Dynamic Maintenance of the Lower Envelope of Pseudo-Lines

Pankaj K. Agarwal[1], Ravid Cohen[2], Dan Halperin[2], and Wolfgang Mulzer[3]

1 Department of Computer Science, Duke University, Durham, NC 27708, USA
pankaj@cs.duke.edu

2 School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel

3 Institut für Informatik, Freie Universität Berlin, D-14195 Berlin, Germany
mulzer@inf.fu-berlin.de

Abstract

We present a fully dynamic data structure for the maintenance of lower envelopes of pseudo-lines. The structure has $O(\log^2 n)$ update time and $O(\log n)$ vertical ray shooting query time. To achieve this performance, we devise a new algorithm for finding the intersection between two lower envelopes of pseudo-lines in $O(\log n)$ time, using *tentative* binary search; the lower envelopes are special in that at $x = -\infty$ any pseudo-line contributing to the first envelope lies below every pseudo-line contributing to the second envelope. The structure requires $O(n)$ storage space.

1 Introduction

A set of pseudo-lines in the plane is a set of infinite x-monotone curves each pair of which intersects at exactly one point. Arrangements of pseudo-lines have been intensively studied in discrete and computational geometry; see the recent survey on arrangements [7] for a review of combinatorial bounds and algorithms for arrangements of pseudo-lines. In this paper we consider the following problem: Given n pseudo-lines in the plane, dynamically maintain their lower envelope such that one can efficiently answer vertical ray shooting queries from $y = -\infty$. The dynamization is under insertions and deletions. If we were given n lines (rather than pseudo-lines) then we could have used any of several efficient data structures for the purpose [3–5, 9, 10]; these are, however, not directly suitable for pseudo-lines. There are several structures that rely on shallow cuttings and can handle pseudo-lines [2,6,8]. The solution that we propose here is, however, considerably more efficient than what these structures offer. We devise a fully dynamic data structure with $O(\log^2 n)$ update-time, $O(\log n)$ vertical ray-shooting query-time, and $O(n)$ space for the maintenance of n pseudo-lines. The structure is a rather involved adaptation of the Overmars-van Leeuwen structure [10] to our setting, which matches the performance of the original algorithm for the case of lines. The key innovation is a new algorithm for finding the intersection between two lower envelopes of planar pseudo-lines in $O(\log n)$ time, using *tentative* binary search (where each pseudo-line in one envelope is “smaller” than every pseudo-line in the other envelope in a sense to be made precise below). To the best of our knowledge this is the most efficient data structure for the case of pseudo-lines to date.

35th European Workshop on Computational Geometry, Utrecht, The Netherlands, March 18–20, 2019.

2 Preliminaries

Let E be a finite family of pseudo-lines in the plane, and let ℓ be a vertical line strictly to the left of the left-most intersection point between lines in E (namely to the left of all the vertices of the arrangement $\mathcal{A}(E)$). The line ℓ defines a total order $\leq$ on the pseudo-lines in E, namely for $e_1, e_2 \in E$, we have $e_1 \leq e_2$ if and only if e_1 intersects ℓ below e_2. Since each pair of pseudo-lines in E crosses exactly once, it follows that if we consider a vertical line ℓ' strictly to the right of the right-most vertex of $\mathcal{A}(E)$, the order of the intersection points between ℓ' and E, from bottom to top, is exactly reversed.

The *lower envelope* $\mathcal{L}(E)$ of E is the x-monotone curve obtained by taking the pointwise minimum of the pseudo-lines in E. Combinatorially, the lower envelope $\mathcal{L}(E)$ is a sequence of connected segments of the pseudo-lines in E, where the first and last segment are unbounded. Two properties are crucial for our data structure: (A) every pseudo-line contributes at most one segment to $\mathcal{L}(E)$; and (B) the order of these segments corresponds exactly to the order $\leq$ on E defined above. In fact, our data structure works for every set of planar curves with properties (A) and (B) (with an appropriate order $\leq$), even if they are not pseudo-lines in the strict sense.

We assume a computational model in which primitive operations on pseudo-lines, such as computing the intersection point of two pseudo-lines or determining the intersection point of a pseudo-line with a vertical line can be performed in constant time.

3 Data structure and operations

The tree structure. Our primary data structure is a balanced binary search tree Ξ. Such a tree data structure supports insert and delete, each in $O(\log n)$ time. The leaves of Ξ contain the pseudo-lines, from left to right in the sorted order defined above. An internal node $v \in \Xi$ represents the lower envelope of the pseudo-lines in its subtree. More precisely, every leaf v of Ξ stores a single pseudo-line $e_v \in E$. For an inner node v of Ξ, we write $E(v)$ for the set of pseudo-lines in the subtree rooted at v. We denote the lower envelope of $E(v)$ by $\mathcal{L}\big(v\big)$. The inner node v has the following variables:

- f, ℓ, r: a pointer to the parent, left child and right child of v, respectively;
- max: the maximum pseudo-line in $E(v)$;
- Λ: a balanced binary search tree that stores the prefix or suffix of $\mathcal{L}(v)$ that is not on the lower envelope $\mathcal{L}(f)$ of the parent (in the root, we store the lower envelope of E). The leaves of Λ store the pseudo-lines that support the segments on the lower envelope, with the endpoints of the segments, sorted from left to right. An inner node of Λ stores the common point of the last segment in the left subtree and the first segment in the right subtree. We will need split and join operations on the binary trees, which can be implemented in $O(\log n)$ time.

Queries. We now describe the query operations available on our data structure. In a *vertical ray-shooting query*, we are given a value $x_0 \in \mathbb{R}$, and we would like to find the pseudo-line $e \in E$ where the vertical line $\ell : x = x_0$ intersects $\mathcal{L}(E)$. Since the root of Ξ explicitly stores $\mathcal{L}(E)$ in a balanced binary search tree, this query can be answered easily in $O(\log n)$ time.

▶ **Lemma 3.1.** *Let $\ell : x = x_0$ be a vertical ray shooting query. We can find the pseudo-line(s) where ℓ intersects $\mathcal{L}(E)$ in $O(\log n)$ time.*

Proof. Let r be the root of Ξ. We perform an explicit search for x_0 in $r.\Lambda$ and return the result. Since $r.\Lambda$ is a balanced binary search tree, this takes $O(\log n)$ time. ◀

Update. To insert or delete a pseudo-line e in Ξ, we follow the method of Overmars and van Leeuwen [10]. We delete or insert a leaf for e in Ξ using standard binary search tree techniques (the $v.\max$ pointers guide the search in Ξ). As we go down, we construct the lower envelopes for the nodes hanging off the search path, using split and join operations on the $v.\Lambda$ trees. Going back up, we recompute the information $v.\Lambda$ and $v.\max$. To update the $v.\Lambda$ trees, we need the following operation: given two lower envelopes $\mathcal{L}_\ell$ and $\mathcal{L}_r$, such that all pseudo-lines in $\mathcal{L}_\ell$ are smaller than all pseudo-lines in $\mathcal{L}_r$, compute the intersection point q of $\mathcal{L}_\ell$ and $\mathcal{L}_r$. In the next section, we will see how to do this in $O(\log n)$ time, where n is the size of E. Since there are $O(\log n)$ nodes in Ξ affected by an update, this procedure takes $O(\log^2 n)$ time. More details can be found in the literature [10, 11].

▶ **Lemma 3.2.** *It takes $O(\log^2 n)$ to insert or remove a pseudo-line in Ξ.*

4 Finding the intersection point of two lower envelopes

Given two lower envelopes $\mathcal{L}_\ell$ and $\mathcal{L}_r$ such that all pseudo-lines in $\mathcal{L}_\ell$ are smaller than all pseudo-lines in $\mathcal{L}_r$, we would like to find the intersection point q between $\mathcal{L}_\ell$ and $\mathcal{L}_r$. We assume that $\mathcal{L}_\ell$ and $\mathcal{L}_r$ are represented as balanced binary search trees. The leaves of $\mathcal{L}_\ell$ and $\mathcal{L}_r$ store the pseudo-line segments on the lower envelopes, sorted from left to right. We assume that the pseudo-line segments in the leaves are half-open, containing their right, but not their left endpoint in $\mathcal{L}_\ell$; and their left, but not their right endpoint in $\mathcal{L}_r$.[1] Thus, it is uniquely determined which leaves of $\mathcal{L}_\ell$ and $\mathcal{L}_r$ contain the intersection point q. A leaf v stores the pseudo-line $\mathcal{L}(v)$ that supports the segment for v, as well as an endpoint $v.p$ of the segment, namely the left endpoint if v is a leaf of $\mathcal{L}_\ell$, and the right endpoint if v is a leaf of $\mathcal{L}_r$.[2] An inner node v stores the intersection point $v.p$ between the largest pseudo-line in the left subtree $v.\ell$ of v and the smallest pseudo-line in the right subtree $v.r$ of v, together with the lower envelope $\mathcal{L}(v)$ of these two pseudo-lines. These trees can be obtained by appropriate split and join operations from the Λ trees stored in Ξ.

Let $u^* \in \mathcal{L}_\ell$ and $v^* \in \mathcal{L}_r$ be the leaves whose segments contain q. Let π_ℓ be the path in $\mathcal{L}_\ell$ from the root to u^* and π_r the path in $\mathcal{L}_r$ from the root to v^*. Our strategy is as follows: we simultaneously descend in $\mathcal{L}_\ell$ and in $\mathcal{L}_r$. Let u be the current node in $\mathcal{L}_\ell$ and v the current node in $\mathcal{L}_r$. In each step, we perform a local test on u and v to decide how to proceed. There are three possible outcomes:

1. $u.p$ is on or above $\mathcal{L}(v)$: the intersection point q is equal to or to the left of $u.p$. If u is an inner node, then u^* cannot lie in $u.r$; if u is a leaf, then u^* lies strictly to the left of u;
2. $v.p$ lies on or above $\mathcal{L}(u)$: the intersection point q is equal to or to the right of $v.p$. If v is an inner node, then v^* cannot lie in $v.\ell$; if v is a leaf, then v^* lies strictly to the right of v;
3. $u.p$ lies below $\mathcal{L}(v)$ and $v.p$ lies below $\mathcal{L}(u)$: then, $u.p$ lies strictly to the left of $v.p$ (since we are dealing with pseudo-lines). It must be the case that $u.p$ is strictly to the left of q or $v.p$ is strictly to the right of q (or both). In the former case, if u is an inner node, u^* lies in or to the right of $u.r$ and if u is a leaf, then u^* is u or a leaf to the right of u. In the latter case, if v is an inner node, v^* lies in or to the left of $v.\ell$ and if v is a leaf, then v^* is v or a leaf to the left of v; see Figure 1.

[1] We actually store both endpoints in the trees, but the intersection algorithm uses only one of them, depending on the role the tree plays in the algorithm.
[2] If the segment is unbounded, the endpoint might not exist. In this case, we use a symbolic endpoint at infinity that lies below every other pseudo-line.

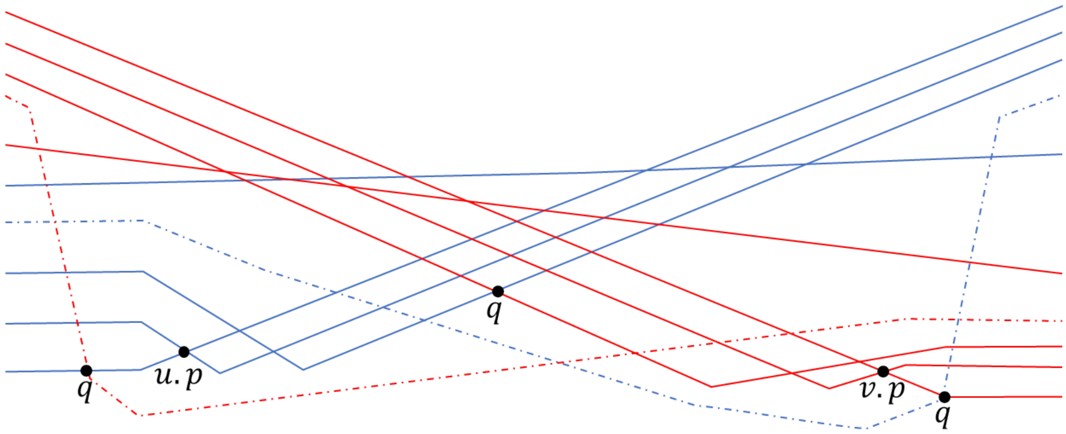


Figure 1 An example of Case 3. $\mathcal{L}_\ell$ is blue; $\mathcal{L}_r$ is red. The solid pseudo-lines are fixed. The dashed pseudo-lines are optional, namely, either none of the dashed pseudo-lines exists or exactly one of them exists. $u.p$ and $v.p$ are the current points; and Case 3 applies. Irrespective of the local situation at u and v, the intersection point can be to the left of $u.p$, between $u.p$ and $v.p$ or to the right of $v.p$, depending on which one of the dashed pseudo-lines exists.

Overmars and van Leeuwen [10, 11] describe a method for the case that $\mathcal{L}_\ell$ and $\mathcal{L}_r$ contain lines. Unfortunately, it is not clear how their strategy applies in the more general setting of pseudo-lines. The reason for this lies in Case 3: in this case, it is not immediately obvious how to proceed, because the correct step might be either to go to $u.r$ or to $v.\ell$. In the case of lines, Overmars and van Leeuwen can solve this ambiguity by comparing the slopes of the relevant lines. For pseudo-lines, however, this does not seem to be possible. For an example, refer to Figure 1, where the local situation at u and v does not determine the position of the intersection point q. Therefore, we present an alternative strategy.

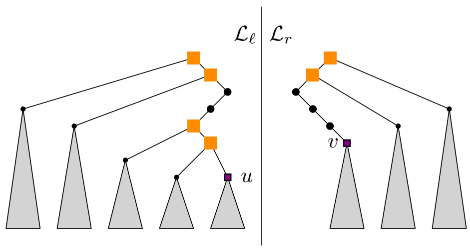


Figure 2 The invariant: the current search nodes are u and v. `uStack` contains all nodes on the path from the root to u where the path goes to a right child (orange squares), `vStack` contains all nodes from the root to v where the path goes to a left child (orange squares). The final leaves u^* and v^* are in one of the gray subtrees; and at least one of them is under u or under v.

We will maintain the invariant that the subtree at u contains u^* or the subtree at v contains v^* (or both). When comparing u and v, one of the three cases occurs. In Case 3, u^* must be in $u.r$, or v^* must be in $v.\ell$; see Figure 3.

We move u to $u.r$ and v to $v.\ell$. One of these moves must be correct, but the other move might be mistaken: we might have gone to $u.r$ even though u^* is in $u.\ell$ or to $v.\ell$ even though v^* is in $v.r$. To correct this, we remember the current u in a stack `uStack` and the current v in a stack `vStack`, so that we can revisit $u.\ell$ or $v.r$ if it becomes necessary. This leads to the general situation shown in Figure 2: u^* is below u or in a left subtree of a node on `uStack`,

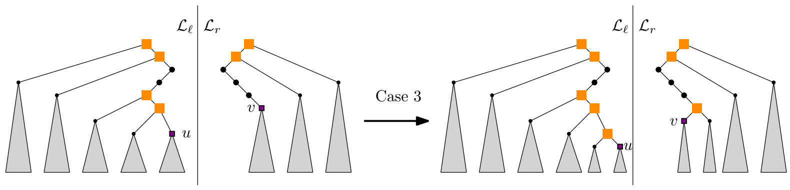


Figure 3 Comparing u to v: in Case 3, we know that u^* is in $u.r$ or v^* is in $v.\ell$; we go to $u.r$ and to $v.\ell$.

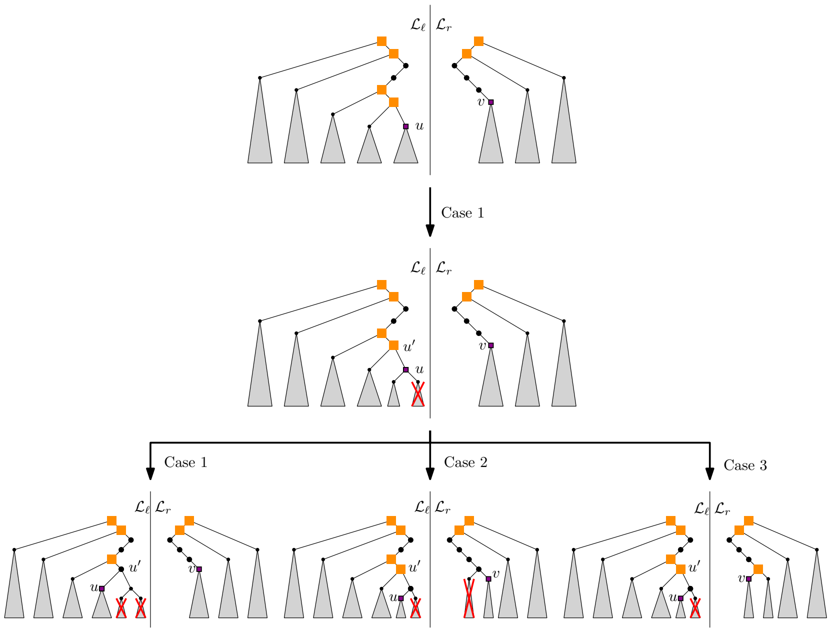
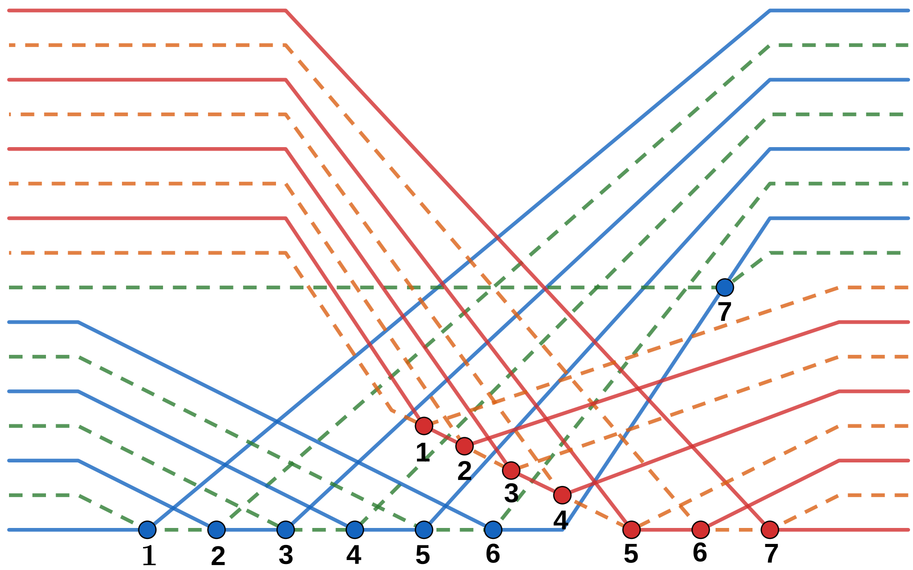
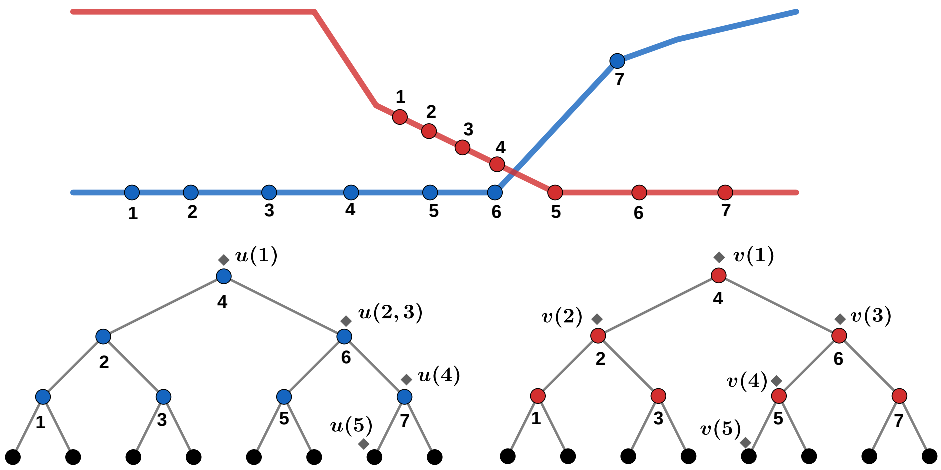


Figure 4 Comparing u to v: in Case 1, we know that u^* cannot be in $u.r$. We compare u' and v to decide how to proceed: in Case 1, we know that u^* cannot be in $u'.r$; we go to $u'.\ell$; in Case 2, we know that u^* cannot be in $u.r$ and that v^* cannot be in $v.\ell$; we go to $u.\ell$ and to $v.r$; in Case 3, we know that u^* is in $u'.r$ (and hence in $u.\ell$) or in $v.\ell$; we go to $u.\ell$ and to $v.\ell$. Case 2 is not shown as it is symmetric.

and v^* is below v or in a right subtree of a node on `vStack`, and at least one of u^* or v^* must be below u or v, respectively. Now, if Case 1 occurs when comparing u to v, we can exclude the possibility that u^* is in $u.r$. Thus, u^* might be in $u.\ell$, or in the left subtree of a node in `uStack`; see Figure 4. To make progress, we now compare u', the top of `uStack`, with v. Again, one of the three cases occurs. In Case 1, we can deduce that going to $u'.r$ was mistaken, and we move u to $u'.\ell$, while v does not move. In the other cases, we cannot rule out that u^* is to the right of u', and we move u to $u.\ell$, keeping the invariant that u^* is either below u or in the left subtree of a node on `uStack`. However, to ensure that the search progresses, we now must also move v. In Case 2, we can rule out $v.\ell$, and we move v to $v.r$. In Case 3, we move v to $v.\ell$. In this way, we keep the invariant and always make progress: in each step, we either discover a new node on the correct search paths, or we pop one erroneous move from one of the two stacks. Since the total length of the correct search paths is $O(\log n)$, and since we push an element onto the stack only when discovering a new correct node, the total search time is $O(\log n)$; see Figure 5 for an example run. For the full pseudocode and the formal proof see [1].



(a) Demonstration of two set of pseudo-lines and their lower envelope: (i) the blue and green pseudo-lines, (ii) the red and orange pseudo-lines. The blue and the red dots represents the intersection points on the lower envelopes.



(b) The top figure shows the lower envelope of (a). The bottom figure shows the the trees which maintain the lower envelopes. $u(i)$ and $v(i)$ shows the position of the pointers u and v at step i, during the search procedure.

Figure 5 Example of finding the intersection point of two lower envelopes:

Step	u	v	uStack	vStack	Procedure case
1	4	4	∅	∅	Case 3
2	6	2	4	4	Case 2 → Case 2
3	6	6	4	∅	Case 3
4	7	5	4, 6	6	Case 1 → Case 3
5	7*	5*	4, 6	6, 5	Case 3 → End

Acknowledgments. We thank Haim Kaplan and Micha Sharir for helpful discussions. Work by P.A. has been supported by NSF under grants CCF-15-13816, CCF-15-46392, and IIS-14-08846, by ARO grant W911NF-15-1-0408, and by grant 2012/229 from the U.S.-Israel Binational Science Foundation. Work by D.H. and R.C. has been supported in part by the Israel Science Foundation (grant no. 825/15), by the Blavatnik Computer Science Research Fund, by the Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, and by grants from Yandex and from Facebook. Work by W.M. has been partially supported by ERC STG 757609 and GIF grant 1367/2016.

References

1. Pankaj K. Agarwal, Ravid Cohen, Dan Halperin, and Wolfgang Mulzer. Dynamic maintenance of the lower envelope of pseudo-lines, 2019. `arXiv:arXiv:1902.09565`.
2. Pankaj K. Agarwal and Jiří Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.
3. Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull with optimal query time. In *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, pages 57–70, 2000.
4. Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS 2002)*, pages 617–626, 2002.
5. Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmaic amortized time. *J. ACM*, 48(1):1–12, 2001.
6. Timothy M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. *J. ACM*, 57(3):16:1–16:15, 2010.
7. Dan Halperin and Micha Sharir. Arrangements. In Jacob E. Goodman, Joseph O'Rourke, and Csaba Tóth, editors, *Handbook of Discrete and Computational Geometry*, chapter 28. Chapman & Hall/CRC, 3rd edition, 2017.
8. Haim Kaplan, Wolfgang Mulzer, Liam Roditty, Paul Seiferth, and Micha Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2495–2504, 2017.
9. Haim Kaplan, Robert Endre Tarjan, and Kostas Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 836–844, 2001.
10. Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, 1981.
11. Franco P. Preparata and Michael Ian Shamos. *Computational Geometry. An Introduction.* Springer-Verlag, New York, 1985.