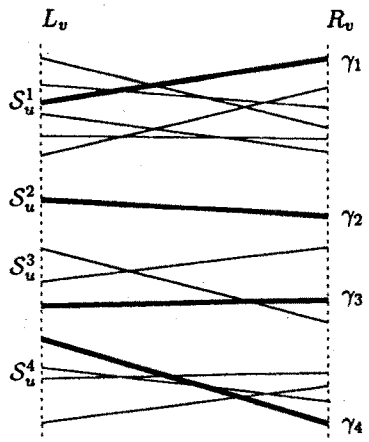

Fig. 2. The long segments $\mathcal{S}_u$, bold edges denote the segments of $\mathcal{R}_u$.

We clip the segments of $\mathcal{S}_u$ and $\mathcal{E}_u$ within I_u. We store two secondary data structures at each node u of T:

(i) We preprocess $\mathcal{S}_u$ so that the colors of all segments of $\mathcal{S}_u$ intersected by a query *segment* contained in I_u can be reported quickly.
(ii) We preprocess $\mathcal{E}_u$ so that the colors of all segments of $\mathcal{E}_u$ intersected by a query *line* can be reported quickly.

5.1. *Preprocessing Long Segments.* Let u be a node of T, and let $\mathcal{S}_u^1, \mathcal{S}_u^2, \ldots, \mathcal{S}_u^t$ be the connected components of $\mathcal{A}(\mathcal{S}_u)$. The segments of the same color (i.e., the segments of the same connected component in $\mathcal{A}(\mathcal{S})$) may split into several connected components of $\mathcal{A}(\mathcal{S}_u)$; see Figure 2. For each $\mathcal{S}_u^i$, we choose an arbitrary segment $r_u^i \in \mathcal{S}_u^i$ as a *representative* of $\mathcal{S}_u^i$. Let $\mathcal{R}_u = \{r_u^1, \ldots, r_u^t\}$, sorted in decreasing order of their heights. The segments of $\mathcal{R}_u$ are pairwise disjoint and partition I_u into trapezoids. The secondary structure, TA_u, associated with $\mathcal{S}_u$ is a minimum height binary tree on the segments of $\mathcal{R}_u$; the ith leftmost leaf of TA_u stores r_u^i. For each node $v \in TA_u$, let $\mathcal{R}_{uv} \subseteq \mathcal{R}_u$ denote the set of segments stored at the leaves of the subtree rooted at v. We store the set of colors of segments of $\mathcal{R}_{uv}$ in a linked list TAP_{uv}. At the ith leftmost leaf of the binary tree, we also preprocess the lines containing the segments of $\mathcal{S}_u^i$ into a linear-size data structure for answering segment intersection detection queries: Let $\mathcal{S}_u^{i*}$ be the set of points dual to the lines supporting the segments of $\mathcal{S}_u^i$. We construct in time $O(|\mathcal{S}_u^i| \log |\mathcal{S}_u^i|)$ a linear-size partition tree $TAQ(\mathcal{S}_u^i)$ that determines whether a query double-wedge W contains any point of $\mathcal{S}_u^{i*}$; see [20]. If $W \cap \mathcal{S}_u^{i*} \neq \emptyset$, then it also returns a point of $\mathcal{S}_u^{i*}$ lying inside W as a witness. The total time spent in preprocessing $\mathcal{S}_u$ is $O(n_u \log n_u)$.

Finally, we preprocess $\mathcal{S}_u^*$, the set of points dual to the lines supporting $\mathcal{S}_u$ in time $O(n_u^{4/3+\varepsilon})$ into a data structure TB_u of size $O(n_u^{4/3+\varepsilon})$ that determines whether a double-wedge contains any of the input points, and returns a witness if the answer is "yes." The query time of this structure is $O(n^{1/3})$, see [20]. This data structure is required only

to preprocess short segments stored in the subtree rooted at u. Once they have been preprocessed, TB_u is discarded.

5.2. *Preprocessing Short Segments.* Next, we describe how to preprocess $\mathcal{E}_u$ so that the colors of $\mathcal{E}_u$ intersected by a query line can be reported quickly. Let $\mathcal{E}_u^1, \ldots, \mathcal{E}_u^t$ be the connected components of $\mathcal{A}(\mathcal{E}_u)$. A line ℓ intersects $\mathcal{E}_u^i$ if and only if it intersects an edge of the convex hull of $\mathcal{E}_u^i$. Moreover, ℓ intersects at most two edges of the convex hull of $\mathcal{E}_u^i$. This suggests the following approach for preprocessing $\mathcal{E}_u$: Compute the convex hull of each connected component $\mathcal{E}_u^i$, preprocess the edges of these convex hulls for answering *line intersection* queries (i.e., preprocess a set of edges into a data structure so that all edges intersected by a query line can be reported efficiently; see [4]), and report a color if ℓ intersects a convex hull edge of a connected component of that color. The problem with this approach is that several connected components in $\mathcal{E}_u$ may have the same color in which case a color will be reported several times. Nevertheless, if $\chi(\mathcal{E}_u^i) = \chi(\mathcal{E}_u^j)$, then there must be a segment in $\mathcal{S}$ (not necessarily a segment of $\mathcal{E}_u^i$) that intersects $\mathcal{E}_u^i$ and the boundary of I_u, and the same holds for $\mathcal{E}_u^j$ (see Figure 3). If both $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ intersect the half-plane lying above (or below) ℓ, then ℓ intersects a segment of $\mathcal{E}_u^i \cup \mathcal{E}_u^j$ if and only if ℓ intersects an edge of $CH(\mathcal{E}_u^i \cup \mathcal{E}_u^j)$ (Figure 3). This is the basic idea of our data structure, which we now describe in detail.

We partition the connected components of $\mathcal{A}(\mathcal{E}_u)$ into three subsets:

- $\mathcal{F}_u^L$: A connected component $\mathcal{E}_u^i \in \mathcal{F}_u^L$ if at least one of the following two conditions hold:
  1. A segment $e_i \in \mathcal{E}_u^i$ intersects the left boundary L_u of I_u.
  2. No segment of $\mathcal{E}_u^i$ intersects the right boundary of I_u, and there is a segment $e \in \mathcal{S}$ that intersects $\mathcal{E}_u^i$ and L_u.

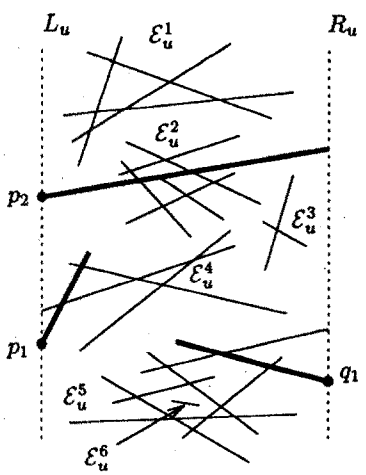

Fig. 3. Short segments; $\mathcal{F}_u^L = \{\mathcal{E}_u^2, \mathcal{E}_u^4\}$, $\mathcal{F}_u^R = \{\mathcal{E}_u^5\}$, and $\mathcal{F}_u^M = \{\mathcal{E}_u^1, \mathcal{E}_u^3, \mathcal{E}_u^6\}$; bold segments are leaders; $\lambda(\mathcal{E}_u^2) \notin \mathcal{E}_u$.

It is easily seen that in the second case, $e \in \mathcal{S}_z$ for some ancestor z of u. We refer to e_i as the *leader* of $\mathcal{E}_u^i$ and denote it by $\lambda(\mathcal{E}_u^i)$. If $e_i \notin \mathcal{E}_u^i$, we clip it within I_u and add it to $\mathcal{E}_u^i$ (and also to $\mathcal{E}_u$), so assume that $\lambda(\mathcal{E}_u^i)$ belongs to $\mathcal{E}_u^i$. If two connected components of $\mathcal{A}(\mathcal{E}_u)$ have the same leader e_i, we merge them into a single component.

- $\mathcal{F}_u^R$: A connected component $\mathcal{E}_u^i \in \mathcal{F}_u^R$ if $\mathcal{E}_u^i \notin \mathcal{F}_u^L$, and if $\mathcal{E}_u^i$ contains a segment e_i that intersects the right boundary R_u of I_u. We refer to e_i as the *leader* of $\mathcal{E}_u^i$ and denote it by $\lambda(\mathcal{E}_u^i)$.
- $\mathcal{F}_u^M$: All the remaining connected components are in $\mathcal{F}_u^M$. The connected components of $\mathcal{A}(\mathcal{F}_u^M)$ are also the connected components of the overall arrangement $\mathcal{A}(\mathcal{S})$, and they lie completely in the interior of I_u. Therefore, the colors of all connected components in $\mathcal{F}_u^M$ are distinct.

The connected components of $\mathcal{E}_u$ can be computed in $O(m_u^{4/3} \log^2 m_u)$ time as mentioned above. If a segment e of $\mathcal{E}_u^i$ intersects the left (resp. right) boundary of I_u, we assign it to $\mathcal{F}_u^L$ (resp. $\mathcal{F}_u^R$), and set $\lambda(\mathcal{E}_u^i) = e$. Let $\mathcal{E}_u^j$ be a component that has not been assigned to any of $\mathcal{F}_u^L$, $\mathcal{F}_u^R$. For every ancestor z of u, we query TB_z with the segments of $\mathcal{E}_u^j$ until we find a segment g of $\mathcal{S}_z$, if any, that intersects $\mathcal{E}_u^j$. If g is found, we clip g within I_u, add a copy of it to $\mathcal{E}_u^j$, assign $\mathcal{E}_u^j$ to $\mathcal{F}_u^L$, and set $\lambda(\mathcal{E}_u^j) = g$; otherwise, we add $\mathcal{E}_u^j$ to $\mathcal{F}_u^M$. The total time spent in this step at u is $O(m_u n^{1/3} \log n)$, and $O(n^{4/3} \log^2 n)$ over all nodes of T.

We process each of $\mathcal{F}_u^L$, $\mathcal{F}_u^R$, $\mathcal{F}_u^M$ into a data structure so that the colors of those segments that intersect a query line can be reported efficiently.

- TC_u: Let p_i be the intersection point of $\lambda(E_u^i)$ and the left boundary of I_u. Let

$$\mathcal{P}_u^L = \{p_i \mid \mathcal{E}_u^i \in \mathcal{F}_u^L\}.$$

Assume that the points in $\mathcal{P}_u^L$ are sorted by their y-coordinates. TC_u is a balanced binary tree whose leaves store $\mathcal{P}_u^i$ in the sorted order. For a node $v \in TC_u$, let $\mathcal{F}_{uv}^L \subseteq \mathcal{F}_u^L$ be the set of connected components $\mathcal{E}_u^j$ such that p_j is stored at a leaf of the subtree rooted at v. For a color c, let $\mathcal{B}_{uv}^c$ denote the set of segments in $\mathcal{F}_{uv}^L$ of color c, defined as

$$\mathcal{B}_{uv}^c = \bigcup \{\mathcal{E}_u^i \mid \mathcal{E}_u^i \in \mathcal{F}_{uv}^L \text{ and } \chi(\mathcal{E}_u^i) = c\}.$$

We compute the convex hull of $\mathcal{B}_{uv}^c$ for each c with $\mathcal{B}_{uv}^c \neq \emptyset$. Let E_{uv}^c denote the set of edges in $CH(\mathcal{B}_{uv}^c)$; see Figure 4. Set $\chi(E_{uv}^c) = c$, $E_{uv} = \bigcup_c E_{uv}^c$, and $m_{uv} = |E_{uv}|$. We store a third-level data structure TCP_{uv} at each node v of TC_u.

- ⋆ TCP_{uv}: We preprocess E_{uv} into a data structure TCP_{uv} of size $O(m_{uv} \log m_{uv})$ for line intersection queries using the algorithm described in [4]. All k segments of E_{uv} intersected by a query line can be reported in time $O(m_{uv}^{1/2+\varepsilon} + k)$. The algorithm constructs a two-level partition tree, each of whose node stores a subset of segments, and the query output consists of $O(m_{uv}^{1/2+\varepsilon})$ canonical subsets. For each canonical subset, we store the set of colors of these segments instead of the segments themselves.

Summing the size of TCP_{uv} of over all nodes of TC_u, the total size of TC_u is $O(m_u \log^2 m_u)$.

- TD_u: Analogous to the previous structure, but for the components of $\mathcal{F}_u^R$.
- TE_u: For each connected component $\mathcal{E}_u^i \in \mathcal{F}_u^M$, we compute the convex hull of its segments. Let E_u^i denote the set of edges in the resulting convex hull, and let

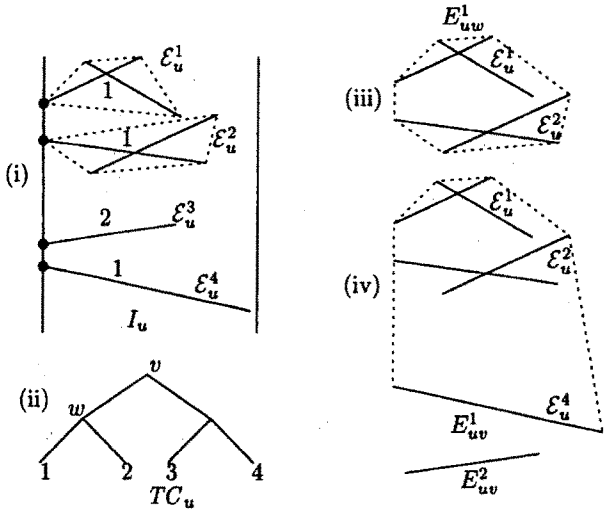

Fig. 4. (i) Short segments at a node u. There are four components in $\mathcal{A}(\mathcal{E}_u)$ and $\chi(\mathcal{E}_u^1) = \chi(\mathcal{E}_u^2) = \chi(\mathcal{E}_u^4) = 1$, $\chi(\mathcal{E}_u^3) = 2$. (ii) The binary tree on $\{p_1, p_2, p_3, p_4\}$, (iii) segments in E_{uw}^1, and (iv) segments in E_{uv}^1, E_{uv}^2.

$E_u = \bigcup_{\mathcal{E}_u^i \in \mathcal{F}_u^M} E_u^i$. We set $\chi(E_u^i) = \chi(\mathcal{E}_u^i)$. We preprocess E_u into a data structure TE_u for line intersection queries, as we preprocessed E_{uv} in TCP_{uv}.

This completes the description of our data structure. Summing over all secondary structures, the total size of the data structure is $O(n \log^3 n)$ and the preprocessing time is $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. Recall that after processing short segments, we discard the secondary structure TB_u, for all $u \in T$.

5.3. *Answering a Query.* Let $\gamma = \overline{pq}$ be the query segment. Without loss of generality assume that p is the left endpoint of γ. Let z be the leaf of T such that $p \in I_z$, and let π_p denote the path from the root of T to z. Similarly, we define the path π_q for q. Let $U(\gamma)$ denote the set of highest nodes u such that $b_u \subseteq \bar{\gamma}$, where $\bar{\gamma}$ is the x-projection of γ. Let ℓ be the line supporting γ. The following lemma is well known:

LEMMA 5.1. *A segment $\gamma = \overline{pq}$ intersects a segment e of $\mathcal{S}$ if and only if there is a node $u \in T$ such that*

1. *$u \in \pi_p \cup \pi_q$, e is stored as a long segment at u, and $\gamma \cap I_u$ intersects the line containing e.*
2. *$u \in U(\gamma)$, e is stored as a short segment at u, and ℓ intersects $e \cap I_u$.*

In view of this lemma, we answer a query in two steps. We first query TA_u, for all nodes $u \in \pi_p \cup \pi_q$, with $\gamma \cap I_u$, and then we query TC_u, TD_u, and TE_u, for all $u \in U(\gamma)$, with ℓ.

SEARCHING AMONG LONG SEGMENTS. Let u be a node on $\pi_p \cup \pi_q$, and let $\hat{\gamma} = \gamma \cap I_u$. Let $\hat{p}, \hat{q}$ be the endpoints of $\hat{\gamma}$. Assume without loss of generality that $\hat{p}$ lies below $\hat{q}$. We query TA_u with $\hat{\gamma}$ and report the colors of the segments in $\mathcal{S}_u$ intersected by $\hat{\gamma}$ as

follows. Searching TA_u with the endpoints $\hat{p}$ and $\hat{q}$ of $\hat{\gamma}$, we compute $O(\log n)$ maximal subtrees of TA_u that lie between the search paths to $\hat{p}$ and $\hat{q}$. At the root v of each such maximal subtree, we report all colors stored in the associated lists TAP_{uv}. Let r_u^i be the segment in $\mathcal{R}_u$ lying immediately below $\hat{p}$. By querying $TAQ(\mathcal{S}_u^i)$ with $\hat{\gamma}^*$, the double-wedge dual to $\hat{\gamma}$, we determine whether $\hat{\gamma}$ intersects any line supporting the segments of $\mathcal{S}_u^i$. If the answer is "yes," we also report the color of r_u^i. Next, we repeat the same procedure for the representative lying immediately above $\hat{q}$. At each node $v \in TA_u$, a color is reported only once, so the total time spent in searching over all nodes in $\pi(p)$ and $\pi(q)$ is $O(n^{1/2+\varepsilon} + k_l \log^2 n)$, where k_l is the number of distinct colors reported.

SEARCHING AMONG SHORT SEGMENTS. Next, let u be a node in $U(\gamma)$. We report the colors of the segments of $\mathcal{E}_u$ intersected by the line ℓ containing γ. Let σ be the intersection point of ℓ and the left boundary of I_u. We search TC_u with σ and determine the leaf z that stores the point lying immediately above σ. The subsets $\mathcal{F}_{uv}^1$, associated with the children v of the nodes on the path from the root of TC_u to z, which do not lie on the path itself, partition the connected components of $\mathcal{A}(\mathcal{F}_u^L)$ into $O(\log n)$ canonical subsets. If v is the left (resp. right) child of its parent, then σ lies below (resp. above) p_j for all $\mathcal{E}_u^j \in \mathcal{F}_{uv}^1$. It is easily seen that ℓ intersects a segment of $\mathcal{B}_{uv}^c$ if and only if ℓ intersects a segment of E_{uv}^c. Therefore we query TCP_{uv} and report the colors of all segments of E_{uv} that intersect ℓ. A color is reported at most twice at each node v, so the time spent at v is $O(m_u^{1/2+\varepsilon} + k_{uv})$, where k_{uv} is the output size. Next, we repeat a similar procedure for TD_u with the intersection point of ℓ and the right boundary of I_u. Finally, we search TE_u with ℓ and report all colors of the convex hull edges intersected by ℓ. Summing over all nodes in $U(\gamma)$, the time spent in querying the short segments is $O(n^{1/2+\varepsilon} + k_s \log^2 n)$, where k_s is the number of distinct colors found. The overall query time is thus $O(n^{1/2+\varepsilon} + k \log^2 n)$.

LEMMA 5.2. *A set of n line segments in the plane can be preprocessed in $O(n^{4/3+\varepsilon})$ time into a data structure of size $O(n \log^3 n)$, so that all k connected components intersecting a given query segment can be reported in $O(n^{1/2+\varepsilon} + k \log^2 n)$ time, for any fixed $\varepsilon > 0$. The space required to construct the data structure is $O(n^{4/3+\varepsilon})$.*

5.4. *Improving the Running Time.* The query time of the above lemma can be improved to $O(n^{1/2+\varepsilon} + k)$ at the expense of a slight increase in the size of the data structure. The space used will be $O(n^{1+\varepsilon})$. The basic idea is to replace all binary trees in the above structure with n^δ-ary trees, for some small constant $0 < \delta < \varepsilon/4$. We describe how to modify TA_u; the other structures can be modified in an analogous manner.

To this end, we replace the binary tree TA_u by a minimum-height n_u^δ-ary tree on the segments in $\mathcal{R}_u$. (Recall that the segments in $\mathcal{R}_u$ are totally ordered, as they are disjoint and their endpoints lie on the boundary of I_u.) The height of TA_u is now $O(1/\delta)$. For an internal node $v \in TA_u$, let $w_1, \ldots, w_t$ ($t \le n_u^\delta$) be the children of v. For each $1 \le i \le j \le t$, let $C_{uv}(i, j)$ denote the set of colors of segments in $\bigcup_{i \le h \le j} \mathcal{R}_{uw_h}$. We store $C_{uv}(i, j)$ in a list at v. The space required by TA_u, including its secondary structures, is easily seen to be $O(n_u^{1+2\delta} \log n_u)$.

In order to compute the colors of segments in $\mathcal{R}_u$ intersected by a segment $\gamma = \overline{pq}$,

we locate its endpoints, in time $O((1/\delta)\log n_u^{\delta}) = O(\log n_u)$, among the segments of $\mathcal{R}_u$. Let π_p (resp. π_q) be the path from the root of TA_u to the leaf that stores the segment of $\mathcal{R}_u$ lying immediately above p (resp. below q). At each node $v \in \pi_p$, if $\{w_i, \ldots, w_j\}$ is the maximal set of children of v such that the segments in $\bigcup_{i \le h \le j} \mathcal{R}_{uw_h}$ intersect γ, we report the colors stored in $C_{uv}(i, j)$. We leave it to the reader to check that all colors of segments in $\mathcal{R}_u$ intersected by γ can be reported in time $O(\log n_u + k)$.

Similarly, the colors of segments in $\mathcal{E}_u$ intersected by a long segment γ can be reported in time $O(m_u^{1/2+\varepsilon} + k)$, using $O(m_u^{1+2\delta}\log^2 m_u)$ space.

Finally, we replace the primary segment tree T by an n^{δ}-ary tree. Let $w_1, \ldots, w_t$, for $t \le n^{\delta}$, denote the children of u. For each $1 \le i \le j \le t$, let $I_u(i, j) = \bigcup_{i \le h \le j} I_{w_h}$; $I_u(i, j)$ is also a vertical strip. We define the set of long and short segments $\mathcal{S}_u(i, j)$ and $\mathcal{E}_u(i, j)$ in a similar way. We then preprocess $\mathcal{S}_u(i, j)$ and $\mathcal{E}_u(i, j)$ as before, but replace binary trees with the trees of constant depth. The space requirement is $O(n^{1+4\delta}\log^3 n) = O(n^{1+\varepsilon})$. Since the depth of each tree is constant, a color will be reported only a constant number of times. Therefore we can obtain the following result.

THEOREM 5.3. *A set $\mathcal{S}$ of n line segments in the plane can be preprocessed in $O(n^{4/3+\varepsilon})$ time, using $O(n^{4/3+\varepsilon})$ space, into a data structure of size $O(n^{1+\varepsilon})$, so that all k connected components of $\mathcal{A}(\mathcal{S})$ that intersect a query segment can be reported in $O(n^{1/2+\varepsilon} + k)$ time, for any fixed $\varepsilon > 0$.*

REMARK 5.4. Using the space/query-time tradeoff for simplex range searching and line intersection data structures [4], [7], we can obtain a space/query-time tradeoff for Theorem 5.3. In particular, for any $n \le N \le n^2$, $\mathcal{S}$ can be preprocessed into a data structure of size $O(N^{1+\varepsilon})$, so that all k connected components of $\mathcal{A}(\mathcal{S})$ that intersect a query segment can be reported in time $O(n^{1+\varepsilon}/\sqrt{N} + k)$. The preprocessing time is $O(n^{4/3+\varepsilon} + N^{1+\varepsilon})$.

6. Intersection Queries for Simple Polygons. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ be a set of simple polygons, and let $\mathcal{S}$ denote the set of n edges of these polygons. We wish to preprocess $\mathcal{P}$ into a data structure so that all the polygons intersecting a query segment can be reported quickly. Again, we wish to return only the labels of polygons intersected by a query segment. By coloring the edges of P_i with color i, we reduce the problem to the colored segment intersection searching problem.

Our basic data structure is the same as in the previous section; i.e., we construct a segment tree T on the segments of $\mathcal{S}$, associate two subsets $\mathcal{S}_u$ and $\mathcal{E}_u$ of segments with each node u of T, and preprocess each of them into a data structure for colored segment intersection queries. The segments in $\mathcal{S}$ of the same color form a connected chain; therefore, if two connected components $\mathcal{E}_u^i$, $\mathcal{E}_u^j$ of $\mathcal{E}_u$ have the same color c, then there is a segment of color c that intersects the boundary of I_u and a segment of $\mathcal{E}_u^i$. We can preprocess the segments of $\mathcal{E}_u$ as described in Section 5.2, but we do not have to find *leaders* at ancestors of u (this results in more efficient preprocessing than in the previous section). We preprocess long segments into a different data structure because, unlike the previous section, segments with different colors may intersect each other.

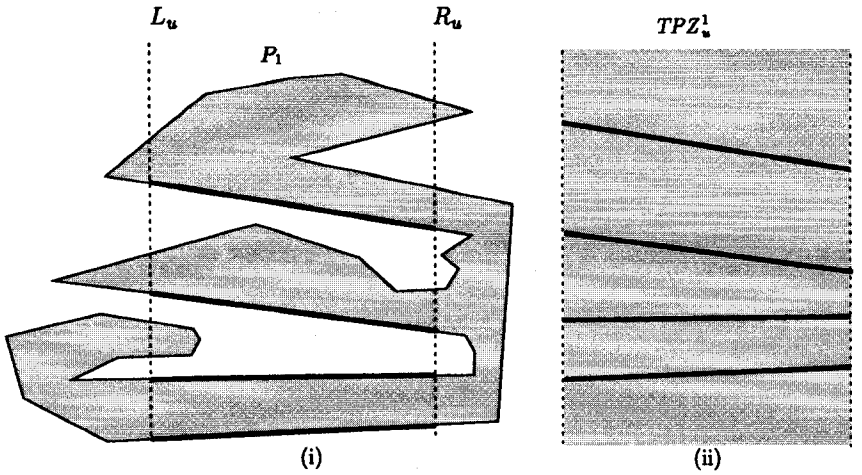

Fig. 5. (i) Partitioning of polygons into long and short segments; bold segments are long segments; (ii) TPZ_u^1.

6.1. *Preprocessing Long Segments.* We want to preprocess $\mathcal{S}_u$ into a data structure so that the colors of segments intersected by a query segment γ can be reported efficiently. (We assume that the segments of $\mathcal{S}_u$ are clipped within I_u.) We now use the fact that the relative interiors of segments of the same color do not intersect except perhaps at their endpoints. Let $\mathcal{S}_u^i \subseteq \mathcal{S}_u$ be the set of segments of color i. The segments of $\mathcal{S}_u^i$ partition I_u into a set TPZ_u^i of trapezoids, of which two are unbounded; see Figure 5. For the sake of convenience we add two edges, one at $-\infty$ and another at $+\infty$, so that all the trapezoids in TPZ_u^i are bounded. We set the color of trapezoids in TPZ_u^i to i. A query segment $\hat{\gamma} = \gamma \cap I_u$ intersects a segment of $\mathcal{S}_u^i$ if and only if the endpoints of $\hat{\gamma}$ lie in different trapezoids of TPZ_u^i. Although we can determine whether the endpoints of $\hat{\gamma}$ lie in different trapezoids of TPZ_u^i by a binary search, doing a binary search for each color separately will be very expensive, so we use a different approach. Set $TPZ_u = \bigcup_i TPZ_u^i$. We construct a two-level data structure TA_u, as described below, to report the trapezoids that contain the upper endpoint of $\hat{\gamma}$ but not the lower endpoint. For every color i, there is only one trapezoid of TPZ_u^i that contains the upper endpoint, so a color will be reported at most once.

Let $\mathcal{S}_u^*$ be the set of points dual to the lines supporting the segments of $\mathcal{S}_u$. We preprocess $\mathcal{S}_u^*$ for double-wedge range queries; see [20]. This algorithm constructs a partition tree on $\mathcal{S}_u^*$; each node v of the tree stores a *canonical subset* $\mathcal{S}_{uv}^* \subseteq \mathcal{S}_u^*$. The points lying (or not lying) in a query double-wedge can be represented as the union of $O(n_u^{1/2+\varepsilon})$ pairwise disjoint canonical subsets. Let $\mathcal{S}_{uv}$ be the set of segments corresponding to the points of $\mathcal{S}_{uv}^*$, and set $\mathcal{S}_{uv}^i = \mathcal{S}_{uv} \cap \mathcal{S}_u^i$. For each segment $e \in \mathcal{S}_{uv}^i$, pick up the edge in $\mathcal{S}_u^i$ that lies immediately above e. Let E_{uv}^i denote the set of lines supporting the segments. Let $E_{uv} = \bigcup_i E_{uv}^i$, and let E_{uv}^* denote the set of points dual to the lines supporting the segments of E_{uv}. Let $TPZ_{uv} \subseteq TPZ_u$ be the set of trapezoids whose top edges are in E_{uv}. We preprocess E_{uv}^* for double-wedge range queries, as described above, and store the resulting structure at v as the second-level structure of v. At each node w of the second-level structure, we store the colors of the segments corresponding to the canonical subset stored at w.

A standard analysis for a partition tree implies that the size of TA_u is $O(n_u \log n_u)$, and that it can be constructed in time $O(n_u \log n_u)$. The total time spent in preprocessing long segments over all nodes of T is thus $O(n^{1+\varepsilon})$.

6.2. *Answering a Query.* Let $\gamma = \overline{pq}$ be a query segment. As in Section 5.3, we query TA_u for all nodes $u \in \pi_p \cup \pi_q$ with the segment $\hat{\gamma} = \gamma \cap I_u$, and we query TC_u, TD_u, TE_u, for all nodes $u \in U(\gamma)$, with the line supporting γ. Since the secondary data structures for short segments are the same as in the previous section, it suffices to describe how to report the colors of segments in S_u that intersect $\hat{\gamma}$. This can be done by reporting the trapezoids of TPZ_u that contain only the upper endpoint of $\hat{\gamma}$. Let γ^* denote the double-wedge dual to $\hat{\gamma}$. We query the first-level structure of TA_u with γ^*. It computes a collection of $O(n_u^{1/2+\varepsilon})$ canonical subsets such that all points in a canonical subset lie in γ^* (which correspond to trapezoids whose bottom edges intersect $\hat{\gamma}$). Let $\mathcal{S}_{uv}^*$ be a canonical subset of the query output. Let τ be a trapezoid of TPZ_{uv}; τ contains the upper endpoint of γ if γ does not intersect the top edge of τ. Such trapezoids of TPZ_{uv} can be retrieved by querying the second-level structure of v with γ^* and reporting the colors of segments corresponding to the points in $\mathcal{S}_{uv}^*$ that do not lie in γ^*. Since each color is reported at most once, all k colors of segments in $\mathcal{S}_u$ intersecting $\hat{\gamma}$ can be reported in time $O(n_u^{1/2+\varepsilon} + k)$.

LEMMA 6.1. *The above procedure reports all k colors of segments in $\mathcal{S}_u$ intersecting a query segment in time $O(n_u^{1/2+\varepsilon} + k \log n_u)$.*

Thus, all k polygons of $\mathcal{P}$ intersecting a query segment can be reported in $O(n^{1/2+\varepsilon} + k \log^2 n)$ time; the query time can be improved to $O(n^{1/2+\varepsilon} + k)$ as in Section 5.4. Hence, we have

THEOREM 6.2. *For any constant $\varepsilon > 0$, a set $\mathcal{P}$ of simple polygons with a total of n edges can be preprocessed in time $O(n^{1+\varepsilon})$ into a data structure of size $O(n^{1+\varepsilon})$ so that all k polygons intersecting a query segment can be reported in time $O(n^{1/2+\varepsilon} + k)$.*

REMARK 6.3. (i) As in the previous section, a space/query-time tradeoff can be obtained by using the standard techniques; see [4] and [7].

(ii) The above algorithm returns only those polygons whose boundaries intersect a query segment. If we also want to report the polygons whose interiors contain γ, we triangulate each polygon and preprocess the set of resulting triangles in a data structure of size $O(n \log^2 n)$ for point location as described in [2]. We query this data structure with one of the endpoints of the query segment and return in time $O(n^{1/2+\varepsilon} + k)$ all k polygons corresponding to triangles that contain the query point.

7. Maintaining the Connected Components of Arbitrary Segments. Next we describe a semidynamic data structure for maintaining the connected components in the arrangement of a set $\mathcal{S}$ of line segments in the plane. The data structure supports the following three operations:

INSERT $(\gamma, \mathcal{S})$: Insert a new segment γ into $\mathcal{S}$.

REPORT-COMPONENT (γ, $\mathcal{S}$): Report (the label of) all connected components in $\mathcal{A}(\mathcal{S})$ that intersect γ.

SAME-COMPONENT (e_1, e_2, $\mathcal{S}$): Given two segments $e_1, e_2 \in \mathcal{S}$, decide whether they lie in the same connected component of $\mathcal{A}(\mathcal{S})$.

We dynamize the static data structure described in Section 5, using the ideas of Section 4 and a number of new ideas. As stated in the introduction, one of the main features of the semidynamic data structure is the "lazy" INSERT operation—a part of work by the insert procedure is postponed for subsequent query procedures without sacrificing the correctness of the query output. Consequently, a specific query may be very expensive, but we show that the amortized query time (over a sequence of INSERT and REPORT-COMPONENT operations) is close to its static counterpart. We analyze the amortized update and query time using the so-called *accounting method*, see [9] and [30]. For our application it means that when we insert a new segment γ into $\mathcal{S}$, certain units of *credits* are charged to the insertion procedure and these credits are deposited at γ for later use. Part of the actual cost of subsequent operations will be paid by these credits. If x units of the cost of an operation is paid by the credits deposited at a segment γ, then the number of credits of γ reduces by x. The number of credits deposited at each segment must always be nonnegative. Suppose that the actual cost of inserting a segment γ is a, that x units of the actual cost is paid by credits assigned to other segments, and that y credits are assigned to γ, then the amortized cost of inserting γ is $a - x + y$. Similarly, if the actual cost of a query procedure is a and x units of the actual cost are paid by credits assigned to other segments, then the amortized cost of the query procedure is $a - x$.

This section is organized as follows. We first describe the overall tree structure T and its adaptation for the maintenance of the connected components of $\mathcal{S}$. Then we describe the secondary structures and the insertion and query procedures for long segments. To describe the secondary structures for short segments, we first present a solution to a special case of a query problem related to connected component searching, in which the query object is a line. We use a variant of this structure to store short segments. The time complexity of update and query procedures is analyzed in Section 7.4.

7.1. *The Global Data Structure.* The structure for maintaining the connected components of a set $\mathcal{S}$ of line segments is basically the same as in Section 5. The main tree T is a segment tree on the x-projections of the segments of $\mathcal{S}$. Every node $u \in T$ stores five secondary structures TA_u, TB_u, TC_u, TD_u, and TE_u. As in Section 5, TA_u and TB_u store the subset of segments that are long at u, and the other three structures store the set of segments that are short at u. One difference is that we replace T and the five secondary structures with dynamic versions of these trees. For the main tree T, we maintain a dynamic segment tree on the segments in $\mathcal{S}$, as described on pp. 212–221 of [21]. As before, for a node $u \in T$, we denote the strip associated with u by I_u, the left boundary of I_u by L_u, the right boundary of I_u by R_u, the subset of short segments by $\mathcal{E}_u$, and the set of segments long at u but short at the parent of u by $\mathcal{S}_u$. All the segments in $\mathcal{E}_u$ and $\mathcal{S}_u$ are clipped within the vertical strip I_u.

As in the data structure for maintaining the connected components of orthogonal segments, described in Section 4, we use a union-find data structure $UF(\mathcal{S})$ on $\mathcal{S}$ to maintain the colors of segments in $\mathcal{S}$. We refer to the standard union and find operations

as COLOR-UNION and COLOR-FIND. COLOR-UNION requires $O(\log n)$ amortized time and COLOR-FIND requires $O(1)$ time. In what follows, by the statement "report the color of a segment e_i," we mean that we perform COLOR-FIND (e_i).

The secondary structures for the long and short segments are described in Sections 7.2.1, 7.3.3, and 7.3.4.

7.1.1. *The Global Query.* Let $\gamma = \overline{pq}$ be a query segment, and let $\overline{\gamma}$ be its xy-projection. Let π_p (resp. π_q) be the path in T from the root to the leaf z such that $p \in I_z$ (resp. $q \in I_z$), and let $U(\gamma)$ be the set of nodes u such that $\bar{\gamma}$ is long at u but not at the parent of u. We query TA_u at nodes $u \in \pi_p \cup \pi_q$ with the segment $\hat{\gamma} = \gamma \cap I_u$, and we query TC_u, TD_u, and TE_u at nodes $u \in U(\gamma)$ with the line ℓ containing γ. The queries in the secondary structures for the long and short segments are described in Sections 7.2.3 and 7.3.6, respectively.

7.1.2. *The Global Insertion.* To add a new segment γ to $\mathcal{S}$, we first have to determine the connected components of $\mathcal{A}(\mathcal{S})$ that γ intersects. This is precisely the REPORT-COMPONENT query. These components, along with γ, now become a single connected component of $\mathcal{A}(\mathcal{S} \cup \{\gamma\})$. If $c_1, \ldots, c_k$ is the set of colors returned by the query, we perform COLOR-UNION (c_1, c_i) for all $2 \le i \le k$.

The actual insertion of $\gamma = \overline{pq}$ into T is performed as described on pp. 212–221 of [21]. This algorithm also finds the sets of nodes π_p, π_q, and $U(\gamma)$ as defined for the query in the previous section. For every node $u \in U(\gamma)$, the segment $\hat{\gamma} = \gamma \cap I_u$ is inserted into TA_u and TB_u as described in Section 7.2.2. For every node $u \in \pi_p \cup \pi_q$, the segment $\hat{\gamma}$ is inserted into one of the structures for short segments TC_u, TD_u, or TE_u, as described in Section 7.3.5.

7.2. *The Long Segments.* In this subsection we first describe how TA_u and TB_u store $\mathcal{S}_u$, and then explain the insertion and query procedures for these structures.

7.2.1. *The Long Segment Structure.* We define $\mathcal{R}_u$, the set of representatives of $\mathcal{S}_u$, as in Section 5—for each connected component of $\mathcal{A}(\mathcal{S}_u)$, choose an arbitrary segment. The segments of $\mathcal{R}_u$ are pairwise disjoint, and their endpoints lie on the boundary of I_v, so they can be ordered by their heights. We construct a balanced binary tree TA_u on $\mathcal{R}_u$; the leaves of TA_u store the segments of $\mathcal{R}_u$, from left to right, in an increasing order of their heights. For each node $v \in TA_u$, let $\mathcal{R}_{uv}$ be the set of segments stored in the subtree rooted at v. Recall that in Section 5 we stored TAP_{uv}, the set of colors of segments in $\mathcal{R}_{uv}$, at v. Since the colors of segments in $\mathcal{S}_u$ will now change dynamically, the list TAP_{uv} now stores (at least) one representative of each color in the set $\mathcal{R}_{uv}$ (i.e., for each color present in $\mathcal{R}_{uv}$, we pick an arbitrary segment of that color and add it to TAP_{uv}). The colors of these segments are obtained using $UF(\mathcal{S})$. As new segments are inserted, some of the connected components of $\mathcal{A}(\mathcal{S})$ will merge into a single component, and some of the segments of $\mathcal{S}$ that had different colors earlier will be assigned the same color. As we see below, we do not update the list TAP_{uv} as soon as the color of a segment in $\mathcal{R}_{uv}$ changes, so there may be more than one representative in TAP_{uv} of one color. We do, however, periodically update TAP_{uv} to keep the amortized query time low; see below for details. Suppose a leaf z of TA_u stores the representative r_u^i. We store at z a dynamic data structure $TAQ(\mathcal{S}_u^i)$ for answering segment intersection-detection queries

(that is, whether a query segment intersects any segment of $\mathcal{S}_u^i$), see [4]. The query and update time for $TAQ(\mathcal{S}_u^i)$ are $O(n_{ui}^{1/2+\varepsilon})$ and $O(n_{ui}^{\varepsilon})$, respectively, where $n_{ui} = |\mathcal{S}_u^i|$.

Let $\mathcal{S}_u^*$ be the set of points dual to the lines supporting the segments of $\mathcal{S}_u$. We preprocess $\mathcal{S}_u^*$ into a dynamic data structure TB_u for determining whether $\mathcal{S}_u^* \cap W = \emptyset$ for a query double-wedge W; see [4] and [20]. If $W \cap \mathcal{S}_u^* \neq \emptyset$, the structure returns a segment corresponding to one of the points in $\mathcal{S}_u^* \cap W$ as a witness; we refer to this query procedure as EMPTY-DOUBLE-WEDGE $(\mathcal{S}_u, W^*)$, where W^* is the segment dual to W. The query time is $O(n_u^{1/2+\varepsilon})$ and the update time is $O(n_u^{\varepsilon})$. This structure is used to find the new leaders of short segments stored in the subtree rooted at a node u of the main tree T.

7.2.2. *The Long Segment Insertion.* Let $\gamma = \overline{pq}$ be a segment such that $p \in L_u$ and $q \in R_u$. We insert γ into TA_u and TB_u as follows. We first show that a representative segment can be inserted into or deleted from TA_u efficiently. Using these procedures as subroutines, we describe how to insert γ into TA_u, TB_u.

Suppose we want to insert a segment r_u^i that does not intersect any segment of $\mathcal{R}_u$. We find the segment r_u^j of $\mathcal{R}_u$ lying immediately below r_u^i. Let v be the node of TA_u storing r_u^j. We create a new leaf z to the right of r_u^j, and store r_u^i at z. For each node v on the path from the root of TA_u to z, we insert r_u^i into TAP_{uv}, in $O(1)$ time. A segment is deleted from TA_u in the same way except that the lists TAP_{uv} are not updated.

The actual insertion of γ is as follows:

(i) We add the point dual to the line containing γ to the segment intersection-detection structure TB_u.

(ii) We search TA_u with the endpoints of γ and find the set $\mathcal{R}(\gamma) \subseteq \mathcal{R}_u$ of all representatives that intersect γ. Let γ^b be the segment of $\mathcal{R}_u$ lying immediately below the lower endpoint of γ. If γ intersects $\mathcal{S}_u^b$, we add γ^b to $\mathcal{R}(\gamma)$. Next, we repeat the same procedure for the representative lying immediately above the upper endpoint of γ.

(iii) We delete all segments of $\mathcal{R}(\gamma)$ from TA_u, but do not update the lists TAP_{uv}. Next, we insert γ into TA_u.

(iv) Let $\mathcal{S}_u^{i+1}, \ldots, \mathcal{S}_u^j$ be the connected components of $\mathcal{S}_u$ whose representatives belong to $\mathcal{R}(\gamma)$. We merge the TAQ-structures for $\mathcal{S}_u^i = \{\gamma\}, \mathcal{S}_u^{i+1}, \ldots, \mathcal{S}_u^j$ by repeatedly inserting the segments of a smaller structure into a larger structure; see Algorithm 1 for the details. We store the resulting structure at the leaf of TA_u that stores γ.

7.2.3. *The Query in Long Segments.* We query TA_u with $\hat{\gamma} = \gamma \cap I_u$ as described in Section 5.3, with one addition. Suppose v is a highest node in TA_u such that all representatives in $\mathcal{R}_{uv}$ intersect $\hat{\gamma}$. We traverse the list TAP_{uv}, report the colors of TAP_{uv}, and remove duplications from TAP_{uv}. In more detail, for each segment $e \in TAP_{uv}$, we find the color $\chi(e)$ of e, and if $\chi(e)$ has not yet been reported, report $\chi(e)$ and mark it "reported." If $\chi(e)$ has already been reported (i.e., it is marked "reported"), we remove e from TAP_{uv}. After traversing TAP_{uv} this way, it is traversed a second time to unmark all marked colors. It is easily seen that the list TAP_{uv} can be updated in time linear in its size.

7.3. *The Short Segments.* In this subsection we describe the dynamic versions of the structures TC_u, TD_u, and TE_u for storing the short segments $\mathcal{E}_u$ at a node $u \in T$. We also present the insertion and query algorithms. Recall that, in Section 5, we computed the con-

Input: $\mathcal{S}_u^i, \ldots, \mathcal{S}_u^j$, a set of groups.
Actions: The structures $TAQ(\mathcal{S}_u^i), \ldots, TAQ(\mathcal{S}_u^j)$ are merged into one *TAQ*-structure.

$\Gamma = \mathcal{S}_u^i$;
for $h = i + 1$ to j do
  if $|\mathcal{S}_u^h| \geq |\Gamma|$ then swap $(\Gamma, \mathcal{S}_u^h)$;
  for each $e \in \mathcal{S}_u^h$ do
    p = point dual to the line supporting e;
    Add p to $TAQ(\Gamma)$;
  end-for
end-for

Algorithm 1. MERGE *TAQ*-STRUCTURE.

nected components $\mathcal{E}_u^1, \ldots, \mathcal{E}_u^t$ of $\mathcal{A}(\mathcal{E}_u)$. Each connected component was assigned to one of three sets $\mathcal{F}_u^L$, $\mathcal{F}_u^R$, and $\mathcal{F}_u^M$. In the dynamic version of our structure, we maintain a partition $\mathcal{E}_u^1, \ldots, \mathcal{E}_u^t$ of $\mathcal{E}_u$ such that, for each i, $\mathcal{A}(\mathcal{E}_u^i)$ is a connected planar graph. However, unlike the static data structure, $\mathcal{A}(\mathcal{E}_u^i)$ is not necessarily a maximal connected component of $\mathcal{A}(\mathcal{E}_u)$ (i.e., $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ may intersect). We call each $\mathcal{E}_u^i$ a *group* of $\mathcal{E}_u$. The groups of $\mathcal{E}_u$ are partitioned into three subsets $\mathcal{F}_u^L$, $\mathcal{F}_u^R$, and $\mathcal{F}_u^M$, as before. Each group stored in $\mathcal{F}_u^L$ and $\mathcal{F}_u^R$ satisfies the same condition as in Section 5.2. If a group $\mathcal{E}_u^i$ belongs to $\mathcal{F}_u^L$, then either there is a segment $e_i \in \mathcal{E}_u^i$ that intersects the left boundary L_u of I_u, or $\mathcal{E}_u^i$ does not intersect the right boundary of I_u, and there is a segment $e \notin \mathcal{E}_u^i$ that intersects L_u as well as $\mathcal{E}_u^i$. The segment $e_i \cap I_u$ is called the *leader* of $\mathcal{E}_u^i$ and is denoted by $\lambda(\mathcal{E}_u^i)$. If $e_i \notin \mathcal{E}_u^i$, we add a clipped copy of e_i to $\mathcal{E}_u^i$. Similarly, every group in $\mathcal{F}_u^R$ intersects the right boundary of I_u; the leaders of groups in $\mathcal{F}_u^R$ are defined in a similar manner. However, unlike the static case, some of the groups of $\mathcal{E}_u$ satisfying these conditions may be in $\mathcal{F}_u^M$, because when we insert a new segment γ into $\mathcal{S}$, it may intersect the left or right boundary of I_u and also a group in $\mathcal{F}_u^M$. It will be too expensive to detect all such groups $\mathcal{E}_u^i$ of $\mathcal{F}_u^M$ and to move them to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$. Instead, $\mathcal{E}_u^i$ will be moved when $\mathcal{E}_u^i$, or another group $\mathcal{E}_u^j$ that intersects $\mathcal{E}_u^i$, is returned by the query procedure. What we gain by postponing the move of $\mathcal{E}_u^i$ is that during a query procedure we can detect whether $\mathcal{E}_u^i$ intersects the boundary of I_u without increasing the asymptotic time complexity of the query procedure, whereas it will be very expensive to detect such groups by the insertion procedure, even if no group is moved. The groups of $\mathcal{F}_u^M$ have the following crucial property (already observed in Section 5.2): If two groups $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ have the same color, then either they lie in the same connected component of $\mathcal{A}(\mathcal{E}_u)$ or the connected components containing them intersect the boundary of I_u.

We first describe a data structure for a related problem, a variant of which is used in the structures TC_u, TD_u, and TE_u.

7.3.1. *Line Intersection Searching.* Let $\mathcal{E}$ be a set of n line segments in the plane, and let $\mathcal{E}^1, \ldots, \mathcal{E}^m$ be a partition of $\mathcal{E}$ into m groups. We develop a data structure $\Upsilon(\mathcal{E})$ for $\mathcal{E}$ that supports the following four operations:

GROUP-REPORT (ℓ): Let ℓ be a line such that, for any $i \leq m$, ℓ intersects $CH(\mathcal{E}^i)$ if and only if it intersects $\mathcal{E}^i$. Report all groups of $\mathcal{E}$ that intersect ℓ. More precisely, for each group $\mathcal{E}^i$ intersected by ℓ, return one of the segments of $\mathcal{E}^i$.

GROUP-INSERT ($\mathcal{E}^\circ$): Given a set of segments $\mathcal{E}^\circ$, create a new group $\mathcal{E}^{m+1} = \mathcal{E}^\circ$.
GROUP-MERGE (i, j): Merge $\mathcal{E}^i$, $\mathcal{E}^j$ into a single group $\mathcal{E}^i_u$; the new group is called $\mathcal{E}^i$.
GROUP-FIND (e): Return i if $e \in \mathcal{E}^i$.

$\Upsilon(\mathcal{E})$ consists of the following three structures:

Semidynamic convex hull structure: For each i, we maintain the convex hull of $\mathcal{E}^i$ using the algorithm of Preparata [25]. A new segment can be inserted in $O(\log n)$ time. Let E^i denote the edges of $CH(\mathcal{E}^i)$. For each segment $g \in E^i$, store some segment of $\mathcal{E}^i$ with g and denote it by $\varphi(g)$. Let $E = \bigcup_{i=1}^m E^i$.

Dynamic partition tree: Using the algorithm of Agarwal and Sharir [4], we maintain the edges of E into a two-level dynamic partition tree $\Psi(E)$ that can report all segments of E intersected by a query line. Moreover, a segment can be inserted into, or deleted from, $\Psi(E)$ in time $O(n^\varepsilon)$, and all k segments of E intersected by a query line can be reported in time $O(n^{1/2+\varepsilon} + k)$. The size of the data structure is $O(n \log n)$.

Union-find structure: We maintain a union-find structure $UF(\mathcal{E})$ on the segments of $\mathcal{E}$ that merges two sets in logarithmic (amortized) time and finds the set containing a segment in $O(1)$ time.

We now describe how to perform the four operations on $\mathcal{E}$. The GROUP-FIND operation is simply the standard FIND operation and takes constant time. The maintenance of $UF(\mathcal{E})$ under GROUP-INSERT and GROUP-MERGE is also standard, the latter being the UNION operation. It follows from [4] that GROUP-INSERT ($\mathcal{E}^\circ$) requires only $O(|\mathcal{E}^\circ|^{1+\varepsilon})$ time. Next, let ℓ be a query line that intersects $CH(\mathcal{E}^i)$, for any $i \le m$, if and only if it intersects $\mathcal{E}^i$. We query $\Psi(E)$ with ℓ and determine all segments of E intersected by ℓ. For each segment $g \in E$ of the query output, we report $\varphi(g)$. Since at most two segments of each group are reported, the total time spent in reporting k groups is $O(n^{1/2+\varepsilon} + k)$.

Finally, suppose we want to merge $\mathcal{E}^i$, $\mathcal{E}^j$. Without loss of generality, assume that $|\mathcal{E}^i| \le |\mathcal{E}^j|$. We insert all segments of $\mathcal{E}^i$ into the convex hull structure of $\mathcal{E}^j$ one by one. Let $g_1, \ldots, g_t$ be the edges of $CH(\mathcal{E}^j)$ not in $CH(\mathcal{E}^i \cup \mathcal{E}^j)$, and let $h_1, \ldots, h_r$, $r \le 4|\mathcal{E}^i|$, be the new edges of $CH(\mathcal{E}^i \cup \mathcal{E}^j)$. We delete $g_1, \ldots, g_t$ from $\Psi(E)$ and insert $h_1, \ldots, h_r$ into $\Psi(E)$. The total time spent is $O((t + r)n^\varepsilon)$.

LEMMA 7.1. *Let $\mathcal{E}$ be a set of n segments in the plane, and let $\mathcal{E}^1, \ldots, \mathcal{E}^m$ be a partition of $\mathcal{E}$. Then $\mathcal{E}$ can be stored in a data structure of size $O(n \log n)$, so that* GROUP-MERGE *operations can be performed in $O(n^\varepsilon)$ amortized time, a set $\mathcal{E}^\circ$ can be inserted in $O(|\mathcal{E}^\circ|^{1+\varepsilon})$ time,* GROUP-REPORT *can be performed in $O(n^{1/2+\varepsilon} + k)$ time (where k is the output size) and* GROUP-FIND *can be performed in constant time.*

PROOF. The size and the query time of the data structure follow from the above discussion, so it remains to bound the insert and merge time. Suppose we perform a sequence of insert and merge operations. Let h be the total number of segments inserted by the GROUP-INSERT procedure into $\mathcal{E}$. Each segment is deleted from E (and therefore from $\Psi(E)$) only once, so we can charge the time $O(n^\varepsilon)$ spent in deleting it from $\Psi(E)$ to its insertion in $\Psi(E)$. Moreover, the insertion of a segment into $\mathcal{E}^j$ (either by GROUP-INSERT

or by GROUP-MERGE) introduces at most four new edges in E. Assuming that a segment is inserted d times, we can pay for all updates in $\Psi(E)$ if we assign $8 \cdot c \cdot d \cdot n^{\varepsilon}$ units of credit to each new segment e added by the INSERT procedure; here c is the constant of proportionality in the update time of $\Psi(E)$. Hence it suffices to bound the value of d. Suppose $e \in \mathcal{E}^i$ just before GROUP-MERGE (i, j) and e is inserted into $\mathcal{E}^j$ during the merge operation. Then $|\mathcal{E}^j| \geq |\mathcal{E}^i|$ or $|\mathcal{E}^i| + |\mathcal{E}^j| \geq 2|\mathcal{E}^i|$, therefore, every time is inserted by the merge operation, the size of the set containing at least doubles. This implies that e will be inserted into convex hull data structures at most $\log(n + h)$ times, i.e., $d \leq \log(n + h)$. Hence, the amortized cost of an insert operation is $O(n^{\varepsilon}) + 8c \log(n + h) n^{\varepsilon} = O(n^{\varepsilon'})$, where ε' is another but arbitrarily small positive constant. This completes the proof of the lemma. □

7.3.2. *A Dynamic Structure for TCP_{uv} and TDP_{uv}.* We use the data structure of Section 7.3.1 to dynamize the structures TCP_{uv} and TDP_{uv}, third-level structures stored at each node of TC_u and TD_u (see Section 5.2). We only describe TCP_{uv} below; TDP_{uv} is completely analogous. TCP_{uv} stores a set of segments, $\mathcal{B}_{uv}$, and a partition $\mathcal{B}_{uv}^1, \ldots, \mathcal{B}_{uv}^s$ of $\mathcal{B}_{uv}$ into groups. $\mathcal{B}_{uv}$ and its partition have the following properties:

(i) Every segment e_i of $\mathcal{B}_{uv}$ has a color, which can be retrieved by COLOR-FIND (e_i) in the structure $UF(\mathcal{S})$ of Section 7.1.
(ii) If two segments e_i and e_j are in the same group, then they have the same color (but the segments of two different groups may have the same color).

TCP_{uv} supports the following operations:

COLOR-REPORT (TCP_{uv}, ℓ): Let ℓ be a line that intersects $CH(\mathcal{B}_{uv}^i)$ if and only if it intersects $\mathcal{B}_{uv}^i$. Report all colors of segments intersecting ℓ.
GROUP-INSERT $(TCP_{uv}, \mathcal{B}^{\circ})$: Create a new group $\mathcal{B}_{uv}^{s+1} = \mathcal{B}^{\circ}$.

TCP_{uv} is basically $\Upsilon(\mathcal{B}_{uv})$, described in Section 7.3.1; therefore, TCP_{uv} supports the GROUP-REPORT, GROUP-INSERT, GROUP-MERGE, and GROUP-FIND operations.

COLOR-REPORT (TCP_{uv}, ℓ) works in four steps. The first step is a call to GROUP-REPORT (ℓ) as described above; it returns a set of segments $e_{h_1}, \ldots, e_{h_2}$, such that there are at most two segments of any group. The second step identifies the groups $\mathcal{B}_{uv}^{l_1}, \ldots, \mathcal{B}_{uv}^{l_2}$ that contain these segments by GROUP-FIND. The third step identifies the colors of $e_{h_1}, \ldots, e_{h_2}$ using COLOR-FIND, and determines which groups of $\mathcal{B}_{uv}^{l_1}, \ldots, \mathcal{B}_{uv}^{l_2}$ have the same color. Finally, in the fourth step, if two groups $\mathcal{B}_{uv}^i, \mathcal{B}_{uv}^j$ have the same color, we merge them by calling GROUP-MERGE (i, j).

7.3.3. *The Short Segment Structures TC_u and TD_u.* This section describes the dynamic counterparts of the structure TC_u described in Section 5.2. The structure TD_u is completely analogous and is not described.

We describe how to preprocess $\mathcal{F}_u^L$, the set of groups that have a connection to the left boundary line L_u. Let $\mathcal{E}_u^1, \ldots, \mathcal{E}_u^t$ be the groups in $\mathcal{F}_u^L$, and let p_j be the left endpoint of the leader $\lambda(\mathcal{E}_u^j)$ of $\mathcal{E}_u^j$. Without loss of generality assume that $p_1, \ldots, p_t$ are ordered from bottom to top. We define the weight of p_j, $w(p_j)$, for $1 \leq j \leq t$, to be the number of segments in $\mathcal{E}_u^j$. We construct a weighted balanced binary search tree TC_u on $p_1, \ldots, p_t$. We define $\mathcal{F}_{uv}^L$ to be the set of connected components $\mathcal{E}_u^j$ such that p_j is stored in a leaf

of the subtree rooted at v. Let

$$\mathcal{B}_{uv} = \bigcup \{\mathcal{E}_u^i \mid \mathcal{E}_u^i \in \mathcal{F}_{uv}^L\}.$$

We maintain a partition $\mathcal{B}_{uv}^1, \ldots, \mathcal{B}_{uv}^s$ of $\mathcal{B}_{uv}$ at v; all segments within each $\mathcal{B}_{uv}^i$ have the same color. Initially, the colors of all groups are distinct, but as the new segments are inserted into $\mathcal{S}$, two different subsets may get the same color. The query procedure periodically merges some of the subsets $\mathcal{B}_{uv}^i$ of the same color. We maintain $\mathcal{B}_{uv}$ in a structure TCP_{uv} using the data structure described in Section 7.3.2.

Furthermore, we store a dynamic segment intersection-searching structure $\Pi(\mathcal{F}_u^L)$ along with TC_u. This structure allows three operations:

SEG-INSERT ($\Pi(\mathcal{F}_u^L)$, γ): Inserts a segment γ into $\Pi(\mathcal{F}_u^L)$.
SEG-DELETE ($\Pi(\mathcal{F}_u^L)$, γ): Deletes a segment γ from $\Pi(\mathcal{F}_u^L)$.
SEG-DETECT (γ, $\mathcal{F}_u^L$): Detects whether γ intersects any segment of $\bigcup \mathcal{F}_u^L$. If so, then it also returns one of the segments intersected by γ.

Such a data structure is given by Agarwal and Sharir [4]. The SEG-INSERT and SEG-DELETE require $O(n_u^\varepsilon)$ amortized time, and the SEG-DETECT requires $O(n_u^{1/2+\varepsilon})$ time. The size of $\Pi(\mathcal{F}_u^L)$ is $O(n_u \log^3 n_u)$.

7.3.4. *The Short Segment Structure TE_u.* Let $\mathcal{E}_u^1, \ldots, \mathcal{E}_u^t$ be the groups of $\mathcal{F}_u^M$ (again, with a slight abuse of notation). We use the data structure of Section 7.3.1 to obtain a dynamic structure TE_u, which replaces the static version described in Section 5.2. The groups in $\mathcal{F}_u^M$ have the following properties (recall that groups are also connected by definition):

(i) Every segment e_i of $\mathcal{E}_u^j$ has a color, which can be retrieved by COLOR-FIND (e_i).
(ii) If two segments e_i and e_j are in the same group, then they have the same color.
(iii) If two groups $\mathcal{E}_u^i$, $\mathcal{E}_u^j$ have the same color, then either they are in the same connected component of $\mathcal{A}(\mathcal{E}_u)$ (and they may be merged), or the connected component of $\mathcal{A}(\mathcal{S})$ that contains both of them intersects the boundary of I_u.

TE_u supports the following three operations:

COLOR-REPORT (TE_u, ℓ): Let ℓ be a line such that ℓ intersects $CH(\mathcal{E}_u^i)$ if and only if it intersects $\mathcal{E}_u^i$. Report all colors of segments intersecting ℓ.
GROUP-INSERT (TE_u, $\mathcal{E}^\circ$): Create a new group $\mathcal{E}_u^{m+1} = \mathcal{E}^\circ$.
GROUP-REMOVE (TE_u, $\mathcal{E}_u^i$): Remove the group $\mathcal{E}_u^i$ from $\mathcal{F}_u^M$.

TE_u is composed of two data structures: $\Upsilon(\mathcal{F}_u^M)$ and a dynamic segment intersection-searching structure $\Pi(\mathcal{F}_u^M)$ similar to the one described in the previous subsection. The size of $\Pi(\mathcal{F}_u^M)$ is $O(n_u \log^3 n_u)$, the update time is $O(n_u^\varepsilon)$, and the query time is $O(n_u^{1/2+\varepsilon})$.

The GROUP-INSERT procedure is basically the same as GROUP-INSERT for TCP_{uv} (see Section 7.3.2), with the extension that the new segments are also inserted into $\Pi(\mathcal{F}_u^M)$. COLOR-REPORT (TE_u, ℓ) is performed in four steps; the first three steps are the same as those for COLOR-REPORT (TCP_{uv}, ℓ) of Section 7.3.2. Let $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ be two groups of the query output that have the same color. The fourth step tests whether $\mathcal{E}_u^i$ and

$\mathcal{E}_u^j$ belong to the same connected component of $\mathcal{F}_u^M$, using a procedure GROUPS-IN-SAME-COMPONENT (i, j). If they belong to the same component, they are merged by GROUP-MERGE (i, j) as described in Section 7.3.1. Otherwise, the smaller of the two groups is removed from $\mathcal{F}_u^M$ by calling the GROUP-REMOVE procedure.

The GROUPS-IN-SAME-COMPONENT (i, j) procedure works as follows. Suppose $|\mathcal{E}_u^i| \leq |\mathcal{E}_u^j|$. For each segment $e \in \mathcal{E}_u^i$, we test whether e intersects any segment of $(\bigcup \mathcal{F}_u^M) - \mathcal{E}_u^i$. If there is no such segment, we return "false," i.e., $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ lie in different components of $\mathcal{A}(\mathcal{E}_u)$. Next, suppose that there is a segment $e \in \mathcal{E}_u^i$ that intersects some segment $g \in (\bigcup \mathcal{F}_u^M) - \mathcal{E}_u^i$. If $g \in \mathcal{E}_u^j$, then we merge $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ into a single group and return "true." If g belongs to some other component $\mathcal{E}_u^h$, then we merge $\mathcal{E}_u^i$ and $\mathcal{E}_u^h$, using GROUP-MERGE (i, h), and determine whether $\mathcal{E}_u^i \cup \mathcal{E}_u^h$ and $\mathcal{E}_u^j$ lie in the same component of $\mathcal{A}(\mathcal{E}_u)$. How we perform the last step depends on $|\mathcal{E}_u^h|$. If $|\mathcal{E}_u^h| > |\mathcal{E}_u^i|$, then we start from scratch. Otherwise, for those segments of $\mathcal{E}_u^i \cup \mathcal{E}_u^h$ that we have not tested so far, we determine whether they intersect any segment of $(\bigcup \mathcal{F}_u^M) - (\mathcal{E}_u^i \cup \mathcal{E}_u^h)$, and repeat the same procedure. Intuitively, this procedure merges $\mathcal{E}_v^i$, $\mathcal{E}_v^j$ into a single group if they belong to the same connected component of $\mathcal{A}(\mathcal{E}_u)$, and in this process some other (but not necessarily all) groups that belong to the same component are also merged with $\mathcal{E}_u^i$. If $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ do not lie in the same connected component of $\mathcal{A}(\mathcal{E}_u)$, then one or both of $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ may be merged with other groups. See Algorithm 2 for a complete description.

7.3.5. *The Short Segment Insertion.* Inserting a short segment is fairly simple. TC_u, TD_u, and TE_u are updated only partially, and the remainder of the work is left to be carried out by subsequent REPORT-COMPONENT queries.

Let $\gamma = \overline{pq}$ be a segment that we want to add to $\mathcal{E}_u$. Assume that γ is clipped within I_u. We create a new set $\mathcal{E}_u^{t+1} = \{\gamma\}$. If γ does not intersect the boundary of I_u, we add $\mathcal{E}_u^{t+1}$ to $\mathcal{F}_u^M$ (i.e., insert $\mathcal{E}_u^{t+1}$ into TE_u by calling GROUP-INSERT $(TE_u, \mathcal{E}_u^{t+1})$ as described in Section 7.3.4).

If γ intersects the left boundary of I_u, we add $\mathcal{E}_u^{t+1}$ to $\mathcal{F}_u^L$. Let p_{t+1} be the left endpoint of γ. We set the weight of p_{t+1} to be 1, and add p_{t+1} to TC_u. Let z be the leaf of TC_u that stores p_{t+1}. For each node v on the path from the root to z, we call GROUP-INSERT $(TCP_{uv}, \{\gamma\})$—it creates a new set $\mathcal{B}_{uv}^{t+1} = \{\gamma\}$, as described in Section 7.3.2. If γ intersects the right boundary of I_u, we insert γ into TD_u in an analogous manner.

7.3.6. *The Query in Short Segments.* We have now come to the description of the REPORT-COMPONENT procedure for short segments. Since a lot of work of the INSERT operation is postponed for later, the query algorithm is fairly involved. The main feature of the procedure is the following. Whenever a color at some node of the secondary structure is reported more than twice it either removes duplications or merges some groups to form larger groups.

Let ℓ be the line supporting the query segment, and let σ be the intersection point of ℓ and the left boundary of I_u. As in Section 5.3, we search TC_u with σ and find the leaf z that stores the highest point lying below σ. For each node v that is a descendant of a node on the path from the root to z, we query the data structure TCP_{uv} constructed on $\mathcal{B}_{uv}$. Since σ either lies below the intersection point of L_v with the leaders of all groups stored in the subtree rooted at v, or it lies above all of them, ℓ intersects a group $\mathcal{B}_{uv}^i$ if and only if it intersects $CH(\mathcal{B}_{uv}^i)$. We can therefore use COLOR-REPORT (TCP_{uv}, ℓ)

Input: $\mathcal{E}_u^i$, $\mathcal{E}_u^j$: two groups in $\mathcal{F}_u^M$ of the same color; $|\mathcal{E}_u^i| \leq |\mathcal{E}_u^j|$.
Output: *true* if $\mathcal{E}_u^i$, $\mathcal{E}_u^j$ are in the same component of $\mathcal{A}(\mathcal{E}_u)$, *false* otherwise.

```
Q = ∅   (* Initialize a stack *)
for all e ∈ E_u^i do
  SEG-DELETE (Π(F_u^M), e), push (e, Q)
end-for
while not empty (Q) do
  e = pop (Q), x = SEG-DETECT (Π(F_u^M), e)
  if an x is returned then
    h = GROUP-FIND (x, F_u^M)
    if h = j then
      for all e ∈ E_u^i do
        SEG-INSERT (Π(F_u^M), e)
      end-for
      GROUP-MERGE (i, j)
      return true
    end-if
    if |E_u^h| > |E_u^i| then
      for all e ∈ E_u^i do
        SEG-INSERT (Π(F_u^M), e)
      end-for
      GROUP-MERGE (i, h)
      return GROUPS-IN-SAME-COMPONENT (i, j)
    end-if
    for all e ∈ E_u^h do
      SEG-DELETE (Π(F_u^M), e), push (e, Q)
    end-for
    GROUP-MERGE (i, h)
  end-if
end-while
return false
```

Algorithm 2. GROUPS-IN-SAME-COMPONENT (i, j).

to report the colors of segments in $\mathcal{B}_{uv}$ that intersect ℓ. Next, we search TD_u with the intersection point of ℓ and the right boundary of I_u, and repeat the same procedure.

After having searched TC_u and TD_u, we search $\mathcal{F}_u^M$ with ℓ. We search TE_u using COLOR-REPORT (TE_u, ℓ) as described in Section 7.3.4. If we find two groups $\mathcal{E}_u^i$, $\mathcal{E}_u^j$ of the same color, we first check whether they are in the same component of $\mathcal{A}(\mathcal{E}_u)$, using the GROUPS-IN-SAME-COMPONENT procedure. If they do not belong to the same component, then one of them is removed from $\mathcal{F}_u^M$ and added to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$, depending on whether they have a connection to the left boundary of I_u or to the right boundary of I_u. $\mathcal{E}_u^i$ is moved to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$ as follows. We first test whether any segment of $\mathcal{E}_u^i$ intersects L_u. If we find such a segment e, then we set $\lambda(\mathcal{E}_u^i) = e$ and insert $\mathcal{E}_u^i$ into TC_u using the procedure MOVE-TO-TC, described below. If no such segment is found, we repeat

Input: A group $\mathcal{E}_u^i$ formerly in $\mathcal{F}_u^M$.
Output: A leader of $\mathcal{E}_u^i$ that connects it to L_u.

```
Q = ∅
for all e ∈ E_u^i do
  push (e, Q)
end for
while not empty (Q) do
  e = pop (Q), z = u
  while z ≠ NULL do
    g = EMPTY-DOUBLE-WEDGE (TB_z, e)
    if a g is found then
      ĝ = g ∩ I_u, λ(E_u^i) = ĝ
      E_u^i = E_u^i ∪ {ĝ}
      return ĝ
      else z = parent (z)
    end-if
  end-while
end-while
```

Algorithm 3. FIND-LEADER.

the same procedure for R_u. If $\mathcal{E}_u^i$ does not intersect R_u either, then we can conclude that there is a segment $g \in \mathcal{S}_w$ for some ancestor w of u that intersects both $\mathcal{E}_u^i$ and L_u. For all ancestors w of u including u itself, we query TB_w with all segments of $\mathcal{E}_u^i$, using the procedure EMPTY-DOUBLE-WEDGE (see Section 7.2.1), until g is found; see Algorithm 3 for details. We set $\lambda(\mathcal{E}_u^i) = g \cap I_u$, insert $g \cap I_u$ into $\Pi(\mathcal{F}_u^L)$, and call MOVE-TO-TC.

Finally, $\mathcal{E}_u^i$ is inserted into TC_u as follows (the MOVE-TO-TC-procedure). Let $p = \lambda(\mathcal{E}_u^i) \cap L_u$; the weight of p is the number of segments in $\mathcal{E}_u^i$. We first insert p into TC_u. (Recall that the leaves of TC_u store the intersection points of the leaders with L_u in increasing order of their y-coordinates.) Let z be the leaf of TC_u storing p. We insert $\mathcal{E}_u^i$ into TCP_{uv} at all ancestors of z in TC_u; see Algorithm 4 for details.

The insertion of $\mathcal{E}_u^i$ into TD_u is completely analogous.

Input: A group $\mathcal{E}_u^i$ formerly in $\mathcal{F}_u^M$ (before it had a leader).
Actions: Insert $\mathcal{E}_u^i$ into TC_u.

```
p = λ(E_u^i) ∩ L_u
INSERT (p, TC_u)
z = leaf of TC_u storing p
while z ≠ NULL do
  GROUP-INSERT (TCP_uz, E_u^i)
  z = parent (z)
end-while
```

Algorithm 4. MOVE-TO-TC.

Table 1. Credits assigned to various copies of a segment.

Structures	Credits
TAP_{uv}	c_1
$TAQ(\mathcal{S}_u^i)$	$c_2 n^\varepsilon \log n$
TCP_{uv}	$c_3 n^\varepsilon \log n$
TDP_{uv}	$c_3 n^\varepsilon \log n$
TE_u	$c_4 n^{1/2+\varepsilon} \log n$

7.4. *The Analysis.* As mentioned earlier, we use the accounting method to bound the amortized update and query time. We assign

$$C(e) = cn^{1/2+\varepsilon} \log^3 n \tag{2}$$

credits to each segment e when it is inserted into $\mathcal{S}$, where c is a sufficiently large positive constant. To simplify the analysis, we distribute these credits among various copies of the segments stored in different secondary structures, as explained in Table 1 (the constants $c_1, \ldots, c_4$ are chosen sufficiently large).

LEMMA 7.2. *The insertion of a segment e into T, excluding the time spent by the* REPORT-COMPONENT *procedure, requires $O(n^{1/2+\varepsilon} \log^3 n)$ amortized time.*

PROOF. By the accounting method, the amortized insertion time is the actual insertion time, plus the number of credits left behind, minus the work paid for by credits.

It is not difficult to see that the actual insertion time for all insertion procedures in *TB*, *TCP*, *TDP*, and *TE* structures is $O(n^\varepsilon \log^2 n)$. Furthermore, the insertion time in all *TA* structures is $O(n^\varepsilon \log n)$ plus the time spent in merging *TAQ* structures. We charge the latter quantity to the credits of the segments in the *TAQ* structures themselves. Whenever two *TAQ* structures are merged, the segments of the smaller one are inserted into the structure of the larger one. Hence, any segment in a *TAQ* structure is inserted into another one at most $\log_2 n$ times. Each insertion requires $O(n^\varepsilon)$ time, so the total charge to any segment is $O(n^\varepsilon \log n)$. Since each segment in a *TAQ* structure was assigned $c_2 n^\varepsilon \log n$ credits when it was inserted for the first time, each segment has enough credits to pay for all insertions into other *TAQ* structures, provided that c_2 is at least as large as the constant hidden in the $O(n^\varepsilon)$ insertion time. Finally, $C(e) = O(n^{1/2+\varepsilon} \log^3 n)$, so the amortized time spent in inserting a segment into the secondary structures is $O(n^{1/2+\varepsilon} \log^3 n)$.

If the secondary structures stored at each node of a dynamic binary search tree can be updated in time $t(n)$, then the amortized update time of the overall data structure is $\sum_{i=1}^{\log n} O(t(2^i))$; see, e.g., [21]. Putting everything together and omitting the straightforward details, we conclude that the amortized insertion time, excluding the time spent by the REPORT-COMPONENT procedure, is $O(n^{1/2+\varepsilon} \log^3 n)$. □

We prove the amortized bound for REPORT-COMPONENT with the help of three lemmas that bound the amortized query time of the three secondary structures.

LEMMA 7.3. *The amortized time for reporting the colors in a list TAP_{uv} is $O(k)$, where k is the number of different colors reported.*

PROOF. Suppose that the list TAP_{uv} contains K segments of k distinct colors. Then the actual time taken by the query is $O(K)$, as the total time spent in processing each segment of TAP_{uv} is $O(1)$. Recall that we have given c_1 credits to each segment in TAP_{uv} when it was inserted. If a segment e is deleted, then the credits assigned to e pay for the cost of processing e. Since e is deleted only once from TAP_{uv}, it will not be charged again. If c_1 is chosen larger than the constant in the big-O of $O(K)$, then e has sufficient credits to pay for the cost charged to it. Hence, the amortized cost is $O(k)$. □

LEMMA 7.4. *The amortized time for* COLOR-REPORT *(TCP_{uv}, ℓ) is $O(n^{1/2+\varepsilon}+k)$, where k is the number of different colors reported.*

PROOF. The actual time for the first three steps of COLOR-REPORT is $O(n^{1/2+\varepsilon}+K)$, where K is the number of groups reported. Let k be the number of different colors that are found. Then the fourth step of COLOR-REPORT performs $K-k$ GROUP-MERGE operations. We charge $O(K-k)$ plus the cost of GROUP-MERGE operations to various segments, as described below, in such a way that each segment of $\mathcal{B}_{uv}$ has enough credits to pay for the cost charged to it. Therefore, the amortized cost of COLOR-REPORT is $O(n^{1/2+\varepsilon}+k)$.

By Lemma 7.1, the cost of all GROUP-MERGE operations can be paid by charging $O(n^{\varepsilon}\log n)$ to each segment of $\mathcal{B}_{uv}$. Next, we charge $O(K-k)$ to segments of $\mathcal{B}_{uv}$ as follows. Recall that a new group of $\mathcal{B}_{uv}$ is created either when a new segment is inserted into $\mathcal{B}_{uv}$ or when a group is moved from $\mathcal{F}_u^M$ to $\mathcal{F}_u^L$. For a group $\mathcal{B}_{uv}^i$, let $\mu(\mathcal{B}_{uv}^i)$ denote one of the segments belonging to $\mathcal{B}_{uv}^i$ when it was created. (If $\mathcal{B}_{uv}^i$ was created by inserting a new segment e, then $\mu(\mathcal{B}_{uv}^i)=e$, and if $\mathcal{B}_{uv}^i$ was created by moving a group $\mathcal{E}_u^i$ from $\mathcal{F}_u^M$, then $\mu(\mathcal{B}_{uv}^i)$ is a segment of $\mathcal{E}_u^i$.) If COLOR-REPORT merges two groups $\mathcal{B}_{uv}^i$, $\mathcal{B}_{uv}^j$ and the new group is called $\mathcal{B}_{uv}^j$, then we charge $\Theta(1)$ cost to $\mu(\mathcal{B}_{uv}^i)$. Since $\mathcal{B}_{uv}^i$ ceases to exist after the merge, $\mu(\mathcal{B}_{uv}^i)$ will not be charged again. Moreover, if COLOR-REPORT reports K groups of k different colors, then $K-k$ groups are merged, so the total cost charged is $\Theta(K-k)$.

The total cost charged to each segment of $\mathcal{B}_{uv}$ is thus $O(n^{\varepsilon}\log n)+O(1)=O(n^{\varepsilon}\log n)$. If the constant c_3 in Table 1 is sufficiently large, then the credits assigned to each segment can pay for the cost charged to it. □

LEMMA 7.5. *The amortized time for* COLOR-REPORT *(TE_u, ℓ) is $O(n^{1/2+\varepsilon}+k)$, where k is the number of different colors reported.*

PROOF. The actual time spent in reporting K groups of k different colors intersected by ℓ is $O(n^{1/2+\varepsilon}+K)$ plus the time spent in the fourth step of the COLOR-REPORT procedure. We show below that we can charge $O(K-k)$ and the time spent in the fourth step of the COLOR-REPORT to various segments of $\bigcup \mathcal{F}_u^M$ in such a way that the credits assigned to each segment are sufficient to pay the cost charged to it, so the amortized cost of the procedure is $O(n^{1/2+\varepsilon}+k)$.

First, as in the previous lemma, we can charge $O(K-k)$ to various segments of $\bigcup \mathcal{F}_u^M$ in such a way that the total cost charged to each segment is only $\Theta(1)$, so we only have to describe how to charge the time spent in the fourth step of the COLOR-REPORT procedure. Consider the GROUPS-IN-SAME-COMPONENT procedure. Observe that every segment e that is pushed on the stack Q will either be moved to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$, or it will end up in a group of $\mathcal{F}_u^M$ whose size is at least twice the size of the group that contained e earlier. Therefore, e is pushed onto Q at most $\log n$ times. By Lemma 7.1, the cost of GROUP-MERGE operations can be paid by charging $O(n^\varepsilon \log n)$ to each segment of $\bigcup \mathcal{F}_u^M$. For each segment e in Q, we spent $O(n^{1/2+\varepsilon})$ in SEG-DETECT, $O(n^\varepsilon)$ time in SEG-DELETE, $O(n^\varepsilon)$ time in SEG-INSERT, and another $O(1)$ time in other procedures. Hence, by charging $O(n^{1/2+\varepsilon} \log n)$ to each segment, we can cover the cost of all calls to the GROUPS-IN-SAME-COMPONENT procedure.

If a group $\mathcal{E}_u^i$ is moved from $\mathcal{F}_u^M$ to $\mathcal{F}_u^L$, we spend $O(|\mathcal{E}_u^i| n^{1/2+\varepsilon} \log n)$ time in finding the leader of $\mathcal{E}_u^i$ (see FIND-LEADER procedure) and $O(n^\varepsilon \log n)$ time to create the new leaf z of TC_u that stores $\mathcal{E}_u^i$. Moreover, $\mathcal{E}_u^i$ is also inserted into TCP_{uv} at all ancestors v of z. By Lemma 7.1, the amortized running time of inserting $\mathcal{E}_u^i$ at v is $O(|\mathcal{E}_u^i| \cdot n^\varepsilon)$. We also assign $c_3 n^\varepsilon \log n$ credits to each segment of $e \in \mathcal{E}_u^i$ stored at v, in order to fulfill the invariant that any segment inserted into TCP_{uv} has $c_3 n^\varepsilon \log n$ credits. Thus, the cost of moving $\mathcal{E}_u^i$ from $\mathcal{F}_u^M$ to $\mathcal{F}_u^L$ can be paid by charging $O(n^{1/2+\varepsilon} \log n)$ to each segment of $\mathcal{E}_u^i$. The same holds if $\mathcal{E}_u^i$ is moved to $\mathcal{F}_u^R$. Clearly, any group of $\mathcal{F}_u^M$ is transferred to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$ only once, thus this charge occurs once. If c_4 is chosen sufficiently large, then the credits assigned to each segment of $\bigcup \mathcal{F}_u^M$ are sufficient to pay for the cost incurred in merging the groups of $\mathcal{F}_u^M$ and in moving the groups of $\mathcal{F}_u^M$ to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$. This completes the proof of the lemma. □

By adding up the amortized query time at all secondary structures and by observing that a REPORT-COMPONENT query does not hand out credits, the previous lemmas lead to the following claim:

LEMMA 7.6. *Let $\mathcal{S}$ be a set of n segments in the plane. $\mathcal{S}$ can be stored into a data structure of size $O(n \log^5 n)$, so that a new segment can be inserted in time $O(n^{1/2+\varepsilon})$, the k connected components of $\mathcal{A}(\mathcal{S})$ intersecting a new segment can be reported in time $O(n^{1/2+\varepsilon} + k \log^2 n)$, and constant time suffices to determine whether two query segments of $\mathcal{S}$ belong to the same component.*

PROOF. The bound on the amortized running time of the REPORT-COMPONENT follows from the previous three lemmas. Next, we obtain the time spent in inserting a segment e. By Lemma 7.2, the amortized time spent by the INSERT procedure, excluding the time spent in REPORT-COMPONENT, is $O(n^{1/2+\varepsilon} \log^3 n) = O(n^{1/2+\varepsilon'})$, for any $\varepsilon' > \varepsilon$. The amortized query time of REPORT-COMPONENT is $O(n^{1/2+\varepsilon} + k \log^2 n)$, where k is the number of components reported. Since all these k components are merged into a single component, we can charge $\Theta(k \log^2 n)$ cost to various segments of $\mathcal{S}$ in such a way that each segment is charged at most $b \log^2 n$ (see, e.g., the proof of Lemma 7.4), here b is a constant. Hence, the total amortized cost of the insert procedure is $O(n^{1/2+\varepsilon}) + O(n^{1/2+\varepsilon} + k \log^2 n) - b \cdot k \log^2 n = O(n^{1/2+\varepsilon})$, provided the constant b is chosen sufficiently large. □

Using the same ideas as in Section 5.4, we can improve the amortized query time to $O(n^{1/2+\varepsilon} + k)$. We omit the straightforward, but rather tedious, details. We thus have

THEOREM 7.7. *Let S be a set of n segments in the plane. S can be stored into a data structure of size $O(n^{1+\varepsilon})$, so that a new segment can be inserted in time $O(n^{1/2+\varepsilon})$, the k connected components of $\mathcal{A}(S)$ intersecting a new segment can be reported in time $O(n^{1/2+\varepsilon} + k)$, and constant time suffices to determine whether two query segments of S belong to the same component.*

Notice that the update time is $O(n^{1/2+\varepsilon})$ because the insertion procedure calls REPORT-COMPONENT and assigns $n^{1/2+\varepsilon}$ credits. If we allow more space, the query time of all structures can be reduced, which will also improve the amortized update time. In particular, if we allow $O(N^{1+\varepsilon})$ space, then the amortized query and update time are $O(n^{1+\varepsilon}/\sqrt{N} + k)$ and $O(n^{1+\varepsilon}/\sqrt{N} + N^{1+\varepsilon}/n)$. This implies an $O(n^{4/3+\varepsilon})$-time on-line algorithm for computing the connected components of $\mathcal{A}(S)$.

COROLLARY 7.8. *The connected components of the arrangement of a set of n segments can be computed by an on-line algorithm in $O(n^{4/3+\varepsilon})$ time.*

8. Conclusions. In this paper we presented several data structures, static as well as semidynamic, for connected-component and simple-polygon intersection searching. For orthogonal segments we presented data structures for the general colored segment intersection-searching problem. We conclude this paper by mentioning a few open problems:

(i) Can one answer the general colored segment intersection query for an arbitrary set of segments in time $O(n^{1/2+\varepsilon} + k)$, where k is the set of output colors, using close-to-linear space? As a first step, one may want to consider the following problem: Preprocess a set S of colored points into a linear-size data structure so that the colors of points lying in a query strip (or a double-wedge) can be reported in time $O(n^{1/2+\varepsilon} + k)$. If the query is a half-space, then it can be solved efficiently: For each color i, compute the convex hull C_i of points of S of color i. Let $C_1, \ldots, C_m$ be the resulting polygons. The problem reduces to determining the set of polygons intersected by a query half-plane. Such a query can be answered in $O(\log n + k)$ time using near-linear space; see [3] and [29]. However, this approach does not extend to strips or to double-wedges.

(ii) No efficient algorithm is known for the counting version of colored segment intersection searching, even for orthogonal segments. Recently, Gupta *et al.* [15] have presented algorithms in some special cases.

(iii) The data structures for connected component intersection searching described here do not support delete operations. One difficulty lies in identifying the new connected components that emerge when we delete a segment.

(iv) Is there a simpler data structure for semidynamic connected-component intersection searching?

References

[1] P. K. Agarwal, Partitioning Arrangements of Lines: II. Applications, *Discrete Comput. Geom.* **5** (1991), 533–573.

[2] P. K. Agarwal, Ray Shooting and Other Applications of Spanning Trees with Low Stabbing Number, *SIAM J. Comput.* **21** (1992), 540–570.

[3] P. K. Agarwal, M. van Kreveld, and M. Overmars, Intersection Queries in Curved Objects, *J. Algorithms* **15** (1993), 229–266.

[4] P. K. Agarwal and M. Sharir, Applications of a New Space Partitioning Technique, *Discrete Comp. Geom.* **9** (1993), 11–38.

[5] B. Aronov, H. Edelsbrunner, L. Guibas, and M. Sharir, Improved Bounds on the Complexity of Many Faces in Arrangements of Segments, *Combinatorica* **12** (1992), 261–274.

[6] R. Bar Yehuda and S. Fogel, Good Splitters with Applications to Ray Shooting, *Algorithmica* **11** (1994), 133–145.

[7] B. Chazelle, M. Sharir, and E. Welzl, Quasi-Optimal Upper Bounds for Simplex Range Searching and New Zone Theorems, *Algorithmica* **8** (1992), 407–430.

[8] S. W. Cheng and R. Janardan, Algorithms for Ray-Shooting and Intersection Searching, *J. Algorithms* **13** (1992), 670–692.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[10] M. Dillencourt, H. Samet, and M. Tammiuen, A General Approach to Connected Component Labeling for Arbitrary Image Representation, *J. Assoc. Comput. Mach.* **39** (1992), 253–280.

[11] D. P. Dobkin and H. Edelsbrunner, Space Searching for Intersecting Objects, *J. Algorithms* **8** (1987), 348–361.

[12] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, 1987.

[13] L. Guibas, Personal communication.

[14] L. Guibas, M. Overmars, and M. Sharir, Ray Shooting, Implicit Point Location, and Related Queries in Arrangements of Segments, Techn. Rep. No. 433, New York University, 1989.

[15] J. Gupta, R. Janardan, and M. Smid, Further Results on Generalized Intersection Searching Problems: Counting, Reporting and Dynamization, *Proc. 3rd Workshop on Algorithms and Data Structures* (1993), Lecture Notes in Computer Science, Vol. 709, Springer-Verlag, Berlin, pp. 361–372.

[16] J. Gupta, R. Janardan, and M. Smid, Efficient Algorithms for Generalized Intersection Searching on Non-Iso-Oriented Objects, *Proc. 10th ACM Symposium on Computational Geometry* (1994), pp. 369–378.

[17] J. Gupta, R. Janardan, and M. Smid, On Intersection Searching Involving Curved Objects, *Proc. 4th Scand. Workshop on Algorithms Theory* (1994), Lecture Notes in Computer Science, Vol. 824, Springer-Verlag, Berlin, pp. 183–194.

[18] H. Imai and T. Asano, Finding the Connected Components and a Maximum Clique of an Intersection Graph of Rectangles in the Plane, *J. Algorithms* **4** (1984), 310–323.

[19] R. Janardan and M. Lopez, Generalized Intersection Searching Problems, *Internat. J. Comput. Geom. Appl.* **3** (1993), 39–70.

[20] J. Matoušek, Range Searching with Efficient Hierarchical Cuttings, *Discrete Comput. Geom.* **10** (1993), 157–182.

[21] K. Mehlhorn, *Data Structures and Algorithms* 3: *Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[22] J. Nievergelt and P. Widmayer, Guard Files: Stabbing and Intersection Queries on Fat Spatial Objects, *Comput. J.* **36** (1993), 107–116.

[23] M. H. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin, 1983.

[24] M. H. Overmars, H. Schipper, and M. Sharir, Storing Line Segments in Partition Trees, *BIT* **30** (1990), 385–403.

[25] F. P. Preparata, A Real Time Algorithm for Convex Hull, *Comm. ACM* **22** (1979), 402–405.

[26] F. P. Preparata and M. I. Shamos, *Computational Geometry—an Introduction*, Springer-Verlag, New York, 1985.

[27] N. Sarnak and R. Tarjan, Planar Point Location Using Persistent Search Trees, *Comm. ACM* **29** (1986), 609–679.

[28] H. W. Scholten and M. H. Overmars, General Methods for Adding Range Restrictions to Decomposable Searching Problems, *J. Symbolic Comput.* **7** (1989), 1–10.

[29] M. Sharir and P. Agarwal, *Davenport–Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, New York, 1995.

[30] R. E. Tarjan, Amortized Time Complexity, *SIAM J. Discrete Math.* **6** (1985), 306–318.

[31] D. E. Willard and G. S. Lueker, Adding Range Restriction Capability to Dynamic Data Structures, *J. Assoc. Comput. Mach.* **32** (1985), 597–617.