# Dynamic location in an arrangement of line segments in the plane

Olivier Devillers, Monique Teillaud, Mariette Yvinec

▶ **To cite this version:**

Olivier Devillers, Monique Teillaud, Mariette Yvinec. Dynamic location in an arrangement of line segments in the plane. Algorithms Review - newsletter of the ESPRIT II Basic Research Action Project no. 3075 (ALCOM) , 1992, 2 (3), pp.89-103. inria-00413506

**HAL Id: inria-00413506**

**https://inria.hal.science/inria-00413506v1**

Submitted on 4 Sep 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Dynamic Location in an Arrangement of Line Segments in the Plane*

**Olivier Devillers**[†] **Monique Teillaud**[†] **Mariette Yvinec**[‡]

**Abstract**

We describe in this paper an algorithm which dynamically maintains the trapezoidal map of an arrangement of line segments. This algorithm is a generalization of the Influence Graph data structure used in [1] to deal with the semi-dynamic case.

Our complexity results are randomized, i.e. the insertion sequences are supposed to be evenly distributed among the $n!$ possible sequences of $n$ segments, and when an object is deleted, we suppose that it can be any of the already inserted objects with the same probability. Under these assumptions the algorithm needs $O(n+a)$ expected space, $O(\log n + \frac{a}{n})$ expected insertion time, $O((1+\frac{a}{n})\log\log n)$ expected deletion time and $O(\log n)$ time to locate any query point in the arrangement, where $a$ is the current size of the arrangement and $n$ the current number of segments.

## 1 Introduction

One of the fundamental topics in computational geometry deals with planar subdivisions. The most common question is, for a planar subdivision of size $n$, to answer queries of the following type : *which is the region containing a given point ?* This problem received a lot of attention in the literature, and has been solved in different ways (see [11] for a survey), in optimal $O(\log n)$ time and $O(n)$ space after an $O(n \log n)$ preprocessing.

If the aim is not only the computation of a location structure to answers queries, but also to be able to modify dynamically the planar subdivision, the deterministic solutions known at this time involve complicated algorithms, and none of them is optimal (the query time [3] or the update time [6] is $O(\log^2 n)$). A very new result

[†]INRIA, 2004 Route des Lucioles, B.P.109, 06561 Valbonne cedex (France), Phone : +33 93 65 77 62, E-mail : odevil or teillaud@alcor.inria.fr. Part of this work was done while these authors were visiting Max Planck Institut

[‡]LIENS, CNRS URA 1327, 45 rue d'Ulm, 75230 Paris cedex 05

**This work has been supported in part by the ESPRIT Basic Research Action Nr. 3075 (ALCOM).*

for deterministic solutions to this problem is due to H. Baumgarten *et al.* [2] and reach querry and insertion time $O(\log n \log\log n)$ and deletion time $O(\log^2 n)$ with linear space.

After the work of Clarkson and Shor [5] studying this problem in a randomized framework, a lot of algorithms based on this principle have been developped. The principal interest of randomized algorithms is their simplicity and their good complexity and the counterpart is that these complexities are only randomized. Namely, for a dynamic algorithm, the update sequence must verify some hypotheses detailed in the sequel.

Concerning the problem of planar location, randomization was first applied to solve the static problem by Clarckson and Shor [5] using the Conflict Graph technique with optimal expected complexity : $O(\log n)$ query time, $O(a + n\log n)$ preprocessing time and $O(a + n)$ space to treat an arrangement of $n$ segments with $a$ intersecting points. [1] developped a new structure, *the Influence Graph*, allowing semi-dynamic algorithms with the same complexity (the whole set of segments has not to be known in advance, but it still must verify the randomized hypothesis). In the special case where the line segments form a known connected planar graph (so $a = n$) the preprocessing can be improved to $O(n\log^\star n)$ [13].

Recently, some results appeared concerning fully dynamic randomized algorithms. The first paper on this topic [7] generalized the Influence Graph structure to be able to remove a site from a Voronoï diagram.

Some results followed : [4, 12] use basically the same idea as [7], while [8, 9] use a completely different approach which is not based on the order of insertion of the points. [12] reaches a complexity of $O(\log^2 n)$ expected update and query time for a set of non intersecting segments. The algorithm in [9] has $O(\log n)$ query time and $O(\log n + \frac{a}{n})$ insertion time and $O(1 + \frac{a}{n})$ deletion time with high probability, where $a$ is the number of intersections between segments.

In this paper we extend the result of [1] to generalize the Influence Graph for the fully dynamic construction of an arrangement of line segments, in the same way as for the Delaunay triangulation in [7]. A line segment is added in $O(\log n + \frac{a}{n})$ expected time and deleted in $O((1 + \frac{a}{n})\log\log n)$ expected time ; the algorithm uses $O(n + a)$ expected space and a query point is located in $O(\log n)$ expected time. The bounds are randomized, i.e. all possible orders for already inserted sites are supposed to be equally likely, and when a site is deleted, it may be any site with the same probability.

## 2 The Influence Graph

Our algorithm is based on the trapezoidal map (or vertical visibility map) often used to compute or triangulate an arrangement of line segments ; this paradigm was already used in a randomized context [5, 1, 13], and we briefly recall the Influence Graph algorithm for this problem.

The trapezoidal map is obtained by drawing a vertical line through each vertex



of the arrangement of the line segments, and by keeping only the portions of the lines extending above and below the corresponding vertex and not intersected by any other segment, see Figure 1. A trapezoid is defined by at most four line segments, and we say that another line segment is *in conflict* with the trapezoid if and only if a part of the line segment is inside the trapezoid (see Figure 2). The subset of the universe of line segments consisting of line segments in conflict with a trapezoid $T$ is called the *influence range* of $T$. A trapezoid has two kinds of neighbors, adjacent trapezoids through vertical edges are called *horizontal neighbors* and adjacent trapezoids through line segments are called *up* or *down neighbors*. A trapezoid has at most four horizontal neighbors, and can have an arbitrary number of up and down neighbors.

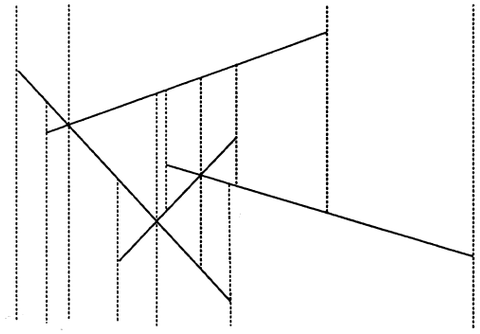

Figure 1: A trapezoidal map

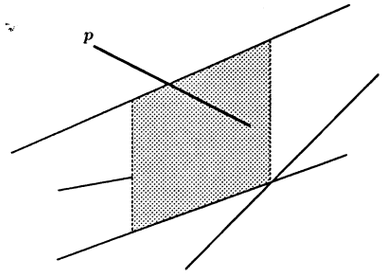


Figure 2: Trapezoid $T$ is in conflict with line segment $p$

The Influence Graph contains all the history of the trapezoidal map during the insertion sequence. When a new segment $p$ is inserted, a graph traversal allows the determination of all trapezoids in conflict with $p$, these trapezoids are called *killed* by $p$. These dead trapezoids are subdivided to form new trapezoids *created* by $p$,

and a dead trapezoid is called *parent* of a new one if they overlap and they are linked to store this relation in the Influence Graph (see Figure 3). Notice that a trapezoid may have several parents, for example in Figure 3 a portion of 4 is merged with a portion of 6 to form $c$ ; thus 4 and 6 are together parents of $c$. These relations between parents and children are used to determine the conflicts with forthcoming segments because if a segment is in conflict with a node, it is in conflict with at least one of its parents.

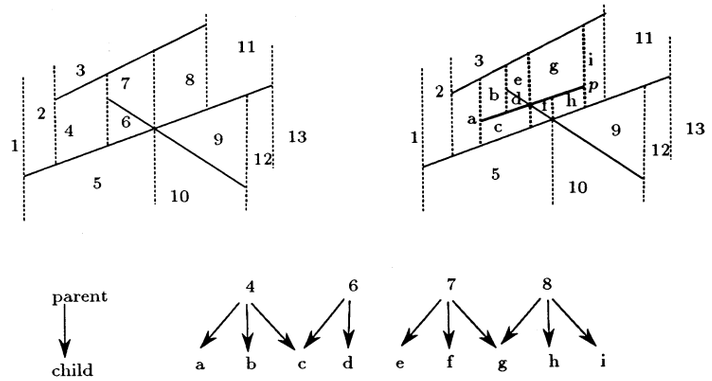


Figure 3: Insertion of $p$ in the Influence Graph

## Definition of a node in the Influence Graph

Let us detail the whole information stored in a node of the graph.

First a node corresponds to a trapezoid $T$, so we store the four segments defining $T$ ; one of them is its creator. In order to merge trapezoids during the insertion of a segment, we need to store horizontal neighbors, that is at most 2 left and 2 right neighbors.

For the removing of a segment (see Section 3), we will need to store for each trapezoid some vertical neighbors : the **up-right** neighbor, namely its up neighbor adjacent to its right side, and the **up-left**, **down-right** and **down-left** neighbors. at the time when the trapezoid was created.
To initialize these neighbors, for a new trapezoid, during an insertion (see Section 2), we need to know all current up and down neighbors of its parents. We have two ways to achieve this : the first one is to maintain all those vertical neighbors for each trapezoid in the map ; the second one is to look for them each time when we need them ; to do so, we only need one of them, and find the other ones using horizontal neighbors. The first solution makes the structure of a node rather heavy, and it has been proved that the second solution does not increase the time complexity and gives all necessary information [5]. So we will consider that we know every up and down neighbor.



We need to introduce an auxiliary kind of entity, consisting of the *corners* of trapezoids (i.e. its vertices). This corners are crucial in the removing process, as will be seen in the sequel. Each corner of the trapezoid is covered by one parent of the trapezoid, we store these at most four parents.

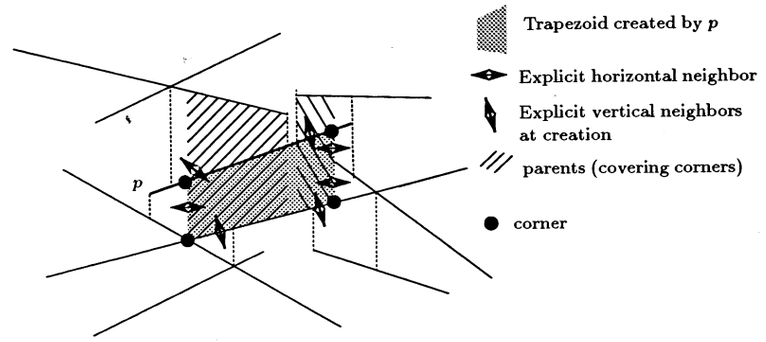


Figure 4: A trapezoid, its neighbors and parents

Let us summarize the structure of a node (see Figure 4) :
- at most 4 segments defining it (one of them is the **creator**)
- the at most 4 horizontal current neighbors
- its at most 4 children
- its killer
- its at most 4 corners
- all current vertical neighbors (they are not explicitely stored)
- one vertical neighbor at creation per corner
- one of parent by corner

If the node is dead, all current neighbors become neighbors at the death, and we store them.

## Insertion of a new segment

The following pseudo-code procedure sums up the algorithm. $p$ is the new segment and $T$ is a node of the Influence Graph. For the first call to **Insert**, $T$ is the root of the Influence Graph.

```
Insert(p, T)
    if p is not in conflict with T
        return ;
    if T is dead
        for each child S of T
            Insert(p, S) ;
```

```
    endfor
else
    p is the killer of T ;
    split T into pieces, that become children of T ;
    deduce the neighbors of the children from the neighbors of T ;
    deduce some corners of the children from the corners of T ;
    create new corners ;
    for each child S of T
        look for a merge with horizontal neighbors of S ;
        for each corner c of S belonging to T
            T is the parent of S in corner c ;
    endfor ;
    update all neighbor relations of the neighbors of T ;
endif ;
```

# 3 Removing a segment

As in [7], when a segment $p$ is deleted, we restore the history as if $p$ had never been inserted. All nodes defined by segment $p$ must be removed from the Influence Graph. These nodes may have children whose definition does not involve $p$, these children are now *unhooked*.

[4] uses the same idea of restoring the history as if $p$ had never been inserted for convex hulls in any dimension. [12] computes the whole history of insertions but also deletions and in this way gets a bigger history that he has to clean up sometimes. [9] uses a different technique which does not involve the history of the construction.

The first part of the removing of $p$ will consist in finding all removed nodes. Then we must reinsert the creators of these nodes, while maintaining at each step the set $A$ of trapezoids in conflict with $p$.

The unhooked nodes will be processed while processing the removed nodes : the influence range of a trapezoid is included in the influence ranges of its children. So, a removed node cannot have only unhooked children. Therefore, an unhooked node must have a removed sibling, and it will be processed in the same time.

## 3.1 Initialization of $A$

$A$ must be initialized with the set of trapezoids killed by $p$. If we assume that we have stored, for each segment, one of the nodes that it created, then by following the parent pointer we find one node killed by $p$, and using the neighbors at the death we complete the initialization of $A$.



## 3.2 Looking for the segments to be reinserted

We do not really look for whole segments to be reinserted. We only look for all parts of them to be reinserted, without trying to connect them. More precisely, what we need to reinsert segments is a list of triplets $(x, T, F)$ consisting of a segment to be reinserted, a node $T$ to be removed, and one parent $F$ of $T$ also to be removed. The reinsertion of such a triplet takes in account only the part of $x$ on the boundary of $T \cap F$ (which can be reduced to an extremity of $x$).

We obtain this list by traversing the Influence Graph, starting from the trapezoids in $A$. Each time we find a node $F$ to be removed, we look at its children, and for each child $T$ defined by $p$ and created by the insertion of a segment $x$, we add $(x, T, F)$ to the list.

Furthermore, the segments need to be reinserted in the same order as they had been inserted first, so we need to sort these triplets (the ordering is on the date of insertion of $x$ only). This can be done in $O(\log \log n)$ time worst case deterministic time (for each triplet) by using a bounded ordered dictionary [14]. The universe for this dictionary is the insertion age of the segments. The required finiteness of the universe can be circumvented using standard dynamization techniques, see for example [10, section5.2].

## 3.3 Corners and bridges

When we reinsert a segment, we must first of all find the trapezoids in the current set $A$ that are in conflict with it. So we need a location structure for $A$.

By definition, $A$ is the set of trapezoids in conflict with the segment removed $p$ at some stage of the history of the construction. So, the segment $p$ crosses all trapezoids in $A$.

Notice that a removed node is defined by $p$, so it has at least one corner on $p$ (except if it is defined by a vertex of $p$, this case is not difficult and can be solved easily).

The general idea is, when we process a triplet $(x, T, F)$, to use the corners of $T$ or $F$ to locate in $A$ the part of $x$ defining $T$ (that is an approximate idea, the details are given below).

So, rather than maintaining the set $A$, we maintain a doubly linked list of *bridges* allowing to deduce the trapezoids of $A$ in conflict with $x$ from the corners of $T$, $F$ or theirs suitable neighbors. The bridges are some corners that appear on $p$. These bridges are ordered by the order in which they appear on $p$. More precisely, a corner of a trapezoid is a bridge, if and only if it is on the common boundary of two trapezoids of $A$, and we store both of them in it (see Figure 5).

In order to initialize the list of bridges, we traverse the set $A$ consisting of all nodes killed by $p$ (see Section 3.1), by using neighborhood relations. For each node $T$ in $A$, and each of its children, the corners $c$ of this child that lie on $p$ now appear as bridges on $p$. $T$ is one of the two trapezoids in $A$ to be stored in $c$.

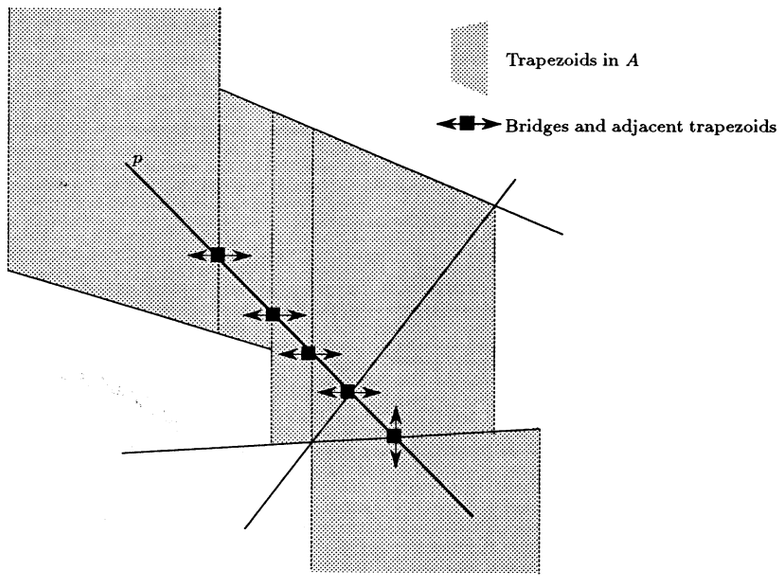


Figure 5: The bridges along $p$

### 3.4 Reinsertion of a triplet $(x, T, F)$

We now describe how a triplet $(x, T, F)$ is reinserted. These such triplets are processed in the order used for the first insertion of segments, the order for triplets having the same $x$ is indifferent.

Let us recall that in the triplet $(x, T, F)$, $x$ is a segment, $T$ is a removed node created by $x$ and $F$ is a removed parent of $T$. At the current stage of the reinsertion, the creator of $F$ is either $p$ or has already been reinserted (because this creator is before $x$ in the insertion order).

In most cases, the removed node $T$ has at least one corner $c$ on the segment $p$ that we delete, we first look for a bridge $b$ *close* to this corner, which means that there is no other bridge on $p$ between $b$ and $c$. Such a bridge $b$ points to a trapezoid in $A$ in conflict with $x$. The main problem is to find $b$ using suitable parent and neighbor pointers. To find $b$ we have to look for trapezoid adjacent to $p$ in the old trapezoidal map, trapezoids above and below give two candidate bridges ($b_1$ and $b_2$) and $b$ is the closest bridge to $c$ among $b_1$ and $b_2$.

Let us assume without loss of generality that $c$ is the down left corner of $T$. There are several cases to examine.

1. If the down left corner of $F$ is $c$ then $c$ is already a bridge, so $c = b$ (see Figure 6).

2. If $c$ is not the intersection $p \cap x$, then $p$ is necessarily the down side of $T$ (otherwise, we are in case 1). $b_1$ is the down left corner of $F$. $b_2$ is the left up corner of $N$, the down neighbor at creation of $T$ in corner $c$ (see Figure 7). $b$ is the closest bridge to $c$ among $b_1$ and $b_2$.

3. If $c = p \cap x$, as in the preeceding case $b_1$ is the down left corner of $F$. Then, we determine as above the trapezoid $N$ adjacent to $c$ below $p$. The down side of $T$ can be $p$ (trapezoid $T'$ in Figure 8) or can be $x$ (trapezoid $T$ in Figure 8). In the first case $N$ is the down neighbor at creation of $T$ in corner $c$ and in the second case $N$ is the down neighbor of the down neighbor at creation of $T$ in corner $c$. $N$ cannot be used directly to determine $b_2$ since the creator of $N$ is also $x$ and it is not possible to ensure that the corner of $N$ is already a bridge. So we use for $b_2$ the left up corner of $N'$, parent of $N$ covering the corner $c = x \cap p$ of $N$. $b$ is the closest bridge to $c$ among $b_1$ and $b_2$.

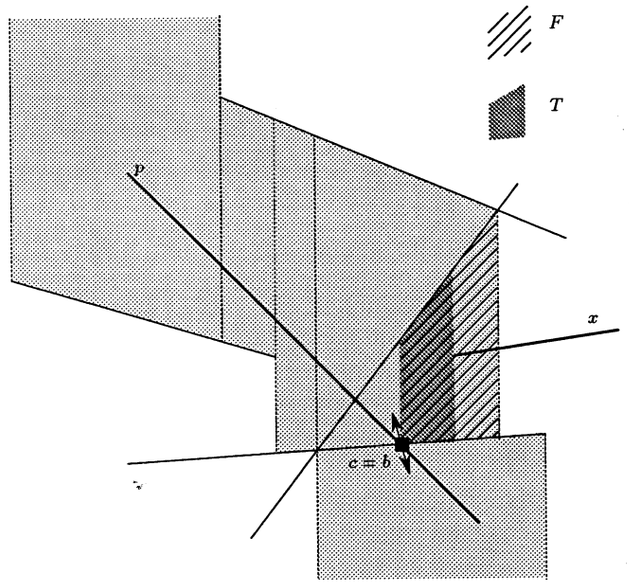


Figure 6: The bridge $b$ is $c$

If the removed node $T$ has no corner on $p$, then it is defined by a vertex of $p$ and this vertex is a suitable bridge $b$.

From $b$ we have a direct access to the two trapezoids of $A$ covering $b$, let $S$ be the one intersecting $T$ (here, the one on the right of the vertical line passing through $b$). This trapezoid is in conflict with $x$, so we update the graph in the following way : first $x$ is the (new) killer of $S$, then $x$ splits $S$ in pieces, these pieces are children of $S$ and some of them that are in conflict with $p$ must be created now and added to $A$. The children of $S$ which are not in conflict with $p$ already exist in the graph and

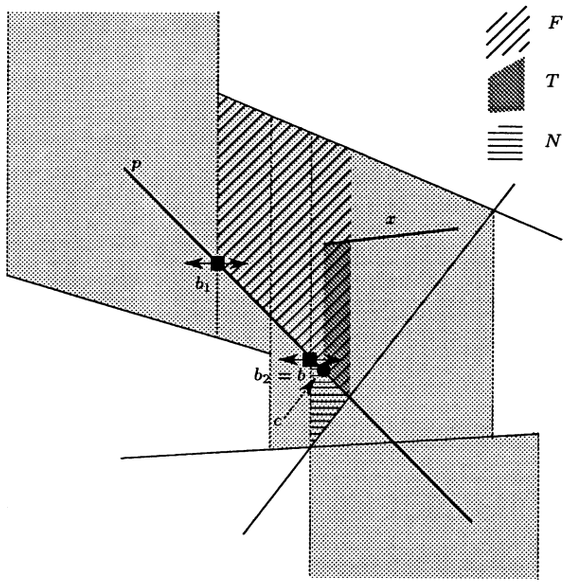


Figure 7: The bridge $b$ is the closest to $c$ among $b_1$ and $b_2$

have not to be created, but they are unhooked and must be hung up to $S$ ; these nodes can be found among the children of $F$. Figure 9 describes these operations in the case of Figure 6. We must also update the list of bridges, the neighborhood relations and possibly merge some newly created nodes.

The following pseudo-code procedure summarizes the reinsertion of a triplet.

Reinsert$(x, T, F)$

{wlog, the corner $c$ of $T$ lying on $p$ is its down left corner}
{Looking for the closest bridge $b$ to $c$ :}
if $c$ is already a bridge then $b = c$
else $b_1$ = left down corner of $F$ ;
if $c \neq x \cap p$ then
$b_2$ = left up corner of the down neighbor at creation of $T$ in corner $c$
else {the down neighbor of $T$ is a sibling of $T$, so its corners are not bridges yet}
$N$ = down neighbor of $T$ in corner $c$
if $p$ do not separate $N$ from $T$
$N$ = down neighbor of $N$ in corner $c$
$b_2$ = left up corner of the parent of $N$ in corner $c$
endif ;
$b$ = closest bridge to $c$ between $b_1$ and $b_2$
endif ;



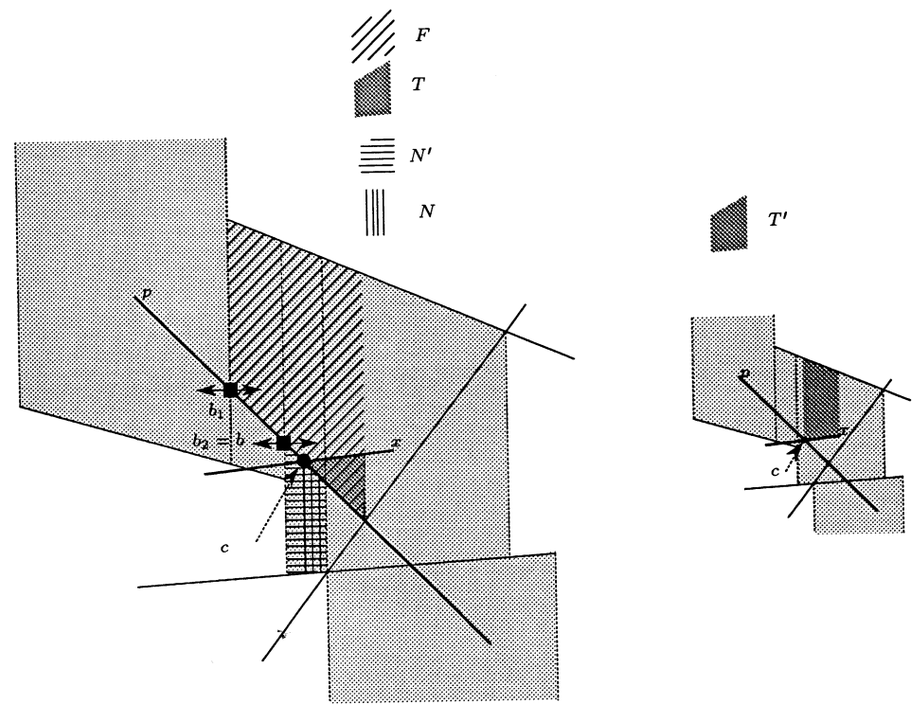


Figure 8: The bridge $b$ is the closest to $c$ among $b_1$ and $b_2$

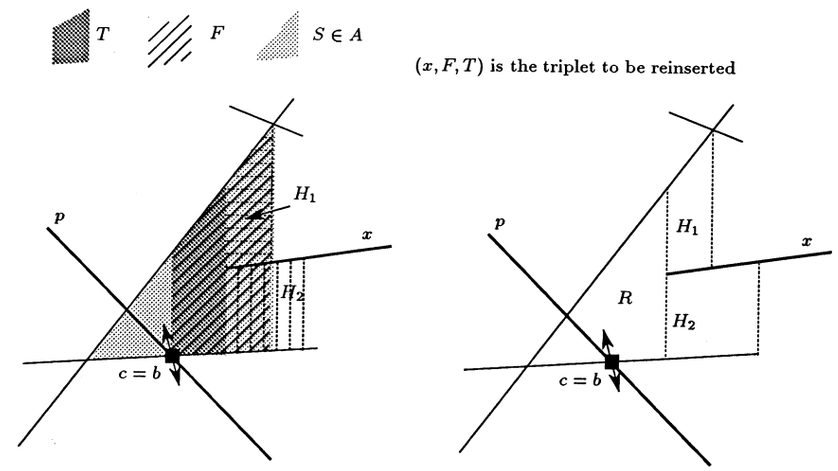


Figure 9: Computing the new trapezoid $R$ and hanging up the unhooked nodes $H_1$ and $H_2$



from $b$, determine the trapezoid $S$ in $A$ which intersects $T$,
    it is in conflict with $x$ ;
$x$ is the killer of $S$ ;
split $S$ with $x$ ;
for each child $R$ of $S$
    if $R$ is in conflict with $p$ {$R$ must be added to $A$}
        if the corner $c$ of $T$ becomes a new bridge (i.e. $b \neq c$)
            create this bridge with $R$ as one of its two trapezoids
            and find its other trapezoid, which is another child of $S$
            insert $c$ in the doubly linked list of bridges as a successor of $b$
        else {$b = c$}
            update $b$ : $R$ is one of its two trapezoids
        endif ;
    else {$R$ must be already existing as an unhooked node}
        find $R$ among the children of $F$ and hang it up to $S$
    endif ;
    update all other fields in $R$, and neighbor relations, as for a usual insertion ;
    look for a possible merge with neighbors as for an insertion
endfor.

# 4 Analysis

The extra work to analyze the deletion of a segment is extremely simple. Let us first recall the result for insertion proved in [1]. $n$ is the number of segments currently present in the structure, $a$ is the number of intersection points between these $n$ line segments.

**Lemma 1** *The expected size of the Influence Graph is* $O(n + a)$.

**Lemma 2** *The expected number of visited nodes during the insertion of the last segment in the Influence Graph is* $O(\log n + \frac{a}{n})$.

**Lemma 3** *The cost of locating a query point in the trapezoidal map is* $O(\log n)$. *This cost is expected over the randomization of the segments (not on the query point).*

This fact, that no hypothesis is needed on the query point is due to the property that a trapezoid is included in the union of its sons. So a location in the Influence Graph never needs back-tracking. Furthermore, as the Influence Graph keeps trace of the history of the insertion, it is possible to answer persistent queries : *what was the trapezoid containing the query point in the trapezoidal map of the line segments inserted before time t (and not deleted).*

We now deal with the analysis of the deletion phase.

**Lemma 4** *The expected number of removed nodes is* $O(1+\frac{a}{n})$.

**Proof :** If the deleted segment $p$ is randomly chosen among the $n$ segments present in the arrangement, the expected number of removed nodes is

$$\begin{aligned} & \sum_{T \text{ trapezoid}} Prob(p \text{ defines } T)\, Prob(T \text{ exists in the Influence Graph}) \\ \leq\ & \frac{4}{n} \times \text{ expected number of nodes of the Influence Graph} \\ =\ & O(1+\frac{a}{n}) \qquad \text{using Lemma 1} \end{aligned}$$

□

Since a node has at most four sons, there are at most four triplets $(x, T, F)$ with the same removed node $F$, thus the above bound aplpy also to the number of triplets processed by the algorithm. These triplets must be sorted according to the age of the insertion of the creator segment, this is done in $O((1+\frac{a}{n})\log\log n)$ expected time using a bounded ordered dictionary (or in $O((1+\frac{a}{n})\log n)$ expected time using a simpler priority queue). The reinsertion of a triplet involves a constant number of operations, so the expected time of deleting a random segment is $O((1+\frac{a}{n})\log\log n)$.

As the Influence Graph is restored as if $p$ was never inserted, Lemmas 1, 2 and 3 still apply, and we obtain the main result of this paper :

**Theorem :** *An arrangement of line segments in the plane can be maintained dynamically in* $O(\log n+\frac{a}{n})$ *time for an insertion and* $O((1+\frac{a}{n})\log\log n)$ *time for a deletion. The space complexity is* $O(n+a)$. *All complexities are expected with respect to the randomized insertion of the line segments. Here* $n$ *denotes the current number of segments present in the arrangement, and* $a$ *denotes the current complexity of the arrangement.*
*The expected cost of the location of any point in the arrangement is* $O(\log n)$, *where the expectation is over the randomization of the order of insertion of the line segments present in the arrangement, these queries can be done persistently with respect to the insertions.*

## Acknowledgements

The authors would like to thank Stefan Meiser for helpfull discussions and Jean-Pierre Merlet for supplying us with his interactive drawing preparation system JPdraw .

# References

[1] J.D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, et M. Yvinec. Applications of Random Sampling to On-line Algorithms in Computational Geometry. *Discrete and Computational Geometry.* To be published. Available as Technical Report INRIA 1285. Abstract published in IMACS 91 in Dublin.



[2] H. Baumgarten, H. Jung, et K. Mehlhorn. Dynamic Point Location in General Subdivisions. Dans *ACM-SIAM Symposium on Discrete Algorithms*, Janvier 1992.

[3] S.W. Cheng et R. Janardan. New Results on Dynamic Planar Point Location. Dans *IEEE Symposium on Foundations of Computer Science*, pages 96–105, 1990.

[4] K.L. Clarkson, K. Mehlhorn, et R. Seidel. Four results on randomized incremental constructions. Juin 1991. Manuscript.

[5] K.L. Clarkson et P.W. Shor. Applications of Random Sampling in Computational Geometry, II. *Discrete and Computational Geometry*, 4(5), 1989.

[6] Y-J. Chiang et R. Tamassia. Dynamization of the Trapezoid Method for Planar Point Location. Dans *7ème ACM Symposium on Computational Geometry à North Conway*, pages 61–70, 1991.

[7] O. Devillers, S. Meiser, et M. Teillaud. *Fully dynamic Delaunay triangulation in logarithmic expected time per operation.* Rapport de recherche 1349, Institut National de Recherche en Informatique et Automatique, (France), Décembre 1990. To be published in CGTA. Abstract published in LNCS 519 (WADS91).

[8] K. Mulmuley et S. Sen. Dynamic Point Location in Arrangements of Hyperplanes. Dans *7ème ACM Symposium on Computational Geometry à North Conway*, pages 132–142, 1991.

[9] K. Mulmuley. Randomized Multidimensional Search Trees : Dynamic Sampling. Dans *7ème ACM Symposium on Computational Geometry à North Conway*, pages 121–131, 1991.

[10] M.H. Overmars. *The design of dynamic data structures. LNCS 156*, Springer-Verlag, 1983.

[11] F.P. Preparata. Planar Point Location Revisited. *International Journal of Foundations of Computer Science*, 1(1):71–86, 1990.

[12] O. Schwarzkopf. Dynamic maintenance of geometric structure made easy. Dans *IEEE Symposium on Foundations of Computer Science*, Octobre 1991. Full paper available as Technical Report B 91-05 Universität Berlin.

[13] R. Seidel. A simple and Fast Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons. *Computational Geometry Theory and Applications*, 1, 1991.

[14] P. van Emde Boas, R. Kaas, et E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Mathematical Sysyems Theory*, 10:99–127, 1977.