Taking a Walk in a Planar Arrangement*

Sariel Har-Peled†

November 27, 2002

Abstract

We present a randomized algorithm for computing portions of an arrangement of n arcs in the plane, each pair of which intersect in at most t points. We use this algorithm to perform online walks inside such an arrangement (i.e., compute all the faces that a curve, given in an online manner crosses), and to compute a level in an arrangement, both in an output-sensitive manner. The expected running time of the algorithm is $O(\lambda_{t+2}(m+n)\log n)$, where m is the number of intersections between the walk and the given arcs.

No similarly efficient algorithm is known for the general case of arcs. For the case of lines and for certain restricted cases involving line segments, our algorithm improves the best known algorithm of [OvL81] by almost a logarithmic factor.

1 Introduction

Let $\hat{S}$ be a set of n x-monotone arcs in the plane, each pair of which intersect in at most t points. Computing the whole (or parts of the) arrangement $\mathcal{A}(\hat{S})$, induced by the arcs of $\hat{S}$, is one of the fundamental problems in computational geometry, and has received a lot of attention in recent years [SA95]. One of the basic techniques used for such constructions is based on *randomized incremental* construction of the *vertical decomposition* of the arrangement (see [BY98]).

If we are interested in only computing parts of the arrangement (e.g., a single face or a zone), the randomized incremental technique can still be used, but it requires non-trivial modifications [CEG+93, dBDS95]. Intuitively, the added complexity is caused by the need to "trim" parts of the plane as the algorithm advances, so that it will not waste energy on regions which are no longer relevant. In fact, this requirement implies that such an algorithm has to know in advance what are the regions we are interested in at any stage during the randomized incremental construction.

*A preliminary version of the paper appeared in the *40th Annual Symposium on Foundations of Computer Science*. This work has been supported by a grant from the U.S.–Israeli Binational Science Foundation, and is part of the author's Ph.D. thesis, prepared at Tel-Aviv University under the supervision of Prof. Micha Sharir.

†Department of Computer Science, D340 Levine Science Research Center; Duke University, Box 90129; Durham, NC 27708-0129; USA; sariel@cs.duke.edu; http://www.cs.duke.edu/~sariel/

A variation of this theme, with which the existing algorithms cannot cope efficiently, is the following *online* scenario: We start from a point $p = p(0) \in \mathbb{R}^2$, and we find the face f of $\mathcal{A}(\hat{S})$ that contains $p(0)$. Now the point p starts moving and traces a connected curve $\{p(t)\}_{t\geq 0}$. As our walk continues, we wish to keep track of the face of $\mathcal{A}(\hat{S})$ that contains the current point $p(t)$. The collection of these faces constitutes the *zone* of the curve $p(t)$. However, the function $p(t)$ is not assumed to be known in advance, and it may change when we cross into a new face or abruptly change direction in the middle of a face (see [BDH99] for an application where such a scenario arises). The only work we are aware of that can deal with this problem efficiently is due to Overmars and van Leeuwen [OvL81], and it only applies to the case of lines (and, with some simple modifications, to certain restricted cases involving line segments as well).[1] It can compute such a walk in (deterministic) $O((n+m)\log^2 n)$ time, inside an arrangement of n lines, where m is the number of intersections of the walk with the lines of $\hat{S}$. This is done by maintaining dynamically the intersection of half-planes that corresponds to the current face.

In this paper, we propose a new randomized algorithm that computes the zone of the walk in a general arrangement of arcs, as above, in $O(\lambda_{t+2}(n+m)\log n)$ expected time, where $\lambda_{t+2}(n+m)$ is the maximum length of a Davenport-Schinzel sequence of order $t+2$ having $n+m$ symbols [SA95]. The new algorithm can be interpreted as a third "online" alternative to the algorithms of [CEG+93, dBDS95]. The algorithm is rather simple and appears to be practical. As a matter of fact, we had implemented and experimented with a variant of the algorithm [AHH+99].

As an application of the new algorithm, we present an algorithm for computing a level in an arrangement of arcs. It computes a single level in $O(\lambda_{t+2}(n+m)\log n)$ expected time, where m is the complexity of the level. We also show how to adapt the main algorithm to obtain a point-location algorithm that locates m points in an arrangement of n arcs, as above, in expected time $O(\lambda_{t+2}(n+m+w)\log n)$, where w is the minimum number of intersections between a spanning tree connecting those query points and the given arcs.

Both results improve by almost a logarithmic factor over the best previous result of [OvL81], for the case of lines (and for certain cases involving line segments).[2] For the case of general arcs, we are not aware of any similarly efficient previous result.

The paper is organized as follows. In Section 2 we describe the algorithm. In Section 3 we analyze its performance. In Section 4 we mention a few applications of the algorithm, including the construction of a single level, and multiple point-location. Concluding remarks are given in Section 6.

2 The Algorithm

In this section, we present the algorithm for performing an online walk inside a planar arrangement.

[1] Recently, an improvement has been given by Chan [Cha99a] and was further improved by Brodal and Jacob [BJ00]; their algorithm can perform an update in $O(\log n \log\log n)$ amortized time, and answer queries (of the kind used in this application) in $O(\log n)$ time per query.

[2] Our results are also asymptotically faster and much simpler to implement than what is yielded by the recent results of Chan [Cha99a, Cha99b], and Brodal and Jacob [BJ00]

Randomized Incremental Construction of the Zone Using an Oracle. Given a set $\hat{S}$ of n x-monotone arcs in the plane, so that any pair of arcs of $\hat{S}$ intersect at most t times (for some fixed constant t), let $\mathcal{A}(\hat{S})$ denote the arrangement of $\hat{S}$; namely, the partition of the plane into faces, edges, and vertices as induced by the arcs of $\hat{S}$ (see [SA95] for details). In the following, we need two basic geometric primitives for splitting and merging vertical trapezoid, `SplitGeom`, `MergeGeom`, illustrated in Figure 1. We assume that $\hat{S}$ is in general position, meaning that no three arcs of $\hat{S}$ have a common point, and that the x-coordinates of the intersections and endpoints of the arcs of $\hat{S}$ are pairwise distinct. The *vertical decomposition* of $\mathcal{A}(\hat{S})$, denoted by $\mathcal{A}_{\mathcal{VD}}(\hat{S})$, is the partition of the plane into vertical pseudo-trapezoids, obtained by erecting two vertical segments up and down from each vertex of $\mathcal{A}(\hat{S})$ (i.e., each point of intersection between a pair of arcs and each endpoint of an arc), and by extending each of them until it either reaches an arc of $\hat{S}$, or otherwise all the way to infinity. See, e.g., [BY98, SA95] for more details concerning vertical decompositions. To simplify (though slightly abuse) the notation, we refer to the cells of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ as *trapezoids*. A *selection* R of $\hat{S}$ is an ordered sequence of distinct elements of $\hat{S}$. By a slight abuse of notation, we also denote by R the unordered set of its elements. Let $\sigma(\hat{S})$ denote the set of all selections of $\hat{S}$. For a permutation S of $\hat{S}$, let S_i denote the subsequence consisting of the first i elements of S, for $i = 0, \ldots, n$.

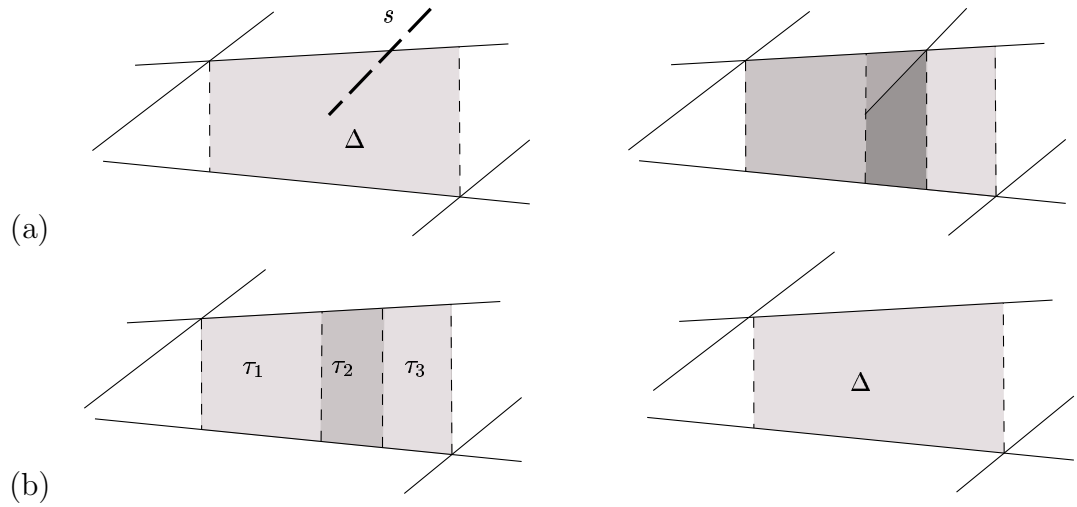

Figure 1: Geometric primitives: (a) `SplitGeom`(Δ, s) splits Δ into a $O(1)$ vertical trapezoids that cover the original trapezoid and their interior is not crossed by s. (b) `MergeGeom`$(\{\tau_1, \tau_2, \tau_3\})$ - merge the adjacent trapezoids τ_1, τ_2, τ_3 with the same top and bottom arcs into a single trapezoid Δ.

Computing the decomposed arrangement $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ can be done in a randomized incremental manner (see [BY98]). Let γ be the curve traced by the walk. For a selection $R \in \sigma(\hat{S})$, let $\mathcal{D}_\gamma(R)$ (resp. $\mathcal{Z}_\gamma(R)$) denote the *district* (resp. *zone*) of γ in $\mathcal{A}(R)$; these are, respectively, the set of all trapezoids of $\mathcal{A}_{\mathcal{VD}}(R)$ and the set of all faces of $\mathcal{A}(R)$ that have a nonempty intersection with γ. Let $\mathcal{A}_{\gamma,\mathcal{VD}}(R)$ denote the set of all trapezoids in $\mathcal{A}_{\mathcal{VD}}(R)$ that cover $\mathcal{Z}_\gamma(R)$. Our goal is to compute $\mathcal{A}_{\gamma,\mathcal{VD}}(\hat{S})$. (Alternatively, we may be interested only in computing the district $\mathcal{D}_\gamma(\hat{S})$ of γ.)

We assume for the moment that we are supplied with an oracle $\mathcal{O}(S_i, \gamma, \Delta)$, that can decide in constant time whether a given vertical trapezoid Δ is in $\mathcal{A}_{\gamma,\mathcal{VD}}(S_i)$. Equipped with this oracle, computing $\mathcal{A}_{\gamma,\mathcal{VD}}(S)$ is fairly easy, using a variant of the randomized incremental construction, outlined above. The algorithm, called `CompZoneWithOracle`, is depicted in Figure 2. We present this algorithm at a conceptual level only, because this is not the algorithm that we shall actually use. It is given to help us to describe and analyze the actual online algorithm that we shall present later.

ALGORITHM `CompZoneWithOracle`($\hat{S}$, γ, $\mathcal{O}$)
`Input:` A set $\hat{S}$ of n arcs, a curve γ, an oracle $\mathcal{O}$
`Output:` $\mathcal{A}_{\gamma,\mathcal{VD}}(\hat{S})$
begin
Choose a random permutation $S = \langle s_1, s_2, \ldots, s_n \rangle$ of $\hat{S}$.
$\mathcal{C}_0 \leftarrow \{\mathbb{R}^2\}$
for i from 1 to n **do**
$\mathcal{D}_i \leftarrow \left\{ \Delta \;\middle|\; \Delta \in \mathcal{C}_{i-1}, int\,\Delta \cap s_i \neq \emptyset \right\}$
$Temp \leftarrow \emptyset$
for each $\Delta \in \mathcal{D}_i$ **do**
$Temp \leftarrow Temp \cup$ `SplitGeom`(Δ, s_i),
where `SplitGeom`(Δ, s) is the operation of splitting a vertical trapezoid Δ crossed by an arc s into a constant number of vertical trapezoids, as in [dBvKOS00], such that the new trapezoids cover Δ, and they do not intersect s in their interior.
end for
Merge all the adjacent trapezoids of $Temp$ that have the same top and bottom arcs. Let $Temp_1$ be the resulting set of trapezoids.
Let $Temp_2$ be the set of all trapezoids of $Temp_1$ that are in $\mathcal{A}_{\gamma,\mathcal{VD}}(S_i)$. Compute this set using $|Temp_1|$ calls to $\mathcal{O}$.
$\mathcal{C}_i \leftarrow (\mathcal{C}_{i-1} \setminus \mathcal{D}_i) \cup Temp_2$
end for
Remove from $\mathcal{C}_n$ all trapezoids not belonging to $\mathcal{A}_{\gamma,\mathcal{VD}}(\hat{S})$, by checking with $\mathcal{O}$ each trapezoid of $\mathcal{C}_n$.
return $\mathcal{C}_n$
end `CompZoneWithOracle`

Figure 2: A randomized incremental algorithm for constructing the zone of a walk in an arrangement of arcs, using an oracle

Note that the set of trapezoids $\mathcal{C}_i$ maintained by the algorithm in the i-th iteration is a superset of $\mathcal{A}_{\gamma,\mathcal{VD}}(S_i)$. There might be trapezoids in $\mathcal{C}_i$ that are no longer in $\mathcal{Z}_\gamma(S_i)$ (typically these are trapezoids that are separated from $\mathcal{Z}_\gamma(S_i)$ by an arc that does not cross their interior and is intersected after they have been created). However, this implies that any such trapezoid will be eliminated the first time an arc that crosses it will be

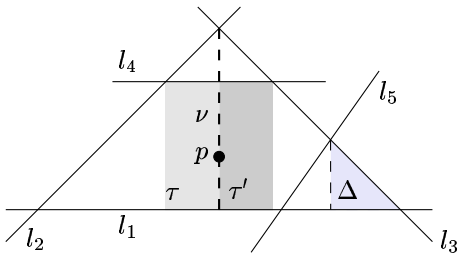

Figure 3: Illustration of the definitions: ν is a transient splitter, and thus τ, τ' are both transient. We have $\text{rank}(\tau) = \text{index}(\tau) = 4$, $\text{rank}(\Delta) = 3$, and $\text{index}(\Delta) = 5$, where $S = \langle l_1, l_2, l_3, l_4, l_5 \rangle$.

handled, or, if no such arc exists, at the final clean-up step of the algorithm. Moreover, the algorithm `CompZoneWithOracle` can be augmented to compute a *history* DAG (as in [SA95]), whose nodes are the trapezoids created by the algorithm and where each trapezoid destroyed during the execution of the algorithm points to the trapezoids that were created from it. Let $\mathcal{HT}_\gamma(S_i)$ denote this structure after the i-th iteration of the algorithm. Note that the out-degree of each node of $\mathcal{HT}_\gamma$ is bounded by a constant that depends on t.

Definitions. A trapezoid created by the `SplitGeom` operation of `CompZoneWithOracle` is called a *transient* trapezoid if it is later merged (in the same iteration) to form a larger trapezoid. A trapezoid generated by `CompZoneWithOracle` is *final* if it is not transient. The *rank* $\text{rank}(\Delta)$ of a trapezoid Δ is the maximum of the indices i, j of the arcs containing the bottom and top edges of Δ in the permutation S. We denote by $D(\Delta)$ the *defining set of a final trapezoid* Δ; this is the minimal set D such that $\Delta \in \mathcal{A}_{\mathcal{VD}}(D)$. It is easy to verify that $|D(\Delta)| \leq 4$. We can also define $D(\Delta)$ for a transient trapezoid Δ, to be the minimal set D such that Δ can be transient during an incremental construction of $\mathcal{A}_{\mathcal{VD}}(D)$. Here it is easy to verify that $|D(\Delta)| \leq 6$. The *index* $\text{index}(\Delta)$ of a trapezoid Δ is the minimum i such that $D(\Delta) \subseteq S_i$. For a trapezoid Δ, we denote by $\text{cl}(\Delta)$ the *conflict list* of Δ; that is, the set of arcs of $\hat{S}$ that intersect Δ in its interior. $\text{next}(\Delta)$ denote the first element of $\text{cl}(\Delta)$, according to the ordering of S.

In the algorithm, whenever we compute a trapezoid, we also compute its conflict list. The overall running time of the algorithm is dominated by the time required to compute and manipulate those conflict lists. Generally speaking, if a trapezoid is created from a parent trapezoid by splitting, we can compute the conflict list, by scanning the parent conflict list and checking for each arc if it intersects the new trapezoid. If a trapezoid is formed by merging several trapezoid, its conflict list can be computed by merging the trapezoids conflict-list. This can be done in linear time in the size of the input conflict lists, as described below. Since a conflict list participate only in a constant number of such merge/split operations, the overall time required to manipulate those conflict lists is proportional to their overall size.

For a trapezoid Δ generated by `CompZoneWithOracle`, which was not merged into a larger trapezoid, we denote by $\text{father}(\Delta)$ the trapezoid that Δ was generated from. A vertical side of a trapezoid Δ is called a *splitter*. A splitter ν is *transient* if it is not incident to the intersection point (or endpoint) that induced the vertical edge that contains ν (this means

that the two trapezoids adjacent to ν are transient, and will be merged into a larger final trapezoid). See Figure 3 for an illustration of some of these definitions. It is easy to verify that a trapezoid Δ is transient if and only if at least one of its bounding splitters is transient. Thus, one can decide whether a trapezoid is transient, by inspecting its splitters, in constant time.

The online algorithm constructs portions of $\mathcal{HT}_\gamma(S)$ incrementally, as they are needed. This is done by performing a sequence of point locations in the arrangement, where each such query constructs the final trapezoid of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that contains the query point, plus all ancestor trapezoids that lie on paths of $\mathcal{HT}_\gamma(S)$ from the root to that trapezoid. Informally, instead of building $\mathcal{HT}_\gamma(S)$ layer-by-layer, as is done by `CompZoneWithOracle`, the online algorithm constructs the DAG in an 'orthogonal' DFS manner. The intuition behind the design and efficiency of the algorithm is that the expected number of nodes in $\mathcal{HT}_\gamma(S)$ is only $O(\lambda_{t+2}(n+m))$, whereas the overall expected running time of `CompZoneWithOracle` is $O(\lambda_{t+2}(n+m)\log n)$ (this is the expected overall size of the conflict lists of the computed nodes, which the algorithm needs to construct and manipulate). Both bounds are easy consequences of known results, and will be discussed in Section 3.2. Thus, in our quest to usurp the oracle, we can afford to pay an extra $O(\log n)$ time during the search for and construction of each node of $\mathcal{HT}_\gamma(S)$. This indeed will be the cost of a point-location query (ignoring the time required for constructing any new node of $\mathcal{HT}_\gamma(S)$ that the query has to pass through). Informally, the online algorithm performs essentially the same operations as the preceding algorithm, except that it executes them in a different order, and, in addition, it may revisit again and again portions of the DAG that have already been constructed as it searches down the DAG while performing point locations. However, since this extra cost is only logarithmic, it does not increase the asymptotic complexity of the algorithm.

An Online Algorithm for Constructing the Zone. Before describing the algorithm in detail, we refer the reader to Appendix A, which provides a pseudo-code for some relevant procedures used by the algorithm, and Appendix B, which presents an example of how the following algorithm works. The reader is encouraged to consult with the Appendices whenever the definitions and the description become too vague.

Let us assume that the random permutation S of $\hat{S}$ has been fixed in advance. Note that S predetermines $\mathcal{HT}_\gamma = \mathcal{HT}_\gamma(S)$. The key observation in the online algorithm is that in order to construct a specific leaf of $\mathcal{HT}_\gamma(S)$ we do not have to maintain the entire DAG, and it suffices to compute only the parts of the DAG that lie on paths connecting the leaf with the root of $\mathcal{HT}_\gamma$ (there might be several such paths, since our structure is a DAG, and not a tree).

To facilitate this computation, we maintain a *partial history* DAG, denoted by T. The nodes of T are of two types: (i) *final nodes*: those are nodes whose corresponding trapezoids appear in $\mathcal{HT}_\gamma = \mathcal{HT}_\gamma(S)$, and (ii) *transient nodes*: these are some of the leaves of T, whose corresponding trapezoids are transient. A transient node can be easily detected since its trapezoid is transient. A final node is simply a node which is not transient. In particular, all the internal nodes of T are copies of identical nodes of $\mathcal{HT}_\gamma$ (whose corresponding trapezoids are final), while some of the leaves of T might be transient. Intuitively, T stores the portion of $\mathcal{HT}_\gamma$ that we have computed explicitly so far. The transient leaves of T delimit portions of

$\mathcal{HT}_\gamma$ that have not been expanded yet. Inside each node of T, we also maintain the conflict list of the corresponding trapezoid Δ and its first element next(Δ).

Suppose we wish to compute a leaf of $\mathcal{HT}_\gamma$ that contains a given point p. We denote this operation by `PointLocate`(p). We first locate the leaf of T that contains p. This is done by traversing a path in T, starting from the root of T, and going downward in each step into the child of the current trapezoid that contains p (each such step requires $O(1)$ time, because, as already noted, the out-degree of any node of $\mathcal{HT}_\gamma$, and thus of T, is bounded by a constant that depends on t). (A technical issue that we face is that usually p lies on some trapezoid boundary, so we need additional local information to determine which child to descend to at each of the above steps — see below for more details) At the end we either reach a final leaf (with an empty conflict list) which is the required leaf of $\mathcal{HT}_\gamma$, or we encounter a leaf v of T. In the latter case, we need to expand T further below v. If v represents a transient trapezoid, then the first step is to replace v by the corresponding node v^* of $\mathcal{HT}_\gamma$, obtained by merging the transient trapezoid of v with adjacent transient trapezoids, with identical top and bottom arcs, to form the final trapezoid associated with v^*. If v is a final node we expand it by splitting it with the first arc that crosses its trapezoid, using steps (iv) and (v) below.

Assume for the moment that we are supplied (for the case where v is transient) with a method (to be described shortly) to generate all those adjacent transient trapezoids, whose union forms the final trapezoid that is stored at v^* in $\mathcal{HT}_\gamma$. Then we do the following: (i) Merge all those transient trapezoids into a new (final) trapezoid Δ; (ii) Compute the conflict list cl(Δ) from the conflict lists of the transient trapezoids; (iii) Compute the first element s_Δ = next(Δ) in cl(Δ) according to the permutation S; (iv) Compute all the transient or final trapezoids generated from Δ by splitting it by s_Δ (this generates $O(1)$ new trapezoids); and (v) Extract from cl(Δ) the conflict list cl(Δ') of each new trapezoid Δ', and compute next(Δ') as well.

Overall, this requires $O(k + l)$ time, where k is the number of transient trapezoids that are merged, and l is the total length of the conflict lists of these transient trapezoids. This is trivial to show for steps (i), (iii), (iv) and (v). To perform in step (ii) the merging of the conflict lists in linear time, one may use a global bit-vector structure. Namely, we initialize before the execution of the algorithm a bit-vector b of size n to be everywhere zero. To merge several conflict lists $L_1, \ldots, L_k$, we scan each list in turn, and for each of its elements s_j, we first test whether $b_j = 0$; if so, we add s_j to the output conflict list, and change b_j to 1. After creating the output conflict list in this manner, we scan the output list, turning off all the bits that got turned on.

In this manner, we have upgraded a transient leaf v of T into a final node v^*. We denote this operation by `Expand`(v). We can now continue going down in T, passing to the child of Δ that contains p and repeating recursively the above procedure at that child, until constructing and reaching the desired leaf of $\mathcal{HT}_\gamma$ that contains p (namely, until we reach a node that is final and had empty conflict list).

To complete the presentation of this point location mechanism, we describe `Expand`(v), the procedure that computes the 'sibling' transient trapezoids that are adjacent to the transient trapezoid of v.

Let τ be the transient trapezoid. Then either the top arc or the bottom arc of τ are the cause of the splitting that generated τ. In particular, next(τ_f) is either the top or bottom

arc of τ, where $\tau_f = \text{father}(\tau)$ denotes the trapezoid that τ was generated (split) from. This also implies that $\text{rank}(\tau) = \text{index}(\tau)$. Since τ is transient, one of the splitters of τ must be transient. Let ν denote such a transient splitter, and let us assume that ν is the right edge of τ. Note also that τ_f must be a final trapezoid, so in particular ν was generated from a final splitter (by the insertion of an arc that separated ν from the vertex that induced the bigger final splitter).

We compute the transient trapezoid τ' that lies to the right of τ and has the same top and bottom arcs, by taking the midpoint p of ν, and by performing a point-location query of p in T using (recursively) the same mechanism described above. During this point-location process, we always go down into the trapezoid Δ that contains p in its interior or on its left edge; see below for details. We stop as soon as we encounter a transient trapezoid τ' that has a left edge identical to ν. This happens when τ and τ' have the same top and bottom edges; namely, we stop when $\text{rank}(\tau) = \text{rank}(\tau')$. (Intuitively, if the trapezoid τ' has rank smaller than $\text{rank}(\tau)$, then either it fully contains ν in its interior, or its left edge is longer than (and contains) ν; the first time when both τ and τ' have the same connecting edge is when their top and bottom edges are identical, namely, when $\text{rank}(\tau) = \text{rank}(\tau')$.) See below for more details in the proof of correctness of the algorithm. We continue this process of collecting adjacent transient trapezoids using point-location queries on midpoints of transient splitters, until the two extreme splitters (the left splitter of the leftmost trapezoid in the sequence and the right splitter of the rightmost trapezoid) are non-transient. We take the union of those trapezoids to be the new expanded trapezoid. See Figure 3 for a scenario where such a merging occurs. A more detailed illustration is given in Appendix B below.

Of course, during this point-location process, we might be forced into going into parts of $\mathcal{HT}_\gamma$ that are rather remote from the point p. In such a case, we will compute those parts in an online manner, by performing `PointLocate` and `Expand` calls on the relevant trapezoids that we encounter while going down T. Thus, the point-location process is recursive, and might be quite substantial. Nevertheless, as will be argued below, the overall cost of the `PointLocate` and `Expand` operations is not excessive, so these operations are efficient in an amortized sense.

Let γ be the curve of the online walk whose zone we wish to compute. We consider γ to be a directed curve, supplied to us by the user through a function $\texttt{EscapePoint}_\gamma$. The function $\texttt{EscapePoint}_\gamma(p, \Delta)$ receives as input a point $p \in \gamma$, and a trapezoid Δ that contains p, and outputs the next intersection point of γ with $\partial\Delta$ following p. If γ ends before we reach $\partial\Delta$, the function returns `nil`. We assume (although this is not crucial for the algorithm) that γ does not intersect itself.

Let G_S denote the adjacency graph of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$. This is a graph having a vertex for each trapezoid in $\mathcal{A}_{\mathcal{VD}}(\hat{S})$, such that an edge connects two vertices if their corresponding trapezoids share a common vertical side. Under general position assumptions, each vertex in G_S has degree at most 4. It is easy to verify that a connected component of G_S corresponds to a face of $\mathcal{A}(\hat{S})$. Given a final leaf-trapezoid Δ, we can compute the face of $\mathcal{A}(\hat{S})$ that contains Δ by performing a DFS in G_S. This is done by performing point-location queries on appropriate points on the splitters of Δ, in a manner similar to that used in the `Expand` operation. This yields all the neighbors of Δ in G_S, and we continue in this manner until the entire connected component of G_S containing Δ is constructed.

Thus, given a walk γ, we can compute its zone by the algorithm `CompZoneOnline` de-

picted in Figure 4. See Appendix A for the pseudo-code of the main subroutines used by `CompZoneOnline`, and Appendix B for an illustration of the execution of `CompZoneOnline`.

As will be shown in Section 3.1, by the time the algorithm terminates, the final parts of T are contained in $\mathcal{HT}_\gamma$. (A proper inclusion might arise; see Remark 3.13.) In analyzing the performance of the algorithm, we first bound the overall expected time required to compute $\mathcal{HT}_\gamma$, which can be done by bounding the expected running time of `CompZoneWithOracle` (in an appropriate model of computation). Next, we will bound the additional time spent by the algorithm in traversing between adjacent trapezoids (i.e., the additional time spent in performing the point-location queries).

Remark 2.1 By skipping the expansion of the face that contains the current point p in `CompZoneOnline`, we get a more efficient algorithm that only computes the district $\mathcal{D}$ of the walk, that is, the collection of trapezoids in $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that γ crosses. There are cases where this will be sufficient; see Section 4 (e.g., in the adaptation of the algorithm for computing a level in an arrangement).

3 Analysis of `CompZoneOnline`

3.1 Correctness

In this subsection, we establish the correctness of `CompZoneOnline`. Before starting, we note that the correctness of `CompZoneWithOracle` is easier to establish, and follows routinely from standard considerations. We therefore omit any further details concerning this issue.

The main technical issues that arise in the proof of correctness have to do with the potentially complex patterns of exploring the DAG T that can arise during the recursive execution of the `PointLocate` and `Expand` routines. The first main step in the proof is to show that the `Expand` routine always terminates properly, with a transient trapezoid that is compatible with the input one. Once this is shown, the rest of the proof is a routine, though somewhat involved, task, which employs induction on the structure of T and on the sequence of steps executed by the algorithm.

Lemma 3.1 *During the execution of* `CompZoneOnline`*, the union of trapezoids of the leaves of T form a pairwise disjoint covering of the plane by vertical trapezoids.*

Proof: By induction on the steps of `CompZoneOnline`, noting that this is true initially, and that each step that modifies T either merges leaf trapezoids into a larger leaf trapezoid or splits a leaf trapezoid into subtrapezoids. ■

Corollary 3.2 *Each conflict list, as computed for a (transient or final) trapezoid Δ by the procedure* `CompZoneOnline`*, is the list of all arcs of S that cross (the interior of) Δ.*

Proof: By induction on the steps of `CompZoneOnline`. Observe that the region(s) that Δ was generated from cover Δ, and thus the union of their conflict lists must contain, by the induction hypothesis, the correct conflict list of Δ, which is thus correctly constructed by the appropriate `Expand` or `PointLocate` step. ■

ALGORITHM `CompZoneOnline`($\hat{S}$, p, $\texttt{EscapePoint}_\gamma$)

`Input:` A set $\hat{S}$ of n arcs, a starting point p of the walk,
and a function $\texttt{EscapePoint}_\gamma$ that represents the walk

`Output:` The decomposed zone $\mathcal{A}_{\gamma,\mathcal{VD}}(\hat{S})$ of γ in $\mathcal{A}(\hat{S})$

begin

Choose a random permutation $S = \langle s_1, s_2, \ldots, s_n \rangle$ of $\hat{S}$.

$T \leftarrow \{(\mathbb{R}^2, S)\}$ - a partial history DAG with a root corresponding to the whole plane; the conflict list of the root is the whole S.

$v \leftarrow \texttt{PointLocate}(p, \cdot)$,
where $\texttt{PointLocate}(p, \cdot)$ is the leaf of $\mathcal{HT}_\gamma$ whose associated trapezoid contains p. (The procedure has an additional flag parameter that we ignore here; it is used in cases where p lies on trapezoid boundaries; see Section A and below.)

/* All the paths in $\mathcal{HT}_\gamma$ from v to the root now exist in T. */

$\mathcal{D} \leftarrow \{\Delta_v\}$ (Δ_v is the trapezoid stored at v.)

while ($p \neq nil$) **do**

$p \leftarrow \texttt{EscapePoint}_\gamma(p, \Delta_v)$

$w \leftarrow \texttt{PointLocate}(p, +)$,
where $(p, +)$ denotes a point p^+ on γ just past p, and w is the next leaf of $\mathcal{HT}_\gamma$, such that $p^+ \in \Delta_w$. This is done by performing a point-location query in T, as described in the text, and expanding T accordingly.

$v \leftarrow w$

$\mathcal{D} \leftarrow \mathcal{D} \cup \{\Delta_v\}$

end while

if only the district of γ needs to be computed **then**

return $\mathcal{D}$ (the district of γ in $\mathcal{A}_{\mathcal{VD}}(\hat{S})$)

$\mathcal{Z} \leftarrow \emptyset$

for each $\Delta \in \mathcal{D}$

Compute the face F of Δ_v in $\mathcal{A}_{\gamma,\mathcal{VD}}(S)$ (if it had not yet been computed).

$\mathcal{Z} \leftarrow \mathcal{Z} \cup F$

end for

return $\mathcal{Z}$.

end `CompZoneOnline`

Figure 4: Algorithm for constructing the zone of a walk in an arrangement of arcs in an online manner. See Appendix A for the pseudo-code of the main subroutines used by `CompZoneOnline`, and Appendix B for an illustration of the execution of `CompZoneOnline`.

Corollary 3.3 *For a trapezoid Δ created by* `CompZoneOnline`*, all the arcs of $D(\Delta)$ appear in S before all the arcs of* $\mathrm{cl}(\Delta)$*.*

Consider an `Expand` operation that is triggered by point location at the midpoint p of a transient splitter ν that bounds a transient trapezoid τ, where τ has already been generated in T and we wish to find the transient trapezoid τ' that shares ν with τ as a common splitter. (It is easily verified that τ' uniquely defined, given the permutation S.) Let i denote the rank of τ. As noted, i is also the index of τ. Clearly s_i must be either the top or the bottom arc of τ. Assume, without loss of generality, that ν is the right splitter of τ and that s_i is the top edge of τ. Let s_j be the bottom arc of τ, with $j < i$.

Lemma 3.4 *During the execution of this* `Expand` *step, ignoring recursive calls, all the trapezoids that are either visited or generated fully contain ν either in their interior or within their left edge. Consequently, for any such trapezoid, except for the last one, either its conflict list contains s_i and s_j, or it contains s_i and s_j is the bottom arc of the trapezoid. Moreover, if this sequence of trapezoids includes a trapezoid that does contain ν on its left edge then all subsequent trapezoids in the sequence have this property.*

Proof: This is shown by induction on the sequence of steps of this (nonrecursive portion of the) execution of `Expand`. For this proof, we assume that during this execution, any recursive call to `Expand` terminates correctly, with a transient trapezoid that is adjacent to the trapezoid that initiated the recursive call and has the same top and bottom arcs. If this is not the case, we assume that the algorithm aborts at that point, and from that point on there is nothing to prove. (We thus modify the algorithm for the purpose of the proof, but a subsequent argument (in Lemma 3.5 below) will show that the algorithm never aborts.)

The first node that the `Expand` procedure visits is the root of T, and the claims clearly hold in this case. Assume they hold for all trapezoids up to and including a trapezoid Δ. Suppose first that Δ is final, and let $s_k = \mathrm{next}(\Delta)$. By induction hypothesis, the conflict list of Δ contains s_i, so we must have $k \leq i$. Then Corollary 3.3 implies that s_k does not cross the relative interior of ν. This implies that the subtrapezoid Δ' of Δ that is split from it by s_k and contains (a point slightly to the right of) the midpoint of ν must fully contain ν in its closure. If Δ contained ν on its left side then clearly this also holds for Δ'.

If Δ is transient then there are two subcases: If the top and bottom edges of Δ are, respectively, s_i and s_j, then the `Expand` procedure terminates and returns Δ; the claims clearly hold in this case. Otherwise, the `Expand` procedure executes a recursive call with the midpoint of some (transient) splitter of Δ. As argued above, we may assume that this recursive call returns a transient trapezoid compatible with Δ, in the above sense. We repeat this step, if needed, until we obtain a sequence of compatible transient trapezoids, including Δ, which cannot be expanded any further. We then merge all these trapezoids into a trapezoid Δ', which clearly must be final. It is obvious, by construction, that Δ' satisfies the first assertion of the lemma.

To prove the second assertion (for transient trapezoids), assume that Δ contains ν on its left edge ν_0. It suffices to show that ν_0 is not transient (which will imply that Δ is not merged with other trapezoids on its left, so that this left splitter will be contained in ν_0), so assume to the contrary that it is transient. As already noted, a transient splitter like ν_0 is generated when we insert an arc s_ℓ that delimits ν_0 and separates it from the vertex w that

induced it. Once this occurs, ν_0 will trigger a call to `Expand` which, if properly terminated, will erase ν_0 as it merges Δ with adjacent transient trapezoids. As argued above, we have $\ell < i$ (because $s_i \in \mathrm{cl}(\Delta)$).

The same argument also implies that the trapezoid τ_f from which τ has been split (by the insertion of s_i) has a non-transient right splitter ν^* that contains ν and extends all the way to the vertex w. Thus, by Corollary 3.2, the conflict list of τ_f contains s_ℓ, so $\mathrm{next}(\tau_f)$ cannot be s_i, contrary to assumption.

This induction step completes the proof of the lemma. ∎

As a consequence, it is easily verified by induction that, during the execution of this `Expand` step, including all recursive calls, no arc s_k with $k > i$ is processed.

We now show that each call to `Expand` terminates correctly.

Lemma 3.5 *Each point-location query at the midpoint of a transient splitter generates a "compatible" transient trapezoid; that is, a transient trapezoid adjacent to the current transient trapezoid, that has the same top and bottom arcs.*

Proof: Suppose the lemma is false, and consider the first call where this happens, where we order the calls in the order of returns from `Expand` (i.e. in postorder on the recursion forest). Let Δ be the transient trapezoid that initiated this call, and assume, without loss of generality, that the call started at the right splitter ν of Δ, and that the top and bottom edges of Δ are, respectively, s_i and s_j, with $i > j$. Arguing as in the proof of Lemma 3.4, we have that during the execution of the nonrecursive portion of this call, the procedure visits or generates a sequence Σ of trapezoids, each of which contains ν, and, by the induction hypothesis, all recursive calls that it executes terminate properly.

Let s_k, s_ℓ, with $k < \ell$, be the two arcs that intersect at the vertex w that induced a splitter that contains ν (the case where w is an endpoint of an ark s_k is handled similarly). Clearly, we must have $k < \ell < i$. Moreover, w must lie above ν, or else the insertion of s_j would have disconnected ν from w; since s_i has not yet been inserted at that stage, it easily follows that ν could not have been formed at all. It is easily seen that any trapezoid $\tau \in \Sigma$ that contains ν in its interior must either contain s_k and s_ℓ in its conflict list or be bounded by s_k and contain s_ℓ in its conflict list. Hence, as can be easily verified, s_k and s_ℓ will eventually be processed in splitting operations during this execution, and will consequently generate trapezoids in Σ that contain ν on their left edge. Moreover, either s_i and s_j appear in the conflict list of any such trapezoid, or s_i appears in the conflict list and s_j bounds the trapezoid on the bottom. Eventually, s_i will thus be inserted, and then the resulting trapezoid will be compatible with Δ and the procedure will terminate correctly, contrary to assumption. ∎

Lemma 3.6 *For any final trapezoid Δ created by the* `Expand` *procedure, during the execution of* `CompZoneOnline`*, Δ is a trapezoid of $\mathcal{A}_{\mathcal{VD}}(S_i)$, where $i = \mathrm{index}(\Delta)$.*

Proof: By induction on the *depth* of the nodes in T, where the depth of a node is defined to be the length of the longest path from the root of T to this node.

The only node of depth 0 is the root of T, which is being computed during the initialization of the algorithm, and is also the only trapezoid in $\mathcal{A}_{\mathcal{VD}}(S_0)$.

Assume that the induction hypothesis holds for all trapezoids of depth $< k$, and let Δ be a final trapezoid of depth k in T. There are two cases to consider: (a) Δ has been generated by splitting a final trapezoid τ by inserting some arc s_i. (b) Δ has been obtained by merging several transient trapezoids.

In case (a), by induction hypothesis, τ is a trapezoid of $\mathcal{A}_{\mathcal{VD}}(S_j)$, where $j = \text{index}(\tau)$. By Corollary 3.2, the conflict list of τ is computed correctly, so no arc s_ℓ, with $j < \ell < i$, crosses τ. Hence τ is also a trapezoid of $\mathcal{A}_{\mathcal{VD}}(S_{i-1})$, and, by construction, any final trapezoid obtained by splitting τ with s_i is a trapezoid of $\mathcal{A}_{\mathcal{VD}}(S_i)$. Since $i = \text{index}(\Delta)$, this completes the induction step in this case.

In case (b), let $\tau_1, \ldots, \tau_m$ be the transient trapezoids whose merging forms Δ. By construction, all of them have the same top arc, say s_i, and the same bottom arc, say s_j. Suppose, without loss of generality, that $i > j$. Since these trapezoids are transient, we have, as argued above, $\text{index}(\tau_\ell) = \text{rank}(\tau_\ell) = i$ for each $\ell = 1, \ldots, m$. In particular, this also holds for the leftmost and rightmost trapezoids in this sequence, which is easily seen to imply that all the arcs in $D(\Delta)$ belong to S_i. Finally, since the conflict lists of the τ_ℓ's are computed correctly, and the conflict list of Δ is the union of these lists, it follows that no arc in that list belongs to S_i. In other words, Δ is a final trapezoid defined by arcs of S_i and no arc of S_i crosses its interior. This readily implies that Δ is a trapezoid of $\mathcal{A}_{\mathcal{VD}}(S_i)$, and this completes the induction step in this case, and thus completes the proof of the lemma. ■

Lemma 3.7 *All the (final) non-leaf nodes computed by* `CompZoneOnline` *appear in* $\mathcal{HT}_\gamma$.

Proof: What we really need to show is that each non-leaf trapezoid Δ in T belongs to $\mathcal{A}_{\gamma,\mathcal{VD}}(S_{i-1})$, where s_i is the arc that has split Δ, thus making it a non-leaf. Indeed, any such trapezoid belongs to $\mathcal{HT}_\gamma$, by construction and by correctness of `CompZoneWithOracle`.

The proof proceeds by induction on the sequence of trapezoid-splitting steps taken by `CompZoneOnline`. That is, a final non-leaf trapezoid Δ will be considered when it is split by an arc (thus becoming a non-leaf). The claim clearly holds initially for the whole plane, stored at the root of T (and of $\mathcal{HT}_\gamma$). Let Δ be a non-leaf final trapezoid generated in T and then split by an arc s_i by `CompZoneOnline`, and suppose that all previously-split non-leaf trapezoids in T appear in $\mathcal{HT}_\gamma$. The trapezoid Δ has been split as part of a point-location query with some point p. Suppose first that $p \in \gamma$ or p lies on a splitter of a final trapezoid of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ (the later case occurs when we expand the district of γ into its zone). Since Δ is split at that point, it follows by construction that $p \in \Delta$ and therefore Δ belongs to $\mathcal{A}_{\gamma,\mathcal{VD}}(S_{i-1})$.

Otherwise, p is the midpoint of some transient splitter ν, and the point location of p is part of some `Expand` operation. Again, p (and in fact the whole segment ν) belongs to Δ. Let τ be the transient trapezoid bounded by ν that has triggered that `Expand` operation, and let τ_0 be the first transient trapezoid (in the execution order of `CompZoneOnline`) in the sequence of compatible trapezoids that includes τ. let τ_f be the father of τ_0, which is a final (non-leaf) trapezoid from which τ_0 has been split by an arc s_j. By induction hypothesis, τ_f belongs to $\mathcal{A}_{\gamma,\mathcal{VD}}(S_{j-1})$. Hence τ_0 is fully contained (as a set) in the zone of γ in $\mathcal{A}(S_{j-1})$. By the preceding analysis, we know that (a) s_j is the top or bottom arc of τ_0 and of τ, and (b) during the whole `Expand` operation that started at τ_0 no arc beyond s_j is inserted. It follows that p belongs to the zone of γ in $\mathcal{A}(S_{j-1})$, and that s_i, the arc that has split Δ, must

satisfy $i \leq j$. Hence Δ belongs to the zone of γ in $\mathcal{A}(S_{i-1})$, which establishes the induction step and thus completes the proof of the lemma. ∎

Lemma 3.8 *All the trapezoids of $\mathcal{A}_{\gamma,\mathcal{VD}}(\hat{S})$ are computed by* `CompZoneOnline`.

Proof: Consider first the sequence of trapezoids of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that constitute the district of γ in the full arrangement, in the order in which they are traversed by γ. The proof first proceeds by induction on this sequence, arguing that each of these trapezoids is produced by `CompZoneOnline`. As implied by the preceding analysis, each point-location query with a point p returns the trapezoid of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that contains p. Hence, if the k-th trapezoid in the above sequence has been generated, the $EscapePoint_\gamma$ function produces a point that lies in the next trapezoid, which will therefore also be produced by `CompZoneOnline`. A similar argument holds for the subsequent stage of the algorithm that expands the district of γ into its zone. ∎

3.2 Running Time

In this subsection, we first analyze the performance of `CompZoneWithOracle`, and then use this analysis to bound the expected running time of `CompZoneOnline`. We assume that `CompZoneWithOracle` maintains for each trapezoid Δ its conflict list $\mathrm{cl}(\Delta)$ that stores the set of arcs that cross it. We also assume that each conflict list $\mathrm{cl}(\Delta)$ stores its minimal element $\mathrm{next}(\Delta)$ in the ordering of S, and that each yet uninserted arc s maintains a list of all current trapezoids Δ for which $\mathrm{next}(\Delta) = s$. Then it is easy to see that the running time of the algorithm is proportional to the overall size of all the conflict lists that it generates. We also assume that a call to the Oracle $\mathcal{O}$ takes $O(1)$ time.

Lemma 3.9 *The algorithm* `CompZoneWithOracle` *computes the zone of γ in $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ in $O(\lambda_{t+2}(n+m)\log n)$ expected time, and the expected number of trapezoids that it generates is $O(\lambda_{t+2}(n+m))$.*

Proof: The proof is a straightforward adaptation of the proof of [CEG+93].[3] Specifically, we first make m cuts at the points where γ crosses the arcs of $\hat{S}$, thereby obtaining a collection of $m+n$ subarcs, so that $\mathcal{Z}_\gamma(\hat{S})$ becomes a single face in the new arrangement. We now insert the original arcs of $\hat{S}$ one by one in the random order S. It is easily checked that the expected number of subarcs of the r random arcs have been inserted is $O\big(r + \frac{m}{n}r\big)$. Thus, the expected number of trapezoids maintained in the r-th iteration is $O\big(\lambda_{t+2}\big(r + \frac{m}{n}r\big)\big)$. Using Clarkson-Shor sampling technique [CS89, Mul94] implies that the overall expected weight of those trapezoids in the r-th iteration is $O(\lambda_{t+2}(n+m))$. However, the expected work in the r-th iteration is the expected weight of the newly created trapezoids, and the probability of a trapezoid (that appears in the set of trapezoids maintained by the algorithm after the r-th iteration) to be created in the r-th iteration is $O(1/r)$. We conclude, that the expected work in the r-th iteration is $O(\lambda_{t+2}(n+m)/r)$. Summing over $r = 2, \ldots, n$, we conclude that the expected overall running time of the algorithm is $O(\lambda_{t+2}(n+m)\log n)$. ∎

[3] The algorithm of [CEG+93] has some additional overhead that is not required in `CompZoneWithOracle`

Lemma 3.10 *The expected number of transient trapezoids generated by* `CompZoneOnline` *is* $O(\lambda_{t+2}(n+m))$*, and the expected total size of their conflict lists is* $O(\lambda_{t+2}(n+m)\log n)$*.*

Proof: Each final trapezoid generated by `CompZoneOnline` might be split into $O(1)$ transient trapezoids. Each final (non-leaf) trapezoid computed by `CompZoneOnline` is also computed by `CompZoneWithOracle`, as follows from Lemma 3.7. By Lemma 3.9, the expected number of such trapezoids is $O(\lambda_{t+2}(n+m))$.

The second part of the lemma follows by a similar argument. ∎

Definition 3.11 A curve γ is *locally x-monotone* in $\mathcal{A}(\hat{S})$, if it can be decomposed inside each face of $\mathcal{A}(\hat{S})$ into a constant number of x-monotone curves.

Theorem 3.12 *The algorithm* `CompZoneOnline` *computes the zone of* γ *in* $\mathcal{A}(\hat{S})$ *in* $O(\lambda_{t+2}(n+m)\log n)$ *expected time, provided that* γ *is a locally x-monotone curve in* $\mathcal{A}(\hat{S})$*.*

Proof: The time spent by `CompZoneOnline` is bounded by the time required to construct the history DAG, by the time spent in maintaining the conflict lists of the trapezoids, and by the time spent on performing point-location queries, as we move from one trapezoid to another in $\mathcal{A}_{\gamma,\mathcal{VD}}(S)$.

By Lemmas 3.9 and 3.10, the expected time spent on maintaining the conflict lists of the trapezoids computed by the algorithm is $O(\lambda_{t+2}(n+m)\log n)$, since the total time spent on handling the conflict lists is proportional to their total length. By Lemma 3.10, the expected total size of those conflict lists is $O(\lambda_{t+2}(n+m)\log n)$.

Moreover, the depth of the history DAG constructed by the algorithm is $O(\log n)$ with a probability polynomially close to 1 [Mul94]. Thus, the expected time spent directly on performing a single point-location query (that is, the number of trapezoids that contain the query point and are visited or generated during this point location step) is $O(\log n)$. The curve γ is locally x-monotone, which implies that it intersects each splitter of any trapezoid of any $\mathcal{A}_{\gamma,\mathcal{VD}}(\hat{S})$ at most $O(1)$ times. Thus, the expected number of point-location queries performed by the algorithm is proportional to the expected number of transient and final trapezoids created, plus $O(m)$. By Lemma 3.10, we have that the expected running time is

$$O\Big(\lambda_{t+2}(n+m)\log n + (\lambda_{t+2}(n+m)+m)\log n\Big) = O(\lambda_{t+2}(n+m)\log n)\,.$$

∎

Remark 3.13 Note that `CompZoneWithOracle` computes the zone of γ in $\mathcal{A}_{\mathcal{VD}}(S_i)$, for each $i = 1,\ldots,n$. As a consequence, it might compute a trapezoid $\Delta \in \mathcal{A}_{\gamma,\mathcal{VD}}(S_i)$ that does not intersect the zone of γ in $\mathcal{A}_{\gamma,\mathcal{VD}}(S)$. In particular, such a trapezoid Δ will not be computed by `CompZoneOnline`. This is a slackness in our analysis that we currently do not know whether it can be exploited to further improve the analysis of the algorithm (we suspect that it cannot improve the above asymptotic bound on the running time).

Remark 3.14 The only classical result of this type that we are aware of, is due to Overmars and van Leeuwen [OvL81]. It maintains dynamically the intersection of n halfplanes in (deterministic) $O(\log^2 n)$ time for each insertion or deletion operation. This procedure can be used to perform walks inside line arrangements in (deterministic) $O((n+m)\log^2 n)$ time,

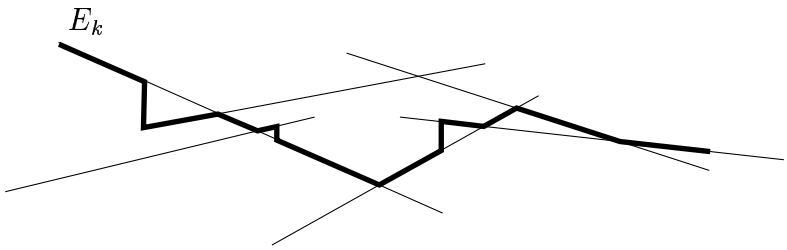

Figure 5: The first level in an arrangement of segments (the vertical edges show the jump discontinuities of the level, but are not part of the level).

where m is the number of intersections of the walk with the lines. Our algorithm is somewhat simpler and is faster than the algorithm of [OvL81] by nearly a logarithmic factor, and, most importantly, applies to more general arrangements.[4]

The technique of [OvL81] can also be used to perform *x-monotone* online walks in arrangement of segments. We simply regard each segment as the full line that contains it, but make it active only when the walk passes vertically above or below the segment. This means that, in addition to the usual updates that the algorithm performs, we also insert (resp. delete) segments when the walk becomes co-vertical with their left (resp. right) endpoint. (If the walk is not x-monotone, this may slow down the algorithm considerably.)

As for general arcs (or non-monotone walks in arrangements of segments), we are not aware of any result of this type in the literature. Of course, if the curve γ is known in advance (and is simple, in the sense that one can compute quickly its intersections with any arc of $\hat{S}$), we can compute the single face containing γ in an appropriately modified arrangement (as in the proof of the general planar Zone Theorem [SA95, Theorem 5.11]; see also the proof of Lemma 3.9 using the algorithms of [dBDS95, CEG$^+$93]. These algorithms (especially the first one) are slightly simpler than the algorithm of Theorem 3.12, although they have the same expected performance. However, these algorithms are useless for online walks, and they are probably slower than our algorithm in practice, as they either maintain additional complicated data-structures [CEG$^+$93], or perform additional redundant computation of regions that lie outside the zone of γ [dBDS95].

Recently, several algorithms for performing online walks were implemented [AHH$^+$99], including a variant of the algorithm presented here, which exhibited satisfactory performance in practice.

4 Applications

In this section we present several applications of the algorithm `CompZoneOnline`.

[4]Recently, Chan [Cha99a] improved the result of [OvL81], providing a data structure for maintaining intersection of halfplanes in $O(\log^{1+\varepsilon} n)$ amortized time for each update. His data structure is considerably more complicated than ours and than that of Overmars and van Leeuwen, and currently seems to be only of theoretical significance. Moreover, our algorithm is still faster than Chan's (by a factor of $O(\log^{\varepsilon} n)$). Very recently, this result was further improved by [BJ00].

4.1 Computing a Level in an Arrangement of Arcs

In this subsection we show how to modify the algorithm of the previous section to compute a level in an arrangement of x-monotone arcs.

Definition 4.1 Let $\hat{S}$ be, as above, a set of n x-monotone arcs in the plane, any pair of which intersect at most t times (for some fixed constant t). We also assume, as above, that $\hat{S}$ is in general position. The *level* of a point in the plane is the number of arcs of $\hat{S}$ lying strictly below it. Consider the closure E_l of the set of all points on the arcs of $\hat{S}$ having level l (for $0 \leq l < n$). E_l is an x-monotone (not necessarily connected) curve (which is polygonal in the case of lines or segments), which is called the *level* l of the arrangement $\mathcal{A}(\hat{S})$. At x-coordinates where a vertical line intersects less than $l+1$ arcs of S, we consider E_l to be undefined.

Levels are a fundamental structure in computational and combinatorial geometry, and have been subject to intensive research in recent years (see [AACS98, Dey98, TT97, TT98]). Tight bounds on the complexity of a single level, even for arrangements of lines, proved to be surprisingly hard to obtain. Currently, the best known upper bound for the case of lines is $O(n(l+1)^{1/3})$ [Dey98], while the lower bound is $\Omega(n \log(l+1))$ [Ede87].[5] See also [AACS98, TT98] for weaker upper bounds for other classes of arcs.

First, note that if $\hat{S}$ is a set of lines, then, once we know the leftmost ray that belongs to E_l, the level l is locally defined: as we move from left to right along E_l, each time we encounter an intersection point (a vertex of $\mathcal{A}(\hat{S})$) we have to change the line that we traverse. (This is also depicted in Figure 5.) In particular, we can compute the level E_l in $O(\lambda_3(n+|E_l|)\log n)$ time, using `CompZoneOnline`. The same procedure can be used to compute a level in an arrangement of more general arcs. The only non-local behavior we have to watch for are jump discontinuities of the level caused when an endpoint of an arc appears below the current level, or when the current level reaches an endpoint of an arc (see Figure 5). See below for details concerning the handling of those jumps.

In the following, let $0 \leq l < n$ be a prescribed parameter. Let E_l denote the level l in the arrangement $\mathcal{A}(\hat{S})$.

The following adaptation of `CompZoneOnline` to our setting is rather straightforward, but we include it for the sake of completeness. We sort the endpoints of the arcs of $\hat{S}$ by their x-coordinates. Each time our walk reaches the x-coordinate of the next endpoint, we update E_l by jumping up or down to the next arc, if needed. This additional work requires $O(n \log n)$ time.

If the level reaches the x-coordinate x_0 of a right endpoint of an arc, past which there are fewer than $l+1$ arcs intersecting a vertical line, then the level lies on the highest arc just to the left of x_0 and it ceases to be defined just to the right of x_0. In this case, our walk climbs to the line $y = +\infty$ and moves along the line (effectively tracing a sequence of topmost trapezoids of $\mathcal{A}(\hat{S})$) until it reaches the x-coordinate of a left endpoint of an arc, following which we might have again $l+1$ arcs crossing a vertical line. If so the walk then descends on the topmost arc and continues to trace the level l. For simplicity, we continue the discussion of the algorithm assuming that E_l is everywhere defined.

[5] Recently, a slightly larger lower bound has been announced by G. Tóth [Tót99].

During the walk, we maintain the invariant that the top edge of the current trapezoid is part of E_l. To compute the first trapezoid in the walk, we compute the intersection of level l with the y-axis (this can be done by sorting the arcs according to their intersections with the y-axis). Let p_0 be this starting point. We perform a point-location query with p_0 in our virtual history DAG to compute the starting trapezoid Δ_0 (containing p_0 on its top arc).

Now, by walking to the right of Δ_0 we can compute the part of E_l lying to the right of the y-axis. Let Δ be the current trapezoid maintained by the algorithm, such that its top edge is a part of E_l. Let $p(\Delta)$ denote the top right vertex of Δ. By performing point-location queries in our partial history DAG T, we can compute all the trapezoids of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that contain $p(\Delta)$ (by our general position assumption, the number of such trapezoids is at most 6; this number materializes when $p(\Delta)$ lies in the intersection of two arcs). By inspecting this set of trapezoids, one can decide where E_l continues to the right of Δ, and determine the next trapezoid having E_l as its roof. The algorithm sets Δ to be this trapezoid.

If the algorithm reaches an x-coordinate of an endpoint of an arc, we have to update E_l by jumping up (if this is the right endpoint of an arc and it lies on or below the level) or down (if it is a left endpoint and lies below the level); namely, we set Δ to be the trapezoid lying above (or below) the current Δ.

The algorithm continues in this manner, until reaching the rightmost edge of E_l. The algorithm then performs a symmetric walk to the left of the y-axis to compute the other portion of the level.

Let `CompLevel` denote this modified algorithm. We summarize our result:

Theorem 4.2 *The algorithm* `CompLevel` *computes the level l in $\mathcal{A}(\hat{S})$ in $O(\lambda_{t+2}(n+|E_l|)\log n)$ expected time.*

Remark 4.3 Since `CompLevel` is online, we can use it to compute the first m' edges or vertices of E_l, in expected $O(\lambda_{t+2}(n+m')\log n)$ time.

Remark 4.4 A straightforward extension of `CompLevel` allows us to compute any connected path within the union of $\hat{S}$ (i.e., we restrict our "walk" to the arcs of $\hat{S}$) in an on-line manner, in randomized expected time $O(\lambda_{t+2}(m+n)\log n)$, where m is the number of vertices of the path. As above, the extended version can also handle vertical jumps between adjacent arcs during the walk.

Remark 4.5 For the case of lines, one can use the algorithm of [OvL81] to construct a level E_l in $O(n\log n + |E_l|\log^2 n)$ deterministic time, as described, e.g., in [EW86]. The same technique, with a simple modification, also works for the case of line segments, with the same complexity bounds. Our algorithm is faster in these cases by nearly a logarithmic factor.

As already mentioned, recently, Chan [Cha99a] presented a faster algorithm for the dynamic maintenance of the intersection of halfplanes, requiring $O(\log^{1+\varepsilon} n)$ amortized time for each operation. Thus, one can compute the level l in $O(n\log n + |E_l|\log^{1+\varepsilon} n)$ deterministic time. Chan [Cha99b] also showed that by using the algorithm of [AdBMS98] one can compute the level l in $O(n+|E_l|\alpha(n)^2\log n)$ randomized expected time. Those results were very recently improved by Brodal and Jacob [BJ00]. We note, however, that our algorithm is still faster and simpler than those two algorithms.

4.2 Other Applications

In this subsection, we provide some additional applications of `CompZoneOnline` and `CompLevel`.

Theorem 4.6 *Let L be a set of n lines in the plane, and let $0 < \varepsilon \leq 1$ be a prescribed constant. Then one can compute a $(1/r)$-cutting of $\mathcal{A}(L)$, having at most $(1+\varepsilon)(8r^2-2r+4)$ trapezoids. The expected running time of the algorithm is $O\big(\big(1+\frac{1}{\varepsilon}\big)\, nr\alpha(n)\log n\big)$, where $\alpha(n)$ is the inverse of the Ackermann function [SA95].*

Proof: Follows by plugging the algorithm of Theorem 4.2 and Remark 4.3 into the algorithm described in [Har00a]. ∎

For a discussion of cuttings of small asymptotic size, and their applications, see [Mat98, Har00a].

Remark 4.7 Theorem 4.6 improves the previous result of [Har00a] by almost a logarithmic factor.

Once we have computed the level l (in an arrangement of general arcs), we can clip the arcs to their portions below the level. Using those clipped arcs as input, we can compute the portion of the arrangement below the level l (i.e., the first l levels of $\mathcal{A}(\hat{S})$) in $O((m+n)\log n + r)$ time, where $m = |E_l|$ is the complexity of the level l, and r is the complexity of the first l levels of $\mathcal{A}(\hat{S})$, using, e.g., the algorithm described in [Mul94]. This improves over the previous result of [ERvK96] that computes this portion of the arrangement of lines in $O(n\log n + nl)$ time. (Note that this running time is *not* output sensitive: It is easy to come up with examples where the complexity of the first l levels is only $O(l^2)$.)

5 Implementation

The algorithm described in this paper was implemented and compared to some other heuristics/algorithms for constructing zone in an online manner for an arrangement of lines, see [AHH+99]. The source code of our program is available at [Har00b]. As a competing algorithm we had implemented the following variant `CompZonePoly`, which differs from `CompZoneOnline` in the following two key points:

- `CompZonePoly` does not perform merging of adjacent compatible regions. Thus, the history DAG is now a tree.
- Each node in the history tree corresponds to a convex polygon of bounded complexity. Namely, if the polygon that corresponds to a node has more than c edges (where c is a prescribed constant), then the polygon is being further split into two regions, and the corresponding node becomes the parent of two new nodes. The motivation for this variant is that the average number of vertices of a face in an arrangement of lines is about 4. Thus, in this representation most faces (and intermediate faces) will correspond to a single node in the history tree.

Currently, we do not have any bounds on the performance of `CompZonePoly` and we leave it as open question for further research. Nevertheless, `CompZonePoly` performs extremely well in practice, and was one of the two fastest algorithms tested in [AHH+99].

5.1 Geometric Filtering

To overcome the problems of robustness and degeneracies, we had used the exact arithmetic as provided by the rational numbers of LEDA [MN95]. Unfortunately, using exact arithmetic in a naive way, slowdown the program by a factor of 20–40 [Har00a]. One possible way to achieve reasonable performance is to use filtering techniques. Here, one uses representation of numbers that maintain the history of the computations that generated them, so that if necessary the computations are recomputed using a higher level of precision. Such arithmetic types are provided by LEDA real, and LEDA rational kernel. While filtering suppose to be a care-free approach to this problem, as one can apply it easily to a program without rewriting, its performance is still inferior compared to the technique described below.

The idea is to implement filtering in the geometric level. Here each geometric entity has two representations: one is a floating-point representation (i.e., inexact) and the other one is an implicit exact representation. For example, a point is represented in its floating-point Cartesian representation and its logical representation; that is, the geometric operations and entities used to create it. For example, a point p might be defined to be the point lying on the line l, and having the same x-coordinate as a point q.

Thus, when a geometric primitive is being called, it is first computed using floating-point arithmetic. If the computation result lies below a certain threshold, then the algorithm recompute the primitive using exact arithmetic. This might require computing the exact representation of the points and lines used in this primitive. While in general this is worse than arithmetic filtering, it performs better in our scenario, as the depth of computation of vertices and edges in planar arrangement is bounded.

Furthermore, since this representation perseveres the combinatorial information, one can use this information to resolve geometric decisions without resorting to exact arithmetic. For example, consider a point p that is defined to be the intersection point between the lines l_1, l_2, and the algorithm calls a primitive `isOnLine` to decide if p lies on the line l_1. Here, after the floating-point predicate had failed, the predicate decides using the logical representation of $p = l_1 \cap l_2$ that it lies on l_1, and thus it return `true`.

Note, this is a scenario where arithmetic filtering will perform badly, for in this case the arithmetic filtering will first carry out the computations using floating-point arithmetic, and after those operations fail the computations will be reperformed using higher level of precision, using some kind of a gap-bound so that it resolves the predicate correctly.

We refer to the above approach as *geometric filtering*. It seems to be the most natural approach to the problem of robustness, although the considerable benefits of this approach in practice are not widely known. For example, in the case of the algorithms of [AHH+99], the usage of geometric-filtering speeded up the algorithms by a factor of 2-3. In practice, virtually all computations are performed using floating-point arithmetic, and only negligible part of the computations resort to exact arithmetic.

A very similar idea was recently implemented by Funke and Mehlhorn [FM00]. For further details, about our approach, see [Har00b].

			CompZoneOnline			CompZonePoly	
Input	lines #	faces #	time	nodes #	nodes created	time	nodes #
test	42	26	0.053	383	457	0.009	203
reg2000	2,004	667	1.449	17,651	21,419	0.228	6,735
rgl2000	2,004	6,271	6.331	93,729	117,324	0.890	36,853
rnd2000	2,004	1,411	1.458	25,072	32,064	0.295	9,844
zon2000	2,004	11,803	10.754	191,089	245,602	1.857	75,067
big2000	2,004	1	1.215	11,649	15,409	0.662	6,779
reg8000	8,004	2,664	6.122	70,409	85,376	1.007	26,841
zon8000	8,002	47,411	49.252	811,471	1,039,358	8.551	318,637
big8000	8,004	1	5.150	46,965	62,184	2.845	27,344

Table 1: Results for the two implemntations. Running times are in seconds. The number of nodes created by `CompZoneOnline` is considerably larger than the final number of nodes in the resulting DAG, as the algorithm merge nodes during its execution.

5.2 Empirical Results

The testing was carried out using the inputs of [AHH+99], and the results are depicted in Table 1. The tests were performed on a dual Pentium II 450MhZ with 512MB memory using Linux. Each entry in the table is the average of 25 executions of the program on this input. As can be seen from the table, `CompZoneOnline` is considerably slower (by a factor 2–8). than `CompZonePoly`.

The disappointing performance provided by `CompZoneOnline` is mainly caused by to the expensive `Expand` operations (involving repeated point-location queries in the DAG). Of course, `CompZonePoly` does not perform `Expand` operations. Furthermore, as testified by Table 1, the usage of vertical trapezoids by `CompZoneOnline` is inherently inefficient as it blows up the number of nodes in the associated history structure by a factor of 2–3 compared to the number of nodes created by `CompZonePoly`.

In addition, the implementation of `CompZonePoly` associates with each line l in the conflict list of a region P, the two edges of ∂P that l intersects. When computing the conflict lists of the children of the node that corresponds to P, one can sometimes compute what conflict lists l belongs to, without executing a single geometric primitive. It is not clear how one can implement a similarly efficient scheme for the computation of the conflict lists of vertical trapezoids.

6 Conclusions

In this paper we have presented a new randomized algorithm for computing a zone in a planar arrangement, in an online fashion. This algorithm is the first efficient algorithm for the case of planar arcs, it performs faster (by nearly a logarithmic factor) than the algorithm of [OvL81] for the case of lines and for the case of an x-monotone walk in an arrangement of segments, and it is considerably simpler. (It is also faster and much simpler than the recent algorithm of [Cha99a].) We also presented an efficient randomized algorithm for computing a level in an arrangement of arcs in the plane, whose expected running time is faster than

any previous algorithm for this problem.

The main result of this paper relies on the application of point-location queries to compute the relevant parts of an "off-line" structure (i.e., the history DAG). The author believes that this technique should have additional applications. In particular, this approach might be useful also for algorithms in higher dimensions. We leave this as an open question for further research.

Although the resulting algorithm seems to be only a minor variant of previous algorithms [dBDS95, CEG+93], the author believes that the new algorithm supersede those algorithms: (i) Implementing the new algorithm was quite easy and does not require any advanced data-structure. In particular, since the algorithm does not keep geometric adjacency information in the vertical decomposition (unlike previous algorithms) its implementation is thus considerably easier. (ii) The algorithm only computes what it *must* compute, while [dBDS95] performs a lot of redundant computations, and (iii) the algorithm provides a powerful data-structure for online computation of parts of an arrangement, where the computation time is the same as a randomized incremental algorithm that uses an oracle. This enables one to compute a portion of an arrangement in a completely arbitrary order, in time identical to the time spent by a optimal randomized incremental algorithm. See [HS01, HI00] for results that use this observation.

It is somewhat surprising, that in most applications that uses Overmars and van Leeuwen [OvL81] (or Chan [Cha99a] improvement) data-structure — which is more flexible than our data-structures since it allows insertion and deletion — one can use our algorithm instead and the resulting algorithms are always faster. See [HS01] for more details.

The empirical results testifies that this algorithm is practical, although it is slower than the heuristic `CompZonePoly` we had also tested. We currently do not have any proof of performance bounds for `CompZonePoly`, and we leave this as an question for further research. Another striking conclusion from the empirical tests, is that using vertical decomposition in practice, is not efficient as using polygons having constant complexity, see [AHH+99, Har00a] for similar results. It seems that planar vertical-decomposition should be avoided in practice, as they give inferior performance. An additional reason to avoid vertical-decompositions in practice, is their vulnerability to degeneracies (for example, several vertices of the arrangement having the save x-coordinate, etc). However, if one computes the zone in arrangement of segments, or of general arcs, it seems that the usage of vertical trapezoids is most natural. We believe that in such scenarios `CompZoneOnline` will perform reasonably well compared to other algorithms.

Acknowledgments

The author wishes to thank Pankaj Agarwal, Danny Halperin, Chaim Linhart and Micha Sharir for helpful discussions concerning the problems studied in this paper and related problems.

References

[AACS98] P. K. Agarwal, B. Aronov, T. M. Chan, and M. Sharir. On levels in arrangements of lines, segments, planes, and triangles. *Discrete Comput. Geom.*, 19:315–331, 1998.

[AdBMS98] P. K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. *SIAM J. Comput.*, 27:654–667, 1998.

[AHH+99] Y. Aharoni, D. Halperin, I. Hanniel, S. Har-Peled, and C. Linhart. On-line zone construction in arrangements of lines in the plane. In *Proc. Workshop on Algorithm Engineering*, pages 139–153, 1999.

[BDH99] K.-F. Böhringer, B. Donald, and D. Halperin. The area bisectors of a polygon and force equilibria in programmable vector fields. *Discrete Comput. Geom.*, 22(2):269–285, 1999.

[BJ00] G. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time and $o(\log n \cdot \log\log n)$ update time. In *Proc. 7th Scand. Workshop Algorithm Theory*, pages 57–70, 2000.

[BY98] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated by H. Brönnimann.

[CEG+93] B. Chzelle, H. Edelsbrunner, L. J. Guibas, M. Sharir, and J. Snoeyink. Computing a face in an arrangement of line segments. *SIAM J. Comput.*, 22:1286–1302, 1993.

[Cha99a] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proc. 40th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 92–99, 1999.

[Cha99b] T. M. Chan. Remarks on k-level algorithms in the plane. manuscript. `http://www.cs.uwaterloo.ca/~tmchan/pub.html`, 1999.

[CS89] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[dBDS95] M. de Berg, K. Dobrindt, and O. Schwarzkopf. On lazy randomized incremental construction. *Discrete Comput. Geom.*, 14:261–286, 1995.

[dBvKOS00] M. de Berg, M. van Kreveld, M. H. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[Dey98] T. K. Dey. Improved bounds for planar k-sets and related problems. *Discrete Comput. Geom.*, 19(3):373–382, 1998.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Heidelberg, 1987.

[ERvK96] H. Everett, J.-M. Robert, and M. van Kreveld. An optimal algorithm for the ($\leq k$)-levels, with applications to separation and transversal problems. *Internat. J. Comput. Geom. Appl.*, 6:247–261, 1996.

[EW86] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15:271–284, 1986.

[FM00] S. Funke and K. Mehlhorn. LOOK - a lazy object-oriented kernel for geometric computation. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 2000.

[Har00a] S. Har-Peled. Constructing planar cuttings in theory and practice. *SIAM J. Comput.*, 29(6):2016–2039, 2000.

[Har00b] S. Har-Peled. Taking a walk in a planar arrangement - the implementation. `http://www.math.tau.ac.il/~ sariel/papers/99/qwalk.html`, 2000.

[HI00] S. Har-Peled and P. Indyk. `http://www.cs.duke.edu/~sariel/papers/99/mst.html` When crossings count - approximating the minimum spanning tree. In *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pages 166–175, 2000.

[HS01] S. Har-Peled and M. Sharir. Online point location in planar arrangements and its applications. *Discrete Comput. Geom.*, 26:19–40, 2001.

[Mat98] J. Matoušek. On constants for cuttings in the plane. *Discrete Comput. Geom.*, 20:427–448, 1998.

[MN95] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38:96–102, 1995.

[Mul94] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[OvL81] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[SA95] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, 1995.

[Tót99] G. Tóth. Point sets with many k-sets. Monte Verita Conference on Discrete and Computational Geometry, 1999.

[TT97] H. Tamaki and T. Tokuyama. A characterization of planar graphs by pseudo-line arrangements. In *Proc. 8th Annu. Internat. Sympos. Algorithms Comput.*, volume 1350 of *Lecture Notes Comput. Sci.*, pages 123–132. Springer-Verlag, 1997.

[TT98] H. Tamaki and T. Tokuyama. How to cut pseudo-parabolas into segments. *Discrete Comput. Geom.*, 19:265–290, 1998.

A Appendix - Pseudo-code for Subroutines of `CompZoneOnline`

ALGORITHM `Split`(v)
`Input:` A final node v in the partial history DAG T
begin
$s \leftarrow$ next(Δ_v), where Δ_v is the trapezoid associated with the node v
$L \leftarrow$ `SplitGeom`(Δ_v, s),
where `SplitGeom`(Δ_v, s), as above, returns the collection of trapezoids that cover Δ_v, so that s does not intersect any of them in its interior.
for each $\tau \in L$ **do**
Create a new node w and attach it as a child of v in T.
Set Δ_w to τ
Compute cl(Δ_w) from cl(Δ_v)
Compute next(Δ_w), the first element of cl(Δ_w)
end for
end `Split`

Figure 6: Splitting a final node in T and creating its children

```
ALGORITHM PointLocateLeftCompatible(v, p, r)
   Input:  v - current node of T
           p - query point
           r - target rank of output trapezoid
   Output: A transient trapezoid of rank r having p on its left splitter
begin
   if rank(v) = r then
      return v
   if isTransient(v) then
      v ← Expand(v)
   if isLeaf(v) then
      Split(v)
   Let w be the child of v, so that Δ_w contains p either in its interior
        or on its left splitter
   return PointLocateLeftCompatible( w, p, r )
end PointLocateLeftCompatible
```

Figure 7: Computing a transient trapezoid in T that is left "compatible" with an input transient trapezoid, by carrying out a point-location query in T. The algorithm also uses a symmetric routine `PointLocateRightCompatible`, whose code is omitted.

```
ALGORITHM  Expand(v)
   Input:  v - current transient leaf node of T
   Output: A final node of T whose trapezoid contains Δ_v
begin
   if isFinal(Δ_v) then
      return v
   L ← {v}

   /* collect the sequence of transient trapezoids adjacent to each other
      to the right of Δ_v */
   temp ← v
   while isTransientRightSplitter(Δ_temp) do
      temp ← PointLocateLeftCompatible( root(T),
                        midPointRightSplitter( Δ_temp), rank(Δ_temp) )
      L ← L ∪ {temp}
   end while

   /* Similarly collect the sequence of transient trapezoids to the left of Δ_v */
   temp ← v
   while isTransientLeftSplitter(Δ_temp) do
      temp ← PointLocateRightCompatible( root(T),
                        midPointLeftSplitter( Δ_temp), rank(Δ_temp) )
      L ← L ∪ {temp}
   end while

   Δ ← ⋃_{u∈L} Δ_u
   Compute cl(Δ) and next(Δ) from the conflict lists of the
      nodes of L, using the global bit-vector technique.
   Add a new leaf node x to the partial history DAG T, mark x as final,
         and replace all nodes of L in T by x.
   Set Δ_x to Δ

   return x
end Expand
```

Figure 8: Expanding a transient leaf trapezoid of T to a final trapezoid containing it

ALGORITHM `PointLocate`(p, $flag$)

`Input:` p - a query point
$flag$ - since p usually lies on the boundary of a trapezoid, $flag$ indicates the side of the splitter or arc that contains p where the point location should take place

`Output:` The trapezoid of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that contains p (in its interior or on the appropriate edge dictated by $flag$)

begin
$v \leftarrow root(T)$

while ($\mathrm{cl}(\Delta_v) \neq \emptyset$) **do**
`Expand`(v)
`Split`(v)
$v \leftarrow$ child of v whose trapezoid contains p; if p lies on the boundary of several children trapezoids, choose the one that is compatible with $flag$
end while

return Δ_v
end `PointLocate`

Figure 9: The function that performs a point-location query; that is, it computes the necessary parts of the partial history DAG T, and returns the trapezoid of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that contains a query point.

B Appendix - Taking a Walk in Ten Easy Figures

In this appendix we illustrate, step by step, the action of processing a single point-location query by `CompZoneOnline`.

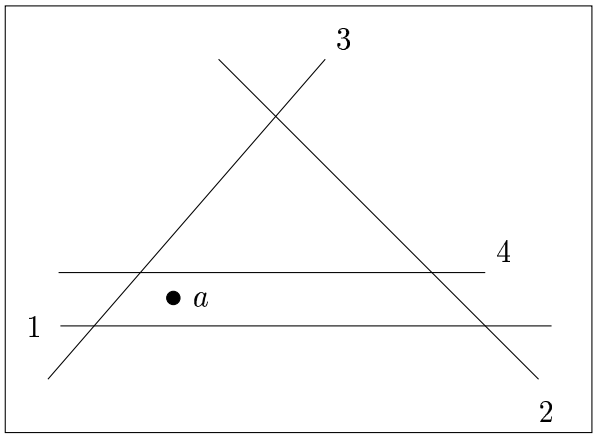

Figure 10: The input for `CompZoneOnline` is the set of segments $\hat{S} = \{1, 2, 3, 4\}$. We assume that the algorithm uses the permutation $S = (1, 2, 3, 4)$. We illustrate how `CompZoneOnline` carries out a point-location query to compute the trapezoid of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that contains the point a. We assume that this is the first query processed by the algorithm.

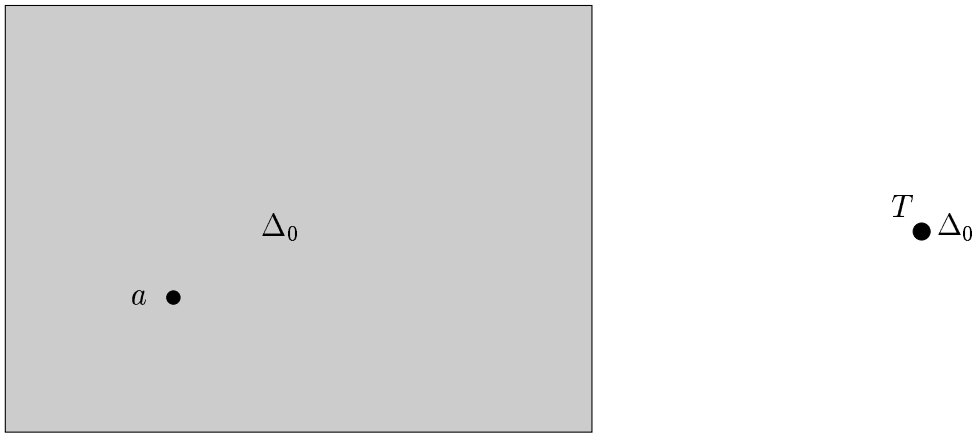

Figure 11: `CompZoneOnline` starts with the trapezoid $\Delta_0 = \mathbb{R}^2$ that corresponds to the root of the partial DAG T.

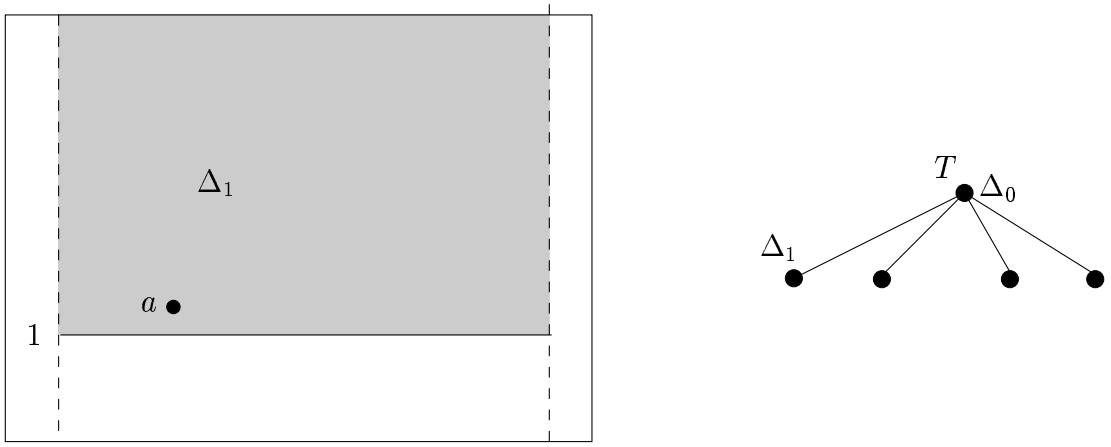

Figure 12: `CompZoneOnline` splits Δ_0 by the segment $\text{next}(\Delta_0) = 1$, and goes down in the DAG into the new node that corresponds to the trapezoid Δ_1. All children of Δ_0 (including Δ_1) are final.

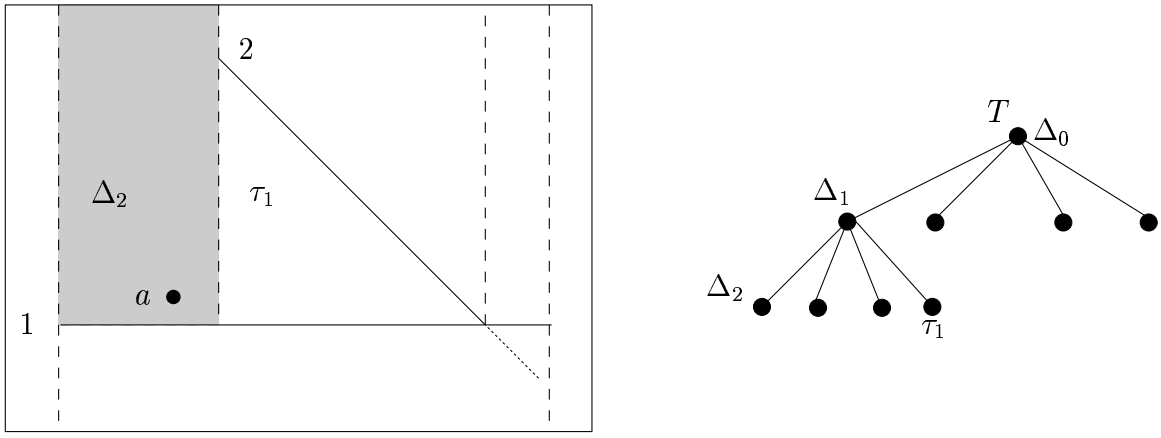

Figure 13: Since Δ_1 is final, it is split by $\text{next}(\Delta_1) = 2$ into four final subtrapezoids, and we go down to the child Δ_2 that contains a.

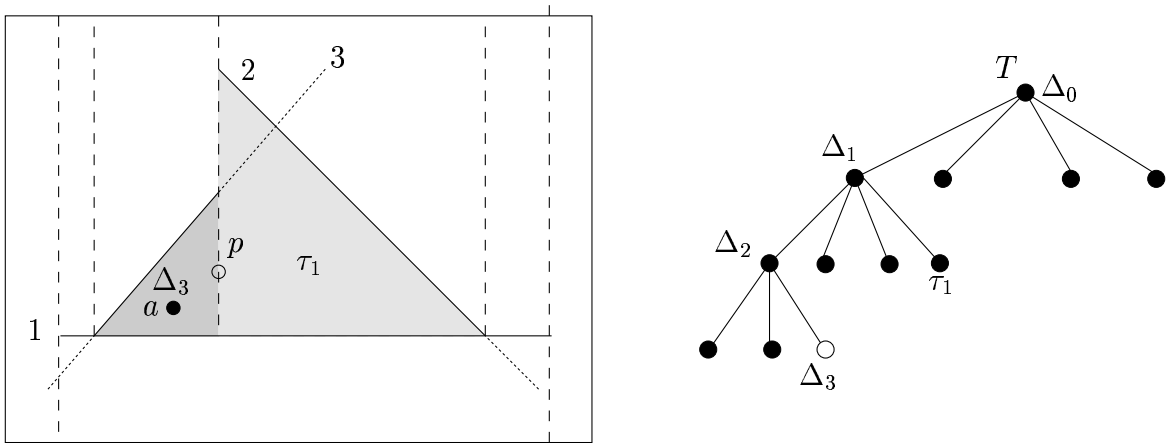

Figure 14: `CompZoneOnline` splits Δ_2 by $\text{next}(\Delta_2) = 3$ into three subtrapezoids. Two of them are final, and the third one, Δ_3, that contains a is transient (its right splitter is transient). We thus execute `Expand`(Δ_3), which performs a point-location query (with the midpoint p of the right splitter). The point location goes down in the partial DAG, through Δ_0 and Δ_1, and reaches the (final) leaf that stores τ_1. Since $\text{rank}(\Delta_3) = 3$ and $\text{rank}(\tau_1) = 2$, those two trapezoids are not compatible (this holds also because τ_1 is final whereas Δ_3 is not), and the algorithm continues by further splitting τ_1 by $\text{next}(\tau_1) = 3$.

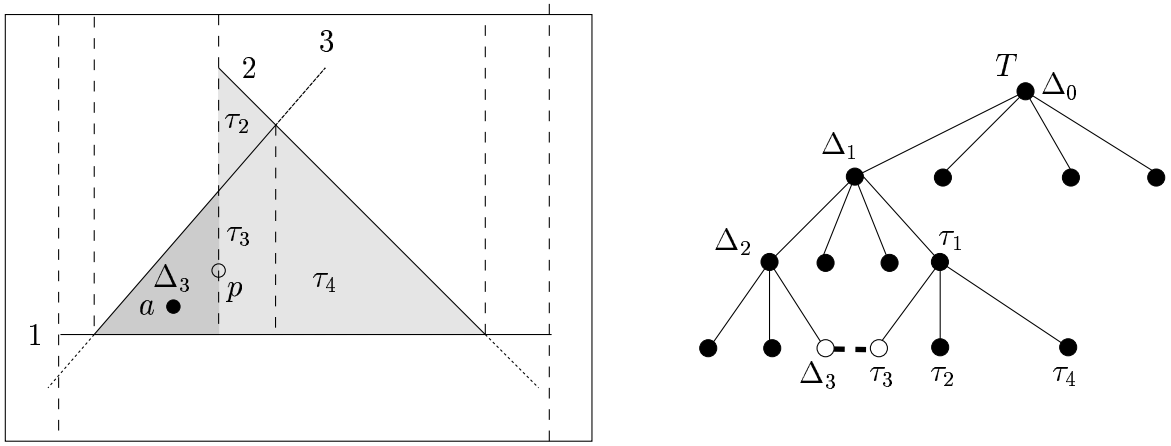

Figure 15: The splitting of τ_1 creates three children τ_2, τ_3, τ_4, of which τ_2 and τ_4 are final, whereas τ_3 is transient. `CompZoneOnline` goes down to the newly created τ_3, which contains p on its left splitter. Since τ_3 is transient and $\text{rank}(\tau_3) = \text{rank}(\Delta_3)$, it is compatible with Δ_3. Since the right splitter of τ_3 is final, and so is the (empty) left splitter of Δ_3 the execution of `Expand`(Δ_3) terminates.

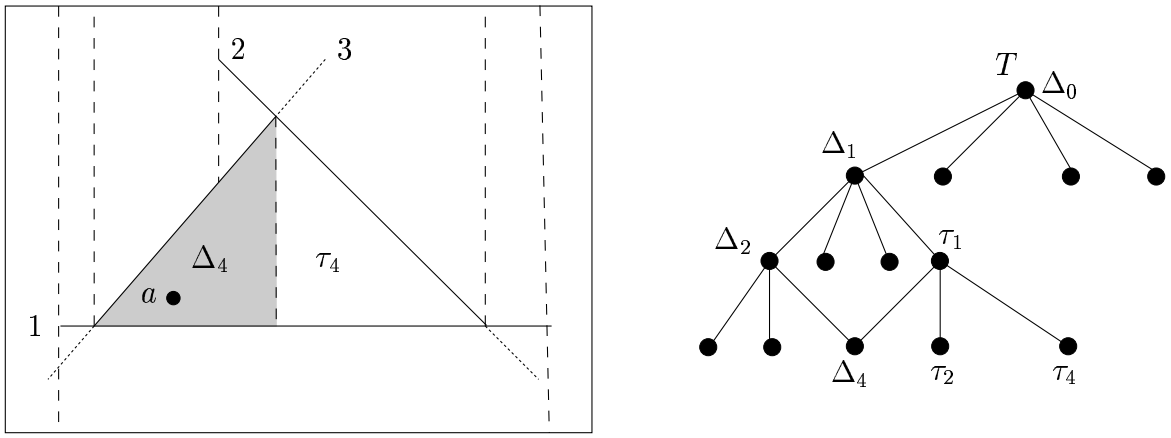

Figure 16: The trapezoids Δ_3 and τ_3, from the previous figure, are merged by `CompZoneOnline` to form the final trapezoid Δ_4. Since $\mathrm{cl}(\Delta_4)$ is not empty, we split it further by $\mathrm{next}(\Delta_4) = 4$.

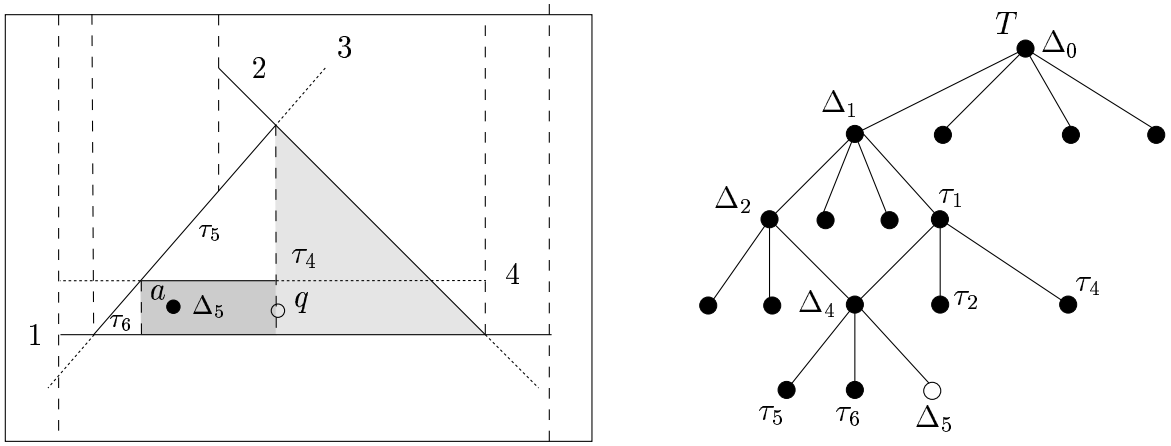

Figure 17: The splitting of Δ_4 creates three children Δ_5, τ_5, τ_6, of which τ_5 and τ_6 are final whereas Δ_5 is transient and contains a. `CompZoneOnline` goes into Δ_5, and since it is transient, `CompZoneOnline` calls `Expand`(Δ_5). A point-location query is performed, at the midpoint q of the right splitter of Δ_5. This query traverses in T the path $(\Delta_0, \Delta_1, \tau_1, \tau_4)$. The (final) trapezoid τ_4 is not compatible with Δ_5 so it is further split by the segment $\mathrm{next}(\tau_4) = 4$.

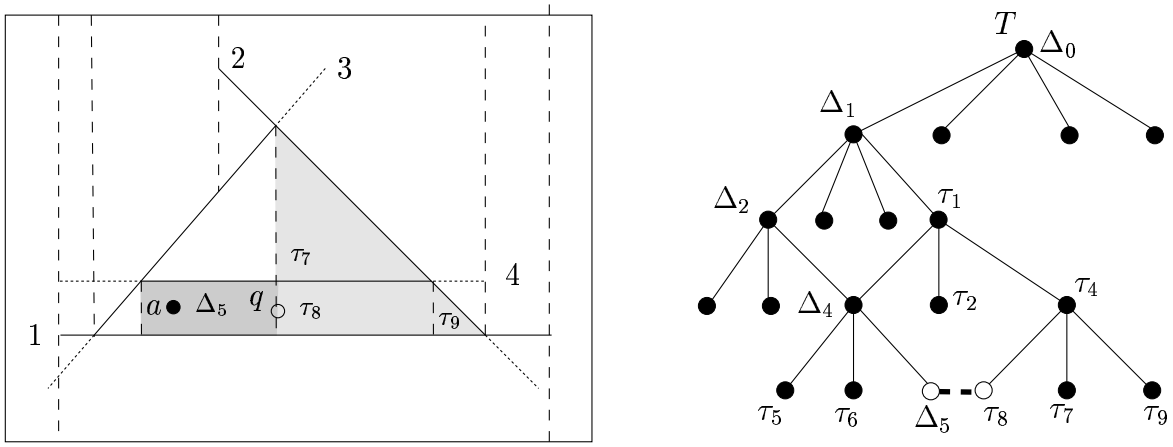

Figure 18: τ_4 is split into τ_7, τ_8, τ_9, of which only τ_8 is transient. `CompZoneOnline` goes into τ_8, which contains q on its left splitter. This trapezoid is compatible with Δ_5, and since both the left splitter of Δ_5 and the right splitter of τ_8 are final, `Expand`(Δ_5) terminates and merges both trapezoids.

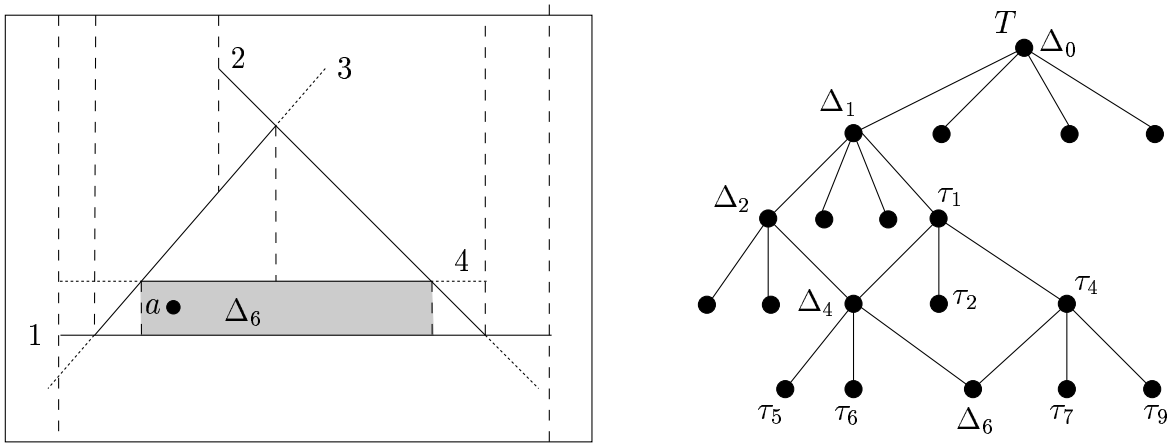

Figure 19: Voila! The newly formed final trapezoid Δ_6 contains our query point a, and its conflict list is empty. Thus, Δ_6 is the trapezoid of $\mathcal{A}_{\mathcal{VD}}(\hat{S})$ that contains a, and is output as such by `CompZoneOnline`.