Computational Geometry: Theory and Applications 1 (1991) 65–78
Elsevier

Off-line dynamic maintenance of the width of a planar point set*

Pankaj K. Agarwal**

Computer Science Department, Duke University, Durham, NC 27706, USA

Micha Sharir***

School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel
Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA

Communicated by Subhash Suri
Submitted 2 October 1990
Accepted 27 March 1991

Abstract

Agarwal, P.K. and M. Sharir, Off-line dynamic maintenance of the width of a planar point set, Computational Geometry: Theory and Applications 1 (1991) 65–78.

In this paper we present an efficient algorithm for the off-line dynamic maintenance of the width of a planar point set in the following restricted case: We are given a real parameter W and a sequence $\Sigma = (\sigma_1, \ldots, \sigma_n)$ of n insert and delete operations on a set S of points in $\mathbb{R}^2$, initially consisting of n points, and we want to determine whether there is an i such that the width of S the ith operation is less than or equal to W. Our algorithm runs in time $O(n \log^3 n)$ and uses $O(n)$ space.

1. Introduction and geometric preliminaries

Let S be a set of n points in $\mathbb{R}^2$. The *width* of S, denoted $\omega(S)$, is the minimum distance between two parallel supporting lines of S. Clearly, $\omega(S)$ is the same as the width of CH(S), the boundary of the convex hull of S.

A pair p, q of vertices of CH(S) are called an *antipodal pair* of vertices if there are two parallel supporting lines of CH(S)—one passing through p and the other

* A preliminary version of this paper has appeared in *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991, pp. 449–458.

** Supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), an NSF Science and Technology Center, under Grant STC-88-09648.

*** Supported by the Office of Naval Research under Grants, N00014-89-J-3042 and N00014-90-J-1284, by the National Science Foundation under Grant CCR-89-01484, by DIMACS, and by grants from the U.S.–Israeli Binational Science Foundation, the Fund for Basic Research administered by the Israeli Academy of Sciences, and the G.I.F., the German–Israeli Foundation for Scientific Research and Development.

through q. For a given set of n points in $\mathbb{R}^2$, there are only $O(n)$ antipodal pairs of vertices, and they can be computed in $O(n \log n)$ time (or even in $O(n)$ time if $CH(S)$ is given [12]). An edge-vertex pair (e, p) of $CH(S)$ is called an *antipodal pair* if there is a supporting line through p parallel to e (and e is not incident to p). Let $W > 0$ be a fixed real parameter. For an edge $e \in CH(S)$, let l_e be the line parallel to e at distance W and lying on the same side of e as S. Let Δ_e be the strip bounded by l_e and the line containing e. It is well known [8] that

Lemma 1.1. *For a set S of points in $\mathbb{R}^2$, $\omega(S) \leq W$ if and only if there is an antipodal edge-vertex pair (e, p) such that $p \in \Delta_e$.*

By this lemma, $\omega(S)$ can be computed in $O(n \log n)$ time (see [8]), or, if $CH(S)$ is known, in linear time.

The *dynamic width* maintenance problem in $\mathbb{R}^2$ is to maintain the width of a set S of points in $\mathbb{R}^2$ as we insert or delete points from S. What is needed here is a suitable data structure from which $\omega(S)$ can be quickly computed, and which can be maintained dynamically as points are added to or removed from S.

Problems involving dynamic maintenance of planar configurations have been studied in several papers. One distinguishes between *on-line* problems, where the sequence of updates is not known in advance and we must complete the processing of an update before the next operation is specified, and *off-line* problems, where the sequence of updates is known in advance. An early paper by Overmars and van Leeuwen [9] shows how to maintain $CH(S)$ itself on-line in $O(\log^2 n)$ time per update. They also give efficient on-line maintenance techniques for several other configurations, including maxima of a set of points in $\mathbb{R}^2$ (see also [4]) and intersection of halfplanes in $\mathbb{R}^2$. Efficient on-line maintenance techniques have also been given for the *diameter* of a set [15], in $O(\sqrt{n} \log n)$ time per update if we allow only deletions, for the *closest pair* [14], in $O(\log^3 n)$ amortized time per update (see also [15]), and for determining whether the radius of the minimum enclosing circle is less than a given parameter r, in time $O(\log^2 n)$ per update [6].

Edelsbrunner and Overmars [3] have shown that if a problem is *decomposable*[1] (see [11] for details), then the off-line dynamic version can be solved almost as efficiently as the static version. For example, the off-line version of the diameter problem can be solved in $O(\log^2 n)$ amortized time per update. A recent result of Hershberger and Suri [7] gives an algorithm for maintaining convex hulls in the off-line model, in $O(\log n)$ amortized time per update. A variant of the off-line model is the *semi on-line* model, where the sequence of insertions is not known in advance but when a point is inserted, we are told when it will be deleted. This model was originally considered by Gabow and Tarjan for the union-find problem

[1] A problem involving optimization over pairs of points in a set S is called *decomposable* if $S \times S$ can be broken into subsets, the optimizing pair in each subset can be computed separately, and the grand optimum can be obtained by combining all these partial results.

[5]. Dobkin and Suri [2] recently considered various problems related to maintaining planar configurations in the semi on-line model. They showed that in this model the diameter as well as the minimum distance can be maintained in $O(\log^2 n)$ amortized time per update (see also [13]).

However, maintaining the width of a set, even in the off-line model, appears to be a harder problem. Intuitively, this is because the width of S is a 'min–max' operator, that is

$$\omega(S) = \min_{e \in \mathrm{CH}(S)} \max_{p \in \mathrm{CH}(S)} \mathrm{dist}(p, e)$$

(where the minimum is taken over hull edges e and the maximum over hull vertices p). In contrast, the other parameters mentioned above (diameter, closest pair, etc.) are all 'max–max' or 'min–min' operators and are thus decomposable.

In this paper, we consider the following off-line and restricted version of the dynamic width problem:

> Given a real parameter $W > 0$, and a sequence $\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ of n insert/delete operations (applied in this order) on a set S of points in $\mathbb{R}^2$, initially consisting of at most n points, determine whether there is an i such that $\omega(S)$ after the ith operation is less than or equal to W.

Of course, we can explictly compute the width of S after each σ_i, but that would take $O(n^2)$ time. In this paper we present a reasonably simple algorithm for the dynamic width problem described above; our algorithm requires $O(n \log^3 n)$ time and linear storage assuming that $\log n$ bits can be stored in $O(1)$ space.

This dynamic width problem arises as a subproblem in a recent study of ours concerning the *two-line center problem,* in which we want to find the smallest W so that a given set of points in the plane can be covered by the union of two strips of width W each. This application, described in a companion paper [1], makes critical use of the algorithm given in this paper.

2. Overview of the algorithm

In this section we describe the overall structure of the algorithm. Let P be the set of points inserted to S (including points initially in S). If a point is inserted more than once, each of its insertions is considered as that of a different point, so P is actually a multi-set. For a point $p \in P$, we define its *life-span* to be the interval $I_p = [i_p, d_p]$, if σ_{i_p} inserts p into S, σ_{d_p} deletes p from S, and no operation in between those two affects p. For initial points $i_p = 0$; for points that are never deleted $d_p = n + 1$. Let $\mathcal{I}$ be the set of life-spans of points in P.

We construct a segment tree $\mathcal{T}$ on the interval $[0, n + 1]$ (see [12, Section 1.2] for details) and store the intervals of $\mathcal{I}$ in $\mathcal{T}$ as in [3]. The kth-leftmost leaf of $\mathcal{T}$, $0 \leq k \leq n$, corresponds to the interval $[k, k + 1]$, and each internal node $v \in \mathcal{T}$ is associated with a canonical interval $[l_v, r_v]$ in the usual way. Let P_v be the set of points whose life-spans are stored at v; let $|P_v| = n_v$. Note that $\sum_v n_v \leq 2n \log n$. By construction, for any point $p \in P_v$, $I_p \supset [l_v, r_v]$, that is, all points of P_v are inserted in S before (or at) σ_{l_v} and deleted from S after (or at) σ_{r_v}. For a node $v \in \mathcal{T}$, let π_v denote the path in $\mathcal{T}$ from the root to v, and let $S_v = \bigcup_{w \in \pi_v} P_w$. Thus, if z_k is the kth-leftmost leaf of $\mathcal{T}$, S_{z_k} coincides with S after the kth operation.

For each node $v \in \mathcal{T}$, we compute $\mathrm{CH}(P_v)$ and $\omega(P_v)$. This can easily be done in overall $O(n \log^2 n)$ time. If $\omega(P_v) > W$, then $\omega(S) > W$ during the entire interval $[l_v, r_v]$. Since we want to determine whether $\omega(S)$ is ever $\leq W$, and we know that this will not occur during this interval, we can discard the subtree rooted at v from further processing (see below for more details). Henceforth, we assume that for all nodes $v \in \mathcal{T}$, $\omega(P_v) \leq W$.

Here is a brief explanation of how the tree $\mathcal{T}$ will be used to maintain information about $\omega(S_z) = \omega(\bigcup_{u \in \pi_z} P_u)$, for each leaf z (namely to determine whether $\omega(S_z) \leq W$ for at least one leaf). We will traverse $\mathcal{T}$ in a depth-first manner and maintain a certain data structure $\mathcal{D}_v$ that provides information about $\omega(S_v)$ at each node v being visited. Suppose we have processed a node v. If $\omega(S_v) > W$, we prune $\mathcal{T}$ at the current node, back up, and continue our traversal through other branches of $\mathcal{T}$. If v is a leaf and $\omega(S_v) \leq W$, we stop the algorithm, since we have found an instance where $\omega(S) \leq W$. Otherwise we complete the traversal of $\mathcal{T}$ without stopping at any leaf and conclude that $\omega(S)$ always remains $> W$.

When we pass from a node u to a child v of u, we will already have $\mathcal{D}_u$ available, and our goal is to produce $\mathcal{D}_v$ by inserting the points of P_v into $\mathcal{D}_u$. We would like to achieve this in time proportional (up to a polylogarithmic factor) to the size of P_v, which will lead to an efficient solution to the overall problem. The point is that in this manner we have reduced our problem to subproblems in which S is to be updated by *insertions* only, which makes them considerably easier to solve.

3. The data structure

In this section we describe the data structure $\mathcal{D}$ that maintains information about the width of a set S, and that can be efficiently maintained as points are inserted into S. We first observe that the edge-vertex distances that affect the width of S need to be computed only between edges lying on the *lower hull* ($\mathrm{LCH}(S)$) of S and vertices of the *upper hull* $\mathrm{UCH}(S)$, or vice versa.

3.1. Description of the data structure

The data structure $\mathcal{D}$ consists of two balanced binary search trees (e.g. red-black trees); $\mathcal{U}$, built on the edges of $\mathrm{UCH}(S)$, and $\mathcal{L}$, built on the edges of $\mathrm{LCH}(S)$. We will describe only $\mathcal{U}$ in detail, since $\mathcal{L}$ is symmetric to $\mathcal{U}$.

The leaves of $\mathcal{U}$ store the edges of UCH(S) ordered from left to right. The internal nodes of $\mathcal{U}$ store secondary structures and auxiliary information, described below. For each edge e of the upper hull, let h_e denote the halfplane lying below the line l_e parallel to e and at distance W below it. For a node $\xi \in \mathcal{U}$, let $\mathcal{U}_\xi$ denote the subtree rooted at ξ. Each node $\xi \in \mathcal{U}$ implicitly stores the following two items:

(i) H_ξ: the intersection of the halfplanes, h_e, over all edges e stored at the leaves of $\mathcal{U}_\xi$.

(ii) A *relative width bit* $\omega_\xi^\star$, which is 0 if there is any edge e stored in $\mathcal{U}_\xi$ such that Δ_e contains the convex hull of S (in other words, the distance between the two supporting lines parallel to e is at most W), and is 1 otherwise. As detailed below, only some of the nodes will actually store this bit.

The first item is stored using the technique of Overmars and van Leeuwen [3]. More precisely, the lines l_e are stored at the leaves of $\mathcal{U}$ alongside the edges e. If δ is the parent of an internal node ξ in $\mathcal{U}$, then ξ stores the edges of H_ξ that do not appear in H_δ as a balanced search tree U_ξ, ordered by the slopes of the edges (e.g. we may use a variant of red-black tree [16]). If ζ, η are respectively the left and right children of ξ, then ξ also stores σ_ξ, the intersection point of ∂H_ζ and ∂H_η (since the lines l_e are ordered by slope, this point is unique). Thus each node ξ stores explicitly only the point σ_ξ and the portion U_ξ of H_ξ. The sets H_ξ (also maintained as appropriate balanced search trees) are computed on demand during our traversal of $\mathcal{U}$. In more detail, when going down in $\mathcal{U}$ from a node ξ to a (say, left) child ζ, we compute H_ζ from H_ξ by splitting H_ξ at σ_ξ and

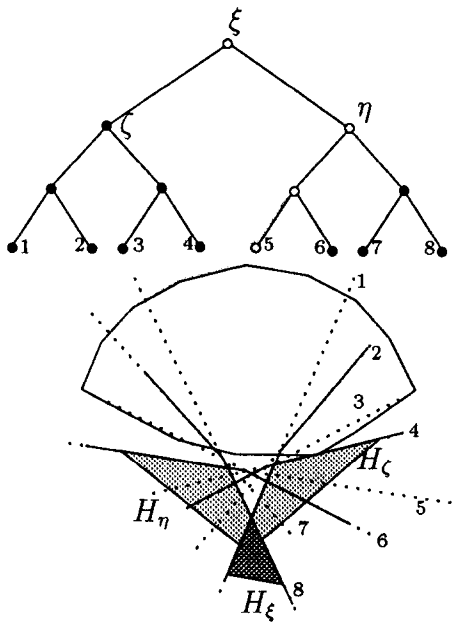

Fig. 1. Upper hill tree $\mathcal{U}$; the relative-width bit is 0 for white nodes and 1 for black nodes.

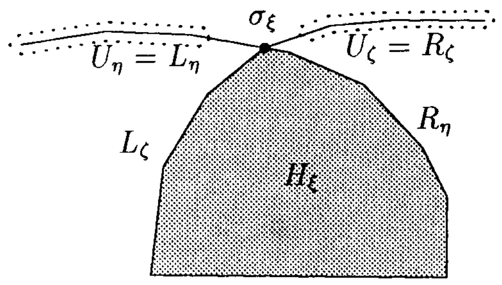

Fig. 2. H_ξ, H_ζ, H_η, U_ζ and U_η.

concatenating U_ξ to the right of the left portion of H_ξ (see Fig. 2). Similarly, when going up in $\mathcal{U}$ to a node ξ from its left and right children, ζ, η, we can compute H_ξ (and also σ_ξ, U_ζ, U_η) from H_ζ, H_η, as follows. Compute σ_ξ, the intersection point of H_ζ and H_η and split H_ζ (resp. H_η) into two subtrees, L_ζ, R_ζ (resp. L_η, R_η) at σ_ξ, where L_ζ, L_η contain the edges that appear to the left of σ_ξ. By definition, we have $U_\zeta = R_\zeta$ and $U_\eta = L_\eta$; H_ξ is the concatenation of L_ζ and R_η. The time needed for either of these steps is $O(\log n)$, as explained in [9]. For a query point q, we can determine in time $O(\log n)$ whether $q \in H_\xi$, provided H_ξ is available as a binary search tree. The space needed to store these structures is linear in n.

Concerning the relative width bits, not every node of $\mathcal{U}$ stores its corresponding bit. This is done for technical reasons that will become clear shortly. Note that if ξ has children ζ, η and the relative width bit is defined for both of them, then, trivially,

$$\omega_\xi^\star = \min\{\omega_\xi^\star, \omega_\eta^\star\}. \tag{3.1}$$

There are several invariants that we will require $\omega_\xi^\star$ to satisfy:

(a) If $\omega_\xi^\star$ is defined, then so are the relative width bits of all ancestors of ξ. Also, if ξ has two children, ζ, η, then either both $\omega_\zeta^\star$, $\omega_\eta^\star$ are defined, or both are undefined.

(b) If $\omega_\xi^\star$ is not defined, then the antipodal vertex (in the lower hull) for all edges stored in $\mathcal{U}_\xi$ is the same (see Fig. 3).

(c) The relative width bit of the root is always defined.

If the relative-width bits of the children of a node ξ are not defined but $\omega_\xi^\star$ is defined, ξ is referred to as a *relative-width leaf.* Our structure maintains an additional bit at each node to mark the relative-width leaves. The width of S is $\leq W$ if the relative width bit of the root of $\mathcal{U}$ or of $\mathcal{L}$ is 0, and it is $>W$ if both of the bits are 1.

3.2. Updating the data structure

Before explaining how $\mathcal{U}$, and its symmetric counterpart $\mathcal{L}$, are updated, we first describe the overall structure of the procedure that inserts a new point p into

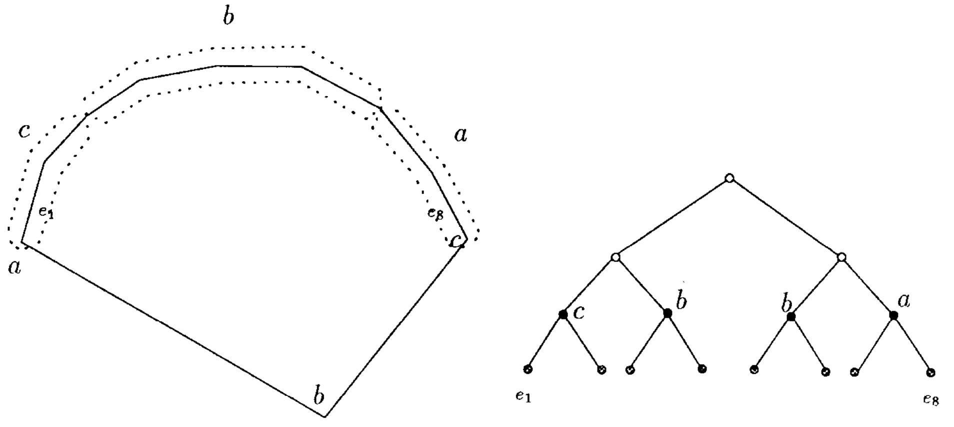

Fig. 3. b is the antipodal vertex for $e_3 \cdots e_6$; black nodes are relative-width leaves.

S. We first add p to the convex hull of S. If p is not a hull vertex, the structure of $\mathcal{U}$ does not change at all. Suppose p becomes a new vertex of the upper hull UCH(S) (the case of the lower hull is treated symmetrically). Let $\theta < \theta'$ be the (rightward-directed) orientations of the two new hull edges, e, e', incident to p. First assume that p is not the rightmost or the leftmost vertex of CH(S). All the previous upper hull edges that had orientations between θ and θ' need to be deleted from $\mathcal{U}$. Consequently, we search with the interval $J = (\theta, \theta')$ in $\mathcal{U}$ to obtain the set of edges $e_i, \ldots, e_k$ that have to be deleted. In addition, we need to insert the edges e, e' into our structure. Finally, for nodes $\xi \in \mathcal{L}$ such that $\mathcal{L}_\xi$ contains an edge whose orientation is in the range $[\theta, \theta']$, $\omega_\xi^{\star}$ needs to be updated, because the antipodal vertex of such an edge has changed (to p). Similarly, for nodes ξ in $\mathcal{U}$ (after the insertions and deletions), whose subtree stores e or e', the bit $\omega_\xi^{\star}$ also has to be updated. (If p becomes the rightmost vertex of CH(S), then e has to be inserted into $\mathcal{U}$ and e' into $\mathcal{L}$, and the edges of $\mathcal{U}$ (resp. $\mathcal{L}$) whose orientations are in the range $(-\pi/2, \theta)$ (resp. θ', $\pi/2$)) have to be deleted. We also have to update the relative-width bits of appropriate nodes in both $\mathcal{U}$ and $\mathcal{L}$. Similar processing is required if p becomes the leftmost vertex.)

Although the edges of LCH(S) whose orientations lie in the range $[\theta, \theta']$ form a contiguous sequence, the leaves whose relative-width bits change due to the insertion of p may alternate (see Fig. 4). As a result, in the worst case, we would have to perform a linear number of updates, if we were to maintain the relative width bit at every node of $\mathcal{U}$. This would obviously be disastrous to the efficiency of the algorithm, which imples that we have to be more selective in maintaining and updating the relative-width bits.

Our approach will be to store relative width bits at only some of the nodes of $\mathcal{U}$, and attempt to store them as high in the tree as possible, while maintaining the invariants (a)–(c). The portion of $\mathcal{U}$ where the relative width bits are undefined is a disjoint union of subtrees, and for each such subtree, $\mathcal{U}_\xi$, there is a

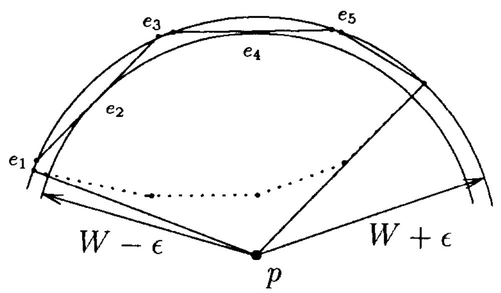

Fig. 4. The relative-width bits are 1 for e_1, e_3, e_5 and 0 for e_2, e_4.

unique vertex q_ξ of $\mathscr{L}$ which is antipodal to all edges in $\mathscr{U}_\xi$; q_ξ can be easily computed in $O(\log n)$ time. This property makes it easy to compute $\omega_\xi^\star$ for such a node ξ 'from scratch', using the structure H_ξ. Specifically, we test whether q_ξ lies in H_ξ. If so, then the relative width bit of ξ is 1, by definition. If not, since q_ξ lies below all the lines supporting the edges of $\mathscr{U}_\xi$, it is easily seen that q_ξ must lie in one of the corresponding strips Δ_e, so $\omega_\xi^\star$ is 0. Hence the important property that we have just noted is that undefined relative width bits can be computed 'on demand' in time $O(\log n)$ per bit, provided H_ξ is available (which indeed will always be the case).

The collection of nodes whose relative width bits are undefined changes during the course of the algorithm. The algorithm makes these changes by occasionally declaring nodes to be relative-width leaves, thereby implicitly making the bits of all their descendants undefined. However the algorithm cannot afford to 'broadcast' this change to each such descendant node. Instead, the operational rule used by the algorithm is that $\omega_\xi^\star$ is undefined if and only if there exists a proper ancestor of ξ that is currently marked as a relative width leaf.

To recap, we need to be able to perform the following three operations on $\mathscr{U}$:

(i) Insert an edge into $\mathscr{U}$.

(ii) Delete a continuous sequence of edges from $\mathscr{U}$.

(iii) Given a sequence of contiguous edges $e_i, \ldots, e_k$ of $\mathscr{U}$, all of which have the same (new) antipodal vertex in the lower hull, update the relative-width data for those nodes $\xi \in \mathscr{U}$ for which the leaves of $\mathscr{U}_\xi$ store some of these edges. (As stated, this is the symmetric operation to the one described above; it is needed when the new vertex lies on the lower hull.)

Inserting an edge. We insert an edge e into $\mathscr{U}$ using any standard insertion algorithm (cf. Tarjan [16]). A typical insertion algorithm first adds a new leaf that stores e, and then rebalances the tree in a bottom-up fashion by following the path, π, from the new leaf to the root. The only additional work that we have to do here is to update $\omega_\xi^\star$, σ_ξ and U_ξ at each node $\xi \in \pi$ (and at siblings of these nodes). This is done in two stages. First, as we walk down along π, we perform the following two steps at each internal node ξ. Let ζ, η be the children of ξ.

(i) Compute H_ζ, H_η from H_ξ, U_η and σ_ξ, as described above.

(ii) If $\omega_\zeta^\star$ and $\omega_\eta^\star$ are undefined, they are computed, as described above, and the nodes ζ, η are marked as relative-width leaves. Moreover if ξ is marked as a relative-width leaf, we unmark it. (As we walk down the path π, it is easy to determine whether $\omega_\zeta^\star$ and $\omega_\eta^\star$ are undefined, using the operational rule mentioned above.)

The first step requires $O(\log n)$ time per node, since H_ξ has already been computed, and the second step also requires $O(\log n)$ time, because when we compute $\omega_\zeta^\star$ and $\omega_\eta^\star$, we already have H_ζ, H_η available.

Next, when we walk up along π, besides rebalancing the tree, we compute the new versions of H_ξ and $\omega_\xi^\star$ at each node $\xi \in \pi$. If ξ is the leaf storing the edge e, $\omega_\xi^\star$ can be easily computed by checking whether l_e intersects the lower convex hull. This can be accomplished in time $O(\log n)$, by searching the tree $\mathcal{L}$ representing LCH(S), using a standard binary search (as in [12]). If the answer is negative, $\omega_\xi^\star$ is set to 0, otherwise it is 1. H_ξ is simply the half-plane lying below l_e. If ξ is an internal node, with children ζ and η, then H_ξ, U_ζ, U_η and σ_ξ are computed, as described above, and $\omega_\xi^\star$ is computed using (3.1) since both $\omega_\zeta^\star$ and $\omega_\eta^\star$ are now defined. It is easily seen that H_ξ can be computed in $O(\log n)$ time and $\omega_\xi^\star$ in constant time. Hence, the total time required to insert an edge into $\mathcal{U}$ is $O(\log^2 n)$.

Our master traversal of the segment tree $\mathcal{T}$ requires us to undo the insert operations when we back up in $\mathcal{T}$ from a node to its parent (see also below); to do this in a space-efficient manner, we store on some stack—(i) the path π, (ii) for every node $\xi \in \pi$ and its child not on π, their colors (red or black), relative width bits, and relative width leaf bits, and (iii) the nodes at which rotations were performed to rebalance the structure, we can reconstruct the original structure as it was before inserting the edge e. We do not have to store H_ξ and σ_ξ, because they can be recomputed, in overall $O(\log^2 n)$ time, by traversing the path π. The total time spent in reconstructing the structure is $O(\log^2 n)$. Since all of the above information can be encoded using $O(\log n)$ bits, we need only $O(1)$ words of storage to store this information.

Deleting a contiguous sequence of edges. Let $(e_i, \ldots, e_k)$ be a contiguous sequence of edges in $\mathcal{U}$. The 'splice' operation that deletes this sequence is performed in three stages. Let π_1 (resp. π_2) be the path from the root to e_i (resp. e_k).

(i) Let $\zeta_1, \zeta_2, \ldots, \zeta_t$, where $t = O(\log n)$, be the left children of nodes on π_1 that do not lie on π_1, ordered from right to left. We merge $\mathcal{U}_{\zeta_1}, \mathcal{U}_{\zeta_2}, \ldots, \mathcal{U}_{\zeta_t}$ by adding these subtrees one at a time in that order. Let h_i be the height of $\mathcal{U}_{\zeta_i}$. It is well known that $\sum_i |h_i - h_{i+1}| = O(\log n)$ and that the height of the tree obtained after merging $\mathcal{U}_{\zeta_i}$ with the preceding subtrees is either h_i or $h_i + 1$. As we will see below, we can merge $\mathcal{U}_{\zeta_i}$ with the preceding subtrees in time $O((|h_i - h_{i+1}| + 1)\log n)$, giving an $O(\log^2 n)$ algorithm for merging all t subtrees.

(ii) Let $\eta_1, \ldots, \eta_s$, where $s = O(\log n)$, be the right children of nodes on π_2 that do not lie on π_2, ordered from left to right. As in (i), we merge $\mathcal{U}_{\eta_1}, \ldots, \mathcal{U}_{\eta_s}$ in $O(\log^2 n)$ time.

(iii) Finally, merge the trees computed in the previous two steps.

It thus suffices to show how to merge two subtrees $\mathcal{U}_1$ and $\mathcal{U}_2$ of $\mathcal{U}$, having heights h_1 and h_2, respectively, in time $O((|h_1 - h_2| + 1)\log n)$. If $h_1 = h_2$, we create a new node δ, add $\mathcal{U}_1$, $\mathcal{U}_2$ as its children, and compute H_δ, $\omega_\delta^{\star}$ and the various related substructures. Now assume that $h_1 > h_2$, and that all edges in $\mathcal{U}_1$ have orientations less than those of $\mathcal{U}_2$. Let θ denote the orientation of the leftmost leaf of $\mathcal{U}_2$. We follow the (rightmost) path π in $\mathcal{U}_1$, as if we are searching for an edge with orientation θ, until we reach a node ζ whose height is $h_2 + 1$. Let $\mathcal{U}_\zeta^r$ denote the right subtree of ζ. We create a new node η, insert it as the right child of ζ, make $\mathcal{U}_2$, $\mathcal{U}_\zeta^r$ the left and right subtrees of η, respectively, and walk up to rebalance the tree. While following the path π down and up, the H_ξ structures and relative-width bits are updated as described in the *insert* operation. Since we spend $O(\log n)$ time at each node and visit only $O(h_1 - h_2 + 1)$ nodes, the total time spent is $O((h_1 - h_2 + 1)\log n)$.

Again, if we want to recover the original structure, we need to store the same information for the nodes that we visited in Step (i)–(iii), as in the previous case. We also need to store the subtrees that we deleted. Since we visit $O(\log n)$ nodes and they are accessed by following pointers from the root, we can encode the former information using $O(1)$ space. As for storing the subtrees, if we removed $m \leq k$ subtrees from the structure, we need $O(m)$ storage to store pointers to the roots of these subtrees. The time spent in reconstructing the structure is again $O(\log^2 n)$.

Remark. The tree merging steps in the above procedure are standard operations on red-black trees. We have described them in some detail only to show that the auxiliary operations that we have to perform, namely updating the H_ξ and $\omega_\xi^{\star}$ data, fit nicely into the tree operations.

Updating the relative-width data. We now explain how to perform the third operation, i.e., given a contiguous sequence $e_i, \ldots, e_k$ of upper hull edges, all having q as their antipodal vertex, update the relative width data for nodes $\xi \in \mathcal{U}$ for which $\mathcal{U}_\xi$ contains an edge from the above sequence. Note that this sequence can be represented as a disjoint union of $t = O(\log n)$ subtrees, rooted at nodes $\xi_1, \ldots, \xi_t$ ordered from left to right (see Fig. 5). Let δ_1 (resp. δ_t) be the parent of ξ_1 (resp. ξ_t), and let π_1 (resp. π_2) be the path from the root to δ_1 (resp. δ_t). The parents of all nodes $\xi_1, \ldots, \xi_t$ lie on π_1 or on π_2. We first walk along π_1 from the root to δ_1 and propagate down the data H_ξ and $\omega_\xi^{\star}$, as described in the *insert* operation. We do the same for π_2. These steps automatically compute relative-width bits for all ξ_i. Finally, $\xi_1, \ldots, \xi_t$ are marked as relative-width

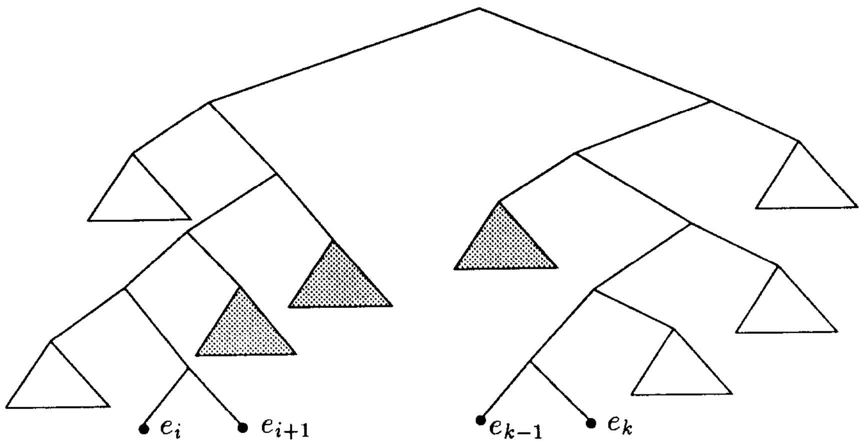

Fig. 5. Shaded subtrees store $e_i, \ldots, e_k$.

leaves, so the relative-width bits for all other nodes in $\mathcal{U}_{\xi_j}$ are implicitly marked undefined. Observe that invariant (b) is preserved by this operation.

To maintain the invariant (a), to satisfy (3.1), and to restore the H_ξ structures, we back up along π_1, π_2 and perform the same steps as in the *insert* operation, that is, for all nodes δ on π_1 update $\omega_\delta^\star$ and compute σ_δ, U_ζ and U_η, where ζ, η are the children of δ. The total time spent is again $O(\log^2 n)$. Again, we can encode the relative width bits of nodes on π_1 and π_2 in $O(1)$ space, so that we can recover the original structure back in $O(\log n)$ time.

Hence, we conclude that inserting a point p into the convex hull requires $O(\log^2 n)$ time. We also observe that our structures use only $O(n)$ storage. Finally, let k be the number of points that were hull vertices before inserting p but not after inserting p, then the above discussion implies that we can store in $O(k+1)$ space the information that we need to recover the original data structure as it was before inserting p. The time spent in recovering the structure is only $O(\log^2 n)$.

To summarize, we have shown how to insert a new point into S. The step that goes from a node u of the segment tree $\mathcal{T}$ to its child v simply consists of adding all the points in P_v one by one into our structures. After adding all the points of P_v, we check whether $\omega(S_v) > W$, that is, whether the relative width bits of the roots of both trees, $\mathcal{U}$ and $\mathcal{L}$, are 1. If $\omega(S_v) > W$, we discard the subtrees rooted at v and back up to its parent u. If $\omega(S_v) \leqslant W$ and v is a leaf, we stop with answer 'yes'. Otherwise, we visit its left and right children recursively, and then back up to its parent u. If we complete the traversal of $\mathcal{T}$ without stopping at any leaf, we can conclude that $\omega(S)$ always remains $>W$. We remember all the changes made in the structure while processing v, using the encoding scheme described above, so that when we later back up from v to u, we can undo all of them, to restore the data structure to its state at u.

The total running time of the algorithm is $O(\log^2 n)$ times the total size of all the sets P_v. As for the space complexity, the data structure requires $O(n)$ space, so it suffices to bound the total space required to store the history of the structure. Note that when we are at a node v, we store only those changes that we made at nodes z on the path π_v. The additional space required to store this information is thus $\sum_{z \in \pi_v} \sum_{p \in P_z} O(k_p + 1)$, where k_p is the number of points that were hull vertices before inserting p but not after inserting p. It is easily seen that, along any path of $\mathcal{T}$, once a point ceases to be a hull vertex, it cannot reappear as a hull vertex, therefore $\sum_{z \in \pi_v} \sum_{p \in P_z} k_p = O(n)$, which gives $O(n)$ bound on the total storage required by the algorithm. Since the total size of all the sets P_v is $O(n \log n)$, we obtain the following theorem.

Theorem 3.1. *Given a real parameter $W > 0$ and a sequence $\Sigma = (\sigma_1, \ldots, \sigma_n)$ of n insert/delete operations on a set S of points in $\mathbb{R}^2$, initially consisting of n points, we can determine, in time $O(n \log^3 n)$, whether there is any i such that $\omega(S) \leq W$ after the operation σ_i. The space required by the algorithm is $O(n)$.*

4. Conclusion

In this paper we have presented an $O(n \log^3 n)$ algorithm for solving the off-line dynamic width maintenance problem, in the restricted sense considered above. The algorithm is not complicated, and uses fairly standard data structures.

Our results raise the following observations and open problems:

(1) Can our technique be modified to run on-line? In particular, can it be modified to allow us also to delete points from S, without having the segment tree structure at hand? On the face of it, this appears to be more difficult, because removing a hull vertex may expose many new edges whose relative width data have to be computed.

(2) We can strengthen our results so that they also apply to the semi on-line model, as defined in the Introduction. We simply have to construct the segment tree $\mathcal{T}$ on the fly in inorder, but otherwise apply essentially the same algorithm described above (see [2] for details).

(3) Can our technique be extended so as to compute the minimum width of S during the sequence of updates, or to report the new width of S after each update? One obvious approach would be to replace the relative width bits by the *relative width* itself, namely the smallest distance between two parallel supporting lines, one of which supports an edge stored in the corresponding subtree. We can then continue to apply (3.1) as above, and in fact almost all the steps of the algorithm generalize nicely, with the exception of the manipulation of the structures H_ξ. We need to replace H_ξ by a structure that supports queries of the form: Given a point p and an angular range I, find the smallest distance from p to

one of the lines supporting an edge whose orientation is in I. Since p lies on the same side of all these lines, it is confined to the upper or lower face in their arrangement. Still this sounds like we need a dynamic version of a nearest-neighbor data structure for the edges of the upper (or lower) face, and we are not aware of any efficient structure of this sort.

(4) One possible way to approach the problem of finding the minimum width of S during the sequence of updates, is to apply Megiddo's parametric search technique [10]. This should be similar to the method given in [1], but the details remain to be worked out.

(5) Finally we mention the problem of maintaining the width of a set S of n moving points, say where each point moves along a straight line at constant speed. It would be interesting to see whether our technique can be extended to handle this 'truly dynamic' case.

Acknowledgments

We wish to thank the referees for their helpful comments.

References

[1] P.K. Agarwal and M. Sharir, Planar geometric location problems, Tech. Report 90-58, DIMACS, Rutgers University, August 1990.
[2] D. Dobkin and S. Suri, Dynamically computing the maxima of decomposable functions with applications. Proceedings 30th Annual IEEE Symposium on Foundations of Computer Science (1989) 488–493.
[3] H. Edelsbrunner and M. Overmars, Batched dynamic solutions to decomposable searching problems, J. Algorithms 6 (1985) 515–542.
[4] G. Frederickson and S. Rodger, A new approach to the dynamic maintenance of maximal points in a plane, Discrete Comput. Geom. 5 (1990) 365–374.
[5] H. Gabow and R. Tarjan, A linear time algorithm for a special case of disjoint set union, J. Comput. System Sci. 30 (1985) 117–122.
[6] J. Hershberger and S. Suri, Finding tailored partitions, Proc. 5th Annual ACM Symp. on Computational Geometry (1989) 255–265.
[7] J. Hershberger and S. Suri, Off-line maintenance of planar configurations, Proceedings 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 32–41.
[8] M. Houle and G. Toussaint, Computing the width of a set, Proceedings 1st Annual Symposium on Computational Geometry (1985) 1–7.
[9] M. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, J. Comput. System Sci. 23 (1981) 166–204.
[10] N. Megiddo, Applying parallel computation algorithms in the design of serial algorithms, J. ACM 30 (1983) 852–865.
[11] K. Melhorn, Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry (Springer, Berlin, 1984).
[12] F. Preparata and M. Shamos, Computational Geometry: An Introduction (Springer, Berlin, 1985).
[13] M. Smid, A worst-case algorithm for semi-online updates on decomposable problems, Tech. Rept. A 03/90, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1990.

[14] M. Smid, Maintaining the minimal distance of a point set in polylogarithmic time, Proceedings 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (1991) 1–6.

[15] K. Supowit, New techniques for some dynamic closest-point and farthest-point problems, Proceedings 1st Annual ACM-SIAM Symposium on Discrete Algorithms (1990) 84–90.

[16] R. Tarjan, Data Structures and Network Algorithms (SIAM Publications, Philadelphia, PA, 1983).