Dynamic Planar Convex Hull with Optimal Query Time and $O(\log n \cdot \log\log n)$ Update Time

Gerth Stølting Brodal* and Riko Jacob*

BRICS**, Department of Computer Science, University of Aarhus
{gerth,rjacob}@brics.dk

Abstract. The dynamic maintenance of the convex hull of a set of points in the plane is one of the most important problems in computational geometry. We present a data structure supporting point insertions in amortized $O(\log n \cdot \log\log\log n)$ time, point deletions in amortized $O(\log n \cdot \log\log n)$ time, and various queries about the convex hull in optimal $O(\log n)$ worst-case time. The data structure requires $O(n)$ space. Applications of the new dynamic convex hull data structure are improved deterministic algorithms for the k-level problem and the red–blue segment intersection problem where all red and all blue segments are connected.

1 Introduction

The problem of maintaining the convex hull of a set of points in the plane under the insertion and deletion of points is one of the foremost important problems in computational geometry [6, 10]. A dynamic data structure for maintaining the convex hull of a point set has numerous applications, e.g. in algorithms solving the k-level problem [7] and the red–blue segment intersection problem where all red and all blue segments are connected [1]. For further applications see [4].

Overmars and van Leeuwen in 1981 gave a solution for the fully dynamic convex hull problem supporting point insertions and deletions in $O(\log^2 n)$ time, where n is the maximum number of points in the set [12]. The data structure of Overmars and van Leeuwen stores the convex hull in a search tree and typical queries on the convex hull are supported in $O(\log n)$ time. Preparata and Vitter gave a simpler approach achieving the same bounds as Overmars and van Leeuwen in [14]. Until recently there was made no progress on improving the update bounds for the general case. First in 1999, Chan presented a data structure that achieves amortized $O(\log^{1+\varepsilon} n)$ update time, where $\varepsilon > 0$ is any arbitrary constant, and $O(\log n)$ query time for various types of queries, e.g. membership and tangent-finding [4].

For special cases better update bounds are known. For the semi-dynamic case where only insertions are allowed, it is easy to achieve $O(\log n)$ insertion

* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

** Basic Research in Computer Science, Centre of the Danish National Research Foundation.

time [13]. For the other semi-dynamic case where only deletions are allowed after preprocessing, Hershberger and Suri achieved $O(n \log n)$ preprocessing time and amortized $O(\log n)$ deletion time [9]. For the off-line case where the sequence of updates is given in advance, a data structure using $O(n \log n)$ time for processing a sequence of n updates was given in [10]. The case where the sequence of updates is random was considered in [11, 15], where it was shown how to achieve expected $O(\log n)$ update time.

In this paper, we first give a new data structure for the semi-dynamic problem where only deletions are allowed after preprocessing, by extending the construction of Hershberger and Suri [9]. Provided that the initial point set is given lexicographically sorted, we achieve amortized $O(n)$ preprocessing time, and amortized $O(\log n \cdot \log\log n)$ deletion time. The data structure requires $O(n)$ space. Our main result for the fully dynamic case is a transformation strategy that combines a fully dynamic data structure with a semi-dynamic data structure for the deletions only case, and generates a new fully dynamic data structure. The construction is based on the construction of Chan [4] combined with several new ideas. Let $U(n)$ and $D(n)$ be two nondecreasing positive functions, where $U(n) \geq \log n$ and $D(n) \geq \log n$. If there exists a fully dynamic data structure with amortized $O(U(n))$ update time and worst-case $O(\log n)$ query time, and a semi-dynamic data structure with $O(n)$ preprocessing time and amortized $O(D(n))$ deletion time, then the transformation yields a data structure with amortized $O(U(\log^4 n) \cdot \log n / \log\log n)$ insertion time, amortized $O(D(n))$ deletion time, and worst-case $O(\log n)$ query time. The queries that can be supported are: find the extreme point on the convex hull in a given direction; report whether a given line intersects the convex hull; report if a given point is contained in the interior of the convex hull; find the two points adjacent to a point on the convex hull; and given an exterior point find the two tangent points on the convex hull from the point.

Combining our semi-dynamic data structure with the fully dynamic data structure of Overmars and van Leeuwen [12], we immediately get amortized $O(\log n \cdot \log\log n)$ deletion and insertion time. By bootstrapping, we can use the resulting data structure as the fully dynamic data structure in the construction and the insertion time reduces to amortized $O(\log n \cdot \log\log\log n)$ time, while the deletion time remains amortized $O(\log n \cdot \log\log n)$.

We note that a semi-dynamic data structure with $O(n)$ preprocessing time and $O(\log n)$ deletion time, would for any constant k imply a fully dynamic data structure with amortized $O(\log n \cdot \log^{(k)} n)$ insertion time and amortized $O(\log n)$ deletion and worst-case $O(\log n)$ query time, by $k - 1$ applications of our transformation strategy and using the data structure of Overmars and van Leeuwen as the initial fully dynamic data structure.[1]

The paper is organized as follows. Section 2 contains a description of the semi-dynamic data structure for the deletions only case, and Sect. 3 and 4 contain the results for the fully dynamic case. Section 5 gives applications of the fully dynamic data structure.

[1] We let $\log^{(1)} n = \log n$, and $\log^{(i+1)} n = \log\log^{(i)} n$ for $i \geq 1$.

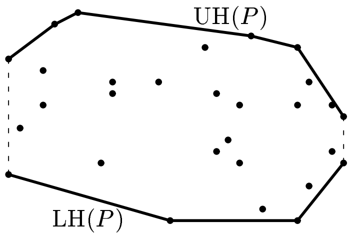

Fig. 1. The convex hull $\mathrm{CH}(P)$ of a set of points P can be partitioned into an upper hull $\mathrm{UH}(P)$, a lower hull $\mathrm{LH}(P)$, and possibly two vertical lines.

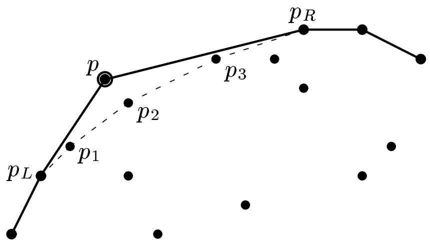

Fig. 2. Deletion of the point p from the upper hull implies that p is replaced by the sequence of points p_1, p_2, p_3.

Notation

Given a set of points P in the Euclidean plane, we let $\mathrm{CH}(P) \subseteq P$ denote the set of points on the convex hull of P, and $\mathrm{UH}(P)$ and $\mathrm{LH}(P)$ denote respectively the upper and lower hull of $\mathrm{CH}(P)$. Figure 1 shows the upper and lower hulls of a set of points. In the following we restrict our attention to the upper hulls of the sets of points, and assume for the sake of simplicity that points are in general position, i.e. all points have distinct x-coordinates and no three points are on a line. The results for the convex hull problems immediately follow from the results on the upper hulls.

2 Semi-Dynamic Data Structure

In this section we give a data structure for the semi-dynamic problem with amortized $O(n)$ preprocessing time, and which supports point deletions in amortized $O(\log n \cdot \log\log n)$ time. To achieve linear preprocessing time we require points to be given lexicographically sorted. The data structure supports the operations:

Build Given a lexicographically sorted set P containing n points, builds a data structure for P and returns the points on $\mathrm{UH}(P)$ from left-to-right.

Delete Deletes a point p from P, and returns the changes to $\mathrm{UH}(P)$, i.e. if p was contained in $\mathrm{UH}(P)$ before the deletion then the sequence of new points on $\mathrm{UH}(P)$ are returned from left-to-right (see Fig. 2).

Our result for the semi-dynamic problem is the following.

Theorem 1. *There exists a data structure supporting* Build *in amortized $O(n)$ time and* Delete *in amortized $O(\log n \cdot \log\log n)$ time. The data structure requires $O(n)$ space.*

In the following we without loss of generality assume $n \geq 4$, such that $\log\log n \geq 1$. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the initial set of points, where $p_i < p_{i+1}$ for $1 \leq i < n$, and let $B = \lceil \log n \rceil$ and $N = \lceil n/B \rceil$. We partition P into a sequence of *blocks* $P_1, \ldots, P_N$, each of size B except for P_N, where $P_i =$

$\{p_{1+(i-1)B}, p_{2+(i-1)B}, \ldots, p_{\min(iB,n)}\}$, for $1 \leq i \leq N$. After a sequence of Delete operations we let $\bar{P} \subseteq P$ denote the set of points which have not been deleted so far, and similarly we for $P_1, \ldots, P_N$ define $\bar{P}_1, \ldots, \bar{P}_N$.

For each block P_i, the points $\bar{P}_i$ are stored in sorted order in a linked list, $\mathrm{UH}(\bar{P}_i)$ is stored as a perfect balanced binary tree, and furthermore the points from left-to-right on $\mathrm{UH}(\bar{P}_i)$ are kept in a doubly linked list.

Since $|\bar{P}_i| \leq B$, the upper hull $\mathrm{UH}(\bar{P}_i)$ can be constructed by a linear sweep of $\mathrm{UH}(\bar{P}_i)$ in $O(B)$ time, see e.g. [2, Sect. 1.1]. The balanced tree and the double linked list storing $\mathrm{UH}(\bar{P}_i)$ can therefore be recomputed in $O(B)$ time, when a point is deleted from block P_i.

The blocks $P_1, \ldots, P_N$ are stored from left-to-right at the leaves of a perfect balanced binary tree T with height $\lceil \log N \rceil$. For each node v in T, we let T_v denote the subtree of T rooted at v, and let $\bar{P}_v$ denote the union of the sets $\bar{P}_i$ stored at the leaves of T_v. It is easy to see that $\mathrm{UH}(\bar{P}_v) \cap \mathrm{UH}(\bar{P}_i)$ is either empty or a consecutive subsequence of $\mathrm{UH}(\bar{P}_i)$. At each node v of T we store $\mathrm{UH}(\bar{P}_v)$ as a doubly linked list L_v of *block-records*, such that for each block P_i contributing to $\mathrm{UH}(\bar{P}_v)$, i.e. $\mathrm{UH}(\bar{P}_v) \cap \mathrm{UH}(\bar{P}_i) \neq \emptyset$, we have a block-record $r_{v,i}$. For each block-record $r_{v,i}$ we store pointers to the leftmost and rightmost points in $\mathrm{UH}(\bar{P}_i)$ which are also in $\mathrm{UH}(\bar{P}_v)$. For a block P_i, let $v_0, v_1, \ldots, v_k$ be the prefix of the nodes in T on the path from the leaf v_0 storing $\bar{P}_i$ to the root, where $\mathrm{UH}(\bar{P}_i) \cap \mathrm{UH}(\bar{P}_{v_j}) \neq \emptyset$, i.e. $r_{v_j,i} \in L_{v_j}$. For $0 \leq j < k$, we with $r_{v_j,i}$ store an *up-pointer* to $r_{v_{j+1},i}$. This representation allows us to efficiently navigate $\mathrm{UH}(\bar{P}_v)$ in both directions from point-to-point and block-to-block in constant time. Note that $\mathrm{UH}(\bar{P})$ is stored at the root of T.

Since each block requires $O(B)$ space the total space for the N blocks is $O(N \cdot B)$. Since P is partitioned into N blocks, the total space for the lists of block-records at each level of T is at most $O(N)$. The total space required is $O(N \cdot B + N \cdot \log N) = O(n)$.

We now turn to the implementation of the operations. For Build the input set P is first partitioned into N blocks, and for each block the upper hull is computed by a sweep line algorithm in $O(B)$ time and each block structure is initialized in $O(B)$ time. The construction time for all blocks is $O(n + N \cdot B) = O(n)$. The tree T is then processed bottom-up level by level. Assume a node v has two children w_1 and w_2, and L_{w_1} and L_{w_2} have already been computed (for a leaf ℓ, we define L_ℓ to only contain one block-record with pointers to the first and last node of $\mathrm{UH}(\bar{P}_\ell)$). First we let L_v be the concatenation of L_{w_1} and L_{w_2}. The resulting list of block-records represents a sequence of points forming a convex curve except for possible at one point, namely the last point from $\mathrm{CH}(\bar{P}_{w_1})$ or the first point from $\mathrm{CH}(\bar{P}_{w_2})$, i.e. there is a pointer to p in one of the block records in L_v.

To fix this problem we apply the standard method used in convex hull construction algorithms: while we have a non-convex point p in the list of points, i.e. p together with its predecessor and successor point in the list form a left-turn, we remove p from the list. Removing p is done as follows: if p is in block P_i, and p is the only point from $\mathrm{UH}(\bar{P}_i)$ in the list, i.e. both pointers in $r_{v,i}$ point to p, we remove $r_{v,i}$ from L_v. Otherwise we replace the pointer to p in $r_{v,i}$ by a pointer to the next point in $\mathrm{UH}(\bar{P}_i)$ in the direction of the point given by the other pointer in $r_{v,i}$, where we utilize that the points in $\mathrm{UH}(\bar{P}_i)$ are kept in a double linked list. We can at most remove a point once in the bottom-up preprocessing of T, and the time for preprocessing one level of T is $O(n)$ plus the time used to eliminate left turns. The total time for constructing all L_v lists becomes $O(n + N \cdot \log N) = O(n)$. It follows that Build takes $O(n)$ time.

Before describing the Delete operation, we observe that only upper hulls actually containing p need to be updated (see Fig. 2). To perform Delete first in $O(\log n)$ time make a binary search locating the block P_i containing p, assuming that P was given as an array of points or that we keep P in a balanced search tree. In $O(B)$ time we check if $p \in \mathrm{UH}(\bar{P}_i)$. If $p \notin \mathrm{UH}(\bar{P}_i)$ then no upper hull needs to be updated and it is sufficient to remove p from the list of points in $\bar{P}_i$ in $O(B)$ time. Otherwise $p \in \mathrm{UH}(\bar{P}_i)$, and let $\overleftarrow{p}$ and $\overrightarrow{p}$ be the predecessor and successor of p in $\mathrm{UH}(\bar{P}_i)$ (if present), and rebuild in $O(B)$ time the data structure for block P_i after p has been deleted from the list of points in $\bar{P}_i$. What remains is to update all the upper hulls which contained p. If $p \in \mathrm{UH}(\bar{P}_v)$ for a node v then $r_{v,i} \in L_v$. But then $r_{v,i}$ is reachable from $\bar{P}_i$ using the stored up-pointers.

The reconstruction of upper hulls is done bottom-up in T. Consider a node v and the effect of deleting p from $\mathrm{UH}(\bar{P}_v)$. Let p_L and p_R be the two points in $\bar{P}_i$ that $r_{v,i}$ has pointers to, where $p_L \leq p_R$. If $p < p_L$ or $p > p_R$ then $p \notin \mathrm{UH}(\bar{P}_v)$ and we are done. If $p_L < p < p_R$ then the changes to $\mathrm{UH}(\bar{P}_v)$ can only be between p_L and p_R, i.e. the updates are done locally in block P_i and no changes are required for L_v. The complicated case is when $p = p_L$ or $p = p_R$. First we need to delete p from the upper hull stored at v. If $p_L = p_R$ then p was the only point from block P_i, and we delete $r_{v,i}$ from L_v. Otherwise we have two cases: if $p = p_L$ then we replace the pointer to p in $r_{v,i}$ by a pointer to $\overrightarrow{p}$, and if $p = p_R$ then we replace the pointer to p in $r_{v,i}$ by a pointer to $\overleftarrow{p}$.

After having deleted p from $\mathrm{UH}(\bar{P}_v)$, we must insert new points onto $\mathrm{UH}(\bar{P}_v)$, as illustrated by Fig. 2. If p was not an endpoint of the *bridge* connecting two points on the two upper hulls stored at the children of v (see Fig. 3), then the changes to $\mathrm{UH}(\bar{P}_v)$ are exactly the changes to $\mathrm{UH}(\bar{P}_w)$, where w is the child of v where $p \in \mathrm{UH}(\bar{P}_w)$ before the deletion. It follows that it is sufficient to create and update existing block-records in L_v with exactly the same pointers to points in blocks as done for L_w.

The final case is when p is an endpoint of the bridge connecting the upper hulls stored at the children of v, ad illustrated in Fig. 3. Assuming the new bridge has been found, then updating L_v with respect to the new points on $\mathrm{UH}(\bar{P}_v)$ consists of inserting a subsequence of the points from each of the upper hulls stored at the children of v, by creating a sequence of new block-records in L_v with the same information as stored at the two children of v and changing at most four pointers in the block-records in L_v corresponding to the ends of the subsequences copied.

To find the new bridge we apply a standard bridge searching algorithm, with minor modifications. The standard bridge searching procedure keeps for

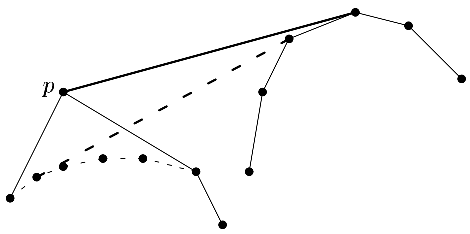

Fig. 3. The bridge between two horizontally separated upper hulls. The dashed lines show the changes to the left upper hull and the new bridge when deleting point p.

the upper hulls two candidate intervals for each of endpoints of the bridge, and performs a "simulations binary search" on both hulls, always halving at least one of the intervals. See e.g. [13, Lemma 3.1] for further details. We replace the binary search by a linear block search on each of the two upper hulls. The linear block search at the left child proceeds left-to-right, always trying to advance one block, whereas the linear block search at the right child proceeds right-to-left. Whenever a search is advanced to the next block a block-record is added to L_v in $O(1)$ time.

The search process for each upper hull first tries to advance a complete block at a time, using the information stored at the block-records at the children of v to always pick the last point in the next block P_i contributing to $\mathrm{UH}(\bar{P}_i)$. After having localized the block P_i containing one endpoint of the new bridge the search then proceeds in a binary fashion using the search tree storing $\mathrm{UH}(\bar{P}_i)$. The total time for finding a bridge becomes linear in the number of block-records created plus $O(\log B)$. The output of Delete can be generated immediately from the changes to $L_{\mathrm{root}(T)}$.

The total time for a deletion becomes $O(B + x + \log N \cdot \log B)$, where x is the total number of new block-records created. Since a deletion at most removes one block-record from each level of T, it follows that D deletions at most delete $D \cdot \log N$ block-records. Since there can at most be $O(N \cdot \log N)$ block-records, it follows that the total time for D deletions is at most $O(D \cdot B + N \cdot \log N + D \cdot \log N + D \cdot \log N \cdot \log B) = O(n + D \cdot \log n \cdot \log\log n)$. Since the $O(n)$ term can be charged to Build, it follows that Build takes amortized $O(n)$ time and each Delete operation amortized $O(\log n \log\log n)$ time.

3 Fully Dynamic Data Structure

For this part of the paper we change the point of view of the exposition to the dual problem and consider upper envelopes instead of upper hulls. This duality, as explained e.g. in [2, p. 167], maps points to lines and vice versa in a way, that preserves above/on/below relations. In this setting a set of points becomes a collection of lines L, and the upper hull transforms to the upper envelope of these lines, i.e. the collection of line segments such that points on a segment are not below any other line. An extreme point query, i.e. given a slope q find the point of the upper hull that has a tangent of slope q, turns into a vertical line query, i.e. given a vertical line with x-coordinate q, report the segment of the upper envelope crossing this line. Note that this is really only a change in point of view. There is no need to perform a computation to go from the original setting to the dual and back.

We apply a standard dynamization technique that divides the current points into sets and keeps one deletion only data structure per set. Additionally there is a more explicit representation of the current upper envelope, namely an interval tree, that allows fast queries without requiring too much work for updates. Inside the interval tree have at each internal node a fully dynamic upper envelope data structure, a so called *secondary structure*. The running time improvement relies on a polylogarithmic bound on the size of the secondary structures.

The description so far fits as well to the data structure proposed in Chan [4]. Compared to that data structure we apply improved deletion only data structures. We also do some explicit grouping of the subenvelopes stemming from the dynamization, such that the number of secondary structure storing segments from one subenvelope is reduced.

The remaining of this section is devoted to proving the following theorem.

Theorem 2. *Let $U(n)$ and $D(n)$ be two nondecreasing positive functions, where $U(n) \geq \log n$ and $D(n) \geq \log n$. Assume there exists a data structure for the dynamic upper envelope problem supporting* Insert *and* Delete *in amortized $O(U(s))$ time, and* Vertical Line Query *in worst-case $O(\log s)$ time, where s is the total number of lines inserted. Assume further that there exists a data structure for semi-dynamic upper envelope problem supporting* Build *on a lexicographically sorted list of n points in amortized $O(n)$ time and* Delete *in amortized $O(D(n))$ time, where n is the number of lines in the structure.*

Then there exists a data structure for the dynamic upper envelope problem supporting Insert *in amortized $O(\log n \cdot U(\log^4 n)/\log\log n)$ time and* Delete *in amortized $O(D(n) + \log n \cdot U(\log^4 n)/\log\log n)$ time, and* Vertical Line Query *in worst-case $O(\log n)$ time, where n is the total number of lines inserted.*

Applying this theorem to the data structure of Overmars and van Leeuwen with $U(s) = \log^2 s$ and the result from Sect. 2 with $D(n) = \log n \cdot \log\log n$, we get Insert in $O(\log n \cdot \log^2(\log^4 n)/\log\log n) = O(\log n \cdot \log\log n)$, and Delete in $O(\log n \cdot \log\log n)$. Applying the theorem again on this new data structure improves Insert to $O(\log n \cdot \log\log\log n)$. The performance of the deletion only data structure is the bottleneck, that renders further applications of the theorem useless.

For the purpose of describing our data structure, we separate it into several layers. We first describe the layers in a top down fashion, we start with a data structure that solves the fully dynamic upper envelope problem using some auxiliary data structures. For the analysis we proceed in a bottom up fashion, i.e. we always analyze the auxiliary data structure first. This avoids any forward references.

3.1 The interfaces

Fully dynamic upper envelopes.

Insert Insert a line, given by the parameters a and b in the representation $y = ax + b$. Return a pointer to a new line data structure.

Delete Given a pointer to a line data structure, delete that structure and the line it represents.

Query Given a value v, report the highest intersection of a line with the vertical line given by $x = v$.

Query structure Q. This data structure combines several independent upper envelopes. It is asserted (and could be easily checked), that the list of line segments in fact form envelopes. It is also asserted, that a line is present in at most one set and has therefore at most one segment.

There is an active set of segments that is considered for queries. For all lists of segments it is asserted, that the segments from this list form an upper envelope. A segment is given by a line and an interval on the x-axis.

Init set with active envelope Given a lexicographically sorted list L of lines and a list $K \subseteq L$ of segments. Initialize a set data structure that can hold upper envelopes stemming from lines in L and insert K into the active set. It is asserted that K forms a complete upper envelope. Return a pointer to a new data structure representing the set.

Delete set Delete a set given by a pointer, removing all segments from the active set.

Replace inside an envelope Given a pointer to a set, pointers to (up to) three segments $\ell_\alpha, \ell, \ell_\omega$, and a lexicographically sorted list of segments K with $K = \ell'_\alpha, \ldots, \ell'_\omega$. Here ℓ_ω and ℓ'_ω are the same segment with a changed left boundary, and ℓ_α and l'_α differ only in the right boundary. It is explicitly allowed that ℓ_α and ℓ_ω are void, with the meaning that ℓ is unbounded to the left and respectively to the right. Replace the three segments by K in the active set. It is asserted that the active set forms an upper envelope after the replacement.

Query Given a value v, report the highest intersection of an active segment with the vertical line given by $x = v$.

Subenvelope structure $\mathcal{T}$. This structure allows queries on a generalization of segments, namely subenvelopes. A subenvelope is an lexicographically sorted list of line segments where neighbors have precisely one point in common. We will maintain a small upper bound on the size of an subenvelope. Again it is asserted that the segments in fact are segments from upper envelopes.

Insert Given a list L of segments, insert the subenvelope formed by L. Return a pointer to the newly created data structure of the subenvelope.

Delete Given a pointer to a subenvelope, delete that subenvelope. Return the segments of the subenvelope.

Query Given a value v, report the highest intersection of an inserted subenvelope with the vertical line given by $x = v$.

3.2 Dynamization

Throughout the following we assume that we know the value of n, the total number of insert operations, in advance. Standard doubling techniques justify this assumption.

Starting from the monotonic data structure presented in Sect. 2, we apply a general dynamization technique for decomposable search problems attributed to Bentley and Saxe [3]. The idea is that we divide the set of lines L into a partition $\mathcal{C}$ based on the order the lines are inserted. More precisely every set $C \in \mathcal{C}$ has a rank. If there are d sets of the same rank i, we merge them into one new set of rank $i + 1$. Sets of rank 0 have size 1. We choose the parameter $d = \lceil \log n \rceil$, leading to at most $r = O(\log_d n) = O(\log n / \log \log n)$ different ranks. This is also an upper bound on the number of times a specific line can participate in the merge of d sets. Furthermore the number $e = |\mathcal{C}|$ of sets is bounded by $e = O(rd) = O(\log^2 n / \log \log n)$. Every set has a deletion only structure and a set in the query structure attached.

The merge operation first deletes all the involved sets from the Query structure Q. Then it orders the lines (dual) according to their slopes, which corresponds to sorting the corresponding (primal) points according to their x coordinates. Here we exploit that the sets we are merging are already sorted in that order. We use a heap of size d to iteratively find the remaining line with smallest slope. Then we invoke the Build operation of the deletion only data structure, and use the reported upper envelope in an Init set operation of the query structure Q. We attach the returned pointer to the new set.

For an Insert(ℓ) we create a new record for ℓ that keeps the coordinates (slope and offset) and also a pointer p_ℓ to the set of $\mathcal{C}$ that currently contains ℓ. Then we create a new set of size 1 and rank 0 and perform necessary merge operations. During the merge operations we update the pointers p_ℓ for all lines we move.

If we want to delete a line ℓ we look up the set $C \in \mathcal{C}$ that contains ℓ, and then we invoke the Delete(ℓ) operation of the deletion only data structure from Sect. 2. This returns a list of new segments, which implicitly gives also the two neighbors of ℓ. With this information we call the Replace inside set operation of Q.

3.3 Grouping

Now we implement the query structure using only a Subenvelope structure. We choose a block size parameter $b = \lceil \log n / \log \log n \rceil$.

The Init set with active envelope operation first deletes all pointers to blocks on the lines of the set. Then it groups the segments of K equally into as few as possible blocks of size at most b. It inserts the resulting subenvelopes and stores the subenvelope pointer at every line.

The Delete set operation walks along the set, deleting blocks pointed to by the lines and deleting the pointers as well.

The Replace inside an envelope operation looks up the blocks where the three lines are stored. Then it deletes the pointed to subenvelopes, building a list L of segments that got deleted. In this list we replace $\ell_\alpha, \ell, \ell_\omega$ by K. Then we group L optimally into blocks of size b. We insert the blocks and update the block pointers.

The query gets directly handed over. This is correct, as all active segments are in some block.

3.4 The interval tree $\mathcal{T}$ for subenvelopes

We implement the subenvelope structure as an interval tree. The interval tree $\mathcal{T}$ is a rooted tree. We assume to know the number M of leaves of $\mathcal{T}$. We choose the degree parameter $B = \lceil \log n \rceil$. We keep $\mathcal{T}$ balanced by maintaining the invariants that the degree of a node is at most $2B - 1$ and at least 2 at the root and at least B for all the other internal nodes. All leaves have the same distance to the root. A leaf ℓ of $\mathcal{T}$ stores a (possibly unbounded) interval I_ℓ, its range. Every internal node v of $\mathcal{T}$ stores its range I_v, the interval that is the (disjoint) union of the ranges of its children. To deal with a non constant degree of a node we maintain a dictionary (balanced tree) of the endpoints of the ranges of its children. For an arbitrary interval I we say that the node u of $\mathcal{T}$ *corresponds* to I if the range of u contains the interval, i.e. $I \subseteq I_u$, and for none of the children v of u it is the case that I is contained in the range I_v of v. Note that there is always a unique node of $\mathcal{T}$ corresponding to an interval. We can find all the intervals containing a certain point p on the path from the root node to the leaf that contains p. We assert that the range of every leaf node contains at most one endpoint of the stored intervals.

We store subenvelopes at the node in $\mathcal{T}$ that corresponds to their interval, i.e. the extent along the x-axis. We store the segments of the subenvelope in the secondary structure at that node, i.e. as lines in a fully dynamic upper envelope structure.

The Insert operation creates a record that has a list of the lines forming the subenvelope, the interval, and a pointer to the node of $\mathcal{T}$. A pointer to this record is returned. It inserts the interval into $\mathcal{T}$ and finds the node u in $\mathcal{T}$ corresponding to the interval and inserts all the lines into the secondary structure S_u. It stores the returned identifiers in a list in the newly created record.

As we have the strong restriction that the range of a leaf should contain at most one endpoint of an interval stored in the tree, we might be forced to split nodes of $\mathcal{T}$ in a bottom up fashion. Assume that node u of $\mathcal{T}$ has too many children. Then we create a new right sibling v of u (creating a new root if u was the root) and move the right half of the children of u to v. We walk through the list of blocks being stored at u. For a block w we take I_w to decide if they should stay at u, get moved to v or moved up to the parent p of u and v. If necessary we delete all the lines of w from the secondary structure S_u of u. If the block moves to v we insert the lines into S_v. If it moves up to p, we keep the block w "on hold", in case that p also gets split. During this we update the pointers between the nodes of $\mathcal{T}$ and the records of blocks.

If a subenvelope has the interval $]-\infty,\infty[$, it gets stored at the root of $\mathcal{T}$, and it cannot cause any splits. We call such a subenvelope trivial. M accounts only for non-trivial subenvelopes.

For the Delete operation we remove all the lines from the secondary structure.

For a Query operation with value x, we determine the path p in $\mathcal{T}$ from the root to the leaf v of $\mathcal{T}$ whose range contains x. For all nodes u on p we perform an upper envelope query for x on the secondary structure S_u. We report the topmost of the answers.

This answer is correct, because the block of the topmost segment at x is stored in one of the parents of the leaf v that contains x.

3.5 Analysis

Bound on the number M of nontrivial subenvelope inserts. We have to bound the number of operations on blocks performed within the query structure Q.

At the init operation we give every line a fractional coin that allows it to participate as a fraction $2/b$ in a non-trivial insert operation, i.e. we need $b/2$ such coins to pay for a non-trivial insert. Then the init operation on a set of size m costs us $\lceil 2m/b \rceil$ non-trivial subenvelope insert operations. If the init operation gives rise to a nontrivial insert, it is paid for.

A replace operation is going to pay for 3 subenvelope deletions and 4 subenvelope insertions. If there are more blocks to be inserted, the blocks are definitely half full, and only 2 blocks on each end contain any lines that have already used their coins. The remaining block insertions can therefore be paid with coins.

Knowing that one line can only cause one replace operation and participate in r init operations, we get a total account of $M = O(n + n \cdot r/b) = O(n + n \cdot \log n/\log\log n \cdot \log\log n/\log n) = O(n)$.

Bound s on the size of secondary structures in $\mathcal{T}$. For every set in $\mathcal{C}$ we have at most B subenvelopes stored at a node v. With the bounds on the size of subenvelope and on $|\mathcal{C}|$ we get $s = O(B \cdot b \cdot e) = O(\log^4 n)$.

A query takes $O(\log M + Q(s) \cdot h) = O(\log n + \log\log n \cdot \log n/\log\log n) = O(\log n)$ time.

Work in the split operations. Every split operation creates at least one new node. We will account on that node for all the insertions and deletions that happened during this single split.

We charge the work of moving a block during a split operation entirely to the newly created node of $\mathcal{T}$. For this we define the level of a node u of $\mathcal{T}$ by stating that leaves have level 0, and that the parent of nodes on level i has level $i+1$. Now we observe that an interval stored at u has both endpoints at some leaf below u. Hence the condition of having at most one endpoint of an interval per leaf implies that we have at most $N_i = (2B)^i$ intervals stored at a node of level i. Now let u be a node on level i. Then u was created by a split operation performed on one of its siblings v. So we know that v is also on level i and the split operation involved at most N_i intervals. Additionally we know $e = O(rd) = O(\log^2 n / \log\log n)$ which means for large n we have $e < B^2$ and that any node in $\mathcal{T}$ stores at most $e \cdot B < B^3$ intervals.

Adding these costs level by level in the tree, we get that the total number of intervals moved because of split operations is bounded by $O((M/B)2B + (M/B^2)4B^2 + (M/B^3)B^3 + (M/B^4)B^3 + (M/B^5)B^3 + \cdots) = O(M) = O(n)$. We conclude that every subenvelope insertion causes in average constantly many moves of a subenvelope during split operations.

Running time of the update operations in $\mathcal{T}$. Given the previous paragraph, we conclude that an update operation of a nontrivial block in $\mathcal{T}$ takes amortized $O(\log M + b \cdot U(s))$ time for finding the correct node in $\mathcal{T}$ and to pay for the insertions and deletions of the segments, including during split operations.

Since $U(s) \geq \log s$, we have $b \cdot U(s) = \Omega(\log n / \log\log n \cdot \log\log n) = \Omega(\log n)$, so the amortized time of a non-trivial block insert operation becomes $O(b \cdot U(s))$.

For trivial blocks it takes amortized $O(U(s))$ time per segment. Note that even so the root node of $\mathcal{T}$ is special, the upper bound s on the number of segments stored there applies as well.

Running time of the Query structure / Fully dynamic structure. In the init operation of the query structure we account for $2/b$ nontrivial block insert operations for every line in the set. We already argued that this is sufficient to pay for the initial insert operation of that line (i.e. when the line appears on the upper envelope of the set we just initialized). Accounting also for the possibility of being inserted as part of a trivial block, we get a per line amortized time of $O(U(s) + b \cdot U(s)/b) = O(U(s))$.

Knowing that every line gets initialized at the worst r times, we get an amortized insert time for the fully dynamic data structure of $O(r \cdot U(s)) = O(\log n / \log\log n \cdot U(s))$ as claimed in Theorem 2.

For the delete operation of the fully dynamic data structure we have to account for the delete operation in the deletion only structure, and for the replace operation in the query structure. As already argued, the replace operation has to account for a constant number of block update operations, yielding an amortized time of $O(D(n) + b \cdot U(s)) = O(D(n) + \log n \cdot U(s) / \log\log n)$, the bound claimed in Theorem 2.

4 Other Queries

With the so far explained data structure for vertical line queries we can efficiently answer a whole class of other queries on the upper envelopes. Assume the query satisfies a so called locality property, that is for a vertical line q we can determine on which side of q the answer lies by solely examine the highest line intersecting q. Then we can use binary search to give an answer with $O(\log n)$ vertical line queries, that is in $O(\log^2 n)$ time. But this overhead is not always necessary. In the next section we will give an important example where the already explained data structure can be used to achieve a $O(\log n)$ query time for a more involved query.

4.1 Arbitrary line queries

The query we address is in the primal setting: given a point p in the plane report the two tangent lines through p touching the convex hull or state that the point is inside the convex hull. This corresponds in the dual to: given an arbitrary line, give the two intersection points of the line with the upper envelope, or "no" if no such intersection exists. The exposition here adopts the dual point of view. The important observation is, that our data structure has the same properties as the data structure in [4], the argument given there applies here as well. We only sketch the query algorithm in our setting.

We use the following fact about arbitrary line queries to navigate in the interval tree of our data structure.

Lemma 1. *Let a and b be to walls and $E' \subseteq E$ a subset of lines s.t. the upper envelope of E' at a and b coincides with the upper envelope of E. Assume that an arbitrary line query for a line ℓ on E' results in the right intersection point t. If t lies between a and b then also the right intersection T of ℓ with E lies between a and b.*

Let ℓ be the line query. The query algorithm starts at the root node of the interval tree. It performs the right intersection query on the secondary structure of the current node, updating the current answer. Then it descends to the child corresponding to the interval the current answer lies in. When it reaches a leaf, the current answer reflects the right intersection of ℓ with the upper envelope of all lines.

Given that our secondary structures support line queries in $O(\log s)$ time, we have an overall query time of $O((\log B + \log s)h) = O(\log B \log n / \log B) = O(\log n)$.

5 Applications

As a prominent example we consider the k-level of n lines, which is dually related to the k-set question on n points. For this problem Edelsbrunner and Welzl [7] gave an algorithm using the data structure of Overmars and van Leeuwen that constructs the k-level in $O(n \cdot \log n + m \cdot \log^2 n)$ time, where m is the size of the k-level. Applying Chan's data structure this improves to $O(n \cdot \log n + m \cdot \log^{1+\varepsilon} n)$ time, and using our data structure this yields an improved $O(n \cdot \log n + m \cdot \log n \cdot \log\log n)$ time bound. A randomized algorithm using expected $O(\lambda_{t+2}(n+m) \cdot$

$\log n)$ time has been given by Har-Peled [8], where $\lambda_{t+2}(n+m)$ is the maximum length of a Davenport-Schinzel sequence of order $t+2$ having $n+m$ symbols.

Basch, Guibas and Ramkumar [1] considered a 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 $R \times B$. Chan [4] reported an improvement from $O((n+m) \cdot \log^3 n)$ time using Overmars and van Leeuwen's data structure to $O((n+m) \cdot \log^{2+\varepsilon} n)$ using Chan's data structure. We get a further improvement to $O((n+m) \cdot \log^2 n \cdot \log\log n)$.

References

1. J. Basch, L. J. Guibas, and G. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proc. 4th European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 302–319. Springer Verlag, Berlin, 1996.
2. M. de Berg, M. van K., M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer-Verlag, Berlin, 1997. Algorithms and applications.
3. J. Bentley and J. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1:301–358, 1980.
4. T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proc. 40th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 92–99, 1999.
5. B. Chazelle. On the convex layers of a planar set. *IEEE Trans. Inform. Theory*, IT-31:509–517, 1985.
6. Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE, Special Issue on Computational Geometry*, 80(9):1412–1434, 1992.
7. H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, Vol. 15, No. 1, 1986.
8. S. Har-Peled. Taking a walk in a planar arrangement. In *Proc. 40th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 100–110, 1999.
9. J. Hershberger and S. Suri. Applications of a semidynamic convex hull algorithm. *BIT*, 32:249–267, 1992.
10. J. Hershberger and S. Suri. Off-line maintenance of planar configurations. *Journal of Algorithms*, 21:453–475, 1996.
11. K. Mulmuley. Randomized multidimensional search trees: lazy balancing and dynamic shuffling. In *Proc. 32nd Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 180–196, 1991.
12. M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.
13. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer Verlag, Berlin, 1985.
14. F. P. Preparata and J. S. Vitter. A simplified technique for hidden-line elimination in terrains. *International Journal of Computational Geometry & Applications*, 3(2):167–181, 1993.
15. O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 197–206, 1991.