Dynamic Planar Convex Hull Operations in Near-Logarithmic Amortized Time*

Timothy M. Chan†

May 31, 2000

Abstract

We give a data structure that allows arbitrary insertions and deletions on a planar point set P and supports basic queries on the convex hull of P, such as membership and tangent-finding. Updates take $O(\log^{1+\varepsilon} n)$ amortized time and queries take $O(\log n)$ time each, where n is the maximum size of P and ε is any fixed positive constant. For some advanced queries such as bridge-finding, both our bounds increase to $O(\log^{3/2} n)$. The only previous fully dynamic solution was by Overmars and van Leeuwen from 1981 and required $O(\log^2 n)$ time per update and $O(\log n)$ time per query.

1 Introduction

Although the algorithmic study of convex hulls is as old as computational geometry itself, the basic problem of optimally maintaining the planar convex hull under insertions and deletions of points [32, 37] remains unsolved and has been regarded by some as one of the foremost open problems in the area [15, 27]. Besides its natural appeal, such a dynamic data structure has a wide range of applications, as it is often used as subroutines for tackling more difficult geometric problems (both dynamic and static); see the references [5, 8, 13, 14, 17, 18, 21, 22, 23, 25, 26, 28, 30, 39] for a mere sampling.

Among the earliest proposed methods for dynamic hull maintenance in the plane was one by Overmars and van Leeuwen [36] and dated back to 1981. The worst-case update time is $O(\log^2 n)$, where n is the maximum number of points. Since this data structure actually stores the vertex sequence of the convex hull in a balanced search tree, all the usual query operations (see below) can be carried out in $O(\log n)$ time. However, Overmars and van Leeuwen's work left open a challenging question: can the update time be brought down to $O(\log n)$?

Unfortunately, no progress was made since then, despite considerable effort by researchers, mostly in the form of special-case results. First, if deletions are not allowed, then it is relatively easy to achieve optimal $O(\log n)$ insertion time [37]. If instead insertions are not allowed after preprocessing,

*A preliminary version of this work appeared in *Proc. 39th IEEE Sympos. Found. Comput. Sci.*, 1999. Portions of this work were performed while the author was at Department of Mathematics and Computer Science, University of Miami.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, tmchan@math.uwaterloo.ca

then a modification of Overmars and van Leeuwen's structure, as described by Chazelle [13] and Hershberger and Suri [26], can perform n deletions on an n-point set in $O(n \log n)$ total time, which is optimal in the amortized sense.

In another paper [27], Hershberger and Suri considered the *off-line* case, where the sequence of insertions and deletions is given in advance, and showed how to process n operations in optimal $O(n \log n)$ time for most types of queries. Alternatively, if the update sequence is *random* in a certain sense, then Mulmuley [33] and Schwarzkopf [40] showed how to achieve $O(\log n)$ expected time per update. Other scenarios were explored; for example, Friedman, Hershberger, and Snoeyink [22] obtained also logarithmic amortized time if updates occur only at the ends of a simple path.

While the specialized results mentioned above indeed have many applications, their usefulness is limited due to their various assumptions. No improvements on Overmars and van Leeuwen's $O(\log^2 n)$ update bound were known for the problem in its fully dynamic setting, except for an unpublished result by Kapoor [29], who achieved an $O(n \log n + D \log^2 n)$ time bound for n insertions and D deletions (unfortunately, in some applications, there are just as many deletions as insertions).

In this paper, we present the first general method to break the $\log^2 n$ barrier: an $O(n)$-space data structure that can be maintained in $O(n \log^{1+\varepsilon} n)$ time over any n on-line updates. Here and throughout, $\varepsilon > 0$ denotes an arbitrarily small constant; factors hidden in O notation may depend on ε. Unlike in Overmars and van Leeuwen's approach, we do not actually store a representation of the vertex sequence of the convex hull. Nevertheless, our data structure contains sufficient information to answer many basic queries on the convex hull. (This query-oriented approach has been advocated by numerous authors; e.g., see [15, 27].)

Our data structure can handle the following types of query operations in $O(\log n)$ time: (i) decide whether a given line intersects the convex hull; (ii) decide whether a given point is inside the convex hull; (iii) find the hull vertex that is extreme in a given direction; (iv) find the two vertices adjacent to a given hull vertex (*gift-wrapping*); and (v) find the two tangent hull vertices from a given exterior point. For example, we can report h consecutive hull vertices in $O(h \log n)$ time by repeated gift-wrappings.

We are less successful in dealing with other classes of queries, which require $O(\log^2 n)$ time: (vi) find the two hull edges (*bridges*) that intersect a given line; and (vii) assuming all points are colored either red or blue, find the two common tangents separating two disjoint convex hulls, one of the red points and one of the blue points. It is possible to balance the cost of updates and queries so that n such operations take $O(n \log^{3/2} n)$ time. (Such a trade-off was posed as a question by Hershberger and Suri [27].) Although this does not quite approach the ideal $O(n \log n)$ bound, we have certainly opened up the possibility of further improvements and extensions through more work.

As it turns out, all that is required to obtain our results is just the right combination of a few standard techniques (all available from the 1980s): a previous deletion-only data structure, a well-known dynamization trick, and a multi-level interval tree, coupled with bootstrapping and a clever choice of parameters. We remark that the resulting combination is nontrivial though and may not be simple enough for practical implementation.

2 The Data Structure

We first restrict ourselves to the simplest kind of queries—finding the hull vertex extreme in a given direction—so as to illustrate the main ideas. This search problem is nice to work with because it is *decomposable*, i.e., the answer to a query on input set $P_1 \cup P_2$ can be inferred in constant time from the answer on P_1 and the answer on P_2. It is sufficient to consider the upper hull alone.

We actually examine an equivalent problem in the *dual* setting [20, 34, 37], transforming upper hulls of points into upper envelopes of lines in the plane. Let L be the current set of lines and $\mathcal{E}(L)$ denote the upper envelope of L. The goal now is to design a data structure that allows insertions and deletions on L and supports *vertical line queries* (or *vertical ray shooting*): intersect $\mathcal{E}(L)$ with a given vertical line q, i.e., find the highest line of L at q.

Suppose that there is already a preliminary method for this problem with the following performance: total update time is $O(mU_0(n))$, query time is $O(\log n)$, and space is $O(n)$. Here, n is the maximum size of L, m is the total number of updates on L (assuming L is initially empty), and $U_0(\cdot)$ is some "well-behaved" function, obeying $U_0(2n) = O(U_0(n))$. We show how to use this "base data structure" to obtain a better data structure.

A deletion-only data structure. Our starting point is a restricted data structure that does not allow insertions. As we have mentioned, the known methods by Chazelle [13] and Hershberger and Suri [26] maintain the upper envelope of n lines through n deletions optimally in $O(n \log n)$ time and $O(n)$ space. Notice that the total amount of changes to the upper envelope here is $O(n)$ (as can be seen by a backwards argument: adding a line creates at most two new vertices and three new edges to the envelope).

Dynamization of the deletion-only structure. To incorporate insertions, we employ a general *dynamization* technique for decomposable search problems, attributed to Bentley and Saxe [6, 15, 32, 34]. This technique uses a simple binary-counting trick and usually slows down update time by a logarithmic factor. To lower the effect of the slow-down, we actually apply a b-ary variant (e.g., see [35]) of the counting trick for a suitably chosen parameter b, as we now describe.

We maintain a partition of L into a collection $\mathcal{C}$ of subsets, where the only operations on a subset are creation and deletion. For each subset $C \in \mathcal{C}$, we maintain the *sub-envelope* $\mathcal{E}(C)$ by the above deletion-only data structure. Each subset is assigned an integer *depth* value. We will ensure the invariant that at most $b-1$ subsets reside at each depth.

To insert an element ℓ to L, we create a new singleton subset $\{\ell\}$ and put it in $\mathcal{C}$ at depth 0. To ensure the invariant, we apply the following rule: whenever there are b subsets $C_1, \ldots, C_b$ at a depth i, remove $C_1, \ldots, C_b$ from $\mathcal{C}$, create a new subset $C_1 \cup \cdots \cup C_b$, and put it in $\mathcal{C}$ at depth $i+1$.

To delete an element ℓ from L, we identify the subset $C \in \mathcal{C}$ containing ℓ and delete ℓ from C.

Analysis of the dynamization technique. By induction, there are at most m/b^i depth-i subsets created over a lifetime of m updates on L (since a depth-i subset is created from b depth-$(i-1)$ subsets). Hence, depth values range from 0 to $\log_b m$. So, the number of subsets at any given time is $O(b \log_b m)$.

Over time, the depth value of the subset containing a fixed element can only increase. Thus, each element belongs to at most $O(\log_b m)$ subsets over time, so the sum of the sizes of all subsets created is $O(m \log_b m)$. It follows that the total cost of maintaining the deletion-only data structures over all subsets created is $O(m \log_b m \log m)$. Furthermore, the total amount of changes to all sub-envelopes over time is $O(m \log_b m)$.

A new problem on line segments. Now, $L = \bigcup_{C \in \mathcal{C}} C$. The standard way to apply the above technique to answer a query on L is simply to query on each of the $O(b \log_b m)$ subsets separately and combine the answers. With each sub-envelope $\mathcal{E}(C)$ available, a query on a subset $C \in \mathcal{C}$ costs $O(\log m)$ time by binary search. Exploiting the decomposability of our problem, we can then answer a vertical line query in $O(b \log_b m \log m)$ time, which unfortunately is suboptimal by a logarithmic factor or worse, regardless of the choice of b. To speed up the process, we need additional structures to search in all sub-envelopes simultaneously.

Our new idea is a simple one. Define a set of line segments S, consisting of the edges of all current sub-envelopes $\{\mathcal{E}(C) \mid C \in \mathcal{C}\}$. We know the following properties about this set S:

(a) Over time, S undergoes M insertions and deletions of line segments, where $M = O(m \log_b m)$.

(b) Any vertical line crosses a maximum of B line segments of S, where $B = O(b \log_b m)$.

(c) The upper envelope of S coincides with $\mathcal{E}(L)$.

We have therefore reduced our problem to one of searching in the upper envelope of S.

Let us introduce a bit of notation at this point: let $\hat{s}$ be the extension of a line segment s to a line, and let $\hat{A} = \{\hat{s} \mid s \in A\}$. In our case, we know that $\hat{S} \subseteq L$.

An interval tree on the line segments. To solve this new problem, we can use a traditional data structure on line segments—the *segment tree* or the *interval tree* [7, 15, 32, 37]. In the former approach, a segment is stored in logarithmically many nodes of the tree, which will compromise our already delicate update bound. We adopt instead the latter approach, storing a segment in exactly one node. Note that in general, interval trees are less suited for augmentation of secondary data structures, which is required here; but our case is different, because every vertical line is known to cross a small number of segments by the above property (b). We also need a B-ary version of the tree instead of the usual binary form; this is dynamized by mimicking standard operations on a *B-tree* (e.g., see [16]).

Each node of our interval tree $\mathcal{T}$ represents a *slab*, i.e., a region enclosed by two vertical lines (*walls*), where the leaf slabs are disjoint and cover the entire plane, and the slab at a node is the union of the slabs at its children. At each node, we can maintain the order of the children slabs by a "mini-tree" (an $O(B)$-sized standard balanced search tree). We store a segment $s \in S$ in the lowest node whose slab contains s. Let S_v be the resulting *canonical subset* of all segments stored in node v. We maintain a base data structure for each $\hat{S}_v$.

We will ensure the following invariants for balance: the degree at the root is between 2 and $2B - 1$, the degree of every non-root internal node is between B and $2B - 1$, and each leaf contains exactly one endpoint; furthermore, every leaf has the same distance to the root.

To insert a new endpoint to the tree $\mathcal{T}$, we find the leaf whose slab contains the endpoint, and split it into two new leaves; its parent now has one more child. To maintain the invariants, we apply the following rule: whenever a node v has $2B$ children $w_1, \ldots, w_{2B}$ (from left to right), we split v into two new nodes v' and v'', where $w_1, \ldots, w_B$ become the children of v' and $w_{B+1}, \ldots, w_{2B}$ become the children of v''; the parent u of v now has v' and v'' as children. To readjust the canonical subsets, we examine all segments in S_v, put those contained in the slab at v' into $S_{v'}$, put those contained in the slab at v'' into $S_{v''}$, and insert the remaining ones into S_u. When the root is split, we make a new root.

To insert a segment s to S, we first insert its two endpoints to the tree as described above. Then we find the lowest node v whose slab contains s and insert s to S_v.

To delete a segment s from S, we identify the node v that stores s and delete s from S_v (retaining lazily the endpoints in the tree).

To answer a vertical line query on L, we find the leaf v whose slab contains the given vertical line q. Say $v_1, \ldots, v_H$ are the ancestors of v. By property (c), the highest line of L at q coincides with the highest segment of S at q. Since each segment of S intersecting q belongs to one of the canonical subsets $S_{v_1}, \ldots, S_{v_H}$, the line that defines the answer belongs to one of $\hat{S}_{v_1}, \ldots, \hat{S}_{v_H}$. Querying the base data structure for each $\hat{S}_{v_i}$ yields the answer.

Analysis of the interval tree. The *height* of a node refers to the distance from the node down to a leaf. By induction, there are $O(M/B^i)$ non-root nodes at height i after M updates on S (since the degree of each non-root node is at least B). By letting H be the overall height of the tree $\mathcal{T}$, this implies that $H = O(\log_B M)$. It also implies that over time, there are $O(M/B^i)$ splits at height i (since each split increments the number of non-root nodes).

Note that the maximum size of each canonical subset S_v is $O(B^2)$ by property (b), because each segment in S_v must cross one of the $O(B)$ walls at the children of v.

We first determine the total number of updates to the canonical subsets over time. The amount of changes that occur from splitting a height-i node v is $O(|S_v|)$, which is bounded by 0 for $i = 0$, $O(B)$ for $i = 1$, and $O(B^2)$ for $i \geq 2$. Consequently, the total amount of changes caused by endpoint insertions is $O((M/B)B + (M/B^2)B^2 + (M/B^3)B^2 + \cdots) = O(M)$. In addition, we have M changes caused by insertions and deletions of segments. Thus, the total cost of maintaining all the base data structures is $O(MU_0(B^2))$.

Similarly, we can bound the total number of updates to the mini-trees by $O(M + (M/B)B + (M/B^2)B + \cdots) = O(M)$ and their total cost by $O(M \log B)$.

In a top-down fashion, we can find a leaf slab containing a given point by H mini-tree searches in $O(H \log B) = O(\log M)$ time. For an insertion, we need to find the lowest slab containing a segment, which can similarly be done in $O(\log M)$ time. We conclude that the total cost of updates over time is $O(M(\log M + U_0(B^2)))$.

A query on L requires queries on H canonical subsets. With the base data structures available, the query time is $O(H \log(B^2)) = O(\log M)$.

Bootstrapping. To summarize, we have a new method for answering vertical line queries with total update time $O(M(\log M + U_0(B^2)))$, query time $O(\log M)$, and space $O(M)$, where $M = O(m \log_b m)$ and $B = O(b \log_b m)$. Since we know S has maximum size n, by a standard trick of rebuilding the tree $\mathcal{T}$ after every n updates on S, we can reduce the total update time to

$$O((M/n) \cdot n(\log n + U_0(B^2))) \;=\; O(m \log_b m \cdot (\log n + U_0((b \log_b m)^2))),$$

query time to $O(\log n)$ and space to $O(n)$. Furthermore, since we know L has maximum size n, by rebuilding the entire data structure after every n updates on L, the total update time becomes

$$O((m/n) \cdot n \log_b n \cdot (\log n + U_0((b \log_b n)^2))).$$

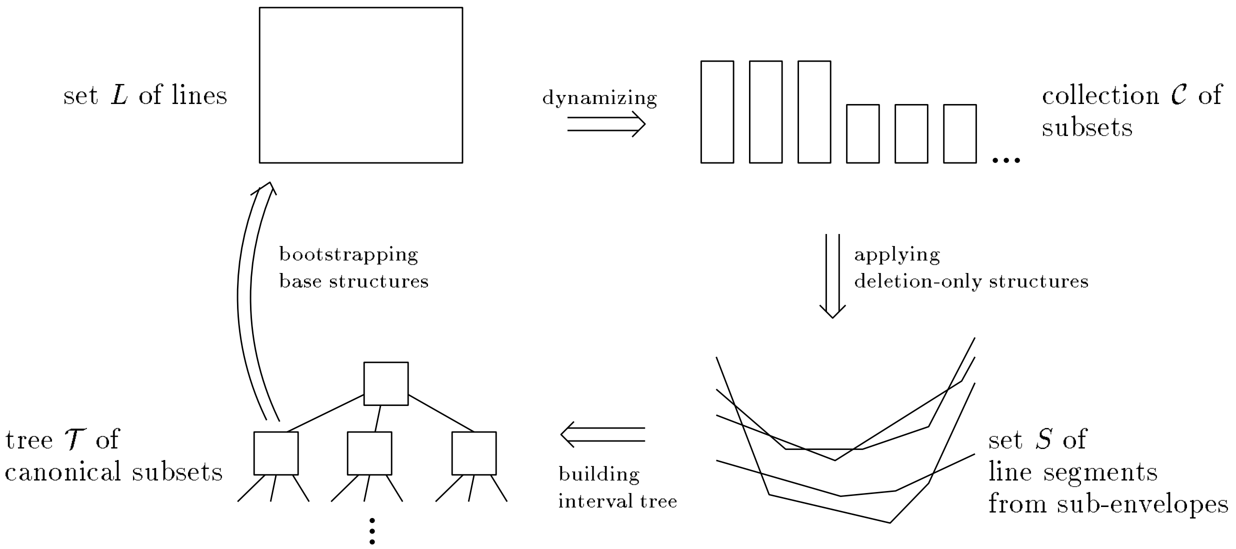

Figure 1: High-Level Picture of Our Data Structure

One can of course start with a trivial base method with $U_0(n) = n^{O(1)}$. With b constant, the total update time is $O(m \log^{O(1)} n)$. By bootstrapping, we can use the new method as the base to get faster and faster methods.

Say $U_0(n) = O(\log^\beta n)$ for some constant β. Then the update time becomes

$$O(m \log_b n \cdot (\log n + \log^\beta(b \log_b n))).$$

Choosing b so that $\log b \approx \log^{1/\beta} n$ yields a total update time of $O(m \log^{2-1/\beta} n)$. Thus, we get a new bound $U_0(n) = O(\log^{2-1/\beta} n)$, in essence replacing β with $2 - 1/\beta$. Repeating a finite number of times reduces β to a constant arbitrarily close to 1.

See Figure 1 for an overall picture of the resulting data structure.

Theorem 2.1 *For any fixed $\varepsilon > 0$, we can maintain an $O(n)$-space data structure that allows insertions and deletions of lines in the plane in $O(\log^{1+\varepsilon} n)$ amortized update time and answers a vertical line query in $O(\log n)$ time, where n is an upper bound on the number of lines.*

Note. Although we have assumed that the value of n is known, we can remove this assumption by a standard "guessing" trick: whenever L is full, double the value of n and rebuild the data structure. It is easy to see that the amortized update time is unchanged.

If we are allowed to initialize L to a given set of size n_0, then the total cost of m updates becomes $O(n_0 \log n + m \log^{1+\varepsilon} n)$: simply keep a separate deletion-only structure for the initial elements.

3 More Queries

With the above data structure for vertical line queries, we can answer other classes of queries in $O(\log^2 n)$ time by the general technique of *parametric searching* [31], assuming a *locality* property—that one can infer on which side of a vertical line q the solution lies by examining the highest line of L at q alone. (In most cases, parametric search can be replaced by binary search.) With a little more care, we show how to obtain better results for particular classes of queries.

Arbitrary line queries. Suppose we want to find the two tangents of the convex hull to an query point, knowing that the point is outside the hull. In the dual, this corresponds to *line queries* (subsuming *ray shooting*): intersect $\mathcal{E}(L)$ with an arbitrary query line q, knowing that the intersection is nonempty. We focus on computing the left intersection point, denoted by $\textsc{ans}(L, q)$. The right point can be determined similarly.

The only change is in the querying part, which requires searching the tree $\mathcal{T}$ along a root-to-leaf path $v_1, \ldots, v_H$, so that $\textsc{ans}(L, q)$ is contained in the slab at each of its nodes v_i. To compute v_{i+1} from v_i, we observe that all segments of S intersecting the walls at the children of v_i belong to $S_{v_1} \cup \cdots \cup S_{v_i}$. By property (c), the highest line of L at each of these walls thus belongs to $\hat{S}_{v_1} \cup \cdots \cup \hat{S}_{v_i}$. In other words, $\mathcal{E}(L)$ coincides with $\mathcal{E}(\hat{S}_{v_1} \cup \cdots \cup \hat{S}_{v_i})$ at these walls. By the locality of our problem, we see that the slab of the child of v_i that contains $\textsc{ans}(L, q)$ is also the slab that contains $\textsc{ans}_i = \textsc{ans}(\hat{S}_{v_1} \cup \cdots \cup \hat{S}_{v_i}, q)$. So v_{i+1} can be deduced from $\textsc{ans}_i$. Since the line that defines $\textsc{ans}(L, q)$ belongs to $\hat{S}_{v_1} \cup \cdots \cup \hat{S}_{v_H}$, the final answer $\textsc{ans}_H$ is correct.

How is $\textsc{ans}_i$ computed? Because the problem is decomposable, we can calculate $\textsc{ans}_i$ from $\textsc{ans}_{i-1}$ and $\textsc{ans}(\hat{S}_{v_i}, q)$ in constant time. The latter requires querying a base data structure for a canonical subset. The whole process requires H queries to the base data structures and the mini-trees, so the query time is $O(H \log(B^2)) = O(\log M)$ as before. The same bootstrapping strategy implies:

Theorem 3.1 *The data structure of Theorem 2.1 supports arbitrary line queries in $O(\log n)$ time.*

Note. All queries (i)–(v) mentioned in the introduction reduce to (iii) extreme-point queries and (v) tangent-finding queries, and can now be solved. For (i), we can decide whether a query line intersects the convex hull by finding the two vertices extreme in the directions parallel to the line. For (ii), we can attempt to find a tangent to the given point and conclude whether the point is outside or inside the convex hull by verifying whether the supposed tangent is correct; to verify a tangent, we can simply find the extremum along a direction parallel to the tangent line. Finally, for (iv), we can gift-wrap at a vertex by finding the tangents at a point close to the vertex.

Linear programming queries. Suppose we want to find the intersection of the convex hull with a vertical line. The dual corresponds to an instance of *linear programming queries*: find the extreme point on $\mathcal{E}(L)$ along a query direction q, i.e., optimize a linear function over an intersection of upper halfplanes. This example illustrates some of the difficulties that arise when we do not have the decomposability property.

Here, we are unable to bootstrap without increasing the query time, and so we only use Overmars and van Leeuwen's data structure as our base method with $U_0(n) = O(\log^2 n)$. As before, let $\textsc{ans}(L, q)$ denote the solution point. Because the locality property is still satisfied, we can use essentially the same query algorithm as for arbitrary line queries, proceeding along a root-to-leaf path and keeping track of $\textsc{ans}_i = \textsc{ans}(\hat{S}_{v_1} \cup \cdots \cup \hat{S}_{v_i}, q)$. However, without decomposability, the computation of $\textsc{ans}_i$ is more involved.

The pair of lines that defines $\textsc{ans}_i$ is either contained in $\hat{S}_{v_1} \cup \cdots \cup \hat{S}_{v_{i-1}}$ or in $\hat{S}_{v_j} \cup \hat{S}_{v_i}$ for some $j \in \{1, \ldots, i-1\}$. It is not difficult to see therefore that $\textsc{ans}_i$ is the least extreme point among $\textsc{ans}_{i-1}$ and $\textsc{ans}(\hat{S}_{v_j} \cup \hat{S}_{v_i})$ over all $j = 1, \ldots, i-1$. Given any pair of convex polygons represented by Overmars and van Leeuwen's structures, we can answer linear programming queries over their intersection in logarithmic time (e.g., see [19]). Thus, each $\textsc{ans}(\hat{S}_{v_j} \cup \hat{S}_{v_i})$ can be found in

$O(\log(B^2))$ time, so $\textsc{ans}_i$ can be calculated from $\textsc{ans}_{i-1}$ in $O(H \log(B^2))$ time. The total cost of a query is $O(H^2 \log(B^2)) = O(\log^2 M / \log B)$.

In the final analysis, we choose the parameter b so that $\log b \approx \log^{1/2} n$ and obtain total update time $O((m/n) \cdot n \log_b n \cdot (\log n + \log^2(b \log_b n))) = O(m \log^{3/2} n)$ and query time $O(\log^2 n / \log(b \log_b n)) = O(\log^{3/2} n)$.

Theorem 3.2 *We can maintain an $O(n)$-space data structure that allows insertions and deletions of upper halfplanes in $O(\log^{3/2} n)$ amortized update time and answers a linear programming query in $O(\log^{3/2} n)$ time, where n is an upper bound on the number of halfplanes.*

Note. The same approach extends to (vi) general bridge-finding queries, as mentioned in the introduction, which seek the intersection of the convex hull with an arbitrary line. In the dual, the objective is now to find the most clockwise (or counterclockwise) point in $\mathcal{E}(L)$ from a given query point $q = (a, b)$. In the calculation of $\textsc{ans}(\hat{S}_{v_j} \cup \hat{S}_{v_i})$, we need to find the most clockwise point in the upper envelope of two convex chains, but by clipping the chains to $x \geq a$ and applying the projective transformation $(x, y) \mapsto (\frac{1}{x-a}, \frac{y-b}{x-a})$, this subproblem reduces to a linear programming query (finding the lowest point) in the intersection of two convex polygons. So, the $O(\log^{3/2} n)$ result still applies.

Within the same bounds, we can also perform linear programming queries on a set containing both upper and lower halfplanes, by incorporating edges from both upper and lower sub-envelopes into the canonical subsets of the interval tree. (Overmars and van Leeuwen's structures can maintain intersections of arbitrary halfplanes.) The separating tangent problem (vii) for two convex hulls dualizes to this.

4 Remarks

It seems appropriate to point out at least some of the applications of our data structure. We just briefly mention those from the literature that catch our attention:

- The *k-level* of n lines in the plane [3, 4, 20, 34] is an important geometric structure (related dually to "k-sets"). Using Overmars and van Leeuwen's dynamic hull structure, Edelsbrunner and Welzl [21] showed that the k-level can be constructed in $O(n \log m + m \log^2 n)$ time (see also [10]), where m denotes the output size. Our data structure improves the bound to $O(n \log m + m \log^{1+\varepsilon} n)$ deterministically. The algorithm uses $O(n)$ space. Very recently, Har-Peled [24] announced a marginally faster $O((n + m)\alpha(n) \log n)$ expected time bound via a randomized approach. Actually, as the author observed [12], an earlier randomized algorithm by Agarwal *et al.* [1] gives almost the same result. However, it is unclear whether the space complexity of either randomized algorithm can be made $O(n)$.

- The *order-k Voronoi diagram* of n points in the plane is another well-known structure in computational geometry, with various applications of its own. An early algorithm of Chazelle and Edelsbrunner [14] used dynamic convex hulls as subroutines. Our result improves their worst-case time bound from $O(n^2 \log^2 n)$ to $O(n^2 \log^{1+\varepsilon} n)$. For smaller values of k, faster methods are known, as the size of the diagram is $O(nk)$. Indeed, by using some complicated machinery (namely, "shallow cuttings" [11]) in conjunction with this new result, we can construct the diagram in time $O(nk \log^{1+\varepsilon} k \cdot (\log n / \log k)^{O(1)})$, which is currently the best determinstic bound. Simpler and slightly faster randomized algorithms are known though [1, 11, 38].

- Basch, Guibas, and Ramkumar [5] examined a natural version of the *segment intersection* problem: given a connected family R of n red line segments and a connected family B of n blue line segments in the plane, report all intersecting pairs from $R \times B$. They (with an observation attributed to de Berg) solved the problem in $O((n+m)\log^3 n)$ time with Overmars and van Leeuwen's data structure, where m stands for the output size. Our result reduces the bound to $O((n+m)\log^{2+\varepsilon} n)$.

- Matoušek [30] studied variants of the *linear programming* problem with violated constraints. One instance is: given n lines in the plane, a linear objective, and a number k, report all local minima of the first k levels. It is known that the number of local minima is $O(k^2)$. His $O(n \log k + k^2 \log^2 n)$ time bound improves to $O(n \log k + k^2 \log^{3/2} n)$ by our data structure.

- Janardan [28], Rote, Schwarz, and Snoeyink [39] used Overmars and van Leeuwen's data structure to maintain an approximate *width* of a dynamic planar point set. With our data structure, a $(1+\delta)$-approximation can be reported in $O((1/\delta)\log^2 n)$ time, and updates take $O(\log^{1+\varepsilon} n)$ amortized time. (We are unable at the moment to obtain a faster reporting time for the kind of queries that occurs in this application.)

For the first three applications above, we can simplify our data structure somewhat (omitting the interval-tree component); because queries coming from these algorithms actually are *monotone* in a certain sense, we do not need the full power of dynamic convex hulls, but rather special structures known as "kinetic priority queues." See a separate note [12] for more details.

There are some situations (e.g., in [17, 18]) where Overmars and van Leeuwen's technique works and ours currently does not. To widen applicability further, it is worthwhile to enlarge the repertoire of query and modifying operations. We should remark that certain queries that are of a global nature, such as reporting the cardinality/area/perimeter of the convex hull, appear particularly difficult with our approach (or any approach that abandons an explicit representation of the hull, like Hershberger and Suri's off-line method [27]). Note that if the total amount of changes to the convex hull over an update sequence is N, then we can maintain the convex hull explicitly in overall time $O(n \log^{1+\varepsilon} n + N \log n)$ by performing gift-wrapping queries after each update.

As convex hulls are so fundamental, our ideas may be helpful for maintaining related geometric configurations in the plane, such as circular hulls and upper envelopes of curves. Dynamic data structures for querying planar Voronoi diagrams and three-dimensional convex hulls are much more involved, and polylogarithmic solutions are currently not known [2].

Finally, it would be desirable to eliminate the extra $\log^{\varepsilon} n$ factor in our update time, or to make our bounds worst-case rather than amortized (the latter is open even in the deletion-only case). Still, we cannot rule out the possibility of a simple optimal solution to the long-standing problem of planar convex hull maintenance.

Added Note. Recently, Brodal and Jacob [9] announced a slightly improved result using the ideas developed in this paper; their data structure supports basic convex hull queries (extreme point and tangent finding) in $O(\log n)$ time and updates in $O(\log n \log\log n)$ amortized time.

Acknowledgements. The author would like to thank the referees for their helpful comments.

References

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

[2] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13:325–345, 1995.

[3] P. K. Agarwal and M. Sharir. Arrangements and their applications. To appear in *Handbook of Computational Geometry* (J. Urrutia and J. Sack, ed.), North-Holland.

[4] A. Andrzejak and E. Welzl. k-sets and j-facets: a tour of discrete geometry. Manuscript, 1997.

[5] J. Basch, L. J. Guibas, and G. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proc. 4th European Sympos. Algorithms*, Lect. Notes in Comput. Sci., vol. 1136, Springer-Verlag, pages 302–319, 1996.

[6] J. Bentley and J. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1:301–358, 1980.

[7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[8] K.-F. Böhringer, B. R. Donald, and D. Halperin. On the area bisectors of a polygon. *Discrete Comput. Geom.*, 22:269–285, 1999.

[9] G. S. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log\log n)$ update time. To appear in *Proc. 7th Scand. Workshop Algorithm Theory*, 2000.

[10] T. M. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. *Discrete Comput. Geom.*, 16:369–387, 1996.

[11] T. M. Chan. Random sampling, halfspace range reporting, and construction of ($\leq k$)-levels in three dimensions. In *Proc. 39th IEEE Sympos. Found. Comput. Sci.*, pages 586–595, 1998. *SIAM J. Comput.*, to appear.

[12] T. M. Chan. Remarks on k-level algorithms in the plane. Manuscript, 1999.

[13] B. Chazelle. On the convex layers of a planar set. *IEEE Trans. Inform. Theory*, IT-31:509–517, 1985.

[14] B. Chazelle and H. Edelsbrunner. An improved algorithm for constructing kth-order Voronoi diagrams. *IEEE Trans. Comput.*, C-36:1349–1354, 1987.

[15] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. of the IEEE*, 80:1412–1434, 1992.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

[17] O. Devillers and M. J. Katz. Optimal line bipartitions of point sets. *Int. J. Comput. Geom. Appl.*, 1:39–51, 1999.

[18] D. Dobkin, D. Eppstein, and D. P. Mitchell. Computing the discrepancy with applications to supersampling patterns. *ACM Trans. on Graphics*, 15:354–376, 1996.

[19] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoret. Comput. Sci.*, 27:241–253, 1983.

[20] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

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

[22] J. Friedman, J. Hershberger, and J. Snoeyink. Efficiently planning compliant motion in the plane. *SIAM J. Comput.*, 25:562–599, 1996.

[23] S. Ghali and A. J. Stewart. Maintenance of the set of segments visible from a moving viewpoint in two dimensions. In *Proc. 12th ACM Sympos. Comput. Geom.*, pages V3–V4, 1996.

[24] S. Har-Peled. Taking a walk in a planar arrangement. In *Proc. 40th IEEE Sympos. Found. Comput. Sci.*, pages 100–110, 1999.

[25] J. Hershberger and S. Suri. Finding tailored partitions. *J. Algorithms*, 12:431–463, 1991.

[26] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992.

[27] J. Hershberger and S. Suri. Off-line maintenance of planar configurations. *J. Algorithms*, 21:453–475, 1996.

[28] R. Janardan. On maintaining the width and diameter of a planar point-set online. *Int. J. Comput. Geom. Appl.*, 3:331–344, 1993.

[29] S. Kapoor. Dynamic maintenance of 2-d convex hulls and order decomposable problems. Manuscript, 1998.

[30] J. Matoušek. On geometric optimization with few violated constraints. *Discrete Comput. Geom.*, 14:365–384, 1995.

[31] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30:852–865, 1983.

[32] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Heidelberg, 1984.

[33] K. Mulmuley. Randomized multidimensional search trees: lazy balancing and dynamic shuffling. In *Proc. 32nd IEEE Sympos. Found. Comput. Sci.*, pages 180–196, 1991.

[34] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, N.J., 1994.

[35] M. H. Overmars. *The Design of Dynamic Data Structures.* Lect. Notes in Comput. Sci., vol. 156, Springer-Verlag, 1983.

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

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

[38] E. Ramos. On range reporting, ray shooting, and k-level construction. In *Proc. 15th ACM Sympos. Comput. Geom.*, pages 390–399, 1999.

[39] G. Rote, C. Schwarz, and J. Snoeyink. Maintaining the approximate width of a set of points in the plane. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 258–263, 1993.

[40] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd IEEE Sympos. Found. Comput. Sci.*, pages 197–206, 1991.