Dynamic Planar Convex Hull

Gerth Stølting Brodal[*,†] Riko Jacob[*]
BRICS[‡], Department of Computer Science, University of Aarhus,
{gerth,rjacob}@brics.dk.

Abstract

In this paper we determine the computational complexity of the dynamic convex hull problem in the planar case. We present a data structure that maintains a finite set of n points in the plane under insertion and deletion of points in amortized $O(\log n)$ time per operation. The space usage of the data structure is $O(n)$. The data structure supports extreme point queries in a given direction, tangent queries through a given point, and queries for the neighboring points on the convex hull in $O(\log n)$ time. The extreme point queries can be used to decide whether or not a given line intersects the convex hull, and the tangent queries to determine whether a given point is inside the convex hull. We give a lower bound on the amortized asymptotic time complexity that matches the performance of this data structure.

Keywords: Planar computational geometry, dynamic convex hull, lower bound, data structure, search trees, finger searches

1 Introduction

The convex hull of a set of points in the plane is one of the most prominent objects in computational geometry. Computing the convex hull of a static set of n point set can be done in optimal $O(n \log n)$ time, e.g., with Graham's scan [9] or Andrew's vertical sweep line variant [1] of it. Optimal output sensitive algorithms are due to Kirkpatrick and Seidel [16] and also to Chan [5], they achieve $O(n \log h)$ running time, where h denotes the number of vertices on the convex hull.

In the dynamic setting we consider a data structure: Given a set S of points in the plane that is changed by insertions and deletions, maintain the convex hull of S. Observing that a single insertion or deletion can change the convex hull of S by $|S| - 2$ points, reporting the changes to the convex hull is in many applications not desirable. Instead of reporting the changes one maintains a data structure that allows queries for points on the convex hull. Typical examples are the extreme point in a given direction, the tangent(s) on the hull that passes through a given point, whether or not a point is inside the convex hull, the edges of the convex hull intersected by a given line, the common tangent(s) between two different convex hulls. Furthermore we might want to report (some consecutive subsequence of) the points on the convex hull or count their cardinality. Overmars and van Leeuwen [17] provide a solution that uses $O(\log^2 n)$ time per update operation and maintains a leaf-linked balanced search tree of the vertices on the convex hull in clockwise order. Such a tree allows all of the above mentioned queries in $O(\log n)$ time. Semidynamic variants of the problems have been considered. There updates are restricted to be either insertions only or deletions only. For the insertion-only problem Preparata [18] gives an $O(\log n)$ worst-case time algorithm that maintains the convex hull in a search tree. The deletion-only problem is solved by Hershberger and Suri in [12], where initializing the data structure (build) with n points and up to n deletions are accomplished in overall $O(n \log n)$ time. Hershberger and Suri in [13] consider the off-line variant of the problem, where both insertions and deletions are allowed, but the times (and by this the order) of all insertions and deletions are known a priori. The algorithm processes a list of insertions and deletions in $O(n \log n)$ time and space, and produces a data structure that can answer extreme point queries for any time using $O(\log n)$ time. Their data structure does not provide an explicit representation of the convex hull in terms of a search tree with the points on the convex hull. The space usage can be reduced to $O(n)$ if the queries are also part of the off-line information.

Chan [7] gives a construction for the fully dynamic problem with $O(\log^{1+\varepsilon} n)$ amortized time for updates (for any constant $\varepsilon > 0$), and $O(\log n)$ time for extreme point queries. His construction does not maintain an explicit representation of the convex hull. It is based on a general dynamization technique attributed to Bentley and

[*]Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).
[†]Supported by the Carlsberg Foundation (contract ANS-0257/20).
[‡]Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation

Saxe [3]. Using the semidynamic deletions only data structure of Hershberger and Suri [12], and a constant number of bootstrapping steps, the construction achieves update times of $O(\log^{1+\varepsilon} n)$ for any constant $\varepsilon > 0$. The construction uses an augmented variant of an interval tree to store the convex hulls of the semidynamic deletion only data structures. This achieves $O(\log n)$ time extreme point queries. The authors [4] and independently Kaplan, Tarjan and Tsioutsiouliklis [10] improve the amortized update time to $O(\log n \log\log n)$. The improved update time in [4] is achieved by constructing a semidynamic data structure that is adapted better to the particular use. More precisely the semidynamic data structure supports build in $O(n)$ time under the assumption that the points are already lexicographically sorted. Deletions cost $O(\log n \log\log n)$ amortized time. This together with a careful choice of the parameters for the interval tree and two bootstrapping steps yields amortized $O(\log n \log\log n)$ update times and worst-case $O(\log n)$ query time. All these data structures have $O(n)$ space usage.

The main result of this paper is fully dynamic planar convex hull data structure that achieves amortized $O(\log n)$ update and query time, as summarized in the following theorem, which is proven in Section 2.

Theorem 1 *There exists a data structure for the fully dynamic planar convex hull problem supporting* INSERT *and* DELETE *in amortized $O(\log n)$ time, and* EXTREME POINT QUERY*,* TANGENT QUERY *and* NEIGHBORING-POINT QUERY *in $O(\log n)$ time, where n denotes the size of the stored set before the operation. The space usage is $O(n)$.*

If we assume that the DELETE operation provides as an argument a pointer to the point in question, we can move all of the deletion cost to the insertion, resulting in amortized $O(1)$ time deletions. From now on we consider only the upper hull of the set of points, the lower hull is completely symmetric. Together they form the convex hull of the set.

For a finite set A of points in the plane, let $\mathrm{UH}(A) \subseteq A$ denote the points on the upper hull of A, $\mathrm{Bd}(A)$ the segments forming the upper hull (including two vertical lines at the leftmost and rightmost point of A), $\mathrm{UC}(A)$ the upper closure of A, i.e., the region of the plane enclosed by $\mathrm{Bd}(A)$, and $\mathrm{UC}_0(A)$ its interior.

Related problems There is a close connection between the upper hull of some points and the lower envelop of some lines. We define (as is standard) the dual transform of point $p = (a, b) \in \mathbb{R}^2$ to be the line $p^* := (a \cdot x - b = y)$. For a set of points S the dual S^* consists of the lines dual to the points in S. Every non-vertical line in the plane is the graph of a linear function. For a finite set L of linear functions the pointwise minimum $m_L(t) = \min_{l \in L} l(t)$ is a piecewise linear function. The graph of m_L is called the *lower envelop* of L. A line $l \in L$ is on the lower envelop of L if it defines one of the linear segments of m_L.

Lemma 2 *Let S be a set of points in the plane. We have $p \in$ $\mathrm{UH}(S)$ iff p^* is on the lower envelop of S^*. The order of points on $\mathrm{UH}(S)$ is the same as the order of the segments of the lower envelop. The extreme-point query on $\mathrm{UH}(S)$ in direction t is equivalent to evaluating $m_{S^*}(t)$.*

A (dynamic) planar lower envelop is frequently understood as a parametric (or kinetic) heap, a generalization of a priority queue. We think of the linear functions as values that change linearly over time. The FIND-MIN operation of the priority queue generalizes to evaluating $m_{S^*}(t)$, the update operations can be implemented using INSERT and DELETE. The data structure of Theorem 1 allows update and query in amortized $O(\log n)$ time. A kinetic heap is a parametric heap with the restriction that the argument (time) of a query may not decrease between the queries. This naturally leads to the notion of a *current time* for queries. In Section 5 we describe a data structure that can answer kinetic queries in amortized $O(1)$ time.

Several geometric algorithms use a parametric (kinetic) heap to store lines. In some cases the function-calls to this data structure dominate the overall execution time. Then our improved data structure immediately improves the algorithm. One such example is the *k-level problem* in the plane. As discussed by Chan [6] we can use two fully dynamic kinetic heaps to produce the k-level of a set of n lines. If we have m segments on the k-level (the output size), the resulting algorithm completes in $O((n + m) \log n)$ time. This improves over the fastest deterministic algorithms, (Edelsbrunner and Welzl [8], using Chan's data structure achieving $O(n \log n + m \log^{1+\varepsilon} n)$ time). It is faster than the expected running time $O((n + m)\alpha(n) \log n)$ of the randomized algorithm of Har-Peled and Sharir [11]. Here $\alpha(n)$ is the slow growing inverse of Ackerman's function.

Lower bounds For the static convex hull computation there is a well known reduction to sorting, presented for example in the textbook by Preparata and Shamos [19]. This establishes together with Ben-Or's [2] result a $\Omega(n \log n)$ lower bound on the real-RAM for computing the convex hull. In the dynamic setting this implies that the sum of the running times of INSERT and QUERY has to be $\Omega(\log n)$. In Section 7 we prove the stronger theorem stated below.

Theorem 3 *Assume there is a data structure implementing the* SEMIDYNAMIC INSERTION-ONLY CONVEX HULL *problem on the real-RAM, that supports extreme point queries in amortized $q(n)$ time, and* INSERT *in amortized $I(n)$ time for size parameter n. Then we have $q(n) = \Omega(\log n)$ and $I(n) = \Omega(\log(n/q(n)))$.*

Similar lower bounds hold if the data structures allows the other mentioned queries. We can use several small instances of the data structures as described in Theorem 1 instead of one big data structure. This asymptotically matches the trade-off formulated in Theorem 3.

Structure of the paper In Section 2 we give a proof of Theorem 1. There we only state the function of the two main components, the geometric merging and the interval-tree. Section 4 and respectively Section 6 give more details of these constructions, Section 5 discusses the somewhat simpler kinetic case. Section 3 focuses on the variant of a search tree we use for the geometric merging, Section 7 discusses the lower bound results. Due to the limited space available in this extended abstract, several details of the construction and its analysis are omitted from this version of our work. The omitted details can be found in Riko Jacob's PhD-Thesis [15], and will appear in a journal version.

2 Outline of the main data structure

This section gives a proof of Theorem 1. Using a doubling technique we regularly rebuild the whole data structure. This allows us to assume that we know in advance the number n of points to be stored in the data structure (up to a constant). We assume that $n = 2^k$ for some integer $k \geq 2$ such that $\log n \geq 2$ and $\log \log n \geq 1$. Throughout the paper $\log$ stands for the binary logarithm.

We keep the points in semidynamic deletions-only data structures, following the ideas from Bentley and Saxe [3]. Insertions create semidynamic sets of rank 1, containing only the inserted point. As soon as we have $\log n$ sets of identical rank r we merge them into one set of rank $r + 1$. This achieves that every point participates in at most $O(\log n / \log \log n)$ merge operations, and that we have at most $O(\log^2 n / \log \log n)$ semidynamic sets simultaneously. This basic approach for the data structure was introduced by Chan [7] (with a different merging degree), and is the same as in [4, 10]. In contrast to the solutions in [4, 7, 10] we do not rebuild the semidynamic data structure from scratch after a merge. (In [4] we already reuse the lexicographical ordering of the points.) Instead we use a data structure that maintains the convex hull of the union of two (merged) sets, provided that the sets themselves are stored in semidynamic data structures. We simulate a degree $\log n$ merging by merging along a balanced binary tree.

We assume that only points on the upper hull of a semidynamic set get deleted. If we want to delete an inside point, we delay its deletion until it becomes part of the upper hull. This does not affect the amortized performance of the data structure.

Definition 1 (Semidynamic Merging Structure) *is a data structure that supports the following operations*

CREATE_SET(p) *Creates a set* $A := \{p\}$, $\mathrm{UH}(A) := (p)$*;*

MERGE(a, b) *Creates a new merging data structure for the set* $C = A \cup B$*. The upper hull of the points stored in* C *can be accessed in left to right order as they are stored in a doubly linked list. The data structures representing* A *and* B *are from now on only accessible from inside the data structure for* C*.*

DELETE(r) *Removes the point* p *referenced to by* r *from all the sets it is stored in. Determines the biggest merging structure* M *that contains* p*. It is assumed that* p *is on the upper hull of* M*. Returns the list* L *of points that replaces* p *on the upper hull of* M*.*

In Section 4 we describe a data structure, its performance is summarized in the following theorem.

Theorem 4 (Semidynamic Merging Structure) *There exists a data structure that implements the operations as described in Definition 1. Let* n *be the number of points stored in the set* $C = A \cup B$ *(not only the size of* $\mathrm{UH}(A) \cup \mathrm{UH}(B)$*). The operation* CREATE_SET(p) *takes* $O(1)$ *time. The operation* MERGE(a, b) *takes amortized time* $O(n)$*. The operation* DELETE(r) *takes amortized* $O(1)$ *time. This time for the* DELETE *operation does not include time spent in the data structures storing* A *and* B*. The space usage of the data structure is* $O(|\mathrm{UH}(A) \cup \mathrm{UH}(B)|)$*. This space usage does not include the space used in the data structures storing* A *and* B*.*

Linear space Using the above merging data structure directly yields a space usage of $O(n \log n)$. We reduce the space usage to $O(n)$ using *separators*. A separator uses the concept of (the first two) convex layers. The idea is that we can delete a point from the recursive data structure(s) as soon as it is on the upper hull of the set we store. This achieves that every point is stored in at most two semidynamic data structures More precisely, a separator uses recursively one semidynamic merging data structure and is itself a semidynamic merging structure, following Definition 1:

INIT(A) Initializes a new data structure B that wraps up the existing semidynamic merging data structure A. Let S be the set of points stored in A. Then B has the interface of a semidynamic merging structure storing the set S. We delete points from A such that no point is stored simultaneously in A and in B.

DELETE(p) Delete the point $p \in \mathrm{UH}(S)$ from the set of points S stored in B. Returns the list L of points that replace p on the upper hull of S. Points of L are deleted from A to guarantee that points are not stored twice.

Using a variant of the merging structure, the operations INIT and DELETE take amortized constant time per element. We will not discuss separators further, details can be found in [15].

Fast queries In order to achieve fast queries, we create an interval tree that allows simultaneous queries to all semi-dynamic sets. We can think of the changes to the semidynamic sets (given by the dynamization technique) as driving the interval tree, which then provides fast queries. The interval tree is easiest to explain in the dual setting, we change our point of view and discuss it in the setting of a lower envelop data structure. As part of the interval tree we use secondary structures, i.e., fully dynamic lower envelop data structures. The gain of the construction is that a secondary set stores only $\log^{O(1)} n$ lines. We require that insertions and queries of the secondary structures already have the aimed-at performance, only for the deletions we get a speed up (by performing a lot of them lazily).

Theorem 5 (Speed up construction) *Let D be a nondecreasing positive function. Assume there exists a dynamic lower envelope data structure supporting* INSERT *in amortized $O(\log n)$ time,* DELETE *in amortized $O(D(n))$ time, and* VERTICAL LINE QUERY *in $O(\log n)$ time, with $O(n)$ space usage, where n is the total number of lines inserted.*

Then there exists a dynamic lower envelope data structure problem supporting INSERT *in amortized $O(\log n)$ time,* DELETE *in amortized $O(D(\log^4 n)^2 + \log n)$ time, and* VERTICAL LINE QUERY *in $O(\log n)$ time, where n is the total number of lines inserted. The space usage of this data structure is $O(n)$.*

We give a proof of this theorem in Section 6. We use it twice to prove Theorem 1. For a first bootstrapping step we use Preparata's data structure with $D(n) = O(n)$. This yields a data structure with deletion time $D(n) = O(\log^8 n)$. In a second bootstrapping step we get a data structure with $D(n) = O(\log^{16}(\log^4 n) + \log n) = O(\log n)$.

3 Finger search trees

We use in several places level-linked-(2,4)-trees [14]. They allow amortized constant extend operations and finger searches that are logarithmic in the distance to the finger. We modify them by suspending the search operation. We do not use arbitrary fingers, but only a finger to the leftmost and rightmost leaf of the tree. We call the resulting data structure a splitter. Suspending the search is especially useful when we search for a point with some geometric properties, and we are not sure to already have such a point. We can begin a search and suspend it as soon as we realize that we do not yet have a good splitting point. Reacting to changes in the geometric situation we can continue the search, not wasting a single comparison.

A *splitter* consists of elements drawn from a completely ordered universe, stored in a level-linked (2,4)-tree. In contrast to the usual situation, searching in this tree should not be understood as finding the predecessor, but as identifying a leaf with a certain property. Every search results in a split operation, we should think of only having an operation SEARCH AND SPLIT. This search is suspended whenever we have to decide how to narrow the interval of possible outcomes (split-points). To implement such a search the splitter has three pointers to elements, namely the *candidate*, the *left guard*, and the *right guard*. The guards identify the current interval of possible split points. The candidate defines two smaller intervals, and the next step of the search is to decide which of them is correct. If we can take this decision, we *advance* the search. This amounts to changing the left or right guard to the candidate and determining a new candidate. If the current situation does not yet allow to advance the search, we keep the search *dangling*. If the situation changes and we now can decide the direction to take from the current candidate, we continue the search (by advancing it). A search is finished by executing a split operation, that has to split between the two guards. All operations we describe are destructive, the data structure is permanently changed by the execution of an operation. For all operations that deal with new elements, we assume that the order of the new elements compared to the old elements is consistent with the operation.

BUILD$(e_1, \ldots, e_k)$ Returns a new splitter containing the elements $e_1, \ldots, e_k$.

EXTEND(s, e) Extends the splitter S that contains the elements $e_1, \ldots, e_k$ to the splitter $e, e_1, \ldots, e_k$ or $e_1, \ldots, e_k, e$.

SHRINK_LEFT/RIGHT(s) The splitter S is changed by deleting its leftmost/rightmost element

INSTANTIATE_DANGLING_SEARCH(s) The guard pointers are set to nil, the candidate pointer is set to an element c stored at the root-node of the (2,4)-tree. (We start a new search that is suspended at the first comparison.) In particular we do not specify an element of the universe to search for.

ADVANCE_DANGLING_SEARCH_LEFT/RIGHT(s) The left (or right) guard is changed to point to the element the candidate pointer is currently pointing to. A new candidate element is determined according to the finger-search procedure (starting from the leftmost or rightmost leaf) in the (2,4)-tree. I.e., we disallow all elements to the right (or left) of the old candidate as possible outcomes of the dangling search.

SPLIT(s, w) The splitter S is split into two splitters S_1 and S_2 according to the value of w, which is either *left guard*, *candidate*, or *right guard*.

JOIN $(s_1, (e_1, \ldots, e_k), s_2)$ The splitters S_1 and S_2 become inaccessible and a new splitter S is created. The splitter S holds all elements from S_1, the new elements $e_1, \ldots, e_k$ and the elements of S_2 in this order. It has an active dangling search, where the left guard is on the rightmost element of S_1 and the right guard on the leftmost element of S_2. The candidate is chosen according to a (binary) search over $e_1, \ldots, e_k$.

Only the operations ADVANCE_DANGLING_SEARCH and SPLIT are allowed for a splitter with an active dangling search, the operations EXTEND, SHRINK and (more importantly) JOIN are only allowed for splitters that currently have no dangling search. We do not implement the JOIN operation as a join of the (2,4)-trees. We rather take it as a wrapper for a delayed extension of S_1 and S_2. Remember that instantiating the dangling search in the situation of the join has the promise built in that we will split at one of the elements $e_1, \ldots, e_k$ before we perform another JOIN operation with this splitter. We place $e_1, \ldots, e_k$ in an auxiliary balanced search tree and use this to guide the dangling search (or in a list and perform a linear scan to guide the search). Not until this search is settled with a SPLIT operation, we EXTEND S_1 (and S_2) with the elements left (and right) of the split point. This meets the interface of the SPLIT operation.

Theorem 6 *The operations of the splitter incur the following amortized execution times: The operations* BUILD *and* JOIN *take amortized $O(k)$ time where k is the number of the new elements ($e_1, \ldots, e_k$). The operations* INSTANTIATE DANGLING SEARCH, EXTEND, SHRINK, *and* SPLIT *take amortized $O(1)$ time. The operation* ADVANCE DANGLING SEARCH *takes a negative constant time in the amortized sense, i.e., it can pay for analyzing a constant sized geometric situation.*

Proof: We use the version of a (2,4)-tree presented in [14], with the modification that searches are suspended. We use $c(n - \ln n)$ as the potential of a splitter of size n. Splitting such a splitter into two splitters of size respectively n_1 and n_2 releases $\Omega(\log \min\{n_1, n_2\})$ potential, achieving the amortized $O(1)$ split operation, including the additional potential when advancing the search. $\square$

4 Geometric merging

The merging data structure is the core of the new approach. Here in particular we refer to [15] for more (and still important) details of the construction.

Let A and B be two sets of points in the plane. Assume that we want to compute $\mathrm{UH}(A \cup B)$ given that we already have $\mathrm{UH}(A)$ and $\mathrm{UH}(B)$ and maintain this under deletion of points. We focus on the task of identifying (and maintaining) the *equality points* of A and B, i.e. the intersections of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$. This identifies the parts of $\mathrm{UH}(A)$ that are inside $\mathrm{UC}(B)$ and vice versa. What remains are several bridge finding tasks, which we can solve efficiently (following ideas from [17]). In the following we focus on the situation (between two equality points) where $\mathrm{Bd}(B)$ is above $\mathrm{Bd}(A)$.

Because deletions are only allowed for points on the merged upper hull, $\mathrm{Bd}(B)$ gets closer to $\mathrm{Bd}(A)$ until it eventually touches in (creating new equality points). Additionally equality points can move to the left or right, or a pair of equality points can disappear. All of these cases have to be addressed, but the core problem is to detect if one or several new pairs of equality points come into existence. We sample both hulls, leading to an over-approximation of the inner hull and under-approximating of the outer hull. It is sufficient to only refine the approximations, sampling more and more points. We never take any points out of the sampling. This monotonicity allows the use of splitters.

For every point $p \in \mathrm{UH}(A)$ we define the concept of a *valid pair of rays* in the following way: Let h and l be two different tangent lines on $\mathrm{Bd}(A)$ through p, i.e. $h \cap l = \{p\}$ and all points of A are on or below h and l. If h and l contain the two adjacent segments of $\mathrm{Bd}(A)$ that are adjacent to p we call this the *canonical pair of rays* rooted at p. Let H and L be the intersection points of h (and respectively of l) with $\mathrm{Bd}(B)$. Then we have a certificate that there is no equality point of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ between the vertical lines through H and L. We maintain only some of these certificates, we require that the maintained certificates are vertically disjoint. A point for which we maintain the certificate is called a *selected point*. For each selected point we decide upon a particular valid pair of rays, its *strong rays*. Once we decided to select a point $p \in A$ it stays selected until $\mathrm{Bd}(B)$ drops below p (and it is therefore no longer a certificate of $\mathrm{Bd}(B)$ being locally above (outside) $\mathrm{Bd}(A)$). The strong rays of a selected point do not change. We maintain the intersections of the strong rays with $\mathrm{Bd}(B)$ explicitly. These intersections and the equality points are the only points of the locally outer hull that are explicitly set in relation with the locally inner hull.

The data structure maintains an inclusion-maximal set of selected points, such that the certificates of two selected points do not overlap: the intersection of two strong rays is outside (above) of $\mathrm{UC}(B)$. For a selected point $p \in \mathrm{UH}(A)$ this requirement disallows the selection of several other points of $\mathrm{UH}(A)$, a range in the left-to-right ordering of $\mathrm{UH}(A)$ around p, the *shadow* of p. The maximal independent set of selected points is characterized by the selected points not being in the shadow of another selected point, and every point of $\mathrm{UH}(A)$ being in the shadow of some selected point.

Deletions of points on $\mathrm{UH}(B)$ induce that some shadows get smaller. We focus on two consecutive selected points p and q where a shadow is changed. We maintain maximality by searching for a point of $\mathrm{UH}(A)$ that is between the shadow of p and the shadow of q. If we find such a point we select it (and perform the same maximality check between the new point and p and q). If we cannot find such a point (the set of selected points is maximal), the shadows of p and q overlap (or touch at least). A point in the intersection of the two shadows is called a *(geometric) valid candidate*. (Geometrically it is easy to verify that a point is a candidate, whereas it seems to require a search to determine the boundaries of the shadows.)

We use a splitter for this search for a selectable point. It stores all the not-selected points of UH(A) between p and q. If the candidate of the dangling search is a geometrically valid candidate, we leave the search dangling. We found a new certificate that the set of selected points is maximal. If another deletion of a point in B changes the shadows further, we do not start over the search, but merely advance it. This use of the splitter is of course the motivation for the otherwise somewhat unusual interface of the splitter. In this way we manage to search for a next point to select, using overall constant time per point in A.

For a candidate point c we determine in which of the two shadows c is contained. This amounts to considering the canonical pair of rays h, l rooted at c. The rays h and l are called *weak rays*. If the right (left) directed strong ray of p (q) intersects l (h) before it intersects Bd(B), then c is in the shadow of p (q). If c is a geometrically valid candidate, this shows that we additionally have a certificate that there is no equality point of Bd(A) and Bd(B) between p and q.

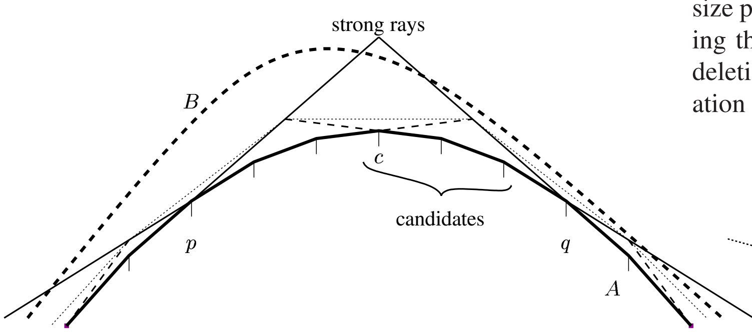

Figure 1. A dangling search. Hull A is below hull B (depicted as a curve, it is here not important that it is a polygon). The points p and q are selected, their strong rays are depicted as a solid line. The point c is the current candidate of the dangling (suspended) search, its weak rays are depicted as dashed lines. The resulting over-approximation of A is depicted as a light, dotted line.

Between an equality point and the first selected point we maintain a certificate that no further point can be selected. In analogy to the dangling search, we call the situation a *half open search*. Geometrically we merely consider the weak ray h originating from the segment defining the equality point. If h intersects the next strong ray inside Bd(B), no further point of UH(A) can be selected. We store the points of UH(A) between the last selected point and the equality point in a splitter that does not have a dangling search.

The certificates defined by selected points and candidates form an over-approximation of A that is entirely inside of UC(B). This situation is exemplified in Figure 1. We only maintain the intersections of Bd(B) with the strong rays explicitly, between two intersections we define a *shortcut*: We consider an appropriate half-plane H and replace in our considerations UC(B) with UC$(B) \cap H$. This situation is exemplified in Figure 2. We can have several such shortcuts. We only have to make sure that the shortcuts do not create new equality points, and that they do not *overlap*, i.e., that any point of B is cut away by at most one shortcut. This ensures that the shortcuts do not make the approximation more complicated than the hull itself. When selecting a point and establishing a new pair of strong rays, we find the intersection with the simplified (shortcut) version of B.

The analysis of the running time of the data structure follows the life-cycle of a point. When p first appears as part of UH(B) (as a result of a deletion on B), it can be inside of UC(A) and is not selected. Then it can become selected. It stays selected until it becomes outside of UC(A) (because of a deletion on A). Once outside it can get cut away by a shortcut once and also be part of a bridge finding once. Finally it might get deleted.

Every deletion requires us to consider only a constant size part of the construction, the deletion can pay for reverting the life-cycle of these constantly many points. Every deletion and every selection of a point can cause the creation of constantly many shortcuts.

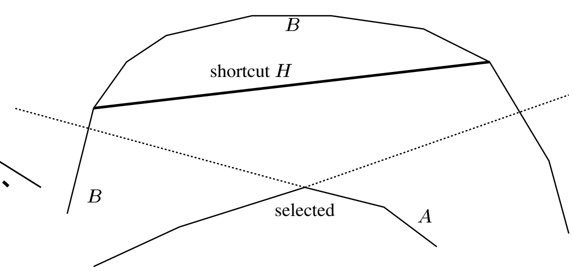

Figure 2. A shortcut. The selected point of A defines two strong rays, depicted as dashed lines.

Processing a deletion of $r \in B$ has to handle different cases: Each of the two deleted segments (in Bd(B) adjacent to r) intersects not, once or twice Bd(A). The corresponding equality points change. We have to examine the position of the new points on the upper hull of B and decide for each of them whether it is inside, outside or on Bd(A). This is done by reestablishing the construction.

We use the splitters and dangling searches if we are looking at a situation where it is still possible that B is above A. If we find a stretch where we now have a stretch with A

above B, we can afford to perform linear scans, both on the new points on UH(B) and on the surfacing part of UH(A). An additional problem is that a deletion can destroy a pair of equality points, then we have to join two regions where A is above B. We can afford to perform a linear scan to detect the situation (the points of A get outside UC(B), on B we have new points), and we can use the JOIN operation on splitters to join the regions. On A the shortcuts achieve that we have to deal with only constantly many old points of B. This situation is exemplified in Figure 3. For a detailed treatment of the different cases see [15].

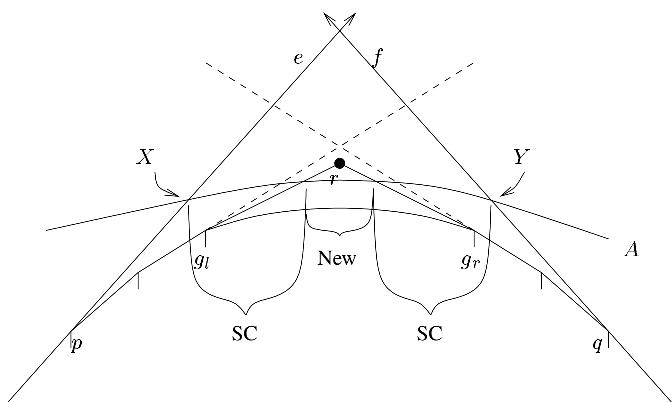

Figure 3. Joining two strips. The point r of hull B gets deleted. The points p and q of B are selected, before the deletion they belong to different regions of the construction. The weak rays of the neighbors of r are depicted as dashed lines. The range of UH(A) marked by SC can be arbitrarily long; here shortcuts achieve that we can regard all of the range of UH(A) between X and Y as new points.

5 Kinetic Heaps

Using the semidynamic geometric merging data structure of Theorem 4 we can design a kinetic heap:

Theorem 7 *There exists a data structure for the fully dynamic kinetic heap problem supporting, for size parameter n,* INSERT *and* DELETE *in amortized $O(\log n)$ time, and* KINETIC FIND MIN *in amortized $O(1)$ time. The space usage of the data structure is $O(n)$.*

Again we could charge the cost of the deletion to the insertion, thus achieving amortized constant time deletions. We give some more details about this construction, as it illustrates several techniques that we also use for the general construction of Theorem 1. The semidynamic data structure of Theorem 4 allows kinetic find-min queries in amortized $O(1)$ time. We perform a linear scan when answering a query, this is charged to the insertion.

Every line l that is part of the lower envelope of its semidynamic structure defines an *activity interval* I_l that contains the query-values for which l is the correct FIND-MIN answer.

Lemma 8 (Kinetic speed up) *Let D be a nondecreasing positive function. Assume there exists a fully dynamic kinetic heap data structure supporting* INSERT *in amortized $O(\log n)$ time,* DELETE *in amortized $O(D(n))$ time, and* KINETIC FIND MIN *in $O(1)$ amortized time, where n is the total number of lines inserted. Assume the space usage of this data structure is $O(n)$.*

Then there exists a fully dynamic kinetic heap data structure supporting INSERT *in amortized $O(\log n)$ time and* DELETE *in amortized $O(D(\log^2 n) + \log n)$ time, and* KINETIC FIND MIN *in amortized $O(1)$ time, where n is the total number of lines inserted. The space usage of this data structure is $O(n)$.*

Proof: We use the already explained dynamization technique with merging degree $\log n$. We use one secondary structure with the assumed performance. This secondary structure stores at most $O(\log^2 n)$ lines simultaneously, the current answer from every (un-merged/top-level) semidynamic structure.

For the current time t we store all the FIND-MIN answers from the semidynamic sets in the secondary structure. Additionally we keep a (2,4)-tree holding the right endpoints of the corresponding activity intervals. We also keep the smallest such endpoint explicitly, it tells how long the current answer remains valid.

For a KINETIC-FIND-MIN query for time t we do the following: We first check with the smallest right endpoint whether the current secondary structure is up-to-date. If so, there is no further change to the data structure. If this is not the case we delete all endpoints from the (2,4)-tree that are smaller than t. We lazily delete the corresponding lines from the secondary structure, i.e. we merely mark them as deleted and remove them only when the data structure is rebuilt because half of the lines stored are marked as deleted. For all semidynamic sets that are no longer represented in the secondary structure we perform a KINETIC-FIND-MIN query for time t. We insert the resulting lines into the secondary structure and insert the right endpoints of the segments into the (2,4)-tree. We update the smallest right endpoint when updating the (2,4)-tree. Now we perform the query on the secondary structure for time t.

For a MERGE operation (stemming from the dynamization technique) we remove the affected current endpoints from the (2,4)-tree and perform lazy deletions of the lines in the secondary structure. We query the new data structure for the current time and insert the resulting line in the secondary structure, and the right endpoint of the segment into the (2,4)-tree.

For a DELETE(l) operation we delete l from the semidynamic data structure it is stored in, and if the deleted line is currently stored in the secondary structure we delete it from the secondary structure (not lazily). We call this situation a *forced* deletion. We also delete the right endpoint from the (2,4)-tree. We query the changed semidynamic data structure for the current time and insert the line into the secondary structure and the right endpoint of the answered segment into the (2,4)-tree.

The run-time analysis keeps accounts for the lines. Every line has to pay for one insertion and deletion into the (2,4)-tree and for one insertion, query, and lazy deletion in the secondary structure for the at most $\log n / \log \log n$ merging levels. This totals to $O(\log n)$ amortized time, charged to the insertion of the line. A deletion pays for the deletion of one line in the secondary structure and for querying and reinserting one right endpoint in the (2,4)-tree and a line into the secondary structure, thus reverting some other lines life-cycle. □

Using Preparata's [18] semidynamic insertion only data structure, we achieve insertions in $O(\log n)$ time and $O(1)$ kinetic heap queries. The $O(n)$ amortized deletions do not only rebuild the data structure, but also pay for advancing the kinetic search over all segments, thus achieving amortized $O(1)$ queries. Using this data structure in Lemma 8 (bootstrapping) we get $O(\log n)$ amortized insertions, $O(1)$ amortized queries and amortized $O(\log^2 n)$ deletions. Bootstrapping one more time reduces the amortized deletion cost to $O(\log n)$, yielding Theorem 7.

6 General queries: Interval Tree

The overall approach to prove Theorem 5 is similar to the use of a secondary structure in the kinetic solution. The main difference is that we need to maintain several secondary structures, each of limited size $\log^{O(1)} n$. To organize them we use the activity intervals of the segments. Geometrically this construction is very similar to Chan's [7], in particular the reasoning for the correctness of the queries is the same, our construction allows the same queries with the same performance.

A traditional interval tree is a data structure that stores intervals in a way that allows efficient containment queries. More precisely for a set J of intervals, the query consists of $x \in \mathbb{R}$ and the answer consists of all intervals $I \in J$ such that $x \in I$. The central idea is to store the intervals at the nodes of a search tree, such that only intervals stored on a standard search-path for x have to be considered for the containment query. We create a secondary structure for every node of $\mathcal{T}$. If we store the lines of the lower envelopes of the semidynamic sets at appropriate nodes of $\mathcal{T}$, as given by their activity interval, we can correctly answer FIND-MIN queries.

The tree structure of an interval tree $\mathcal{T}$ is that of a search tree storing at the leafs the endpoints of the intervals. For every interval I exists a *canonical node* of $\mathcal{T}$, defined as the node v of $\mathcal{T}$ where both endpoints of I are (would be) leafs below v, but none of the children of v enjoys this property. Like Chan we choose the underlying tree structure to be that of an insertion-only B-tree. In contrast to Chan we use degree parameter of $\log n$, independent of the bootstrapping. With this choice the height of $\mathcal{T}$ is bounded by $O(\log n / \log \log n)$. Even if we allow secondary structures of size $\log^{O(1)} n$, we achieve vertical line queries in $O(\log n)$ time. Unlike Chan we allow the lines to be stored anywhere on the path in $\mathcal{T}$ from the root to the canonical node. This does not compromise the correctness of the queries. Like in [4] this allows us to save time when determining for a line the appropriate node of $\mathcal{T}$ to store it. Additionally, this freedom allows us to perform the movement of lines lazily (as a result of changed lower envelopes in the semidynamic sets, i.e., because of merge or delete operations). Chan's argument bounding the work spent in node-split operation carries over.

Like in the kinetic case we can allow every line to be inserted into a secondary structure once as the result of a merge operation of the dynamization technique. Here we also have to determine appropriate nodes of $\mathcal{T}$ where we should insert the lines. We address this problem by forming *chunks* of $\log n / \log \log n$ consecutive segments on the lower envelope. Now we determine for the complete chunk one interval and find the appropriate node in $\mathcal{T}$, taking $O(\log n)$ time. This costs per segment $O(\log \log n)$ time, the same as for inserting the line into the secondary structure. This chunk size is small enough to move the chunk in $O(\log n)$ time, i.e. to insert all lines into a different secondary structure. This allows us to maintain the chunks under deletions of lines and also bounds the work when we split nodes of the interval tree.

As part of moving a line from one secondary structure to another, we also need to delete the line from the secondary structure it is currently stored in. We will perform these deletions lazily, delaying the insertion of the line into the new secondary structure as well. We call this concept a *lazy movement*. When half of a secondary structure consists of lazily moved lines, we rebuild it from scratch. Only then we execute lazy movements, i.e., we insert the lines in the secondary structure they belong.

If a line l is part of the merging of semidynamic sets, its interval shrinks because the line l is now competing with more lines for a place on the lower envelope. This means that the canonical node of the interval of a line will in general be closer to the leafs. We do not really need to move the line, it is still stored on the path from the root to its canonical node. Even if the line l is no longer on the lower envelope (but is still not deleted from S) we can store l at *any* node in the interval tree without compromising the correctness of the queries.

In contrast to this, a deletion of line $l \in S$ can extend activity intervals, first of all the two segments that are neighbors of l on the lower envelope of the semidynamic set of l. But also a line h that becomes part of the lower envelope and is still stored somewhere in $\mathcal{T}$ get a new activity interval. This can be bigger (not a subset) than the last non-empty activity interval I_h (the one justifying the position of h in $\mathcal{T}$), only if I_h was defined by an intersection point of h and l. This happens for at most two lines per merging l participated in. When the activity interval of a line extends, we might have to immediately move it to a different secondary structure. As we do not allow multiple copies of one line, we have to delete the line from a secondary structure using the (slow) DELETE operation, we say that we perform a *forced move*. It is essential for our analysis to bound the number of forced moves.

We distinguish two cases depending on the performance of the deletions of the secondary structure. The size of a secondary structure is bounded by $O(\log n \cdot \frac{\log n}{\log\log n} \cdot \frac{\log^2 n}{\log\log n})$, the terms stemming respectively from the degree of $\mathcal{T}$, the chunk size, and the number of semidynamic sets. We simplify this bound to $O(\log^4 n)$.

Assume $D(\log^4 n) = \Omega(\log n)$ (e.g. $D(n) = O(n)$ in the first bootstrapping step). Every line participates in at most $O(\log n/\log\log n)$ merge operations, this also bounds the number of forced moves a deletion can cause. This term is by assumption bounded by $O(D(\log^4 n))$, resulting in a total amortized cost of $O(D(\log^4 n)^2)$ as claimed in Theorem 5.

Otherwise we have $D(\log^4 n) = O(\log n)$ (e.g. $D(n) = O(\log^8 n)$ in the second bootstrapping step). In this situation we move some of the costs for forced moves to the insertions without changing their asymptotic performance. To do so, we introduce *barrier levels* of the merging in the dynamization technique. At a barrier level, all lines in the created semidynamic set get paid a forced move. Choosing the parameter $b(n) = \log n/D(\log^4 n)$, we charge $b(n)$ forced moves to every insertion. Now a deletion has only to pay for forced moves back to the last barrier level, leaving us with less than $\log n/b(n)$ forced moves to be paid by the deletion. This charges $O(b(n) \cdot D(\log^4 n)) = O(\log n)$ time to the insertion, thus not changing the asymptotic performance. The forced moves a single deletion has to pay is $O(D(\log^4 n)\log n/b(n)) = O(D(\log^4 n)^2)$. This yields Theorem 5.

7 Lower bounds

In this section we derive lower bounds on running times that asymptotically matches the quality of the data structures we presented in the previous sections. The model of computation is the algebraic real-RAM. The lower bounds on the decision problems hold for algebraic computation trees. A real-RAM algorithm can be understood as generating a family of decision trees, the height of the tree corresponds to the worst-case execution time of the algorithm. This is the model used in the seminal paper by Ben-Or, from where we take the main theorem [2, Theorem 3] that bounds the depth of a computation tree in terms of the number of connected components of the decided set. We consider the following decision problem, a variant of element-distinctness.

Definition 2 *For a vector $z = (x_1, \ldots, x_n, y_1, \ldots, y_k) \in \mathbb{R}^{n+k}$ we have $z \in \textsc{DisjointSet}^+_{n,k} \subset \mathbb{R}^{n+k}$ if and only if $y_1 \le y_2 \le \cdots \le y_k$ and for all i and j we have $x_i \neq y_j$.*

Lemma 9 *For $8 < k \le n$ the depth h of an algebraic computation tree (the running time of a real-RAM algorithm) deciding the set $\textsc{DisjointSet}^+_{n,k}$ is lower bounded by $h \ge c \cdot n \log k$ for some $c > 0$.*

Proof: (sketch) There are $(k+1)^n$ ways of distributing the values x_i into the intervals formed by the y_i, no two of them can be in the same connected components of $\textsc{DisjointSet}^+_{n,k}$. Using [2, Theorem 3] we get $2^h 3^{n+k+h} \ge (k+1)^n$ implying the claimed lower bound. □

The running time functions in the following theorems are assumed to be non-decreasing with the size of the data structure. As these functions are used as upper bounds on running times, this is no loss of generality.

Definition 3 SEMIDYNAMIC KINETIC MEMBERSHIP *asks for a data structure that maintains a set S of real numbers under insertions, and allows for a value x the query $x \in S$, provided that x is not smaller than any previously performed query.*

Theorem 10 *Let $\mathcal{A}$ be a data structure that implements* SEMIDYNAMIC KINETIC MEMBERSHIP. *For size parameter n assume that the amortized running time of the* INSERT *operation of $\mathcal{A}$ be bounded by $I(n)$ and the amortized running time for the* KINETIC-FIND-MIN *query be bounded by $q(n)$. Then we have $I(n) = \Omega\left(\log \frac{n}{q(n)}\right)$.*

Proof: By reduction from $\textsc{DisjointSet}^+_{n,k}$. We choose the parameter $k = \lfloor n/q(n) \rfloor$. Let the vector $z = (a_1, a_2 \ldots, a_n, b_1, b_2 \ldots, b_k) \in \mathbb{R}^{n+k}$ be some input to $\textsc{DisjointSet}^+_{n,k}$. We check in linear time whether we have $b_1 \le b_2 \le \cdots \le b_k$. If this is not the case, we reject. We insert all the values a_i into $\mathcal{A}$. Then we perform queries for all the b_j (in the natural order). If one of the queries returns $b_j \in S$, i.e., $b_j = a_i$ for some i and j, we reject. Otherwise we accept. This correctly solves the $\textsc{DisjointSet}^+_{n,k}$ problem.

The reduction takes linear time. By Lemma 9 the running time of this algorithm is bounded by $(I(n)+d)\cdot n + q(n)\cdot k \geq c\cdot n\cdot \log k$ for some constants c and d. Using our choice of k we get $(I(n)+d)\cdot n + n \geq c\cdot n\cdot \log\lfloor n/q(n)\rfloor$. Dividing by n and rearranging terms yields $I(n)\geq c\cdot\log(\lfloor n/q(n)\rfloor) - 1 - d$. □

Note that for $q(n) = O(n^{1-\varepsilon})$, Theorem 10 implies $I(n) = \Omega(\log n)$. Another example is that $I(n) = O(\log\log n)$ yields $q(n) = \Omega(n/(\log n)^{O(1)})$.

Theorem 11 *Consider a data structure implementing the* SEMIDYNAMIC MEMBERSHIP *problem on the real-RAM that supports* MEMBER *queries in amortized $q(n)$ time, for size parameter n. Then we have $q(n) = \Omega(\log n)$.*

Proof: Reduction from $\text{DISJOINTSET}^{+}_{n,k}$. Let $I(k)$ be an upper bound on the amortized insertion time for one element when creating a data structure holding k elements. We choose the parameter $n = k\cdot(1+I(k))$. Let the vector $z = (a_1,\ldots,a_n,b_1,\ldots,b_k)\in\mathbb{R}^{n+k}$ be an input to $\text{DISJOINTSET}^{+}_{n,k}$. We check in time k whether we have $b_1\leq b_2\leq\cdots\leq b_k$. If this is not the case, we reject. We insert the values $b_1,\ldots,b_k$ into the data structure. Now we perform queries for the values $a_1,\ldots,a_n$. By Lemma 9 we get for sufficiently large k and some constant c the inequality $k\cdot(1+I(k)) + n\cdot q(k)\geq c\cdot n\log k$. □

Corollary 12 *Consider a kinetic heap data structure. Assume that for size parameter n the amortized running time of the* INSERT *operation is bounded by $I(n)$ and the amortized running time for the* KINETIC-FIND-MIN *query is bounded by $q(n)$. Then we have $I(n) = \Omega\left(\log\frac{n}{q(n)}\right)$.*

Proof: We use the data structure to solve the SEMIDYNAMIC KINETIC MEMBERSHIP. For an insertion of a_i we insert the tangent on $y = -x^2/2$ at the point $(a_i, -a_i^2/2)$. For a member query b_i we perform KINETIC-FIND-MIN(b_i). If the query returns the tangent line, we answer "$b_i\in S$". The corollary follows from Theorem 10. □

Finally we can also conclude the main theorem:
Proof: (of Theorem 3) A semidynamic insertion-only convex-hull data-structure can be used as a kinetic heap (duality), Corollary 12 provides the bound on the insertions. The bound on the queries relies on Theorem 11, with the same geometric reduction as in Corollary 12. □

8 Open problems

It remains open whether a data structure achieving worst-case $O(\log n)$ update times exists. It is also unclear if other queries (like the segment of the convex hull intersected by a line) can also be achieved in $O(\log n)$ time. Furthermore it would be desirable to come up with a simpler data structure achieving the same running times.

References

[1] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[2] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. 15th Annual ACM Symposium on Theory of Computing, 80 – 86*, 1983.

[3] J. L. Bentley and J. B. Saxe. Decomposable searching problems. I. Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[4] G. S. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time and $O(\log n\cdot\log\log n)$ update time. In *Proc. 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 57–70. Springer, 2000.

[5] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete Comput. Geom.*, 16(4):361–368, 1996. Eleventh Annual Symposium on Computational Geometry (Vancouver, BC, 1995).

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

[7] T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *Journal of the ACM*, 48(1):1–12, January 2001.

[8] H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, 15(1):271–284, 1986.

[9] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[10] K. H., R. Tarjan, and T. K. Faster kinetic heaps and their use in broadcast scheduling. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 836–844, 2001.

[11] S. Har-Peled and M. Sharir. On-line point location in planar arrangements and its applications. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 57–66, 2001.

[12] J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32(2):249–267, 1992.

[13] J. Hershberger and S. Suri. Off-line maintenance of planar configurations. *J. Algorithms*, 21(3):453–475, 1996.

[14] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time using level-linked search trees. *Information and Control (now Information and Computation)*, 68(1-3):170–184, 1986.

[15] R. Jacob. *Dynamic Planar Convex Hull*. PhD thesis, BRICS, Dept. Comput. Sci., University of Aarhus, 2002. http://www.brics.dk/˜rjacob/Diss.

[16] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15(1):287–299, 1986.

[17] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. System Sci.*, 23(2):166–204, 1981.

[18] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Comm. ACM*, 22(7):402–405, 1979.

[19] F. P. Preparata and M. I. Shamos. *Computational geometry, An introduction*. Springer-Verlag, New York, 1985.