Connected Component and Simple Polygon Intersection Searching

P.K. Agarwal and M. van Kreveld

UU-CS-1994-10
February 1994

Connected Component and Simple Polygon Intersection Searching

P.K. Agarwal and M. van Kreveld

Technical Report UU-CS-1994-10
February 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Connected Component and Simple Polygon Intersection Searching[*]

Pankaj K. Agarwal[†] Marc van Kreveld[‡]

Abstract

Efficient data structures are given for the following two query problems: 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 $O(n^{1+\epsilon})$ that can answer a query in time $O(n^{1/2+\epsilon}+k)$, 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 $O(n^{1/2+\epsilon})$ and $O(n^{1/2+\epsilon} + k)$ respectively. If we allow $O(n^{4/3+\epsilon})$ space, the amortized update and query time can be improved to $O(n^{1/3+\epsilon})$ and $O(n^{1/3+\epsilon} + k)$, respectively. For orthogonal segments the amortized update and query time are $O(\log^2 n)$ and $O(\log^2 n + k \log n)$. Some other related results are also mentioned.

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 a 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 done to date 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

[*]Part of this work was done while the second author was visiting the first author at Duke University on a grant of 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.

[†]Computer Science Department, Duke University, Durham, NC 27706, U.S.A.

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

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 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 the applications, one can process the set of simple objects that define the input objects for intersection searching, 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 these problems where the input objects are of not simple shape. We assume that the objects are defined by segments and the query is again a segment. We study the following 'abstract' problem, which we call the *colored segment intersection* problem:

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

By coloring the segments of each object with the same color, we can reduce the original intersection searching problem to the colored segment intersection problem. The following examples illustrate the 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; we just want to report the indices of these polygons. By assigning the color i to the edges of P_i, one can reduce this problem to colored segment intersection searching.

2. **Connected component intersection searching:** The *connected components* in the arrangement of a set $\mathcal{S}$ of segments are the connected components of the planar graph formed by $\mathcal{S}$. More formally, two segments of $\mathcal{S}$ are in the same connected component if there is a path between them along the edges of the arrangement of $\mathcal{S}$. The connected components form a partition $\mathcal{S}^1, \ldots, \mathcal{S}^m$ of $\mathcal{S}$. We want to preprocess $\mathcal{S}$ so that the connected components of $\mathcal{S}$ intersecting a query segment can be reported quickly. Again for each $\mathcal{S}^i$, we do not want to report all the segments of the connected components; 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 $\mathcal{S}$ 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. [9]. The *segment intersection searching problem,* where 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 [16], Gupta et al. [14], and Nievergelt and Widmayer [19].

In the last few years much work has been done on the segment intersection searching problem [2, 3, 5, 7, 10, 13, 21]. Agarwal and Sharir [3] showed that $\mathcal{S}$ can be preprocessed using $O(n^{1+\epsilon})$ space and time, so that all k segments of $\mathcal{S}$ intersecting a query segment can be reported in time $O(n^{1/2+\epsilon} + k)$.[1] (The n^ϵ factor in the size and query time can be reduced to $\log^{O(1)} n$ factor using a more sophisticated partition tree due to Matoušek.) Roughly speaking it stores a family of subsets of $\mathcal{S}$, called *canonical subsets*, so that the segments of $\mathcal{S}$ intersecting a query segment can be represented as the union of $O(n^{1/2+\epsilon})$ pairwise disjoint canonical subsets, and they can be computed in $O(n^{1/2+\epsilon})$ time. One can easily extend this algorithm 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 $O((1+k)n^{1/2+\epsilon})$, which is much worse than the desired bound of $O(n^{1/2+\epsilon} + k)$.

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 [15]. 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 [15]. None of these algorithms work for reporting the connected components that intersect a query segment. If the query is a line, one can answer a connected component intersection query as follows (see also Janardan and Lopez [16]): For each connected component of $\mathcal{A}(\mathcal{S})$, find its convex hull and preprocess the convex hull edges for segment intersection searching. Since a connected component of $\mathcal{A}(\mathcal{S})$ 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 $\mathcal{S}$ intersecting a query line can be reported in time $O(n^{1/2+\epsilon} + k)$. Observe that the above approach works for colored segment intersection searching (assuming that the query is again a line) too as long as the segments of the same color form a connected graph. Thus one can also answer polygon intersection queries for lines using the same approach. However this approach does not work when query is a segment.

In this paper we begin with a relatively simple algorithm for colored segment intersection searching for the case where 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 [16].

In Section 3, we present a dynamic solution for colored segment intersection searching for the case where the segments of $\mathcal{S}$ 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,

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

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 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 connected component searching problem for arbitrary segments. For any fixed $\epsilon > 0$, the preprocessing time is $O(n^{4/3+\epsilon})$, the space is $O(n^{1+\epsilon})$ and a query takes $O(n^{1/2+\epsilon} + k)$ time, where k is the output size. One can improve the query time by increasing the size of the data structure, as in standard segment intersection searching. In particular, for a parameter $n \le N \le n^2$, one can answer a query in time $O(\frac{n^{1+\epsilon}}{N} + k)$ using $O(N^{1+\epsilon})$ 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 for connected components can be modified so that new segments can be inserted efficiently. The semi-dynamic structure we obtain allows for insertions in $O(n^{1/2+\epsilon})$ amortized time. The amortized query time of the new structure is $O(n^{1/2+\epsilon} + k)$. The insertion of a segment is fairly expensive because before inserting a segment we need to determine the connected components that it intersects, which comes down to answering a query. If we allow the size of the data structure to be $O(n^{4/3+\epsilon})$, the update and query time can be improved to $O(n^{1/3+\epsilon})$ and $O(n^{1/3+\epsilon}+k)$. This leads to an $O(n^{4/3+\epsilon})$ 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 $\mathcal{S}$ such 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 small number of times, usually constant. This approach assures that no color is reported often, and thus query time will be low. An interesting feature of our dynamic algorithms is the *lazy* update of data structures. Part of the work is performed only later by the query algorithms. However, a query is answered correctly at all times. The idea is reminiscent of the union-find structure with path compression [8]. 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 Problem

In this section we consider the colored segment intersection problem for orthogonal segments, i.e., given a set $\mathcal{S}$ of orthogonal segments and a color assignment $\chi : \mathcal{S} \to \{1, \ldots, m\}$, we preprocess $\mathcal{S}$ into a data structure so that the set of colors of the segments intersected by a query (orthogonal) segment γ can be reported efficiently. If γ is vertical (horizontal), then it suffices to search among the horizontal (resp. vertical) segments of $\mathcal{S}$. As a result, we can preprocess horizontal and vertical segments separately. To find the horizontal segments that intersect a horizontal query segment, we can use the same structure. We query with the endpoints of the query segment in the structure that stores horizontal segments, which is possible because a point is just a degenerate vertical segment. We also need a data structure on the endpoints of the horizontal segments, but this is just a simple variation of the structure to be described next, so we omit the description. Similarly, we can find the vertical segments intersecting a vertical query segment. We will only describe how to preprocess a set $\mathcal{S}$ of horizontal (colored) segments for queries with vertical segments.

For simplicity, we assume that all horizontal segments of $\mathcal{S}$ have different y-coordinates, and that they are sorted in increasing order of their heights. We construct a balanced binary tree T on the segments of $\mathcal{S}$. The i^{th} leftmost leaf of T is associated with the i^{th} segment of $\mathcal{S}$. For each node v of T, let $\mathcal{S}_v \subseteq \mathcal{S}$ denote the set of segments associated with 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 $\ell_v : y = y_v$ with v. At each node other than the root of T, we store the secondary structure described below.

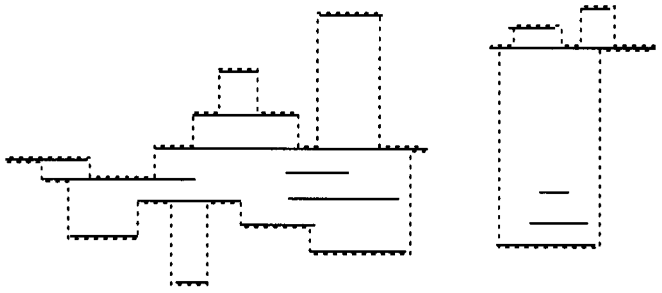

Figure 1: Upper and lower envelopes of a set of horizontal segments.

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 (not necessarily continuous) linear 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 sky-line* (see Figure 1),[2] and every *break-point* 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 non-empty subsets of node v. Clearly $m_v \leq n_v$. Assume that v is a left child of its parent. For each i, we 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 with a different color. Let $x_1 < \cdots < x_r$ be the x-coordinates of all break-points of the histograms in U_v. We have $r \leq 2n_v$. The vertical ordering of the histograms in U_v remains the same in every interval (x_j, x_{j+1}), and changes only at the break-points. We store the vertical ordering of U_v for all x-coordinates, using the persistent data structure of Sarnak and Tarjan [24]. 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 which has its topmost endpoint above all histograms, we can report all k histograms that intersect e in time $O(\log(m_v + r) + k)$. In more detail, we compute the persistent data structure by sweeping a vertical line from left to right. At each break-point we stop and update the structure. Assume that the histogram U_v^i has a break-point at some x_j. Then we perform the following two operations:

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

(i) If U_v^i was 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 tree using the y-coordinate of U_v^i at (x_j, x_{j+1}) as the key.

Since $r, m \leq 2n_v$, 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 store a similar secondary structure at v, but replace the upper envelopes U_v^i with 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 ℓ_z intersects γ. 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 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 $\mathcal{S}$ intersecting γ.

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

Theorem 2.2 *A set $\mathcal{S}$ of n orthogonal line segments in the plane, and a color assignment $\chi : \mathcal{S} \to \{1, \ldots, m\}$ of $\mathcal{S}$, can be preprocessed in time $O(n \log^2 n)$, into a data structure of size $O(n \log n)$, such 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 $O(n \log n)$ [15], 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 $O(n \log n)$ into a data structure of size $O(n \log n)$, such 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 $O(n \log n)$, such that all k polygons intersecting an orthogonal query segment can be found in $O(\log n + k)$ time.*

3 Dynamic Orthogonal Colored Segment Intersection

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 ($\mathcal{S}$, γ): 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 only consider the situation where $\mathcal{S}$ consists of only horizontal segments, 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. Mehlhorn [18, pp. 192–199]. 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. We maintain three secondary structures on $\mathcal{B}_v$.

(i) TCL_v: It stores the set of colors of intervals in $\mathcal{B}_v$ as a balanced binary search tree (e.g. red-black tree). Each node of the tree, storing the color i, in turn stores two lists L_v^i and R_v^i of left resp. right endpoints of intervals of color i. They are also stored as balanced binary search trees.

(ii) TL_v: A balanced binary tree on the colors of intervals of $\mathcal{B}_v$.

(iii) TR_v: A balanced binary tree on the colors of intervals of $\mathcal{B}_v$.

We perform three operations on the interval tree: (i) INSERT (b, T, i), (ii) DELETE (b, T, i), and (iii) REPORT (x, T). The third operation reports the colors of intervals that intersect 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 [18] 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 (or r) is the leftmost (resp. rightmost) endpoint of the intervals in $\mathcal{B}_v$ of color i, we update TL_v (resp. TR_v).

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 [18]). An interval is deleted in $O(\log n)$ time using a similar approach. We leave it to the reader to fill in the details.

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. Furthermore, a binary search tree supports the max-operation in constant time, and 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 all k colors of intervals containing a query point can be reported in time $O((k + 1) \log n)$.*

Semi-dynamic case: If we perform only insert operations, the secondary data structure 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 $\mathcal{B}$ at each node of T, 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 should 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 answered in the same way as earlier except that we perform an additional step. If we find t intervals $b_1, \ldots, b_t$ of the same color, say i, we delete them from the corresponding secondary structures. Let $b = \bigcup_{j=1}^{t} b_j$; b is a single connected interval, because $x \in b_j$ for all $j \leq t$. We insert b into T. If the above procedure returns k' intervals of k distinct colors, then the actual running time for reporting and updating is

$$O(k' + \log n) + O((k' - k + 1)\log n) = O(\log n + k) + O((k' - k)\log n)\,.$$

Since there are at most twice as many deletions as there are insertions, and any interval can be deleted at most once, we charge the $O(\log n)$ time spent in deleting an interval to its insertion, so the amortized insert and query time are $O(\log n)$ and $O(\log n + k)$, respectively. Hence, we have

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 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. [25, 27]. Basically, this comes down to maintaining a balanced binary tree on the y-coordinates of the segments, and every node 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 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 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 $\mathcal{A}(S)$, because insertion of a segment can merge several connected components into one. Therefore, if we label the segments in the i^{th} connected component by color i, then the insertion of segment can change the colors of many segments explicitly. It will be very expensive to update the colors of all these segments. 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 ($\mathcal{S}$, γ): 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 i^{th} connected component of the arrangement. Let $\mathcal{S}^i \subseteq \mathcal{S}$ denote the set of segments in the i^{th} connected component of $\mathcal{S}$ (or the segments of color i). The connected components now change dynamically as we insert segments. Since it will be very expensive to change the color of every segment explicitly, we will 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 [8]. Throughout this section, we will use $UF(\mathcal{S})$ to find the color of a segment, and by the phrase 'find the color of a segment e' we will 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 semi-dynamic case in Section 3.1. 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 implies that the amortized query time is $O(\log^2 n + k \log n)$. 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 actually 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 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 [15].

Corollary 4.2 *There exists an on-line algorithm 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 $\mathcal{S}^1, \dots, \mathcal{S}^m$ 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 colored segment intersection queries. However, unlike in the colored segment intersection searching problem, where the color of each segment is given, we have to compute the color of each segment in $\mathcal{S}$.

Let $G(\mathcal{S})$ be a graph with n vertices $\{1, \dots, n\}$ 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 $e_i, e_j \in S$ 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 $G(\mathcal{S})$ [12]. 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. [4], 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 [23] 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 L_u, and the right bounding line R_u. A segment $e \in \mathcal{S}$ is associated with a node u if $b_u \subseteq \bar{e}$ and $b_{\mathrm{parent}(u)} \not\subseteq \bar{e}$, 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 $\mathcal{S}_u, \mathcal{E}_u$ will be referred to as *long* and *short* segments, respectively. We have:

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

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 at node u

Let $\mathcal{S}_u^1, \mathcal{S}_u^2, \dots, \mathcal{S}_u^t$ 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 $\mathcal{S}_u$. 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 $\mathcal{R}_u = \{r_u^1, \dots, r_u^t\}$, 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 associated with $\mathcal{S}_u$, TA_u, is a minimum height binary tree on the segments of $\mathcal{R}_u$; the i^{th} 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 i^{th} 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 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 stores $\mathcal{S}_u^{i*}$ that determines whether a query double-wedge W contains any point of $\mathcal{S}_u^{i*}$; see [17]. 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 also preprocess $\mathcal{S}_u^*$, the set of points dual to the lines supporting $\mathcal{S}_u$ in time $O(n_u^{4/3+\epsilon})$ into a data structure TB_u of size $O(n_u^{4/3+\epsilon})$, 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 [17]. 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.

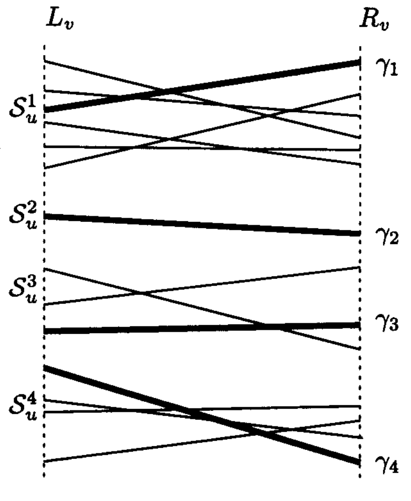

Figure 2: The long segments $\mathcal{S}_u$, bold edges denote the segments of $\mathcal{R}_u$.

5.2 Preprocessing short segments at node u

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 $\mathcal{E}_u$. 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 [3]), 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. However, observe that 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). Moreover, 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.

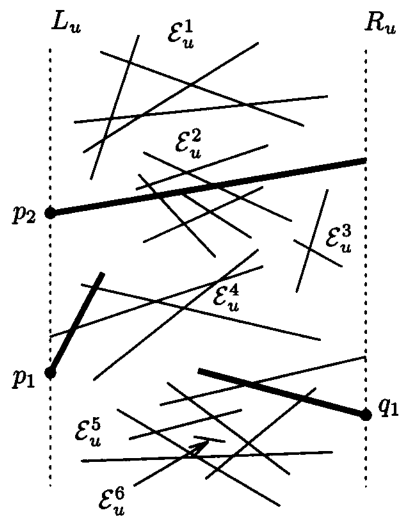

Figure 3: Short segments; $\mathcal{F}_u^L = \{\mathcal{E}_u^2, \mathcal{E}_u^4\}$, $\mathcal{F}_u^R = \{\mathcal{E}_u^5\}$, and $\mathcal{F}_u^M = \{\mathcal{E}_u^1, \mathcal{E}_u^3, \mathcal{E}_u^6\}$; bold segments are leaders; $\lambda(\mathcal{E}_u^2) \notin \mathcal{E}_u$.

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

- $\mathcal{F}_u^L$: A connected component $\mathcal{E}_u^i \in \mathcal{F}_u^L$ if a segment $e_i \in \mathcal{E}_u^i$ intersects the left boundary L_u of I_u, or if there is a segment $e \in S$ that intersects $\mathcal{E}_u^i$ as well as L_u. It is easily seen that, in the latter case, $e \in S_z$ for some ancestor z of u. We will 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 $\mathcal{E}_u^i$.
- $\mathcal{F}_u^R$: 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 $\mathcal{F}_u^M$ are also the connected components of the whole set 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 which has not be 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 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.

After having computed $\mathcal{F}_u^L, \mathcal{F}_u^R$, and $\mathcal{F}_u^M$, we process each of them separately, so that the colors of the segments of each subset intersected by 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 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 $CH(\mathcal{B}_{uv}^c)$. Set $\chi(E_{uv}^c) = c$, $E_{uv} = \bigcup_c E_{uv}^c$, and $m_{uv} = |E_{uv}|$.

  ⋆ 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 [3]. All k segments of E_{uv} intersected by a query line can be reported in time $O(m_{uv}^{1/2+\epsilon} + k)$. It constructs a two-level partition tree of which each node stores a subset of segments, and the query output consists of $O(m_{uv}^{1/2+\epsilon})$ canonical subsets. For each canonical subset, we store the set of colors of these segments instead of the segments themselves.

  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 $E_u = \bigcup_{\mathcal{E}_u^i \in \mathcal{F}_u^M} E_u^i$. 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 $O(n^{4/3+\epsilon})$, for any fixed $\epsilon > 0$.

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, and 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 γ intersects a segment e of 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, or*

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. First, we query TA_u for all nodes $u \in \pi_p \cup \pi_q$ with $\gamma \cap I_u$, and then we query with ℓ in TC_u, TD_u and TE_u for all $u \in U(\gamma)$.

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 $\hat{\gamma}$. 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 with the endpoints $\hat{p}$ and $\hat{q}$ of $\hat{\gamma}$ in TA_u, we compute $O(\log n)$ maximal subtrees of TA_u that lie between the search paths to $\hat{p}$ and $\hat{q}$. Assume without loss of generality that $\hat{p}$ lies below $\hat{q}$. 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 long segments is $O(n^{1/2+\epsilon} + k_l \log^2 n)$, where k_l is the number of 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 descendants v of the nodes on the path from the root of TC_u to z, partition the connected components of $\mathcal{F}_u^L$ into $O(\log n)$ canonical subsets such that, either σ lies below p_j for all $\mathcal{E}_u^j \in \mathcal{F}_{uv}^1$, or it lies above all of them. It is easily seen that ℓ intersects a segment of $\mathcal{B}_{uv}^c$ if and only if ℓ intersects an edge of E_{uv}^c. Therefore we query TCP_{uv} and report the colors of all segments of E_{uv} that intersect ℓ. At each node v, a color is reported at most twice, so the time spent at v is $O(m_u^{1/2+\epsilon} + k_{uv})$, where k_{uv} is the output size. Next, we repeat a similar procedure for TD_u with 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 $O(n^{1/2+\epsilon} + k_s \log^2 n)$, where k_s is the number of colors found. The overall query time is thus $O(n^{1/2+\epsilon} + k \log^2 n)$.

Lemma 5.2 *A set of n line segments in the plane can be preprocessed in $O(n^{4/3+\epsilon})$ time and space 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 $O(n^{1/2+\epsilon} + k \log^2 n)$ time, for any fixed $\epsilon > 0$.*

5.4 Improving the running time

The query time of the above lemma can be improved to $O(n^{1/2+\epsilon} + k)$ at the expense of slight increase in the size of the data structure. The space used will be $O(n^{1+\epsilon})$. The basic idea is to replace all binary trees in the above structure with n^δ-ary trees, for some small constant $0 < \delta < \epsilon/4$.

To this end, we replace the binary tree TA_u by a minimum height n_u^δ-ary tree on the segments in $\mathcal{R}_u$. The height of TA_u is now $O(1/\delta)$. For an internal node $v \in TA_u$, let $w_1, \ldots, w_t$ $(t \leq n_u^\delta)$ be the children of v. For each $1 \leq i \leq j \leq t$, let $C_{uv}(i,j)$ denote the set of colors of segments in $\bigcup_{i \leq h \leq j} \mathcal{R}_{uw_h}$. 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 γ, we locate its endpoints, in time $O(\frac{1}{\delta} \log n_u^\delta) = O(\log n_u)$, among the segments of $\mathcal{R}_u$. Let π_p, π_q be the same as defined before. 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 $\bigcup_{i \leq h \leq j} \mathcal{R}_{uw_h}$ intersect γ, we now report the colors stored in $C_{uv}(i,j)$. We leave it to 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, one can show that colors of segments in $\mathcal{E}_u$ intersected by a long segment γ can be reported in time $O({m_u}^{1/2+\epsilon} + k)$, 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 $w_1, \ldots, w_t$, $t \leq n^\delta$, denote the children of u. For each $1 \leq i \leq j \leq t$, let the strip $I_u(i,j) = \bigcup_{i \leq h \leq j} I_{w_h}$. 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 $O(n^{1+4\delta} \log^3 n) = O(n^{1+\epsilon})$. Since the depth of each tree is constant, a color will be reported only a constant number of times. Therefore we can conclude

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

Remark 5.4: Using the space/query time tradeoff for simplex range searching and line intersection data structures [3, 6], one can obtain a space/query-time tradeoff for Theorem 5.3. In particular, for any $n \leq N \leq n^2$, one can preprocess $\mathcal{S}$ into a data structure of size $O(N^{1+\epsilon})$, so that all k connected components intersected by a query segment can be reported in time $O(\frac{n^{1+\epsilon}}{\sqrt{N}} + k)$. The preprocessing time is $O(n^{4/3+\epsilon})$ for $N < n^{4/3}$ and $O(N^{1+\epsilon})$ for $N \geq n^{4/3}$.

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 only wish to return the indices 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 $\mathcal{E}_u^i$. Hence, 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). However, we have to preprocess long segments into a different data structure, because, unlike the previous section, segments with different colors may intersect each other.

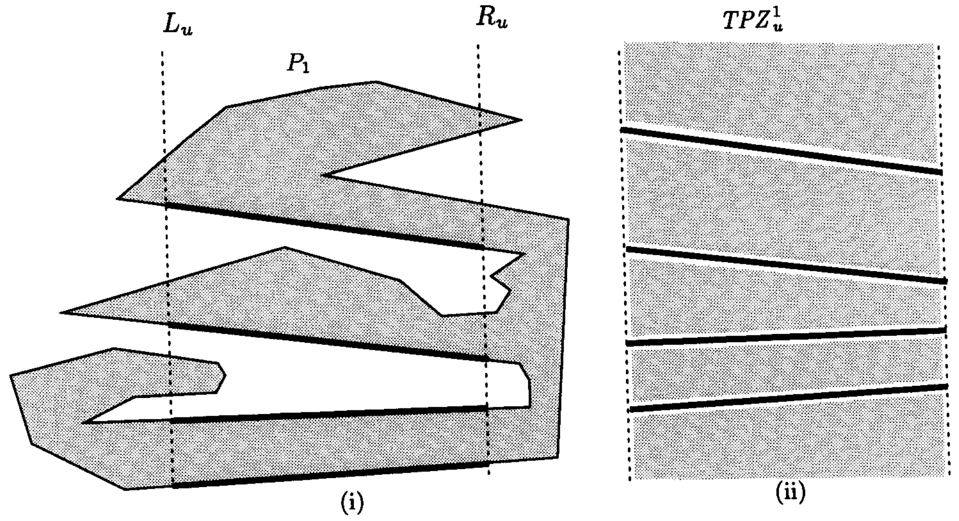

Figure 4: (i) Partitioning of polygons into long and short segments; bold segments are long segments; (ii) TPZ_u^1.

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, two of which are unbounded. For the sake of convenience we add two edges, one at $-\infty$ and another at $+\infty$, so that all trapezoids in TPZ_u^i become 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. We can determine whether the endpoints of $\hat{\gamma}$ lie in different trapezoids of TPZ_u^i by binary search, but we cannot afford to do binary search for each color separately. Instead, we set $TPZ_u = \bigcup_i TPZ_u^i$, and construct a two-level data structure TA_u to report the trapezoids that contain the upper endpoint of $\hat{\gamma}$, but not the lower endpoint of $\hat{\gamma}$. For any color, there is precisely one trapezoid 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 Matoušek [17]. This algorithm constructs a partition tree on $\mathcal{S}_u^*$, each of whose node v 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 $O({n_u}^{1/2+\epsilon})$ 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^i_{uv} denote the set of resulting segments and $E_{uv} = \bigcup_i E^i_{uv}$. Let E^*_{uv} denote the set of points dual to the lines supporting the segments of 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 of w.

The standard analysis for multi-level partition tree implies that TA_u requires $O(n_u \log n_u)$ space and time. The total time spent in preprocessing long segments over all nodes of T is thus $O(n^{1+\epsilon})$, for any constant $\epsilon > 0$.

6.2 Answering a query

Let $\gamma = \overline{pq}$ be a query segment. We only have to describe how to report the trapezoids of TPZ^i_u, for some node $u \in \pi_p \cup \pi_q$, that contain the upper endpoint of $\hat{\gamma} = \gamma \cap I_u$. 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 $O(n_u^{1/2+\epsilon})$ canonical subsets such that all points in a canonical subset lie in γ^* (which correspond to trapezoids whose bottom edges intersect $\hat{\gamma}$). Let S^*_{uv} be a canonical subset of the query output. A trapezoid τ whose bottom edge is a segment corresponding to a point of S^*_{uv} contains the upper endpoint of γ, if γ does not intersect the top edge of τ. This can be accomplished by querying the second level structure of v with γ^* and reporting the colors of segments corresponding to the points in S^*_{uv} that do not lie in γ^*. Since each color is reported at most once, all k colors of segments in S_u intersecting $\hat{\gamma}$ can be reported in time $O({n_u}^{1/2+\epsilon} + k)$.

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

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

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

Remark 6.3: (i) As in the previous section, one can obtain a space/query-time tradeoff by using the standard techniques; see [3, 6].

(ii) The above algorithm returns only those polygons whose boundaries intersect a query segment. If one also wants to report the segments 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 $O(n^{1/2+\epsilon} + k)$ all k polygons corresponding to triangles that contain the query point.

7 Maintaining the Connected Components of Arbitrary Segments

Next we describe a semi-dynamic data structure for maintaining the connected components in the arrangement of a set $\mathcal{S}$ of arbitrary line segments in the plane under insertions. The data structure supports the following three operations:

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

REPORT-COMPONENT $(\gamma, \mathcal{S})$: Report (the label of) all connected components in $\mathcal{S}$ 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 will dynamize the static data structure described in Section 5 using the ideas of Section 4 and some new ideas. As stated in Introduction, one of the main features of the semi-dynamic data structure is the 'lazy' INSERT operation — a lot 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 will show that the amortized query time (over a sequence of INSERT and REPORT-COMPONENT operations) is close to its static counterpart. We will analyze the amortized update and query time using the so called *accounting method*, see [8, 26]. Translated to our application, it means that when we insert a new segment γ into $\mathcal{S}$, certain units of cost are assigned to γ as *credits*. Part of the actual cost of subsequent operations may be paid by these credits. If x credits of an operation are charged to a segment γ, then the credit of γ reduces by x. The credit of each segment must always be nonnegative. The amortized insertion time of a new segment γ is the difference in the actual time and the cost paid by credits, plus the credits assigned to γ, and the amortized time for a query is the difference of the actual running time and the cost paid by credits.

This section is organized as follows. First, we 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 for long segments completely, with the insertion and query procedures. 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, where the query object is a line. We use a variant of this structure for the secondary structures for short segments. The time complexity of update and query procedures is analyzed in Section 7.4.

7.1 The global solution

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 on the segments of $\mathcal{S}$, and 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, 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 of the differences 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 in Mehlhorn [18, pp. 212–221]. As before, 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 segments short at u by $\mathcal{E}_u$, and the set of segments long at u by $\mathcal{S}_u$. All the segments in $\mathcal{E}_u$ and $\mathcal{S}_u$ are clipped within the vertical strip I_u.

Another difference with the static structure, but analogous to the data structure for maintaining the connected components of orthogonal segments described in Section 4, we will use a union-find data structure $UF(\mathcal{S})$ on $\mathcal{S}$ to maintain the colors of segments in $\mathcal{S}$. We will 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 will be 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. As in Section 5.3, 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. $q \in I_z$). Let $U(\gamma)$ be the set of nodes u such that $\bar{\gamma}$ is long at u, but not at the parent of u (where $\bar{\gamma}$ is the x-projection of γ. 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 will be 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 larger component. 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 $2 \leq i \leq k$.

The actual insertion of $\gamma = \overline{pq}$ into T is performed as described in Mehlhorn [18, pp. 212–221]. 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

This section describes the structures, insertions and queries for the structures TA_u and TB_u, which store the subset $\mathcal{S}_u$ of segments that are long at a node $u \in T$.

7.2.1 The long segment structure

As in Section 5.1, we construct a balanced binary tree TA_u on the representatives of each connected component of $\mathcal{A}(\mathcal{S}_u)$. Since the colors of representatives change dynamically, instead of storing the colors of $\mathcal{R}_{uv}$, 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 we will 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. At each leaf of TA_u, storing the representative r_u^i, we store $\mathcal{S}_u^i$ in a dynamic data structure $TAQ(\mathcal{S}_u^i)$ for segment intersection detection queries, see [3]. The query time is $O(n_{ui}^{1/2+\epsilon})$ and amortized update time for $TAQ(\mathcal{S}_u^i)$ is $O(n_{ui}^{\epsilon})$, 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 [3, 17]. 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 will 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 $O(n_u^{1/2+\epsilon})$ and the update time is $O(n_u^{\epsilon})$. This structure will be 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

We update the secondary structures TA_u and TB_u as follows. Let $\gamma = \overline{pq}$ be the segment to be inserted. We first show that a representative segment can be inserted into or deleted from TA_u efficiently. Using this as a subroutine, we describe the actual insertion of γ.

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} at the front of the list in $O(1)$ time. A segment is deleted 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 $\hat{\gamma}$ and find the set $\mathcal{R}(\gamma) \subseteq \mathcal{R}_u$ of all representatives that intersect $\hat{\gamma}$. Let γ^b be the segment of $\mathcal{R}_u$ lying immediately below the lower endpoint of $\hat{\gamma}$. 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 $\hat{\gamma}$.

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

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

Algorithm 1: MERGE TAQ-STRUCTURE

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
```

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

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 partitioned the segments of $\mathcal{E}_u$ into the connected components $\mathcal{E}_u^1, \ldots, \mathcal{E}_u^t$ inside I_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 each $\mathcal{A}(\mathcal{E}_u^i)$ 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 will 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 $\mathcal{F}_u^L$ satisfies the same condition as in Section 5.2, i.e., either there is a segment $e_i \in \mathcal{E}_u^i$ that intersects the left boundary L_u of I_u, or 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 copy of it to $\mathcal{E}_u^i$. All groups in $\mathcal{F}_u^R$ intersect the right boundary of I_u. However, some of the groups satisfying these conditions may be in $\mathcal{F}_u^M$. Intuitively, the reason is that 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 of $\mathcal{F}_u^M$ and to move them to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$. Instead, we wait until TE_u is visited by the query procedure. If the query procedure detects that certain groups of $\mathcal{F}_u^M$ can be moved to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$, it will do so. What we gain is that the query procedure can detect this for free, whereas the insertion would require a lot of work for detection, even if no groups can be 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 $\mathcal{E}_u$, or the connected components containing them touch the boundary of I_u.

This section begins, however, with 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 segment $\mathcal{E}^\circ$, 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 $\mathcal{E}_u^i$; 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:

Semi-dynamic convex hull structure: For each i, we maintain the convex hull of $\mathcal{E}^i$ using the algorithm of Preparata [22]. 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 $\phi(g)$. Let $E = \bigcup_{i=1}^m E^i$.

Dynamic partition tree: Using the algorithm of Agarwal and Sharir [3], we maintain the edges of E into a two-level dynamic partition tree $\Psi(E)$ that can report all segments intersected by a query line. Moreover, a segment can be inserted into or deleted from $\Psi(E)$ in time $O(n^\epsilon)$, and all k segments of E intersected by a query line can be reported in time $O(n^{1/2+\epsilon} + k)$. 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 [3] that GROUP-INSERT $(\mathcal{E}^\circ)$ requires only $O(|\mathcal{E}^\circ|^{1+\epsilon})$ time. Next, let ℓ be a query line such that ℓ intersects $CH(\mathcal{E}^i)$, for any $i \leq m$, if and only if ℓ 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 $\phi(g)$. It is easily seen at most two segments of a group are reported, so the total time spent in reporting k groups is $O(n^{1/2+\epsilon} + k)$.

Finally, suppose we want to merge $\mathcal{E}^i, \mathcal{E}^j$. Without loss of generality, assume that $|\mathcal{E}^i| \leq |\mathcal{E}^j|$. 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 $h_1, \ldots, h_r$, $r \leq 4|\mathcal{E}^i|$, 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 $h_l, \ldots, h_r$ into $\Psi(E)$. The total time spent is obviously $O((t + r)n^\epsilon)$.

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 maintained in a data structure of size $O(n \log n)$, so that* GROUP-MERGE *operations can be performed in $O(n^\epsilon)$ amortized time, a set $\mathcal{E}^\circ$ can be inserted in $O(|\mathcal{E}^\circ|^{1+\epsilon})$ time,* GROUP-REPORT *can be performed in $O(n^{1/2+\epsilon} + k)$ 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 are obvious 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 $O(n^\epsilon)$ 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 the GROUP-INSERT or by the GROUP-MERGE) introduces at most 4 new edges in E. Assuming that a segment is inserted d times, we can pay for all updates in $\Psi(E)$ if we assign $8 \cdot c \cdot d \cdot n^\epsilon$ 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 a segment is inserted by the merge operation, the size of the set containing it at least doubles. This implies that e will be inserted into convex hull data structures at most $\log(n + h)$ times, thus $d \leq \log(n + h)$. Hence, the amortized cost of an insert operation is $O(n^\epsilon) + 8c\log(n + h)n^\epsilon = O(n^{\epsilon'})$, where ϵ' is another but arbitrarily small positive constant. This completes the proof of the lemma. $\square$

7.3.2 A dynamic structure for TCP_{uv} and TDP_{uv}

We will 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, TD_u (see Section 5.2). We only describe TCP_{uv} below, TDP_{uv} is completely analogous. The structure stores a set $\mathcal{B}_{uv}$ of segments, and its partitioning $\mathcal{B}^1_{uv}, \ldots, \mathcal{B}^s_{uv}$ into groups.

The structure TCP_{uv} has the following additional properties:

(i) Every segment e_i of $\mathcal{B}_{uv}$ 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 have the same color, then they can be merged, because the property that if a query line ℓ intersects $\mathcal{B}^i_{uv}$ if and only if ℓ intersects $CH(\mathcal{B}^i_{uv})$ will continue to hold after the merge.

TCP_{uv} supports the following operations:

COLOR-REPORT (TCP_{uv}, ℓ): Let ℓ be a line such that for any $i \le s$, ℓ intersects $CH(\mathcal{E}^i)$ if and only if it intersects $\mathcal{B}^i_{uv}$. Report all colors of segments intersecting ℓ.

GROUP-INSERT $(TCP_{uv}, \mathcal{B}^\circ)$: Create a new group $\mathcal{B}^{s+1}_{uv} = \mathcal{B}^\circ$.

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 $e_{h_1}, \dots, e_{h_2}$, such that there are at most two segments of any group. The second step identifies the groups $\mathcal{B}^{l_1}_{uv}, \dots, \mathcal{B}^{l_2}_{uv}$ that contain these segments by GROUP-FIND. The third step identifies the colors of $e_{h_1}, \dots, e_{h_2}$ using COLOR-FIND, and determines which groups of $\mathcal{B}^{l_1}_{uv}, \dots, \mathcal{B}^{l_2}_{uv}$ have the same color. Any two such groups are merged by GROUP-MERGE (i, j) in the fourth step, as described above.

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 7.3. The structure TD_u is completely analogous and will not be described.

We describe how to preprocess $\mathcal{F}^L_u$, the set of groups that have a connection to the left boundary line L_u. With a slight abuse of notation, let $\mathcal{E}^1_u, \dots, \mathcal{E}^t_u$ be the groups of $\mathcal{F}^L_u$, and let p_j be the left endpoint of the leader $\lambda(\mathcal{E}^j_u)$ of $\mathcal{E}^j_u$. Without loss of generality assume that $p_1, \dots, p_t$ are ordered from bottom to top. We define the weight of p_j to be the number of segments in $\mathcal{E}^j_u$. We construct a weighted balanced binary search tree TC_u on $p_1, \dots, p_t$. We define $\mathcal{F}^L_{uv}$ to be the set of connected components $\mathcal{E}^j_u$ such that p_j is stored in a leaf of the subtree rooted at v. Let

$$\mathcal{B}_{uv} = \{e \mid e \in \mathcal{E}^i_u \text{ and } \mathcal{E}^i_u \in \mathcal{F}^L_{uv}\}.$$

We maintain a partition $\mathcal{B}^1_{uv}, \dots, \mathcal{B}^s_{uv}$ of $\mathcal{B}_{uv}$ at v; all segments within each $\mathcal{B}^i_{uv}$ have the same color. Initially, the colors of all subsets 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 $\mathcal{B}^i_{uv}$ 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 $\Pi(\mathcal{F}^L_u)$ along with TC_u. This structure allows three operations:

SEG-INSERT $(\Pi(\mathcal{F}^L_u), \gamma)$ Inserts a segment γ into $\Pi(\mathcal{F}^L_u)$.

SEG-DELETE $(\Pi(\mathcal{F}^L_u), \gamma)$ Deletes a segment γ from $\Pi(\mathcal{F}^L_u)$.

SEG-DETECT $(\gamma, \mathcal{F}_u^L)$: 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 [3]. The SEG-INSERT and SEG-DELETE require $O(n_u^\epsilon)$ amortized time and the SEG-DETECT requires $O(n_u^{1/2+\epsilon})$ 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 a variant of the data structure of Section 7.3.1 to obtain a dynamic structure TE_u, which replaces its static version described in Section 5.2. TE_u has the following properties:

(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 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 both groups $\mathcal{E}_u^i, \mathcal{E}_u^j$ can be removed from $\mathcal{F}_u^M$.

TE_u supports the following three operations:

COLOR-REPORT (TE_u, ℓ): Let ℓ be a line such that for any $i \leq m$, ℓ intersects $CH(\mathcal{E}_u^i)$ if and only if it intersects $\mathcal{E}^i$. 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 $\mathcal{F}_u^M$.

To be able to perform these operations, we let the data structure TE_u consist of four structures, among which the three structures of $\Upsilon(\mathcal{F}_u^M)$ are as described above. The fourth structure is 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 $O(n_u^\epsilon)$, and the query time is $O(n_u^{1/2+\epsilon})$.

The GROUP-INSERT procedure is basically the same as the GROUP-INSERT for TCP_{uv} (cf. Section 7.3.2), with the extension that the new segments are also inserted into $\Pi(\mathcal{F}_u^M)$. The COLOR-REPORT (TE_u, ℓ) operation is performed in four steps; the first three steps are the same as 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 $\mathcal{F}_u^M$ of the same color that intersect ℓ. The fourth step tests whether $\mathcal{E}_u^i$ and $\mathcal{E}_u^j$ are in the same connected component of $\mathcal{F}_u^M$ using a procedure GROUPS-IN-SAME-COMPONENT (i, j). If so, they are merged by GROUP-MERGE (i, j) as described in Section 7.3.1. Otherwise, the smaller one is deleted by GROUP-REMOVE $(TE_u, \mathcal{E}_u^i)$ or GROUP-REMOVE $(TE_u, \mathcal{E}_u^j)$.

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 components $\mathcal{E}_u^h$, 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 preform the last step depends on $|\mathcal{E}_u^h|$. If $|\mathcal{E}_u^h| > |\mathcal{E}_u^i|$, then we start from the scratch, otherwise we test those segments of $\mathcal{E}_u^i \cup \mathcal{E}_u^h$, that we have not tested so far, 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. See Algorithm 2 for a more detailed description.

7.3.5 The short segment insertion

The INSERT-operation for the short segments is fairly simple, because it performs the updating of the structures TC_u, TD_u and TE_u only partially, and leaves the remainder of the work to be carried out by the subsequent REPORT-COMPONENT queries.

Let $\gamma = \overline{pq}$ be a segment that we want to add to $\mathcal{E}_u$. We create a new set $\mathcal{E}_u^{t+1} = \{\hat{\gamma}\}$. If $\hat{\gamma}$ does not intersect the left or right 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 $\hat{\gamma}$ 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 $\hat{\gamma}$. 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 $(TCP_{uv}, \{\hat{\gamma}\})$ — it creates a new set $\mathcal{B}_{uv}^{t+1} = \{\hat{\gamma}\}$, as described in Section 7.3.2. If $\hat{\gamma}$ intersects the right boundary of I_u, we perform similar actions on TD_u.

7.3.6 The query in short segments

We now have 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. Its main feature is the following. Whenever a color at some node of the secondary structure is reported more than once or twice, depending on the secondary structure, it either removes duplications or merges some groups to form larger groups.

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, which is a descendant of a node on the path from the root to z, we query the data structure TCP_{uv} constructed on $\mathcal{B}_{uv}$. We report the segments of $\mathcal{B}_{uv}$ intersected by ℓ using COLOR-REPORT (TCP_{uv}, 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, 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 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 $g \in \mathcal{S}_w$

Algorithm 2: Groups-in-Same-Component (i, j)

Input: $\mathcal{E}_u^i, \mathcal{E}_u^j$: two groups in $\mathcal{F}_u^M$ of the same color; $|\mathcal{E}_u^i| \leq |\mathcal{E}_u^j|$.
Actions: If $\mathcal{E}_u^i, \mathcal{E}_u^j$ are in the same component of $\mathcal{A}(\mathcal{E}_u)$, they are merged, and possibly other groups in that component. Otherwise, $\mathcal{E}_u^i$ is merged with all other groups in the component to which $\mathcal{E}_u^i$ belongs.
Output: *true* if $\mathcal{E}_u^i, \mathcal{E}_u^j$ are in the same component, *false* otherwise.

```
$Q = \emptyset$    (*Initialize a stack*)
for all $e \in \mathcal{E}_u^i$ do
    Seg-Delete $(\Pi(\mathcal{F}_u^M), e)$, push $(e, Q)$
end-for
while not empty $(Q)$ do
    $e =$ pop $(Q)$, $x =$ Seg-Detect $(\Pi(\mathcal{F}_u^M), e)$
    if an $x$ is returned then
        $h =$ Group-Find $(x, \mathcal{F}_u^M)$
        if $h = j$ then
            for all $e \in \mathcal{E}_u^i$ do
                Seg-Insert $(\Pi(\mathcal{F}_u^M), e)$
            end-for
            Group-Merge $(i, j)$
            return $true$
        end-if
        if $|\mathcal{E}_u^h| > |\mathcal{E}_u^i|$ then
            for all $e \in \mathcal{E}_u^i$ do
                Seg-Insert $(\Pi(\mathcal{F}_u^M), e)$
            end-for
            Group-Merge $(i, h)$
            return Groups-in-Same-Component $(i, j)$
        end-if
        for all $e \in \mathcal{E}_u^h$ do
            Seg-Delete $(\Pi(\mathcal{F}_u^M), e)$, push $(e, Q)$
        end-for
        Group-Merge $(i, h)$
    end-if
end-while
return $false$
```

Algorithm 3: Find-Leader

```
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 $L_u$.

$Q = \emptyset$
for all $e \in \mathcal{E}_u^i$ do
    push $(e, Q)$
end for

while not empty $(Q)$ do
    $e =$ pop $(Q)$, $z = u$
    while $z \neq$ NULL do
        $g =$ Empty-Double-Wedge $(TB_z, e)$
        if a $g$ is found then
            $\hat{g} = g \cap I_u$, $\lambda(\mathcal{E}_u^i) = \hat{g}$
            $\mathcal{E}_u^i = \mathcal{E}_u^i \cup \{\hat{g}\}$
            return $\hat{g}$
        else $z =$ parent $(z)$
        end-if
    end-while
end-while
```

for some ancestor w of u that intersects both $\mathcal{E}_u^i$ and L_u. For all ancestors w of u, 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 into $\Pi(TC_u)$ 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.

7.4 The analysis

As mentioned in the beginning of the section, we will use the accounting method to bound the amortized update and query time. We assign

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

credit 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 this credit among various copies of the segments stored in different secondary structures. For each copy of a segment e in the following structures, we assign the credits as shown in Table 1 (the constants $c_1, \ldots, c_4$ are chosen sufficiently large):

Algorithm 4: MOVE-TO-TC

```
Input:  A group $\mathcal{E}_u^i$ formerly in $\mathcal{F}_u^M$.
Actions:  Insert $\mathcal{E}_u^i$ into $TC_u$.

$p = \lambda(\mathcal{E}_u^i) \cap I_u$
INSERT ($p, TC_u$)
$z$ = leaf of $TC_u$ storing $p$
while $z \neq$ NULL do    (* $z$ has not passed the root *)
    GROUP-INSERT ($TCP_{uz}, \mathcal{E}_u^i$)
    $z$ = parent ($z$)
end-while
```

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

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

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

Proof: By the accounting method, the amortized insertion time is the actual insertion time, plus the 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 $O(n^\epsilon \log^2 n)$. Furthermore, the insertion time in all *TA*-structures is $O(n^\epsilon \log n)$ 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 $O(n^\epsilon)$ time, so the total charge to any segment is $O(n^\epsilon \log n)$. Since each segment in a *TAQ*-structure was assigned $c_2 n^\epsilon \log n$ credits when it was inserted (or when a subtree with it was last reconstructed), each segment has enough credits to pay for all insertions into other *TAQ*-structures if c_2 is chosen at least as large as the constant hidden in the $O(n^\epsilon)$ insertion time. Finally, during the insertion of e, $O(n^{1/2+\epsilon} \log^3 n)$ credits are assigned in total to all its occurrences.

If the secondary structures stored at each node of a dynamic binary search tree can be updated in time t, then the amortized update time of the overall data structure is $O(t \log n)$, see e.g. Mehlhorn [18]. It is straightforward to show that the construction time, inclusive of the distribution of credits to all the occurrences of segments, is $O(n^{3/2+\epsilon} \log^2 n)$.

Putting everything together and omitting all the straightforward details, we can conclude the amortized insertion time, excluding the time spent by the REPORT-COMPONENT procedure, is $O(n^{1/2+\epsilon}\log^3 n)$. □

We prove the amortized bound for REPORT-COMPONENT with the help of three lemmas that bound the amortized time in three of the 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 which k are distinct. Then the actual time taken by the query is $O(K)$, since the total time spent in processing each segment of TAP_{uv} is $O(1)$. Recall than 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 $O(n^{1/2+\epsilon}+k)$, where k is the number of different colors reported.*

Proof: The actual time for the first three steps of COLOR-REPORT is $O(n^{1/2+\epsilon}+K)$ time, 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 so 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 $O(n^{1/2+\epsilon}+k)$.

By Lemma 7.1, the cost of all GROUP-MERGE operations can be paid by charging $O(n^\epsilon \log n)$ 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 charged is $\Theta(K-k)$.

The total cost ever charged to each segment of $\mathcal{B}_{uv}$ is thus $O(n^\epsilon \log n)+O(1)=O(n^\epsilon \log n)$. If the constant c_3 in Table 1 is chosen 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 $O(n^{1/2+\epsilon}+k)$, where k is the number of different colors reported.*

Proof: As in the previous lemma, the time spent in reporting K groups with k different colors intersected by ℓ is $O(n^{1/2+\epsilon} + K)$ plus the time spent in the fourth step of the COLOR-REPORT procedure. We charge $O(K-k)$ and the time spent in the fourth step of the COLOR-REPORT to various segments of $\bigcup \mathcal{F}_u^M$, as described below, so the amortized cost of the procedure is $O(n^{1/2+\epsilon} + k)$.

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 so that each segment is charged only $\Theta(1)$ and it is charged only once, 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 will end up in a group of $\mathcal{F}_u^M$ at least twice the size as before the procedure. 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 $O(n^\epsilon \log n)$ to each segment of $\bigcup \mathcal{F}_u^M$. For each segment e in Q, we spent $O(n^{1/2+\epsilon})$ in SEG-DETECT, $O(n^\epsilon)$ time in SEG-DELETE, $O(n^\epsilon)$ time in SEG-INSERT, and another $O(1)$ time in other procedures. Hence, by charging $O(n^{1/2+\epsilon}\log n)$ to each segment, we can cover the cost of all calls to the GROUPS-IN-SAME-COMPONENT procedure.

Finally, if a group $\mathcal{E}_u^i$ is moved from $\mathcal{F}_u^M$ to $\mathcal{F}_u^L$, we spend $O(|\mathcal{E}_u^i| n^{1/2+\epsilon}\log n)$ time in finding the leader of $\mathcal{E}_u^i$ (cf. FIND-LEADER procedure), and $O(n^\epsilon \log n)$ time to create the new leaf z of TC_u that stores $\mathcal{E}_u^i$. $\mathcal{E}_u^i$ is also inserted into TCP_{uv} at all ancestors v of z. By Lemma 7.1, the amortized running time of the insertion procedure is $O(|\mathcal{E}_u^i| \cdot n^\epsilon)$. We also assign $c_3 n^\epsilon \log n$ credits to each segment of $e \in \mathcal{E}_u^i$ stored at v to fulfill the invariant that any segment inserted into TCP_{uv} has $c_3 n^\epsilon \log n$ 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 $O(n^{1/2+\epsilon}\log n)$ 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 $\mathcal{F}_u^M$ can go to $\mathcal{F}_u^L$ or $\mathcal{F}_u^R$ only once, and never go back. 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 $\mathcal{F}_u^L, \mathcal{F}_u^R$. 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:

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 $O(n^{1/2+\epsilon}\log^3 n)$, and the connected components of $\mathcal{A}(\mathcal{S})$ intersecting a new segment can be reported in time $O(n^{1/2+\epsilon}\log^2 n + k\log^2 n)$, and constant time suffices to determine whether two query segments of $\mathcal{S}$ are in the same component.*

Proof: The bound on the amortized running time of the REPORT-COMPONENT follows from the previous three lemmas. As for 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-COLOR is $O(n^{1/2+\epsilon})$. The amortized query time of REPORT-COLOR is $O(n^{1/2+\epsilon} + k\log^2 n)$, where k is the number of components reported. Since all these k components are merged into a single component, we can charge $\Theta((K-k)\log^2 n)$

cost to various segments of $\mathcal{S}$ in such a way so that each segment is charged at most $O(\log^2 n)$ (cf. Lemma 7.4). Hence, the total amortized cost of the insert procedure is $O(n^{1/2+\epsilon}) + O(n^{1/2+\epsilon} + k\log^2 n) - b \cdot k\log^2 n = O(n^{1/2+\epsilon})$, provided that the constant b is chosen sufficiently large. $\square$

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

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

Notice that update time is $O(n^{1/2+\epsilon})$ because of the call to REPORT-COMPONENT and assigning $n^{1/2+\epsilon}$ credit for searching segment intersection structures. 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 $O(N^{1+\epsilon})$ space, then a query can be answered in amortized time $O(\frac{n^{1+\epsilon}}{\sqrt{N}} + k)$ and the update time is $O(\frac{n^{1+\epsilon}}{\sqrt{N}})$ if $N \leq n^{4/3}$ and $O(\frac{N^{1+\epsilon}}{n})$ if $N > n^{4/3}$. Again we leave out the details. This implies an $O(n^{4/3+\epsilon})$ time on-line algorithm for computing the connected components of $\mathcal{A}(\mathcal{S})$.

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

8 Conclusions

In this paper we presented several data structures, static as well as semi-dynamic, for connected component intersection searching 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 some open problems:

(i) Can one answer the general colored segment intersection query for an arbitrary set of segments in time $O(n^{1/2+\epsilon} + k)$, 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 $O(n^{1/2+\epsilon} + k)$.

(ii) No efficient algorithm is known for the counting version of colored segment intersection searching, even for orthogonal segments. Recently, Gupta et al. [14] 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 because of the deletion of a segment.

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

References

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

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

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

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

[5] Bar Yehuda, R., and S. Fogel, Good Splitters with Applications to Ray Shooting, *Proc. 2nd Canadian Conference on Computational Geometry* (1990), pp. 81–85.

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

[7] Cheng, S. W., and R. Janardan, Space-Efficient Ray-Shooting and Intersection Searching: Algorithms, Dynamization and Applications, *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms* (1991), pp. 7–16.

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

[9] Dillencourt M., H. Samet, and M. Tammiuen, A General Approach to Connected Component Labeling for Arbitrary Image Representation, *JACM* **39** (1992), pp. 253–280.

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

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

[12] Guibas, L., *personal communication.*

[13] Guibas, L., 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.

[14] Gupta, J., R. Janardan, and M. Smid, Further Results on Generalized Intersection Searching Problems: Counting, Reporting and Dynamization, *3rd Workshop on Algorithms and Data Structures,* 1993, to appear.

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

[16] Janardan, R., and M. Lopez, Generalized Intersection Searching Problems, *Int. J. Comp. Geom. & Appl.* 3 (1993), 39–70.

[17] Matoušek, J., Efficient Partition Trees, *Proc. 7th Ann. Symp. on Comp. Geometry* (1990), pp. 1–9.

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

[19] Nievergelt, J. and P. Widmayer, Guard files: Stabbing and Intersection Queries on Fat Spatial Objects *The Computer Journal* **36** (1993), pp. 107–116.

[20] Overmars, M. H., *The Design of Dynamic Data Structures*, Lect. Notes on Comp. Science 156, Springer-Verlag, Berlin, 1983.

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

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

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

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

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

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

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