MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Semi-dynamic Maintenance of the Width of a Planar Point Set

Christian Schwarz

MPI–I–92–153 December 1992

Im Stadtwald
66123 Saarbrücken
Germany

Semi-dynamic Maintenance of the Width of a Planar Point Set

Christian Schwarz

MPI–I–92–153 December 1992

Semi-dynamic Maintenance of the Width of a Planar Point Set*

Christian Schwarz

Max-Planck-Institut für Informatik
W-6600 Saarbrücken, Germany
schwarz@mpi-sb.mpg.de

December 6, 1992

Abstract

We give an algorithm that maintains an approximation of the width of a set of n points in the plane in $O(\alpha \log n)$ amortized time, if only insertions or only deletions are performed. The data structure allows for reporting the approximation in $O(\alpha \log n \log\log n)$ time. α is a parameter that expresses the quality of the approximation. Our data structure is based on a method of Janardan that maintains an approximation of the width under insertions and deletions using $O(\alpha \log^2 n)$ time for the width query and the updates.

1 Introduction

Let S be a set of n points in the plane. The width of S, denoted by $W(S)$, is defined as the smallest distance of two parallel lines enclosing S and has applications in collision-avoidance problems and in the approximation of polygonal curves [5]. The width of S can also be characterized using the convex hull of S: Let $CH(S)$ denote the convex hull of S, and let e and v be an edge and a vertex of $CH(S)$, respectively. Furthermore, let ℓ_e be the supporting line of e and ℓ_v the line through v that is parallel to ℓ_e. The pair (e, v) is called an *antipodal pair* of $CH(S)$ if ℓ_e and ℓ_v support $CH(S)$. (The supporting line of a line segment e is the line containing e, and the supporting line of a convex polygon P is a tangent of P.) Using this terminology, $W(S)$ is the minimum distance between lines ℓ_e and ℓ_v, taken over all antipodal pairs (e, v) of $CH(S)$, see [5]. In [5], it is shown how to compute $W(S)$ for a given set S in $O(n \log n)$ time and $O(n)$ space: first, compute the convex hull $CH(S)$, and then generate the — only $O(n)$ — antipodal (edge-vertex) pairs in linear time, using a procedure similar to the one given in [11, pp. 179-181] for antipodal point-point pairs. Thus, if the convex hull is at hand, $W(S)$ can be computed in $O(n)$ time by generating and checking all antipodal edge-vertex pairs on the hull.

The dynamic width problem is to maintain $W(S)$ while the set S is modified by insertions and deletions of points. In this paper, we consider the *semi-dynamic* width problem. We want to maintain the width of S when the update sequence consists either of insertions only or of deletions only. We require the updates to be performed *on-line*, i.e. each update operation

*This work was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

must be completed before the next update request is specified. The goal is to find algorithms that beat the trivial $O(n)$ bound obtained by updating the convex hull using any efficient dynamic or semi-dynamic convex hull algorithm and then recomputing the width in linear time by generating and checking all antipodal pairs as described above.

We now summarize previous work on dynamic width maintainance. When looking for a dynamic solution, the property of a problem being decomposable [8] has proved to be very useful. For decomposable problems, efficient dynamization techniques have been developed ([8, 3]). The width problem is not decomposable. The above definition of width using antipodal pairs can be formulated as follows (see [1]):

$$W(S) = \min_{e \in CH(S)} \max_{v \in CH(S)} dist(\ell_e, \ell_v),$$

where ℓ_e and ℓ_v are the parallel lines containing e and v, respectively, as described above. That is, $W(S)$ is a "min-max"-operator, which is not decomposable, in contrast to related problems such as diameter ("max-max") or closest pair ("min-min"). Therefore, the techniques of [8, 3] do not work here.

A restricted off-line variant of dynamic maintenance was discussed by Agarwal and Sharir [1]: Given a set S, a positive real number $\mathcal{W}$ and a sequence of n updates on S, does the case "$W(S) \leq \mathcal{W}$" occur during the execution of the sequence? In [1], an $O(n \log^3 n)$ algorithm is given for this problem.

The first on-line algorithm was given by Janardan [6]. He considered the dynamic maintenance of an *approximation* of $W(S)$ and gave insertion and deletion algorithms that run in $O(\alpha \log^2 n)$ time. The data structure uses $O(\alpha n)$ space. The approximation can be made arbitrarily close to optimal, and its quality depends on the parameter α. Reasonable approximations are already available for constant values of α, see Theorem 1 below. As noted by Janardan [6], approximate solutions are quite appropriate for problems where already the input data are not exact, but only known up to a certain precision, like it is the case for example in statistical applications. We cite Janardan's result:

Theorem 1 (Janardan [6]) *Let S be a set of n points in the plane. Let $W(S)$ be the width of S and let $\alpha \geq 1$ be an integer-valued parameter. There is a data structure that uses $O(\alpha(n))$ space and supports the following operations in $O(\alpha \log^2 n)$ time:* insert a point into S, delete a point from S *and* report $\widetilde{W}(S)$, *where* $\widetilde{W}(S)/W(S) \leq \sqrt{1 + \tan^2 \frac{\pi}{4\alpha}}$. *For instance,* $\widetilde{W}(S)/W(S) \leq 1.01$ *for* $\alpha = 5$, *and* $\widetilde{W}(S)/W(S) \leq 1.0001$ *for* $\alpha = 16$. *All bounds are worst case.*

In this paper, we give a data structure that maintains the approximation $\widetilde{W}(S)$ of Theorem 1 in $O(\alpha \log n)$ amortized time if only insertions or only deletions are performed. Reporting $\widetilde{W}(S)$ takes time $O(\alpha \log n \log \log n)$. The paper is organized as follows. Since our data structure is based on the one given by Janardan, we will start by reviewing his solution in Section 2. Section 3 contains the description of the improved data structure for the semi-dynamic case.

2 Dynamic maintenance of the width approximation

As already mentioned in the introduction, we can dynamically maintain the width of S as follows: Maintain the convex hull $CH(S)$ by using the algorithm of [9], and then search for the

antipodal edge-vertex pair that yields the minimal distance between two parallel lines enclosing S. Unfortunately, it is not known how to perform this task efficiently, i.e. without essentially checking all antipodal pairs.

There is yet another algorithm given in [5] to compute the width. Instead of finding antipodal edge-vertex pairs directly, it finds the corresponding parallel lines of support. The algorithm is implemented using the duality transform, to be described in the next subsection. Having found the antipodal pairs in this way, the algorithm finds the one with the minimum distance in linear time as before. As already mentioned, we want to avoid checking all antipodal pairs after an update.

Janardan observed that, as opposed to the distance of parallel lines of support in primal space, the interpretation of this distance in dual space has a nice monotonicity property along the convex hull of the set, and this makes efficient searching for the minimum possible. But by transforming the problem into dual space, the distance between parallel lines can be distorted considerably, so the result computed in dual space may be meaningless. The amount of distortion depends on the slope of the lines. So, by maintaining a family of coordinate systems instead of only one, there will always be one system where the error is kept small.

We now discuss the duality transform and its application to the width problem ([5, 6]) in detail.

2.1 Interpretation of antipodal pairs in dual space

Let us consider the following duality transform that converts points into lines and non-vertical lines into points:

$$\begin{aligned} p = (a, b) &\longmapsto \mathcal{F}(p) : y = ax + b \\ \ell : y = mx + c &\longmapsto \mathcal{F}(\ell) = (-m, c) \end{aligned}$$

An important property of duality is that it is *incidence-preserving*: p is above (resp. on) ℓ if and only if $\mathcal{F}(p)$ is above (resp. on) $\mathcal{F}(\ell)$. Also, parallel lines are mapped to points with the same x-coordinate.

We now use this transform to map the points of S into dual space. We obtain a set of lines $\mathcal{L}$. We orient each non-horizontal line of $\mathcal{L}$ upwards and each horizontal line rightwards. By this orientation, we obtain a set of halfplanes $\mathcal{H}_{left}(\mathcal{L})$ and $\mathcal{H}_{right}(\mathcal{L})$, defined to be the halfplanes that lie to the left and to the right of the oriented lines of $\mathcal{L}$, respectively. Now, the following geometric objects are of interest to us:

Consider $L := \partial \bigcap_{\ell \in \mathcal{L}} \mathcal{H}_{left}(\mathcal{L})$, the boundary of the intersection of the halfplanes in $\mathcal{H}_{left}(\mathcal{L})$. It is a convex chain $(e_0, v_1, e_1, \ldots, v_L, e_L)$, with vertices $v_1, \ldots, v_L$ and edges $e_0, \ldots, e_L$, where $e_1, \ldots, e_{L-1}$ are line segments and e_0, e_L are half-infinite rays. Similarly, we define $R := \partial \bigcap_{\ell \in \mathcal{L}} \mathcal{H}_{right}(\mathcal{L})$.

These convex chains correspond to the convex hull $CH(S)$ of S, as follows: Let U (resp. D) be the upper (resp. lower) convex hull of S, i.e. the upward (resp. downward) convex chain connecting the leftmost and righmost vertex of S.

Fact 1 For each vertex v of U, $\mathcal{F}(v)$ is a supporting line of an edge of L. In particular, the supporting line of an edge (u, v) of U maps to the vertex $\mathcal{F}(u) \cap \mathcal{F}(v)$ of L.

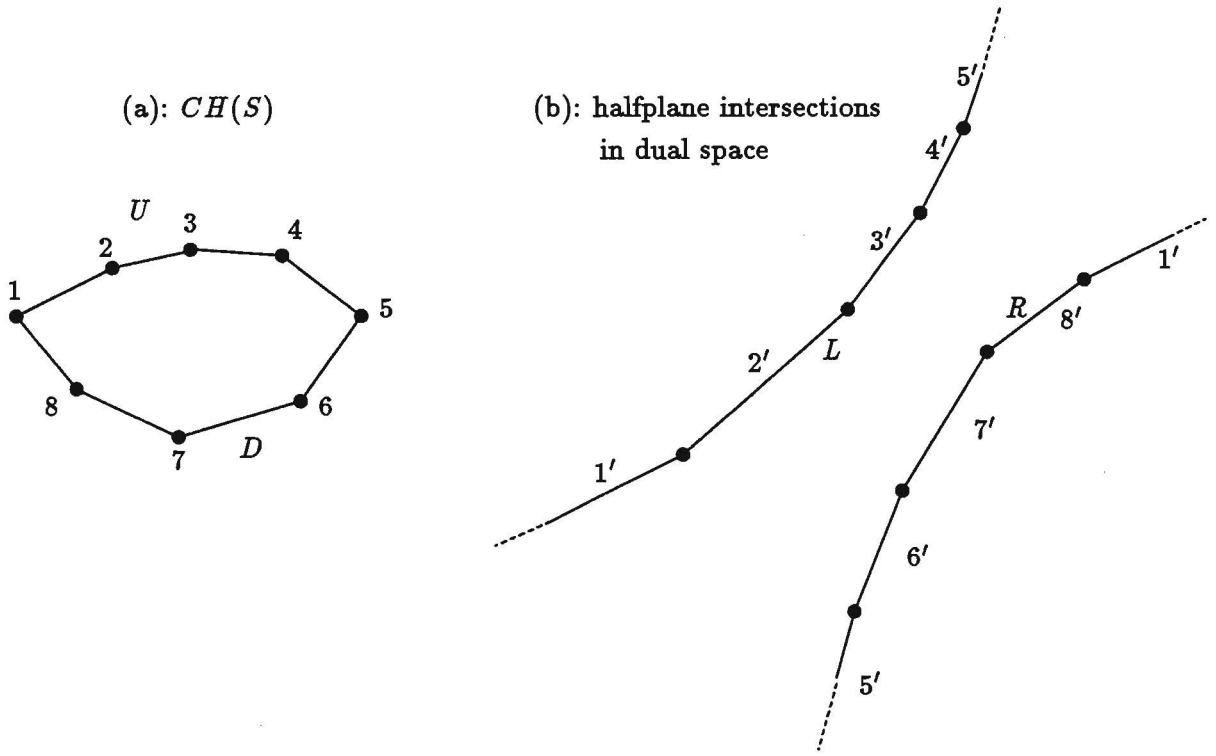

Figure 1: The duality transform $\mathcal{F}$. (a): vertices 1-5 of $CH(S)$ belong to U, and vertices 5-8 belong to D. (b): The dual chains L and R of U resp. D. For each vertex i of $CH(S)$, i' denotes the edge(s) supported by line $\mathcal{F}(i)$.

An analogous statement holds for $v \in D$ and $\mathcal{F}(v) \in R$. Figure 1 shows an example. The dual of vertex 2 is the supporting line of the edge $2'$; the dual of the supporting line of $(2,3)$ is the common endpoint of the edges labeled $2'$ and $3'$.

Now we note further correspondences between the upper and lower part of $CH(S)$ and the convex chains L and R in dual space. Let v be a vertex of $CH(S)$, w.l.o.g. $v \in D$. Let $e_1 = (w,v), e_2 = (v,z)$ be the edges incident on v and let ℓ_1, ℓ_2 be their supporting lines. Consider the supporting lines ℓ of $CH(S)$ that are obtained by rotating around v, starting with $\ell = \ell_1$ and ending with $\ell = \ell_2$. Let $\mathcal{L}(v, \ell_1, \ell_2)$ be the set of these lines. Since the duality transform is incidence-preserving, every line through v must map to a point on $\mathcal{F}(v)$.

If v is a non-extreme vertex of D, then the neighboring vertices w and z are also on D. By Fact 1, $\mathcal{F}(w)$ and $\mathcal{F}(z)$ are supporting lines of edges of R, as is $\mathcal{F}(v)$. Furthermore, ℓ_1, the supporting line of $e_1 = (w,v)$, maps to point $\mathcal{F}(w) \cap \mathcal{F}(v)$, and ℓ_2 maps to $\mathcal{F}(v) \cap \mathcal{F}(z)$. Therefore,

Fact 2 *If v is a non-extreme vertex of D, the set of lines $\mathcal{L}(v, \ell_1, \ell_2)$ maps to the line segment $\mathcal{F}(v) \cap R$, with endpoints $\mathcal{F}(\ell_1)$ and $\mathcal{F}(\ell_2)$.*

(In Figure 1, if $v = 7$, then $\mathcal{F}(7) \cap R$ is the segment labeled $7'$.)

If v is an extreme vertex of D, then v is also on U. W.l.o.g. let be $w \in D$, i.e. $e_1 \in D$. Then $z \in U$ and therefore $e_2 \in U$. Now $\mathcal{F}(v)$ is a supporting line of an edge of R and also of an edge of L. $\mathcal{F}(w)$ is a supporting line of an edge of R and $\mathcal{F}(z)$ is a supporting line of an edge of L.

$\mathcal{F}(\ell_1)$ is a vertex of R, namely $\mathcal{F}(v) \cap \mathcal{F}(w)$, and $\mathcal{F}(\ell_2)$ is a vertex of L, namely $\mathcal{F}(v) \cap \mathcal{F}(z)$. We have

Fact 3 *If v is an extreme vertex of D, then the set $\mathcal{L}(v, \ell_1, \ell_2)$ maps to the rays $\mathcal{F}(v) \cap R$ and $\mathcal{F}(v) \cap L$, i.e. the part of $\mathcal{F}(v)$ that is obtained by excluding the line segment from $\mathcal{F}(\ell_1)$ to $\mathcal{F}(\ell_2)$.*

As an example, consider $v = 5$ in Figure 1. The rays $\mathcal{F}(5) \cap L$ and $\mathcal{F}(5) \cap R$ are the ones labeled $5'$.

Let (e, v) be an antipodal pair of $CH(S)$, w.l.o.g. let $e \in U$. Then $v \in D$, because U is convex. Let ℓ_e and ℓ_v be the parallel lines through e and v, respectively, that support $CH(S)$. From the above facts, it follows that

1. $\mathcal{F}(\ell_v)$ is a point on the segment $\mathcal{F}(v) \cap R$,
2. $\mathcal{F}(\ell_e)$ is a vertex of L,
3. $\mathcal{F}(\ell_v)$ and $\mathcal{F}(\ell_e)$ have the same x-coordinate.

On the other hand, let v' be a vertex of L and let p' be the point of R with the same x-coordinate as v'. Define $e := U \cap \mathcal{F}^{-1}(v')$ and $v := D \cap \mathcal{F}^{-1}(p')$.

Claim: (e, v) is an antipodal pair of S.

Proof: By the definition of v, $\mathcal{F}(v)$ contains p', i.e. p' is on a ray or segment $\mathcal{F}(v) \cap R$. Following the notation used to establish Facts 2 and 3, let w and z be the vertices that are adjacent to v on $CH(S)$. We want to prove that $\ell_v := \mathcal{F}^{-1}(p')$ is a tangent of $CH(S)$. To do this, it suffices to prove that it belongs to the set $\mathcal{L}(v, \ell_1, \ell_2)$, where ℓ_1, ℓ_2 are the supporting lines of $e_1 = (w, v)$ and $e_2 = (v, z)$, respectively. Since the duality transform $\mathcal{F}$ is injective, Facts 2 and 3 imply that every point on a segment or ray $\mathcal{F}(v) \cap R$ must belong to $\mathcal{L}(v, \ell_1, \ell_2)$.∎

We therefore have

Fact 4 *Let $e \in U, v \in D$, and let ℓ_e and ℓ_v be the parallel lines through e and v, respectively. Then (e, v) is an antipodal pair of $CH(S)$ if and only if $\mathcal{F}(\ell_v)$ is the point where the vertical line through vertex $\mathcal{F}(\ell_e)$ in L intersects the chain R.*

In Figure 1, if $e = (3, 4)$, then $v = 7$ and $\mathcal{F}(\ell_7)$ is the point on R that lies vertically below the common endpoint of the edges $3'$ and $4'$. If $e \in D$ and $v \in U$, then claims analogous to the ones in Facts 2-4 hold.

Fact 4 establishes a one-to-one correspondence between the antipodal edge-vertex pair (e, v) in primal space and the vertex-point pair $(\mathcal{F}(\ell_e), \mathcal{F}(\ell_v))$ in dual space. Accordingly, in dual space, the distance between the vertex $\mathcal{F}(\ell_e)$ and the point $\mathcal{F}(\ell_v)$ will play the role of the distance of the parallel lines of support belonging to (e, v). Thus, the search for the minimum, over the vertices of both chains L and R, of the vertical distance of a vertex to the other chain in dual space corresponds to the search for the antipodal pair (e, v) with minimal distance between ℓ_e and ℓ_v in primal space.

2.2 The basic search and update algorithms

Let p be a vertex on L. The distance of p to R, denoted by $\rho(p, R)$, is defined to be the length of the vertical segment starting at p and ending at a point of R. The distance of a point $p \in R$ to L is defined analogously. The distance function ρ has the following monotonicity property which effects an efficient searching algorithm.

Lemma 1 (Janardan [6]) *Let $p_0 \in L$ such that $\rho(p_0, R) = \min_{p \in L} \rho(p, R)$. Index the vertices on L with subsequent integers such that the vertices to the right of p_0 get positive indices and the vertices to the left of p_0 get negative indices, i.e. $L = \ldots, p_{-2}, p_{-1}, p_0, p_1, p_2, \ldots$. Then there are indices $k^- \leq 0$ and $k^+ \geq 0$ such that $\rho(p_i, R) = \rho(p_0, R)$ for $k^- \leq i \leq k^+$, $\rho(p_i, R) > \rho(p_{i-1}, R)$ for $i > k^+$ and $\rho(p_i, R) > \rho(p_{i+1}, R)$ for $i < k^-$. An anologous statement holds for the points on R.*

Using Lemma 1, p_0 can be found by binary search, as shown in Figure 2.

```
(1)   p_mid ← middle vertex of L;
(2)   p_l ← left neighbor of p_mid in L; p_r ← right neighbor of p_mid in L;
(3)   find the intersections of R with the vertical lines through p_mid, p_l and p_r;
(4)   compute ρ(p_mid, R), ρ(p_l, R) and ρ(p_r, R) ;
(5)   if ρ(p_mid, R) ≤ ρ(p_l, R) and ρ(p_mid, R) ≤ ρ(p_r, R) then
(6)       p_0 ← p_mid; stop
(7)   else
(8)       if ρ(p_mid, R) > ρ(p_l, R) then
(9)           p_l ← new rightmost vertex of L to be considered
(10)      else
(11)          p_r ← new leftmost vertex of L to be considered
(12)      fi
(13)  fi;
(14)  goto (1);
```

Figure 2: The searching algorithm to find the vertex $p_0 \in L$ which minimizes the vertical distance to chain R.

How about the implementation of this algorithm? In [9], it is shown that the intersection of n halfplanes can be maintained in $O(n)$ space and $O(\log^2 n)$ update time such that the halfplanes contributing to the boundary — these are the duals of the points of the upper/lower hull — are stored at the leaves of a balanced binary search tree, in sorted order of their slopes. An internal node v of the tree corresponds to the part of the convex chain represented by the leaves of v's subtree. Also, if each internal node v is augmented with pointers to leftmost and rightmost leaf in its subtree, this does not affect the bounds on space and update time of the structure.

The binary search is thus implemented by walking down a root-to-leaf path in the tree. When we are at node v, then the vertex formed by the intersection of the boundary of the halfplane stored at the rightmost leaf of v's left subtree and the boundary of the halfplane stored at the leftmost leaf of v's right subtree is chosen to be the "middle" vertex p_{mid} in

line (1). Using the pointers described before, line (1) can be executed in $O(1)$ time. Knowing p_{mid}, its neighbors p_l and p_r are computed in constant time in line (2). Discarding parts of the convex chain L in lines (10) and (12) is implicitly performed in $O(1)$ time by continuing the search in the left or right child of the current node v.

Since $|L|+|R| \leq n+2$ if n is the current size of the point set, the algorithm makes $O(\log n)$ iterations. Each line of the algorithm takes constant time, except line (3). The intersections of R and the vertical lines through p_{mid}, p_l and p_r are also computed by binary search, in $O(\log n)$ time. It follows that the total time to find the point $p_0 \in L$ with minimal distance to chain R is $O(\log^2 n)$.

Now consider the following algorithm to report the width of S: Maintain L and R in separate instances of the above structure and use the search algorithm of Figure 2 to search L (and similarly R) for the vertex that minimizes the distance to the other chain. Out of the two resulting vertices, pick the one, say v, with smaller vertical distance to point p on the other chain, and report the distance of the primal parallel lines corresponding to v and p. The running time of this algorithm is $O(\log^2 n)$. However, it does not compute the width $W(S)$, because the duality transform $\mathcal{F}$ does not preserve the distance between parallel lines. More specifically,

Lemma 2 (Janardan [6]) *Let g and h be parallel lines in the plane, with slope m. The distance between the points $\mathcal{F}(g)$ and $\mathcal{F}(h)$ is $\sqrt{1+m^2}$ times the distance between g and h.*

2.3 The approximation algorithm

From Lemma 2, we infer that the quality of the result decreases with rising absolute value of the slope of the optimal supporting lines.

This observation is used as follows. Let α be a positive integer. Instead of one coordinate system, a family $C_i = (X_i, Y_i), 0 \leq i \leq \alpha$, with positive x- and y-axes X_i and Y_i, respectively, is maintained. X_0 is horizontal and the angle between X_i and X_{i-1} is $\frac{\pi}{2\alpha}$ for $1 \leq i \leq \alpha$.

Then, it is shown in [6] that for any line ℓ in the plane, there is an $i \in \{0, \ldots, \alpha\}$, such that the absolute value of ℓ's slope in the system C_i is at most $\tan \frac{\pi}{4\alpha}$. Therefore, there is a coordinate system in which the slope of the optimal pair of supporting lines is small.

The update and query algorithms are modified as follows. For each coordinate system C_i, keep the data structure described before, i.e. two instances L_i, R_i of the halfplane intersection structure of [9], for the intersection of the left and the intersection of the right halfplanes, respectively. To insert or delete a point p, compute p_i, the coordinate of p in C_i, for $0 \leq i \leq \alpha$, and perform the update with p_i in L_i and R_i for all i, as described before. The running time for the update operations is $O(\alpha \log^2 n)$, where n is the current size of the point set.

To answer a width query, we use the search algorithm given before, for each i, and find $2(\alpha+1)$ point pairs. Out of these pairs, we pick the one for which the distance of the corresponding pair of parallel lines in primal space is mimimal and report this distance. The running time of this algorithm is $O(\alpha \log^2 n)$.

Let $\widetilde{W}(S)$ be the distance reported in the query algorithm. To obtain a bound on the reported distance, consider the parallel lines that give rise to this distance. Using the fact that for any line ℓ, there is a system C_i where the slope of ℓ is at most $\tan \frac{\pi}{4\alpha}$, and that for parallel lines g and h with slope m, the distance between the points $\mathcal{F}(g)$ and $\mathcal{F}(h)$ is $\sqrt{1+m^2}$ times

the distance between g and h (Lemma 2), it is shown in [6] that $\widetilde{W}(S) \leq W(S) \cdot \sqrt{1 + \tan^2 \frac{\pi}{4\alpha}}$. This completes the proof of Theorem 1.

3 Semi-dynamic maintenance of the width approximation

In this section, we consider the case where the point set is modified by one type of update operation (insert or delete) only. We will show how to modify the method of Section 2 to achieve the following result.

Theorem 2 *Let S be a set of n points in the plane, and let α be a positive integer. There is a data structure that supports the operation* report $\widetilde{W}(S)$ *in $O(\alpha \log n \log\log n)$ worst case time, where $\widetilde{W}(S)$ satisfies the property described in Theorem 1. Furthermore, the data structure supports the operation* insert p into S *in $O(\alpha \log n)$ amortized time. There is a related structure with the same bound for reporting $\widetilde{W}(S)$, namely $O(\alpha \log n \log\log n)$ that, instead of insertions, supports deletions of points in $O(\alpha \log n)$ amortized time. Both data structures use $O(\alpha n)$ space.*

From now on, we concentrate on improving the data structure for one fixed coordinate system given in Subsection 2.2. Theorem 2 then follows by applying the method of maintaining a family of coordinate systems described in Subsection 2.3.

3.1 Potential improvements for the semi-dynamic case

Let us examine the running time of the operations. The cost of an update operation is $O(\log^2 n)$, determined by the time to update the structure representing the chains L and R, which are instances of the halfplane intersection structure of [9]. The dynamic halfplane intersection problem is dual to the dynamic convex hull problem, and the data structures for these problems are basically the same. So, a faster convex hull algorithm would decrease the update time. Indeed, in the semi-dynamic case, there are algorithms that update the convex hull in $O(\log n)$ amortized time: see [4] for deletions and [10] for insertions. The time bound for insertions in [10] is worst case.

The running time for a query in the structure of Subsection 2.2 is $O(\log^2 n)$, because it is a twofold binary search: a vertex v_0, say on L, is searched such that the vertical distance of v_0 to the other chain is minimal. For each vertex v on L that is tested, a binary search on R is performed to find the segment that intersects the vertical line through v. This is the inner binary search. We want to speed-up the query by replacing the inner binary search by a faster method.

The segment on R that intersects the vertical line through v is found by identifying one of its incident vertices. We call these vertices the left resp. right neighbor of v in R. Note that one of them may not exist. Assume e.g. that the right neighbor does not exist. In this case, the segment opposite v is the ray at the right end of the chain. We can easily handle this special case. If the right neighbor, say p', exists, then the segment on R that is opposite v is adjacent to p' and is therefore accessible once p' is found.

Note that the vertices are ordered by x-coordinate on both chains L and R. So, in an abstract sense, our problems is as follows: We have two ordered lists of real numbers, and we are given a number in one list. Our goal is to *localize* this number in the other list, where

localizing means to find the position (i.e. right neighbor) that the number would have in the other list.

The problem that we want to solve turns out to be an instance of a well-known data-structuring problem called *union-split-find problem*, see e.g. [7, 2]. We will address this problem in Subsection 3.2, and then, in Subsection 3.3, we will apply it to our situation.

The data structure in Subsection 2.2 is organized such that the two chains L and R are kept separately. To perform the above described search for a neighbor of a vertex in the other list, we maintain an additional structure, the *mixed L, R-structure*, to be described in Subsection 3.3, where the vertices of both chains are stored together.

In this structure, each vertex that, for example, vanishes from one of the chains L and R due to an update that changes the convex hull, has to be deleted explicitly. Note that the number of vertices changing on L and R in dual space corresponds to the number of edges that are destroyed and established on the convex hull in primal space, which in turn differs by at most two from the number of points that change on the convex hull. That is, if k is the number of points changing on the hull due to an update, $c(n)$ is the time to update the convex hull and $u(n)$ is the time to update the mixed L, R-structure (for one vertex of L or R), then the running time for an update operation will be proportional to $c(n) + k \cdot u(n)$.

In the semi-dynamic case we know that, summed over all operations, the number of points deleted from or inserted into the hull is $O(n)$. In the deletions only case, n is the size of the initial set, and in the insertions only case, n is the size of the final set.

Therefore, we can amortize the cost, i.e., if k_i is the number of vertices that change on the hull due to update $\#i$, then the total update cost is proportional to

$$\sum_{i=1}^{n} c(n) + k_i \cdot u(n) = O\left(n\left(c(n) + u(n)\right)\right). \tag{1}$$

For this reason, the method is not applicable to the fully dynamic case, where the total number of points deleted from or inserted into the hull is no longer related to the size of the point set itself.

3.2 The union-split-find problem

Consider the following problem. Let B be a linear list of elements some of which are marked. The marked elements partition the list into intervals. We want to carry out the following operations:

Find(x : element): return the next marked element y to the right of x in B (including x);

Split(x : unmarked element): turn x into a marked element;

Union(x : marked element): turn x into a unmarked element;

Add(y, x : element): insert a new unmarked element y before x into B;

Erase(x : unmarked element): remove x from B;

In [7] it is shown that each of the above five operations can be supported in time $O(\log\log|B|)$ and space $O(|B|)$, where the time bound is worst case for Union, Split and Find and amortized for Add and Erase.

3.3 The mixed L, R-structure

We shall now describe the mixed L, R-structure that stores the vertices of the chains L and R together in order to speed up queries.

From now on, we treat L and R as lists of items, ordered by a real-valued key. (This key is the x-coordinate of the point stored in the chain L or R.) We will maintain the ordered list $B := L \cup R$ such that we can execute dictionary operations and are also able to support the repertory described in Subsection 3.2. The mixed L, R-structure is composed of the following parts:

1. A balanced binary tree $T(B)$ that stores the elements of B and supports insertions, deletions and searches of elements in $O(\log |B|)$ time.

2. A structure $USF(LR)$, which is an instance of the union-split-find structure of Subsection 3.2, that stores the list $B = L \cup R$ such that the elements of L are unmarked and the elements of R are marked.

3. A structure $USF(RL)$, analogous to $USF(LR)$, where the elements of L are marked and the elements of R are unmarked.

Furthermore, we link the occurrences of elements in $USF(LR)$ and $USF(RL)$ to the corresponding occurrence in $T(B)$.

We want to use the structure for the following query:

Neighbor$(v : list_1) : list_2$

given $v \in list_1$, find the right neighbor of v in $list_2$, where $list_1$ and $list_2$ are L and R or vice versa.

We can implement this operation by executing Find(v), using either the structure $USF(LR)$, if $v \in L$, or the structure $USF(RL)$, if $v \in R$.

We also want to update the data structure under insertions and deletions in L and R such that the above mentioned marking invariants are preserved. Figure 3 shows how to update the mixed L, R-structure under insertions into and deletions from L. Insertions and deletions of elements in R are performed analogously. (We denote an operation on a certain data structure by adding the name of the structure as a prefix.)
Note that inserting or deleting $v \in L$ needs one additional operation in the structure $USF(RL)$, because there, L is the set of marked items.

Lemma 3 *Let $n = |L| + |R|$. Then the Neighbor query takes time $O(\log \log n)$ and the amortized time of the update operations is bounded by $O(\log n)$. The data structure uses $O(n)$ space.*

Proof: The time bound for the query follows from the description of the union-split-find structure above. Also, the structures $USF(B)$ and $T(B)$ both use linear space. Now let us turn to the update algorithms in Figure 3. The search time in the tree $T(B)$ in line (3) of the insertion algorithm is $O(\log n)$. The other operations are performed on the union-split-find structure and take $O(\log \log n)$ amortized time. Therefore, the amortized update time is bounded by $O(\log n)$. ∎

```
(1)  Algorithm insert v into L
(2)  begin
(3)      find the position of v in the list B = L ∪ R, by binary search in the tree T(B);
         this gives us an element w ∈ B such that v's place in B is directly in front of w;
(4)      T(B)Insert(v, w);
(5)      USF(LR)Add(v, w);
(6)      USF(RL)Add(v, w);
(7)      USF(RL)Split(v);
(8)  end;

(1)  Algorithm delete v from L
(2)  begin
(3)      T(B)Delete(v);
(4)      USF(LR)Erase(v);
(5)      USF(RL)Union(v);
(6)      USF(RL)Erase(v);
(7)  end;
```

Figure 3: The algorithms to update the mixed L, R-structure for an insertion into or a deletion from the chain L.

3.4 The complete algorithm

Having described the mixed L, R-structure, we can add the structures that keep the convex chains L and R separately to obtain our complete data structure:

1. the structure that maintains the convex chain L,
2. the structure that maintains the convex chain R,
3. the mixed L, R-structure.

We have to add some details. As already mentioned, to store the convex chains L and R, we use either one of the semi-dynamic convex hull structures of [4] or [10]. In each case, the results of these papers admit to maintain the following information in $O(\log n)$ amortized time per insertion or deletion of a point: there are balanced binary trees storing the points of the current upper and lower hull (U and D) in their leaves, sorted by x-coordinate. Also, each internal node in these trees has pointers to the leftmost and the rightmost leaf in its subtree, respectively. We call these trees *hull trees*. Applying the duality rules given in Subsection 2.1, we obtain semi-dynamic structures for maintaining the intersection of halfplanes such that the halfplanes currently forming the boundary are stored at the leaves of a search tree. Each vertex on U resp. D corresponds to a halfplane that contributes to the boundary of the chain L resp. R. Moreover, two neighboring vertices on U resp. D specify a vertex on L resp. R. So, as discussed in Subsection 2.2, each subsequent pair of leaves in the hull trees corresponds to a

vertex of L or R. Since we need to store the vertices of L and R explicitly in the mixed L, R-structure and also want to connect this structure with the hull trees, we let each subsequent pair of leaves in a hull tree point to the occurrence (in $T(B)$) of the vertex in L or R that they determine.

We can now implement the operations *insert p into S*, *delete p from S* and *report_width* and thus prove Theorem 2.

The update operations are very similar. In Figure 4, we formulate a general update algorithm and mention the details that are different later. Let L' resp. R' denote the convex chains L and R after the update.

```
(1)   Algorithm Update(p);
(2)   begin
(3)       modify the hull tree representing U (and L);
(4)       modify the hull tree representing D (and R);
(5)       out := {v : v ∈ L ∪ R  and  v ∉ L' ∪ R'};
(6)       in := {v : v ∉ L ∪ R  and  v ∈ L' ∪ R'};
(7)       forall v ∈ out do
(8)           delete v from the mixed L, R-structure as described in Figure 3
(9)       od;
(10)      forall v ∈ in do
(11)          insert v into the mixed L, R-structure as described in Figure 3
(12)      od;
(13)  end;
```

Figure 4: The algorithm to insert/delete a point p.

We obtain Insert(p) and Delete(p) from Update(p) by applying the appropriate semi-dynamic convex hull algorithm in lines (3) and (4). In lines (5) and (6), the sets *in* and *out* are determined. These are the points that change in L and R due to the changes on the convex hull $CH(S)$.

Lemma 4 *The operations Insert(p) and Delete(p) correctly maintain the data structure and run in $O(\log n)$ amortized time.*

Proof: The correctness is obvious. The running time of lines (3) and (4) is $O(\log n)$ amortized. The cost of lines (5) and (6) is proportional to k, where k is the number of points newly arising or vanishing on L or R. Finally, lines (7)-(12) take time $O(k \log n)$. As discussed in Subsection 3.1, the sum of the k's over all updates is $O(n)$. So, according to Equation (1), the amortized update time is proportional to

$$\frac{1}{n}\sum_{i=1}^{n} c(n) + k_i \cdot u(n) = O(c(n) + u(n)),$$

where $c(n)$ is the time to update the hull, k_i is the cardinality of $in \cup out$ and $u(n)$ is the time to update the mixed L, R-structure (for one vertex). We have $c(n) = O(\log n)$ amortized and $u(n) = O(\log n)$ amortized, and therefore the amortized update time is $O(\log n)$. ∎

It remains to describe the operation report_width. We only discuss the algorithm to find $p_0 \in L$ with minimal distance to chain R. To do this, we simply use the search algorithm of Figure 2, which walks down a root-to-leaf path in the hull tree for the points of U (resp. for the halfplanes contributing to L), with the following details added: Suppose we are at node v of the hull tree. We find p_{mid} and its neighbors p_l and p_r in L in lines (1) and (2) as before. Each of these vertices is defined by the intersection of the boundaries of the two halfplanes which are stored in adjacent leaves of the hull tree. From each pair of leaves defining vertex $p \in \{p_l, p_{mid}, p_r\}$, pointers are available to the occurrence of p in the tree $T(B)$, and from there to the occurrence of p in the structures $USF(LR)$ and $USF(RL)$.

Now, for $p \in \{p_l, p_{mid}, p_r\}$, we perform the query Neighbor(p) in the structure $USF(LR)$ to obtain p's next neighbor to the right in R, if it exists. From Lemma 3, this takes time $O(\log\log n)$. Given this vertex of R, we can compute the segment vertically below p and the distance of p to this segment in constant time. If the right neighbor does not exist, then the segment vertically below p must be the ray at the right end of the chain. We can access this ray easily.

The rest of the algorithm is as in Figure 2. We have the following lemma:

Lemma 5 *The running time of operation report_width is* $O(\log n \log\log n)$.

Note that the data structure discussed in this section corresponds to the one given in Subsection 2.2, i.e. the query does not really report the width, but a distorted value that depends on the slope of the lines that give rise to the reported distance. However, as discussed at the beginning of this section, by applying the approximation technique of Subsection 2.3, we obtain the semi-dynamic approximation result claimed in Theorem 2.

4 Conclusion and open problems

We have shown how to maintain an approximation of the width of a set of n points in the plane, when either only insertions or only deletions of points are performed, such that the amortized update time is $O(\alpha \log n)$ and the time to retrieve the approximate width is $O(\alpha \log n \log\log n)$. The data structure uses $O(\alpha n)$ space. The parameter α expresses the quality of the approximation.

There are still a lot of open problems left. First, one could improve the running times of the operations for the approximation problem. Since it seems that the algorithms at least have to maintain the convex hull, improving the time for the approximate width query to $O(\alpha \log n)$ would be the goal.

Furthermore, as already mentioned in the introduction, there is still no exact algorithm for dynamic maintenance with a sublinear running time for both updates and width query.

Acknowledgements

The author would like to thank Rajeev Raman for helpful discussions and Michiel Smid for reading a preliminary draft of this paper.

References

[1] P.K. Agarwal and M. Sharir. *Off-line dynamic maintenance of the width of a planar point set.* Computational Geometry, Theory and Applications **1**, 1991, pp. 65-78.

[2] K. Mehlhorn, St. Näher. *Dynamic fractional cascading.* Algorithmica **5**, Nr. 2, 1990, pp. 215-241.

[3] D. Dobkin and S. Suri. *Maintenance of geometric extrema.* Journal of the ACM **38**, 1991, pp. 275-298.

[4] J. Hershberger and S. Suri. *Applications of a semi-dynamic convex hull algorithm.* Proc. Second Scand. Workshop on Alg. Theory (SWAT), 1990, pp. 380-392.

[5] M. Houle and G. Toussaint. *Computing the width of a set.* Proc. First Annual ACM Symp. on Computational Geometry, 1985, pp. 1-7.

[6] R. Janardan. *On maintaining the width and diameter of a planar point set online.* International Symp. on Algorithms, 1991, LNCS Vol. 557, pp. 137-149.

[7] K. Mehlhorn. *Datenstrukturen und effiziente Algorithmen 1.* B.G. Teubner, Stuttgart, Germany, 1986

[8] M. Overmars. *Dynamization of order-decomposable set problems.* Journal of Algorithms **2**, 1981, pp. 245-260.

[9] M. Overmars and J. van Leeuwen. *Maintenance of configurations in the plane.* Journal of Computer and System Sciences **23**, 1981, pp. 166-204.

[10] F.P. Preparata. *An optimal real time algorithm for planar convex hulls.* Comm. of the ACM **22**, 1979, pp. 402-405.

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