Dynamic Planar Convex Hull

Gerth Stølting Brodal [1,2]

BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. [3]

Riko Jacob [1]

Database and Information Systems Group, University of Munich (LMU), Institute for Computer Science, Oettingenstr. 67, 80538 München, Germany.

Abstract

In this paper we determine the amortized 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.

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

Email addresses: `gerth@brics.dk` (Gerth Stølting Brodal), `rjacob@brics.dk` (Riko Jacob).

URLs: `www.brics.dk/~gerth` (Gerth Stølting Brodal), `www.brics.dk/~rjacob` (Riko Jacob).

[1] Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Work done while staying at BRICS.

[2] Supported by the Carlsberg Foundation (contract number ANS-0257/20).

[3] BRICS: Basic Research in Computer Science, `www.brics.dk`, funded by the Danish National Research Foundation.

1 Introduction

The convex hull of a set of points in the plane is one of the most prominent objects in computational geometry. It is defined as the convex polygon having vertices on points of the set, such that all the points of the set are inside the polygon. Computing the convex hull of a static set of n points can be done in optimal $O(n \log n)$ time, e.g., with Graham's scan (1) or Andrew's vertical sweep line variant (2) of it. Optimal output sensitive algorithms are due to Kirkpatrick and Seidel (3) and also to Chan (4), who achieve $O(n \log h)$ time, where h denotes the number of vertices on the convex hull.

In the dynamic setting we consider a set S of points in the plane that is changed by insertions and deletions. 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 $\vec{a}$, the tangent(s) on the hull that passes through a given point p_e, whether or not a point p_c is inside the convex hull, the segments of the convex hull intersected by a given line ℓ_b, the bridges (common tangents) between another convex hull C. These queries are illustrated in Figure 1. Furthermore we might want to report (some consecutive subsequence of) the points on the convex hull or count their cardinality.

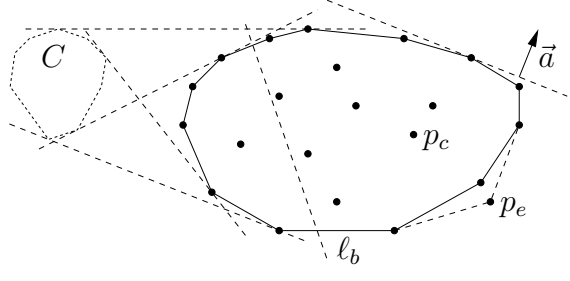

Fig. 1. Different queries on the convex hull of a set of points.

Overmars and van Leeuwen (5) 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. The leaf-links allow to report k consecutive points on the convex hull (between two directions, tangent lines or alike) in $O(\log n + k)$ time. Semidynamic variants of the problem have been considered. There, updates are restricted to be either insertions only or deletions only. For the insertion-only problem Preparata (6) gives an $O(\log n)$ worst-case time algorithm that maintains the vertices of the convex hull in a search tree. The deletion-only problem is solved by Hershberger and Suri (7), 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 (8) also

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 (9) 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 Saxe (10). Using the semidynamic deletions-only data structure of Hershberger and Suri (7), and a constant number of bootstrapping steps, the construction achieves update times of $O(\log^{1+\varepsilon} n)$ for any constant $\varepsilon > 0$. To support queries this construction uses an augmented variant of an interval tree to store the convex hulls of the semidynamic deletion only data structures. The authors of the present paper in (11) and independently Kaplan, Tarjan and Tsioutsiouliklis (12) improve the amortized update time to $O(\log n \log\log n)$. The improved update time in (11) is achieved by reconsidering the framework of Chan (9), and by constructing a semidynamic deletion-only 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. All these data structures have $O(n)$ space usage.

The main result of this paper is a fully dynamic planar convex hull data structure:

Theorem 1 *There exists a data structure for the fully dynamic planar convex hull problem supporting insertions and deletions in amortized $O(\log n)$ time, and extreme point queries, tangent queries and neighboring-point queries in $O(\log n)$ time, where n denotes the size of the stored set before the operation. The space usage is $O(n)$.*

This is complemented with a matching lower bound: [4]

Theorem 2 *Assume there is a semidynamic insertion-only convex hull data structure on the real-RAM, that supports extreme point queries in amortized $q(n)$ time, and insertions in amortized $I(n)$ time for size parameter n. Assume that q and I are non-decreasing functions.*

[4] Both results are published in a conference version (13), and in the PhD-Thesis of Riko Jacob (14).

Then we have $q(n) = \Omega(\log n)$ *and* $I(n) = \Omega(\log(n/q(n)))$.

This implies that queries have to take (amortized) $\Omega(\log n)$ time and that insertions also have to take amortized $\Omega(\log n)$ time, as long as queries take amortized time $O(n^{1-\varepsilon})$. Note that this result is stronger than just applying the well known lower bound $\Omega(n \log n)$ for the static convex hull computation, as presented for example in the textbook by Preparata and Shamos (15). It implies in the dynamic setting only that the sum of the amortized running times of insertion and next-neighbor query is $\Omega(\log n)$.

For a finite set A of points in the plane, the convex hull of A naturally decomposes into an upper and a lower part of the hull. In the remaining of the paper we work only with the upper hull, the lower hull is completely symmetrical. To represent the convex hull we store every point of A in two data structures, one that maintains access to the upper hull, and one for the lower hull. This allows us to answer the mentioned queries on the convex hull. To simplify the exposition, we extend the upper hull of A to also include two vertical half-lines as segments, one extending from the leftmost point of A vertically downward, and another one extending downward from the rightmost point. The set of points that are on segments and vertices of the upper hull of A are denoted as $\mathrm{Bd}(A)$, the vertices of the upper hull as $\mathrm{UV}(A) \subseteq A$, and the interior of the upper hull as $\mathrm{UC}_0(A)$, and the upper hull together with its interior as $\mathrm{UC}(A)$—the upper closure of A.

1.1 Duality, kinetic heaps and applications

There is a close connection between the upper hull of some points and the lower envelope of some corresponding lines. We define (as is standard, see e.g. (16)) the dual transform of point $p = (a, b) \in \mathbb{R}^2$ to be the line $p^* := (y = a \cdot x - b)$. For a set of points S the dual S^* consists of the lines dual to the points in S. This concept is illustrated in Figure 2. The slope of a non-vertical line $(y = a \cdot x + b)$ is the value of the parameter a.

Every non-vertical line in the plane is the graph of a linear function. For a finite set L of linear functions the point-wise minimum $m_L(t) = \min_{l \in L} l(t)$ is a piecewise linear function. The graph of m_L is called the *lower envelope* of L. A line $l \in L$ is on the lower envelope of L if it defines one of the linear segments of m_L. The essential properties of the duality transformation are captured in the following Lemma.

Lemma 3 *Let S be a set of points in the plane. We have $p \in \mathrm{UV}(S)$ if and only if p^* is on the lower envelope of S^*. The left-to-right order of points on $\mathrm{UV}(S)$ is the same as the right-to-left order of the segments of the lower envelope. The extreme-point query on $\mathrm{UV}(S)$ in direction $q = (-\alpha, 1)$ (the*

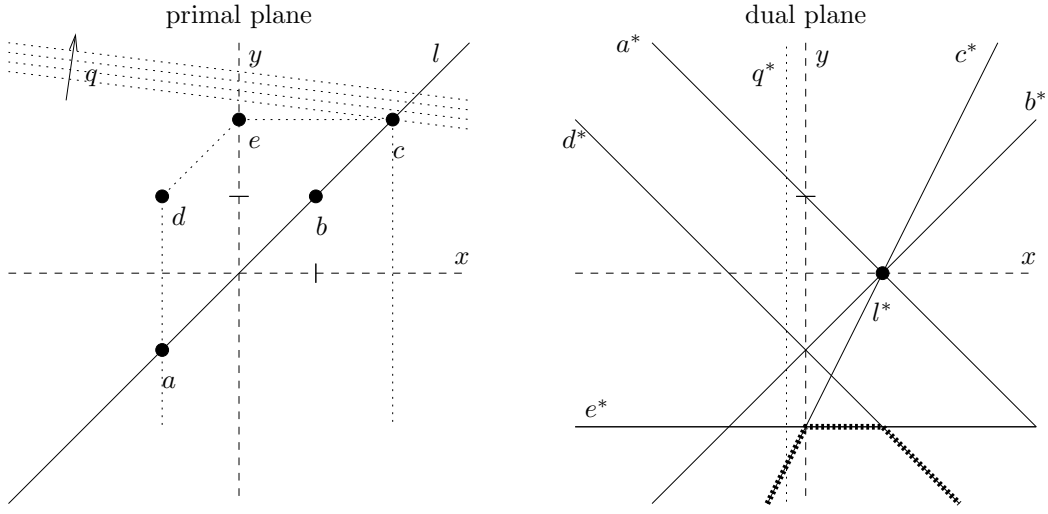

Fig. 2. Duality of points and lines. q stands for an extreme point query in the primal setting, q^* is the corresponding query in the dual setting.

answer tangent line has slope α) is equivalent to evaluating $m_{S^*}(\alpha)$.

A (dynamic) planar lower envelope is frequently understood as a *parametric 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 amount to insertions and deletions of lines. The data structure we summarize in 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 (kinetic) queries may not decrease between two queries. This naturally leads to the notion of a *current time* for queries. In Section 8 we describe a data structure that can answer kinetic queries in amortized $O(1)$ time and updates in amortized $O(\log n)$ 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 algorithm by Edelsbrunner and Welzl (17) solving the *k-level problem* in the plane. The problem is in the dual setting and is given by a set S of n non-vertical lines in the plane. For every vertical line we are interested in the k-th lowest intersection with a line of S. The answer is given by a collection of line-segments from lines of S. This generalizes the notion of a lower envelope ($k = 1$) and an upper envelope ($k = n$). The situation is exemplified in Figure 3.

As discussed by Chan (18) 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 algorithm using the data structure of the present paper

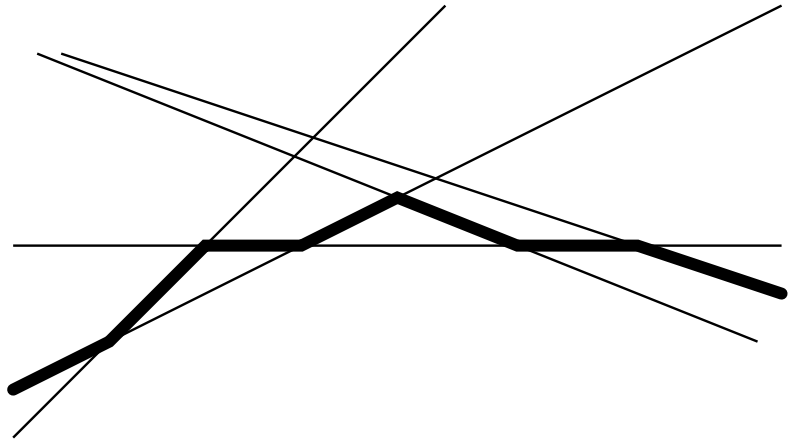

Fig. 3. The 2-level of 5 lines in the plane. Note that the 2-level consists of 7 segments, two of the lines define two separate segments.

completes in $O((n+m)\log n)$ time. This improves over the fastest deterministic algorithms, (Edelsbrunner and Welzl (17), 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 (19). Here $\alpha(n)$ is the slow growing inverse of Ackerman's function.

1.2 Algorithmic model

The model of computation is the algebraic real-RAM, as for example introduced in the textbook by Preparata and Shamos (15). The presented algorithm/data structure is formulated for the real-RAM, but it actually only requires only to evaluate the sign of constant degree polynomials on the coordinates of the input points. (Auxiliary lines are defined by two input points, hence every decision considers only a constant number of input points.)

We simplify the exposition here by assuming that the points are in general position, i.e., no three input points are on one line, and no three lines defined by input points meet in one point. This avoids most of the special cases, for example a point can only be above or below a line defined by two other points, but not on it. This assumption really only simplifies the exposition, we can always treat the degenerate cases explicitly, extending the described data structure and algorithms in a straightforward manner. We discuss further aspects of this in Section 7.

1.3 Structure of the paper

Section 2 gives the proof of Theorem 1, describing the overall performance of the proposed data structure. There we only state the function of the two main components, the geometric merging and the interval-tree. Sections 4,5,6 describe the geometric merging. Section 9 describes the interval-tree, preceded by Section 8 that discusses the somewhat simpler case of a kinetic heap. Section 3 focuses on the variant of a search tree we use for the geometric merging, Section 7 discusses issues of avoiding the general-position assumption, and Section 10 discusses the lower bound results.

2 Outline of the main data structure

The core of the present paper is a data structure for the dynamic planar convex hull problem. This section describes how different components of the data structure work together to achieve Theorem 1.

Using a doubling technique we regularly rebuild the whole data structure. This allows us to assume that we know in advance some n such that the number of points to be stored in the data structure is between $n/4$ and n. We assume $n = 2^k$ 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, using the logarithmic method of Bentley and Saxe (10). 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 first used by Chan (9) (with a different merging degree), and is the same as in (11; 12).

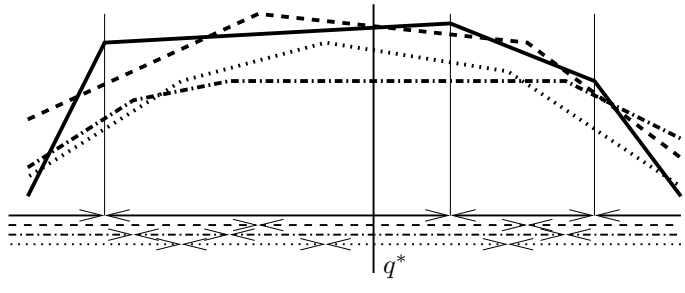

Fig. 4. Illustrating the task of joining several lower envelopes for the purpose of a fast query, and the connection to intervals.

To achieve $O(\log n)$ queries we use a version of an interval tree to simultaneously query the dual envelopes of the upper hulls of the sets stored in the semidynamic data structures. This is illustrated in Figure 4. We give the full functionality off the interval-tree in Section 2.2, and the full description in Section 9. The different pieces of the construction, and how they interact is schematically depicted in Figure 5. The whole construction is a speed-up construction for the deletion-time, the interval-tree uses secondary structures, fully dynamic planar convex hull data structures, that store polylogarithmically many points. Two bootstrapping steps achieve the claimed amortized deletion time of $O(\log n)$.

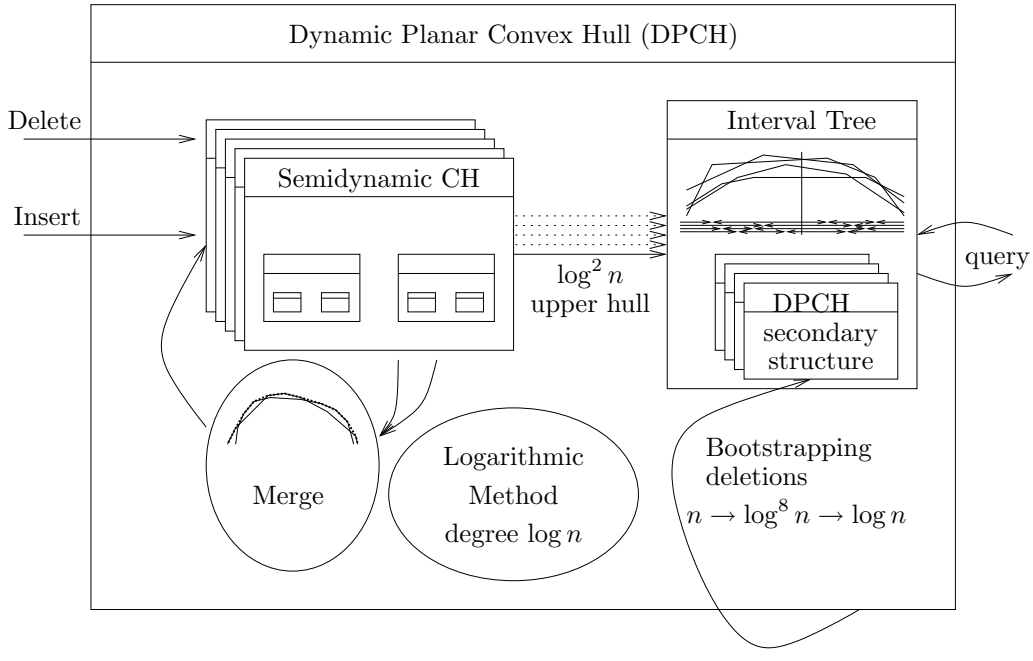

Fig. 5. A schematic depiction of the different components of the fully dynamic convex hull data structure.

The merging technique for the semidynamic sets is the real novelty of the present paper. We simulate a merging of $\log n$ sets by binary merging along a balanced binary tree of height $O(\log \log n)$. When merging two semidynamic sets we do not only reuse some information about the stored points (like the lexicographical ordering of the points as in (11)), but we continue to use the whole semidynamic data structures. We avoid storing the points simultaneously in several semidynamic binary merging data structures by extracting the current upper hull from the already existing data structures that get merged. We give more details on this process in Section 2.1, describing the geometric concepts common to merging and extracting in Section 4, and finally describe the algorithm for extracting in Section 5 and for binary merging in Section 6.

We assume that only points on the upper hull of one of the semidynamic sets get deleted. If the point we want to delete is not on the upper hull, we delay its deletion until it becomes part of the upper hull (or the complete

data structure is rebuild). This does not affect the amortized performance of the data structure, because the test for being on the upper hull of the semidynamic set is easy, since the upper hulls of the semidynamic sets are explicitly maintained.

2.1 Semidynamic binary merging data structure

We consider a data structure that supports the following operations

Definition 1 (Semidynamic Merging Structure)

CREATE_SET(p) *The point p in the plane is given by its (x, y) coordinates. Creates a set $A := \{p\}$, $\mathrm{UV}(A) := (p)$; Returns a pointer to the data structure representing A and a pointer to the representation of the point p, its* base record.

MERGE(A, B) *The sets A and B are given by a pointer to their merging data structures. 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 in a doubly linked list. The data structures representing A and B are from now on only accessible from inside the newly created data structure for C.*

DELETE(r) *Removes the point p referenced to by r from the merging-structure. Returns the list L of points that replaces p on the upper hull.*

The attribute "semidynamic" for this data structure has to be taken with a grain of salt: we can insert a point into data structure A by first creating a set B containing it, and then merging A and B. But this is not the intended use of the data structure, its efficiency relies upon the merging being along a balanced tree (as resulting from the dynamization technique), by this limiting the number of MERGE operations a point participates in.

More concretely every semidynamic set has a history of merge operations that defines the binary merging tree T in the following way. The data structure stemming form the operation CREATE_SET(p) corresponds to a leave u with the set $A_u = \{p\}$. The resulting data structure of MERGE(U, V), where U and V correspond to the nodes u and v of the merging tree, corresponds to the node w of T. The children of w are u and v, and we define $A_w = A_u \cup A_v$. At the root r of T we have the set of all (ever inserted) points of the semidynamic set, A_r.

If points get deleted they are by assumption on the upper hull of the semidynamic set, hence stored at the root node. Such a deletion is not different from an extraction of a point because it is on the upper hull of a merged set. Let R be the set of all deleted points on the overall upper hull of the semidynamic

set, not including points extracted from lower levels.

Now we describe the process of extracting the points of the upper hull from the used data structures. As part of of the MERGE(U, V) operation we determine the upper hull $L = \mathrm{UV}(A_w \setminus R)$. Then we remove the points of L respectively from U and V, leading to an update of their upper hulls, and by this possibly to deletions on data structure for the children (and recursively the descendants) of u and v.

Let D_v be the set of points that provided as the upper hull by the data structure corresponding to node v. Denote with P_v the set of all nodes of T that are ancestors of v, i.e., on the path from the parent of v to the root. Then we have

$$D_v = \mathrm{UV}(A_v \setminus (R \cup \bigcup_{u \in P_v} D_u)) \,,$$

recursively defined, starting at the root, where we have $D_r = \mathrm{UV}(A_r \setminus R)$.

For T_v denoting the set of nodes of the subtree rooted at v, we have $D_v = \mathrm{UV}(\bigcup_{u \in T_v} D_u)$, i.e., the upper hull of the points stored at nodes of T_v is the set D_v.

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

Theorem 4 (Semidynamic Binary Merging Structure)
There exists a data structure that implements the operations as described in Definition 1. Let L denote the upper hull of the The operation CREATE_SET(p) *takes $O(1)$ time. The operation* MERGE(a, b) *takes amortized time $O(|A|+|B|)$. The operation* DELETE(r) *takes amortized $O(d(r))$ time, where $d(r)$ is the depth in the merging tree of the leaf storing $\{r\}$, when r gets on the upper hull. The space usage of the data structure is linear in the number of stored points, including the space used in the data structures storing A and B.*

The complicated description of $d(r)$ stems from the delayed deletion of points that are not yet on the overall upper hull. It is no problem because the doubling techniques gives us access to the number of points n, and the logarithmic method gives rise to a binary merging tree of height $O(\log n)$.

This data structure has two components. The *geometric merging* is responsible for computing and maintaining the upper hull of the two recursive data structures. The *extractor* is responsible for maintaining the upper hulls D_v, and deleting all points on the upper hull from the merging part (and by this from the recursive data structures).

2.2 Fast queries

In order to achieve fast queries, we create an interval tree that allows simultaneous queries to all semidynamic sets. We can think of the changes to the semidynamic sets (given by deletions and the dynamization technique) as driving the interval tree, which then provides fast queries. The interval tree is easiest to explain in the dual setting. Hence we change our point of view and discuss it in the setting of a lower envelope data structure. As part of the interval tree we use secondary structures, i.e., fully dynamic upper hull/lower envelope data structures. The gain of the construction is that secondary structures store only $O(\log^4 n)$ lines. We require that insertions and queries of the secondary structures already have the aimed-at performance of $O(\log n)$, only for the deletions we get a speed up.

Theorem 5 (Speed up construction) *Let D be a nondecreasing positive function. Assume there exists a fully 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 9. We apply it twice to prove Theorem 1. For a first bootstrapping step we use a variant of Preparata's data structure with $D(n) = O(n)$, by already reusing the lexicographic order of the points. Theorem 5 yields a data structure with deletion time $D(n) = O(\log^8 n)$. In a second bootstrapping step we get a data structure with $D(n) = O(\log^{16}(\log^4 n) + \log n) = O(\log n)$, and Theorem 1 follows. We address the possiblity to use the interval-tree for other queries in Section 9.1.

2.3 Representation issues

For every input point p we create a *base record* that stores the coordinates of the point (the only real numbers used in our algorithm), and pointers into the data structures that use p, more precisely the record defines the role of p in the data structure. This is possible because our construction places p only in constantly many data structures. The benefit of this is, that passing points as arguments in a function-call or as a result can be handled with constant overhead by passing a pointer to the base record.

The coordinates of a completely deleted point might still be used in the construction to define auxiliary lines. The space usage for this is accounted for in the data structure that uses it. For this kind of usage there is no pointer from the base record to the using data structure. We delete the base record of a point only when it is no longer used in this way.

3 Finger search trees

Essential to our binary hull merges and separators are the application of finger search trees in the form of level-linked-(2,4)-trees (20). The property of (2,4)-trees we exploit is that they allow amortized constant extend operations, finger searches that are logarithmic in the distance to the finger, and splits that are logarithmic in the smaller resulting subtree. We do not use arbitrary fingers, but only a finger to the leftmost and rightmost leaf of the tree. Suspending the search is especially useful when we search for a point with a certain geometric property, and we are not sure to already have such a point. We can begin the search and suspend it as soon as we realize that the geometric situation does not allow us to decide in which direction to advance the search. Reacting to changes in the geometric situation we can later continue the search, not wasting a single comparison.

A *splitter* consists of elements drawn from a 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 a combined (atomic) 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 and the candidate is some element in this interval 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 *suspended*. If later the situation changes and we now can decide the direction (left/right) to take from the current candidate, we continue the search (by advancing it). A search is finished by executing a split operation. The user of the data structure has to ensure that this split is performed between the two guards. This allows in particular a state, where there is no further candidate because the guards are already neighbors, but the split operation is not (yet) performed. 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.

We will use splitters to hold subsets of the points stored in the upper-hull data structure, the ordering is given by the x-coordinates, no two points will have the same x-coordinate. We distinguish two major states of a splitter, depending on whether or not there is a suspended search. If there is a suspended search, the only operations allowed are to continue the search or to terminate the search by performing a split operation. Otherwise, if there is no suspended search, we can modify the set stored in the splitter (extend, shrink, and limited join of splitters which produces a splitter with a suspended search).

BUILD$(e_1, \ldots, e_k)$ Returns a new splitter containing the elements $e_1, \ldots, e_k$, with no suspended search.

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 $e_1, \ldots, e_k, e$. The splitter is required to not have a suspended search.

SHRINK_LEFT/RIGHT(S) The splitter S is changed by deleting its leftmost (respectively rightmost) element The splitter is required to not have a suspended search.

INSTANTIATE_SUSPENDED_SEARCH(S) We start a new search that is suspended at the first comparison step. The guard pointers of S are set to `nil`, the candidate pointer is set to an element c stored at the root-node of the (2,4)-tree. In particular it is not necessary to supply this function call with an element of the universe to search for. We use the convention that a `nil`-pointer for a guard stands for no restriction on the possible split point in this direction. The splitter must not have a suspended search before using this function, but it has afterward.

ADVANCE_SUSPENDED_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 suspended search (and by this as possible split-points). If the guards become neighbors, there is no new candidate, but the search remains suspended. The splitter is required to have a suspended search, and there is a candidate, i.e. the guards are not neighbors.

SPLIT(S, w) The splitter S is split into two splitters S_1 and S_2 according to the flag w, which is either *left guard*, *candidate*, or *right guard*. The split point is not part of S_1 or S_2, unless the guards are neighbors. This operation finishes a suspended search of the splitter.

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 a suspended 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 $e_1, \ldots, e_k$. It is required that none of the participating splitters S_1 and S_2 has a suspended search.

We do not implement the JOIN operation as a join of the (2,4)-trees. Instead we perform a delayed extension of S_1 and S_2. Instantiating the suspended 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 merely place $e_1, \ldots, e_k$ in an auxiliary balanced search tree (or an array) and use this to guide the suspended search. Not until this search is settled with a SPLIT operation, we EXTEND S_1 and S_2 with the elements left (and respectively 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 SUSPENDED SEARCH, EXTEND, SHRINK, *and* SPLIT *take amortized $O(1)$ time.*
- *The operation* ADVANCE SUSPENDED 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 (20), 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 $\Omega(c \cdot \log \min\{n_1, n_2\})$ potential, achieving the amortized $O(1)$ split operation, including the additional (negative) potential $\Theta(c)$ when advancing the search. □

The splitter is an interesting data structure, even if we never suspend a search. Then we can use it to cut a list into smaller and smaller pieces, determining every cut-point by a search. The performance of the splitter guarantees that the overall time used is linear in the length of the list we started with.

4 Geometric merging

In this section we describe the geometric binary merging data structure. First we show how different subproblems come into existence and how to combine

them into a solution of the problem. Then we describe the details of the solutions to the subproblems.

Let A and B be two sets of points in the plane. Assume that we want to compute $\mathrm{UV}(A \cup B)$ given that we already have $\mathrm{UV}(A)$ and $\mathrm{UV}(B)$. More precisely we want a data structure that gives a list of the points $\mathrm{UV}(A \cup B)$ in left-to-right order. This list is maintained under deletions of points from $\mathrm{UV}(A \cup B)$, under the assumption that we have two data structures maintaining $\mathrm{UV}(A)$ and respectively $\mathrm{UV}(B)$ as two lists of points.

This situation naturally leads to the following life-cycle of a point $p \in A$: (see Figure 6)

(1) p is inside A, i.e. not part of $\mathrm{UV}(A)$.
(2) p becomes part of $\mathrm{UV}(A)$, and $p \in \mathrm{UC}(B)$.
(3) Delete on B: $p \notin \mathrm{UC}(B)$, but $p \in \mathrm{UC}_0(A \cup B)$: p is hidden by a *bridge*.
(4) p gets on $UV(C)$.
(5) p gets deleted.

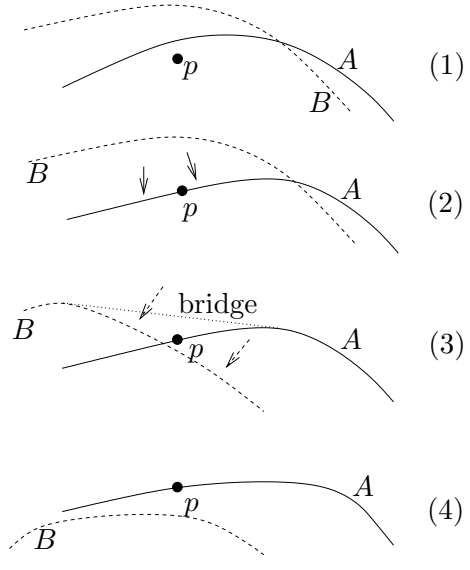

Fig. 6. The different stages of the life-cycle of a point in the merging.

To solve our algorithmic problem, it is sufficient to keep track of the stage in the life-cycle for every point in A and B. We have this strict life-cycle because we only allow deletions of points on the overall upper hull. The data structure maintains a partition of the upper hulls into stretches of points in the same stage of the life-cycle. It turns out, that the bridge finding is an independent problem. We will address it in Section 6.7.

The main difference for points of $\mathrm{UV}(A)$ is whether or not they are inside $\mathrm{UC}(B)$, i.e., we should distinguish between points in stage (2) of their life-cycle and points in stage (3). Theses stretches to be distinguished are sep-

arated by the *equality points*, i.e., the intersection points of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$. The general position assumption guarantees that $\mathrm{Bd}(A) \cap \mathrm{Bd}(B)$ consists of isolated points.

Our algorithmic solution is to keep track of all equality points by making them explicit in the data structure. This amounts to detect possible changes to equality points during deletions, and to adjust the data structure accordingly. The possible changes are moving equality points, and pairs of equality points that come into existence or disappear. Given that equality points are represented explicitly, it is easy to detect if equality points move or disappear. The more complicated task, and hence the focus of our algorithms, is to detect if a pair of new equality points comes into existence (this can be more than one pair per deletion). Our data structure can be seen as a construction that allows us to efficiently detect such a formation of equality points, without being too inefficient in the cases of moving and disappearing equality points.

For an extractore we define the set B to be the points on the upper hull, and the set A the other points, stored further down in the merging tree. There we do not have equality points (and no recursive data structure for B), the extractor itself has to identify the points to move from A to B. In this situation the life-cycle of a point is reduced to the stages (2) and (4).

To achieve a correct geometric construction, it is often helpful to keep a geometric proof of correctness. In our case we make explicit several auxiliary points that basically constitute a geometric proof that there can not be equality points that our construction did not identify. This (the auxiliary points and lines) is what we call a *certificate*. One such certificate addresses the range between two equality point, which we call a *streak*. Such a streak has a *polarity*

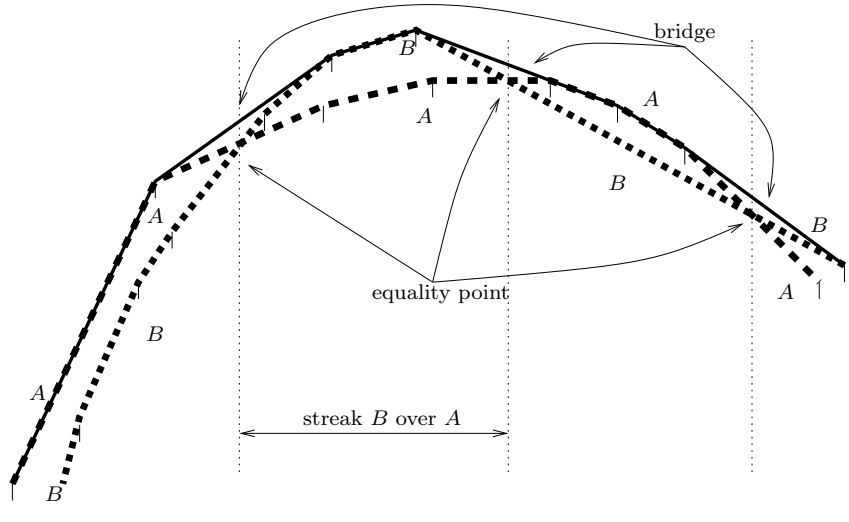

Fig. 7. The task of merging two hulls boils down to identifying equality points and bridges.

(as illustrated in Figure 7), either $\mathrm{Bd}(A)$ is above $\mathrm{Bd}(B)$ inside the streak,

or vice versa. This leads to the notion of the *locally inside* and *locally outside* upper hull. The certificate we maintain is based on a sampling of points on the locally inner hull, $\mathrm{Bd}(A)$, the so called *selected points*. We introduce one more stage in the life-cycle of a point between (2) and (3):

(2′) A point on $p \in \mathrm{UV}(A)$ can get (and stay) selected as long as it is inside $\mathrm{Bd}(B)$, i.e., once a point $p \in \mathrm{UV}(A)$ gets selected it stays selected until $\mathrm{Bd}(B)$ has changed "below" p.

4.1 Geometric concepts at a point: Rays, certificates

For every point $p \in \mathrm{UV}(A)$ we define the concept of a *valid pair of rays* in the following way: Let S and T be two tangent lines on $\mathrm{Bd}(A)$ through p, where the slope of S is smaller than the slope of T. Let h be a line through p that has a slope between the slopes of S and T, and let H be the closed half-plane above h. Then the two rays $s = S \cap H$ and $t = T \cap H$ form a valid pair of rays *rooted* at p. We say that s is left-directed (all of it is to the left of p) and t is right-directed ray (all of it is to the right of p). The noteworthy property of such a valid pair (s, t) of rays is, that any line that intersects both s and t, does not intersect $\mathrm{UC}_0(A)$. If S and T contain the two segments of $\mathrm{Bd}(A)$ that are adjacent to p we call this the *canonical pair of rays* rooted at p. (This achieves the smallest allowed angle between s and t.) See Figure 8 for an illustration.

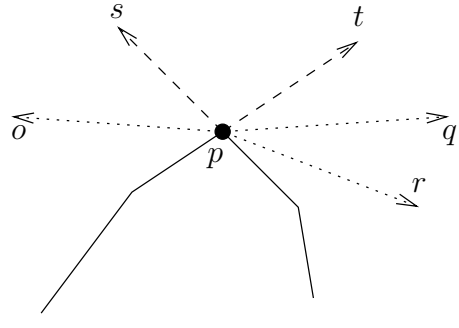

Fig. 8. Valid rays rooted at the point p: The canonical pair of rays is (s, t). The pair (o, q) is a valid pair of rays, and so is (s, r), (o, t), and (s, q). The pair (o, r) is not valid, a line that intersects o and r is below p.

4.1.1 Strong certificates

Our data structure maintains a set $Q_A \subseteq \mathrm{UV}(A) \cap \mathrm{UC}_0(B)$ of *selected points*. Symmetrically there is $Q_B \subseteq \mathrm{UV}(B) \cap \mathrm{UC}_0(A)$. We will introduce some conditions and invariants we maintain for these sets. For each selected point $p \in Q_A$ we decide upon a particular valid pair of rays (s_p, t_p), its *strong rays*, collected in the set R_A. When p gets selected we take the canonical pair of rays to

be (s_p, t_p). We do not change (s_p, t_p), even if they no longer are the canonical pair of rays rooted at p. This happens if a neighbor of p on $\mathrm{Bd}(A)$ gets deleted.

For all $p \in Q_A$ the data structure maintains the intersections of s_p and t_p with $\mathrm{Bd}(B)$ explicitly. We call these two intersection points u and v *strong ray intersections*. By the geometry of valid rays, we are sure that there is no equality point of $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$ (in the vertical slab) between u and v. See Figure 9 for an illustration. We call p, u, v a *strong (separation) certificate* that extents (horizontally) from u to v. Besides the equality points, this kind of intersections are the only points of the locally outer hull $\mathrm{Bd}(B)$ that are explicitly set in relation with the locally inner hull $\mathrm{Bd}(A)$.

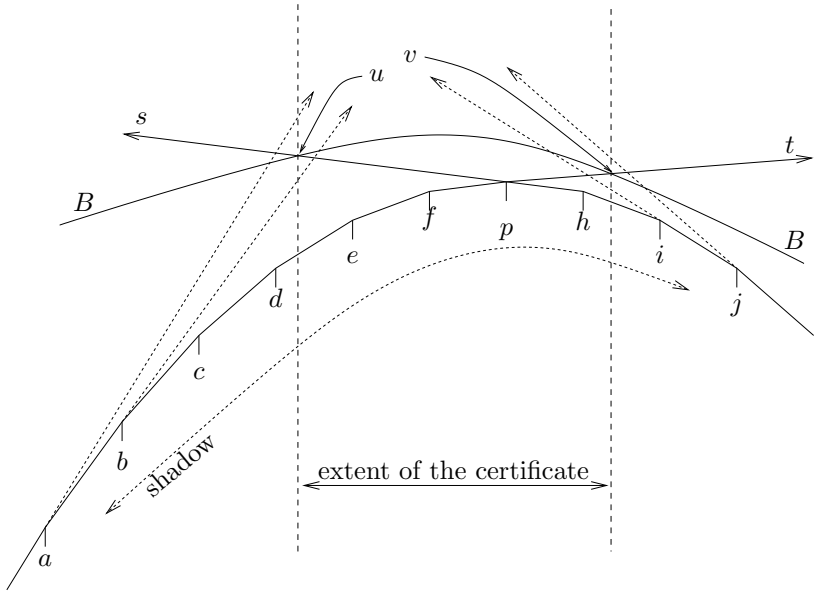

Fig. 9. The shadow around the selected point p, given by the strong rays s and t (t is not canonical). The points b–i are in the shadow of p, the points a and j are not.

The most important invariant about the set Q_A of selected points is, that we require that two strong separation certificates do not overlap, i.e., that they are horizontally disjoint. We say that the two strong certificates enjoy the *disjointness condition*. For a selected point $p \in \mathrm{UV}(A)$ this requirement disallows the selection of several other points of $\mathrm{UV}(A)$, a range in the left-to-right ordering of $\mathrm{UV}(A)$ around p, the *shadow* of p, as illustrated in Figure 9. This policy makes sure that we do not select too many points, but it also introduces a gap between any two consecutive strong separation certificates.

As we will only consider dynamic aspects of the construction later, here a parenthesis on the dynamic of certificates and shadows. Deletions on $\mathrm{UV}(B)$ move the strong ray intersections u and v closer to p, the horizontal extent of the separation certificate shrinks, so does the shadow of p.

4.1.2 Light separation certificates

Consider the gap between two neighboring two strong separation certificates, manifested by the two selected points $p, q \in Q_A$, wher p is to the left of q, such that no point in the slab between p and q is selected (is in Q_A). Let f be the right-directed strong ray rooted at p with strong ray intersection v, and e be the left directed strong ray rooted at q with strong ray intersection u. The situation is illustrated in Figure 10. The disjointness condition states that v is left of u.

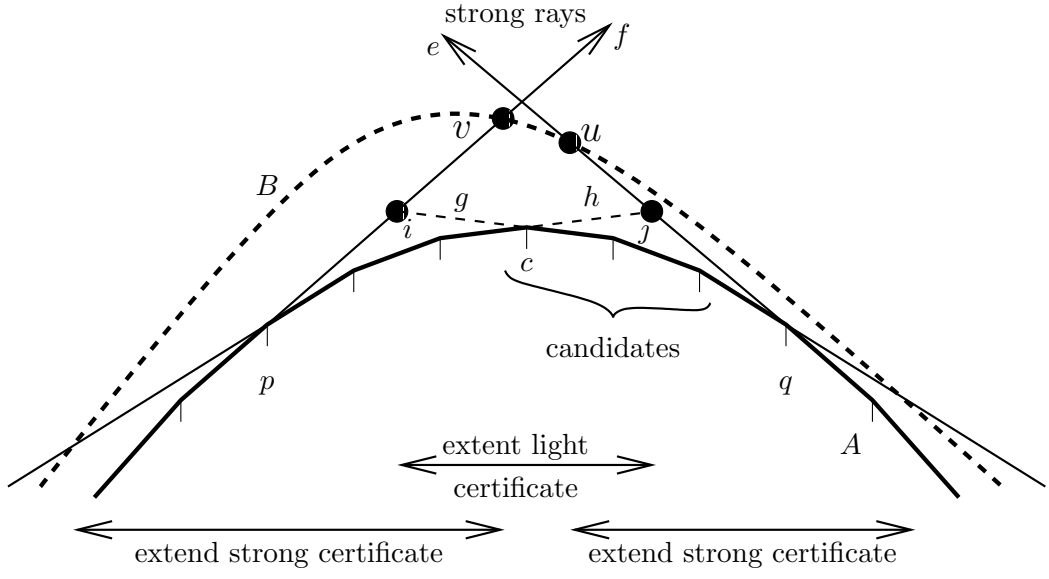

Fig. 10. A light certificate. Hull A is below hull B (depicted as a curve, it is here not important that it is a polygon). The points p and q are selected, their strong rays are depicted as a solid line. The gap between the strong certificates is between u and v. The point c is a candidate, its weak rays (g, h) are depicted as dashed lines. The light certificate extents from i to j, thus covering the gap.

To close the gap in separation certificate that is between u and v, we use a another type of certificate that is based on canonical rays. For a point $c \in \mathrm{UV}(A)$ we consider the pair of canonical rays g and h, here used as *weak rays*. For g and h we do not determine the intersection with $\mathrm{Bd}(B)$, but we only check if v is above g (as a line) and u is above h. If this is the case, we conclude that the intersections with $\mathrm{Bd}(B)$ are further away from c than v and u respectively, and that there cannot be an equality point between u and v. The valid *light certificate* (c, g, h) closes the gap between the two neighboring strong certificates of p and q.

Usually there will be more than one point $c \in \mathrm{UV}(A)$, that has a light certificate that closes the gap, we are free to choose any such point. More precisely, any point that is simultaneously in the shadow of p and of q defines a valid light certificate. Another (somehow degenerate) possibility is, that the two

shadows touch precisely, i.e., that there is no point between the shadows, but also no point in the intersection of the shadows. For this situation we use a special version of a light certificate. Let a be the rightmost point of the left shadow, and b the leftmost point of the right shadow. By assumption is $\overline{ab}$ a segment of $\mathrm{Bd}(A)$. Instead of a pair of canonical rays we use the line defined by $\overline{ab}$. Because a and b are in their respective shadows, this line intersects the next strong rays inside of $\mathrm{Bd}(B)$. Under the non-degeneracy assumption, this forms also a valid light certificate.

The third possibility is, that the shadows do not overlap. Then there cannot be a light certificate. Instead there is a point that is allowed to be selected because its strong certificate (using canonical rays) will not overlap the existing strong certificates. We call such a point a *selectable point*.

As we will come to the data structure aspects only later, here a small parenthesis on the connection to the splitter. We store the not-selected points of $\mathrm{UV}(A)$ between the selected points p and q in a splitter. If the candidate of the suspended search defines a valid light certificate, we leave the search suspended. If the deletion of a point in B shrinks the shadows further and the light certificate is no longer valid, we do not start the search over, but merely advance it. Instead of leading to a new valid light certificate, the search might lead us to a selectable point that is allowed to be selected (outside of the shadows). In this case we end the suspended search with a split operation and select the point. This use of the splitter is of course the motivation for the otherwise somewhat unusual interface of the splitter, the way to search for a next point to select needs only overall constant time per point in A. In this way every splitter has two strong rays and strong ray intersections that guide its suspended search, in Figure 10 the strong rays e and f.

4.1.3 Boundary certificates

A strong certificate usually does not extend to an equality point. To achieve a complete certificate, we introduce the so called *boundary certificate*. Let e be an equality point and $p \in \mathrm{UV}(A)$ a selected point, such that there is no selected point between e and p. Then we take a weak ray h rooted at e towards p. Being a tangent on $\mathrm{Bd}(A)$ the line of h is given by the segment containing e. If h intersects the strong ray rooted at p inside $\mathrm{Bd}(B)$, there cannot be another equality point between e and p. Additionally we know that the shadow of p reaches over e, and we cannot select any of the points on $\mathrm{UV}(A)$ between p and e. Otherwise, if there is no valid boundary certificate, the point i is selectable, if it gets selected there is a trivially valid boundary certificate between e and i.

We also have to address the case if p is the rightmost (or symmetrically left-

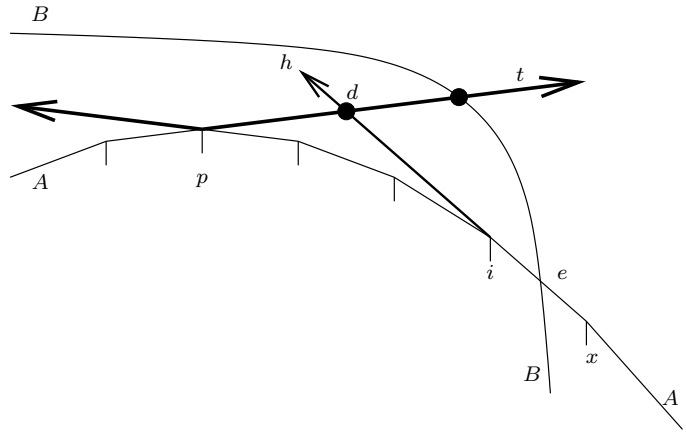

Fig. 11. A boundary certificate: p is the rightmost selected point in the streak, e is the equality point, t the relevant strong ray, and i is the point defining the direction of the weak ray. The intersection d of t and the weak ray h shows that the boundary certificate is valid.

most) selected point of $\mathrm{UV}(A)$. If the rightmost point r_A of A is further to the right than the rightmost point r_B of B, we have an ordinary boundary certificate (only that one of the segments is the special rightmost vertical segment of B). Otherwise (r_B right of r_A) we have a boundary certificate that shows that there can not be another equality point between p and r_A. We take the vertical upward ray h rooted at r_A as a light ray. If h intersects the strong ray t at p inside $\mathrm{Bd}(B)$, this certificate is valid. This is actually not really a special case since h can be seen as the left-directed canonical ray at r_A (also if r_A is selected). The situation can be understood as an ordinary boundary certificate if we define that the vertical segment of $\mathrm{Bd}(A)$ at r_A intersects the vertical segment of $\mathrm{Bd}(B)$ at r_B at an downward infinity point. We further define that such a vertical boundary certificate extends all the way to right-infinity. In the case r_A right of r_B (first case), this definition leads to a *trivial streak* (locally lower hull consists of one segment only) that extends to right-infinity. This way a complete certificate always horizontally covers everything from left-infinity to right-infinity.

4.2 Shortcuts

We simplify the locally outer hull by applying *shortcuts*. This reduces work and unifies the description of the algorithm in the context of finding strong ray intersections.

We consider a non-vertical line l and the half-plane H below l and replace in our considerations $\mathrm{UC}(B)$ with $\mathrm{UC}(B) \cap H$. This situation is exemplified in Figure 12. For a line l we determine the line-segment $l \cap \mathrm{UC}(B)$, the *shortcut*. The shortcut replaces the part of $\mathrm{Bd}(B)$ that is above l. The two *cutting points*

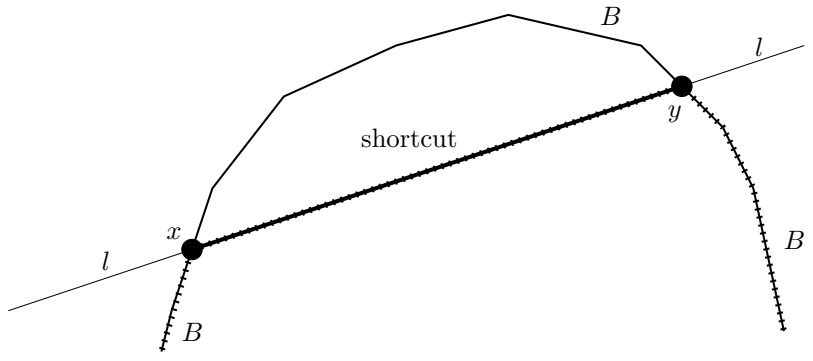

Fig. 12. A shortcut defined by the line l with cutting points x and y. The resulting shortcut version B' of B is indicated as an additional doted polygon.

of $l \cap \mathrm{Bd}(B)$ replace all the points of B above l.

We use shortcuts to simplify the locally outer hull between points that are explicitely set in relation with the locally inner hull. They allow us to reduce the number of segments on the locally outer hull between points that are connected to the inner hull, i.e. strong ray intersections and equality points. Let u and v two neigboring such points. Then there always is a (strong, light or boundary) certificate, that ensures that the line connecting u and v is outside the locally inner hull. As discussed algorithmically later, we use a shortcut "close" to $\overline{uv}$, close enough that the shortcut version of the locally outer hull has only a constant number of segments between u and v.

We will have many shortcuts, for which we make sure that they do not *overlap*, i.e., they are all (horizontally) disjoint, the intersection point of two shortcut defining lines is outside of $\mathrm{UC}_0(B)$.

For a set H of non-vertical lines that intersect $\mathrm{UC}_0(B)$ we define the shortcut version of B to be the set of points (usually called B')

$$\mathrm{SC}_H(B) = \Big(\mathrm{UV}(B) \cap \bigcap_{l \in H} h_l\Big) \cup \bigcup_{l \in H} (l \cap \mathrm{Bd}(B)) \ ,$$

where h_l denotes the half-space below the line l. We will only use sets H of shortcuts that do not introduce new equality points with $\mathrm{Bd}(A)$, i.e. $\mathrm{Bd}(A) \cap \mathrm{Bd}(B) = \mathrm{Bd}(A) \cap \mathrm{Bd}(\mathrm{SC}_H(B))$, i.e., all the introduced segments (shortcuts) are above $\mathrm{Bd}(A)$. We also use only lines that are defined by two input points (possibly already deleted).

4.3 The complete certificate

Summarizing the just explained concepts, we show how a complete certificate looks like, and how it is represented in the data structure. This certificate

provides a proof that all the equality points are made explicit. All the sets referred to in the next requirement are made explicit in the data structure as explained in Section 4.6. Note that all the conditions in this certificate are local, they can be verified by only considering a constant number of input and auxiliary points and lines.

All together this has the flavor of an induction hypothesis or a loop-invariant. The requirement describes the state of the data structure before we process the deletion of a point, and the state we hence should reach by reacting to the the deletion.

Definition 2 (Complete Certificate)
The tuple $C = (A, B, E, Q_A, Q_B, R_A, R_B, D_A, D_B, L_A, L_B, H_A, H_B, C_A, C_B)$, *where* A *and* B *are two finite sets of points in the plane, and we have the set of identified equality points* E, *sets of selected points* $Q_A \subseteq \mathrm{UV}(A)$ *and* $Q_B \subseteq \mathrm{UV}(B)$, *sets of strong rays* R_A *and* R_B, *the sets* D_A *and* D_B *of strong ray intersections, the sets of candidates (light certificates)* $L_A \subseteq \mathrm{UV}(A)$ *and* $L_B \subseteq \mathrm{UV}(B)$, *the sets of shortcuts* H_A *and* H_B, *and the sets of cutting points* C_A *and* C_B. C *forms a complete certificate if the following conditions are met:*

(1) $E \subseteq \mathrm{Bd}(A) \cap \mathrm{Bd}(B)$, *and two neighboring equality points in* E *define the same polarity for the streak between them.*
(2) The cutting points stem from the shortcuts, $C_B = \mathrm{Bd}(B) \cap \bigcup_{l \in H_B} l$.
(3) All shortcuts $l \in H_B$ *are* effective, $l \cap \mathrm{UC}_0(B) \neq \emptyset$ *(*$|l \cap C_B| = 2$*).*
(4) All shortcuts $l \in H_B$ *are* conservative, *i.e., l does not introduce an equality point with* $\mathrm{Bd}(A)$, $l \cap \mathrm{UC}_0(B) \cap \mathrm{UC}(A) = \emptyset$.
(5) Shortcuts $l \in H_B$ *and* $f \in H_B$ *are* disjoint, $l \cap f \cap \mathrm{UC}_0(B) = \emptyset$ *(order on the cutting points).*
(6) R_A *forms valid pairs of rays on* $\mathrm{Bd}(A)$ *and these pairs of rays are rooted at the points of* Q_A, *and form the strong ray intersections* D_A *with* $\mathrm{Bd}(\mathrm{SC}_{H_B}(B))$.
(7) $Q_A \subset \mathrm{UC}_0(B)$ *and* $Q_B \subset \mathrm{UC}_0(A)$, *as proved by the strong ray intersections* D_A.
(8) Let s *be a consecutive sequence of segments from* $\mathrm{Bd}(\mathrm{SC}_{H_B}(B))$, *such that* s *does not intersect any strong ray of* R_A *or* $\mathrm{Bd}(B)$. *Then* s *consists of at most 3 segments.*
(9) strong certificates (implied by R_A *and* D_A*) do not overlap horizontally.*

For any streak of polarity A *below* B *between the identified equality points* $u, v \in E$ *we have:*

(10) If there is a point of $\mathrm{UV}(A)$ *between* u *and* v, *then some point of* $\mathrm{UV}(A)$ *between* u *and* v *is selected.*
(11) Between two strong certificates (selected points) there is a point $c \in L_A$ *that forms valid light certificate.*
(12) Between u *(*v*) and the leftmost (rightmost) selected point* $p \in \mathrm{UV}(A)$ *is*

a valid boundary certificate.

All the above conditions are also valid with A and B interchanged.

By the nature of a separation certificate and the fact that all certificates together (strong, light and boundary) cover everything from left to right except the equality points in E, we are sure to have identified all equality points.

A streak of the certificate, that has as the locally inner hull only one segment, is called *trivial streak*.

4.4 Limited impact

Because we will consider deletions of a point $r \in \mathrm{UV}(B)$, $r \notin \mathrm{UC}_0(A)$ it is useful to collect some properties of a complete certificate around such a point.

Lemma 7
Let $C = (A, B, E, Q_A, Q_B, R_A, R_B, D_A, D_B, L_A, L_B, H_A, H_B, C_A, C_B)$ be a complete certificate, and let $\overline{xr}$ and $\overline{ry}$ be two consecutive segments of $\mathrm{Bd}(B)$.

Then $\overline{xr}$ and $\overline{ry}$ intersect at most 6 strong rays of R_A, and these strong rays are rooted at no more than 4 selected points of Q_A.
Let $H \subseteq H_B$ be shortcuts intersecting $\overline{xr}$ and $\overline{ry}$, then $|H| \leq 3$ and at most 7 selected points and 12 strong rays intersect segments of $H \cup \{\overline{xr}, \overline{ry}\}$

PROOF. A worst case situation for two segments is depicted in Figure 13. Because of the strong ray separation, it is impossible that 4 consecutive strong rays of A intersect a segment of $\mathrm{Bd}(B)$. With the names given in Figure 13 we have that the slope of strong ray 0 must be larger than (or equal) that of strong ray 3. Because ray 3 is right-directed and intersects $\overline{xr}$, it is impossible that the left-directed ray 0 also intersects $\overline{xr}$. The bound on the number of selected points stems from the strong ray intersections following the order of the strong rays. Each of $\overline{xr}$ and $\overline{ry}$ intersects at most two different shortcuts, one of these possibly 4 shortcuts is shared, $|H| \leq 3$. In the worst case we look at two pairs of segments (real segment and shortcut) as just argued for. (The third possible shortcut between $\overline{xr}$ and $\overline{ry}$ cannot contribute.) □

Note that two consecutive segments $\overline{xr}$ and $\overline{ry}$ of $\mathrm{Bd}(B)$ can have at most 4 intersections with $\mathrm{Bd}(A)$ (equality points).

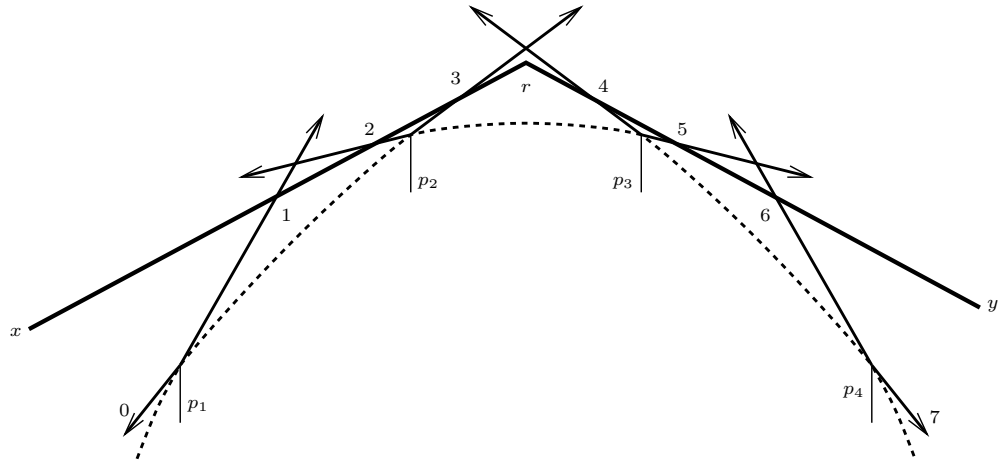

Fig. 13. A worst-case of 6 strong rays intersecting $\overline{xr}$ and $\overline{ry}$, rooted at 4 selected points.

4.5 The generic algorithm

Assume we are in the situation that we identified a pair of equality points, and know that there are no further equality points in this streak. This is for example the case when we initialize an extractor or a merging data structure. Even if this sounds like a somewhat boring situation (after all we already know all the equality points), this is the generic algorithm when (re-)establishing a separation certificate. The other algorithms are really only extensions (with small modifications) of this algorithm.

We start by describing the algorithm in high-level pseudo-code. Then we explain the underlying data structures and give some more details and explanations, especially on a typical run.

In general (if the new streak has space for more than one strong certificate) the algorithm will start by selecting the leftmost point p and the rightmost point q of $\mathrm{UV}(A)$ in the new streak. Then the point of $\mathrm{UV}(A)$ between p and q are stored in a lasting splitter M, the points of $\mathrm{UV}(B)$ between t_p and s_q are stored in one temporary splitter T_M. This is when "the loop" really starts, the search facility of M either leads to a next point r to select, creating two copies of the situation, one between p and r, and another between r and q. Alternatively it might be that the search stays suspended because we found a valid light certificate between the strong certificates of p and q.

To advance the search on M, we determine for a candidate point $c \in \mathrm{UV}(A)$ whether or not it is in the shadow of p, and also whether or not it is in the shadow of q. If it is in both shadows, the light certificate of c is valid. If in contrast it is in neither of the shadows, the point c is selectable.

Generic algorithm "Establish certificate between two equality points"

Assume: Bd(A) below Bd(B) between equality points e and f
Assume: there is a lasting splitter L for UV(A) between e and f
Assume: there is a temporary splitter T for UV(B) between e and f
while(there is a lasting splitter M not defining a valid light/boundary certificate)
  advance suspended search trying to adjust light certificate
  if(selectable point $p \in$ UV(A) found)
    then select p
      split M into M_l and M_r at p
      determine strong rays t_p and s_p as canonical rays
      search for $s_p \cap$ Bd(B) splitting T_M into T_{M_l} and T
      search for $t_p \cap$ Bd(B) splitting T into X and T_{M_r}
      add to H_B the shortcut above t_p and s_p (points in X)
  if(candidate of suspended search on M defines valid light certificate)
    add to H_B the shortcut above M

If it is in the shadow of p, but not in the shadow of q, we advance the search to the right. The point c becomes the new left guard of the (suspended) search of M. Such a left guard of the (suspended) search is also geometrically interesting, it is a point not in the shadow of q (also not in the future since the shadow of q does not extend) where we are allowed to split the splitter if we want to finish the search.

4.5.1 Introducing shortcuts

When we create a shortcut, we usually have two strong-ray intersections u and v on two segments a and b of the shortcut version Bd(B') of B. (Otherwise u or v or both can be an equality point.) In general it is possible that the strong ray intersections are on shortcuts. The situation is illustrated in Figure 14. If a and b are adjacent or there are only three segments between them, there is no need for a shortcut. We define x to be the right endpoint of a if a is an segment of Bd(B), or otherwise the leftmost point of UV(B) that is to the right of the shortcut a. We define y symmetrically as the rightmost point of UV(B) to the left of b. We take the line l defined by x and y as the new shortcut.

This new shortcut might overlap with existing ones, but none of these can intersect a strong ray (they are completely between a and b). To achieve shortcut separation, we hence delete all these interfering shortcuts. More precisely we scan Bd(B') above l, marking segments of UV(B) as hidden by the shortcut, and deleting existing shortcuts (but still using them to scan fast, segments that were behind a shortcut need not be visited). Then we instantiate the new shortcut. This achieves the aggressive shortcut policy. It also complies

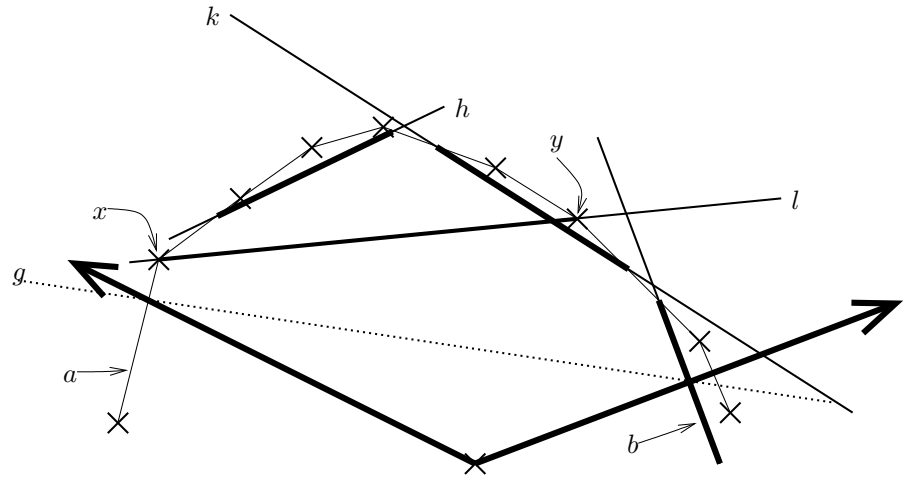

Fig. 14. The situation of creating a new shortcut. The naive and most aggressive choice g is forbidden by shortcut separation. The line l provides a shortcut that results in monotonic strong ray intersections and is defined directly by two input points. We delete the shortcuts h and k.

with the principle, that strong ray intersections only get closer to the selected point, because none of the deleted shortcuts defines a strong ray intersection.

4.6 Representation issues

Our data structure introduces several types of auxiliary points:

- Equality points on both hulls,
- strong ray intersections, and
- cutting points on the locally outer hull.

The equality points and the strong certificates connect the two hulls. This connection is made explicit by pointers. On the locally outer hull, shortcuts achieve that strong ray intersections (from each other) and equality points are only a constant number of segments apart. On the locally inner hull, the points between two selected points are stored in *lasting splitters* (with a suspended search to define a valid light certificate). Between an equality point and a selected point the points are stored in a lasting splitter, that has no active suspended search.

To allow the necessary navigation, all (auxiliary) points have a record, identifying the coordinates of the point by pointers to the base-record of the input points necessary to define them. We should not delete base-records that are referenced to in this way. Alternatively we could store the coordinates, but we prefer not to copy real numbers.

At every (auxiliary) point we have pointers to the following objects:

- To the strong ray intersections if the point is selected
- To the selected point from the strong ray intersections
- Between selected points and adjacent splitter
- To the next point to the left / right (auxiliary or not)
- From one cutting point to the other cutting point and to the shortcut record (and back)
- Between equality point and adjacent splitter, both directions

We also have a flag that indicates if the point is cut away by a shortcut.

Our construction with extractors achieves that every point is stored and used only in constantly many data structures. Hence we can store the mentioned pointers directly in the base record of the point. This allows us to simply pass a pointer to the base record if we want to pass a point between merging data structures and extractors.

4.7 data structure in the re-establishing: temporary splitter

One of the main ingredients to the data structure and the generic algorithm is to connect the locally outer and locally inner hull, not only at the already identified equality points, but also at the strong certificates. In the data structure we reduce the locally outer hull to less than 4 segments by introducing shortcuts. In the generic algorithm we store the points on the locally outer hull in a piece of the temporary splitter. We will stick to this principle of having a lasting splitter on one side matched with either only a constant number of points, or a temporary splitter.

4.8 Preselected points

The process of a selecting a point is a two-stage process: First we find a selectable point, usually without knowing the precise extent of its strong certificate precisely we have evidence that it will not overlap already existing strong certificates. We call such a point preselected, data structure wise we have to select this point to pay for the search that lead us to it. Then we instantiate the strong rays and strong ray intersections, making the new strong certificate explicit.

In some of the algorithms of Sections 5 and 6 to re-establish a complete certificate, we will identify preselected points, and defer the instantiation of strong rays to a later stage, for example because we do not yet have a good repre-

sentation of the other upper hull, a temporary splitter that would allow us to determine the strong ray intersections.

When we (later) select such a preselected point $p \in A$, we are in the situation that the shortcut version of the other hull is represented in a temporary splitter (or is only a constant sized part). We use the temporary splitter to find the segment of $\mathrm{Bd}(B')$ that is intersecting the vertical line through p. From there we can perform a scan to find the intersections with the canonical rays at p, now used as strong rays. We instantiate a shortcut above the new strong rays.

We will also see the situation that a preselected point turns out to be already in stage (3) of its life-cycle. Then we will make sure that the point gets its proper role in the new streak of opposite polarity.

4.9 Deselecting points

Assume we have a point $p \in \mathrm{UV}(A)$ that is selected, but after a deletion on $\mathrm{UV}(B)$ we have $p \notin \mathrm{UC}_0(B)$. In this situation p does no longer give rise to a strong certificate, and we have to reflect this fact in the data structure. One effect is that p should not be the boundary of a lasting splitter, and we should shrink the lasting splitter and remove points that surface just like p. The splitter is only allowed to shrink if it has no suspended search (in geometry the concept of a light certificate is also void, there is no gap between strong certificates).

Let us focus on the lasting splitter M to the left of p. We pre-select the right guard g of M. (If g is `nil`, we can just finish the suspended search, there have only been $O(1)$ advances of the search) This is an allowed splitting point of M, so it is fine for the data structure and accounting. Additionally we know a-priori (without even determining strong rays or anything), that the strong certificate of g (if it exists at all) does not overlap with any other strong certificate, as we will argue now, we are allowed to pre-select it. Let q be the selected point to the left of M (M is between q and p). By being the right guard, g is not in the shadow of q, and hence a strong certificate at g will not overlap the strong certificate at q. Additionally no strong certificate of A below B can extend over p, as we have there A over B (or at least equality). Hence the certificate of g cannot overlap with another strong certificate to the right. If instead g is already in stage (3) in its life-cycle, no harm is done by pre-selecting it.

4.10 Accounting for the generic algorithm

The search for valid light certificates or selectable points is paid for by the splitter. The selection of one point takes constant time, which is accounted for by the life-cycle of the point (it can be selected only once). We introduce shortcuts only after selecting a point, the scanning over the segments is again paid by the life-cycle.

5 Linear space: Extractors

As already explained in Section 2.1, we initialize an extractor as the last step of merging two semidynamic sets. We copy the list of upper vertices into a representation of B, and then delete the vertices in B (recursively) from A. Finally we use the generic algorithm (from Section 4.5) to create a complete certificate. A separator has by construction only one streak of polarity B over A, and no actual equality points (we only have the two equality points at infinity). A complete certificate consists only of selected points and strong rays on A, and shortcuts only on B, the sets E, Q_B, R_B and H_A are empty.

5.1 Re-establishing an extractor

In the following we discuss the situation where the set of points C, of which the extractor makes the upper vertices explicit, changes from C_1 to C_2 by the deletion of the point $r \in \mathrm{UV}(C_1)$. The purpose of the extractor is to maintain the partition of C into $B = \mathrm{UV}(C)$ and $A = C \setminus B$. The data structure represents $B_1 = \mathrm{UV}(C_1)$ and $A_1 = C_1 \setminus B_1$ and a complete certificate as described in Section 4.3 (Definition 2) between them. The algorithmic task is to react to the deletion of $r \in B_1$ by determining $B_2 = \mathrm{UV}(C_2)$ and $A_2 = A_1 \setminus B_2$, and to construct (repair) the complete certificate between A_2 and B_2.

We observe some geometric properties and give the following names: The change from B_1 to B_2—as lists representing $\mathrm{UV}(C_1)$ and $\mathrm{UV}(C_2)$—is that r gets replaced with a (possibly empty) list $L \subseteq \mathrm{UV}(A_1)$ of points. Let x be the left neighbor of r on $\mathrm{UV}(B_1)$, y its right neighbor. We define $\hat{B}_1 = B_1 \setminus \{r\}$. The difference between $\mathrm{Bd}(B_1)$ and $\mathrm{Bd}(\hat{B}_1)$ is, that the triangle xry is replaced by the segment $\overline{xy}$. Let u be the point of $\mathrm{Bd}(A_1)$ that has a tangent (on A_1) through x and is right of x, symmetrically let $v \in \mathrm{Bd}(A_1)$ be left of y and have a tangent through y. Then the list L consists of points of $\mathrm{UV}(A_1)$ between u and v, as for example in Figure 17. Alternatively, L is empty if $\overline{xy}$

is above $\mathrm{Bd}(A_1)$. We say that the points of L are *surfacing*. On $\mathrm{UV}(A_2)$ the list L is replaced by a consecutive list of points J, defined by deleting the points of L from A_1.

It is possible that r is the righmost (or symmetrically the leftmost) point of B. If x is the rightmost point of C_2 we set $L = \emptyset$ and only need to adjust the boundary certificate (generic algorithm). Otherwise the rightmost point of A_1 surfaces, we determine L by a right-to-left scan of $\mathrm{UV}(A_2)$, and use the generic algorithm to complete the certificate (after shrinking a lasting splitter and possibly deselecting some points of Q_A).

We can check whether a point $p \in \mathrm{UV}(A_1)$ is surfacing by examining only its neighbors u and v on $\mathrm{UV}(A_1)$ and x and y: If and only if p is to the right of x and the left of y and $p \in \mathrm{UV}(\{p, x, y, u, v\})$, we have $p \in L$.

5.2 The algorithm

Our approach is to use the existing complete certificate between A_1 and B_1 to identify L. Once we know L, the sets B_2 and A_2 are defined, and we can access $\mathrm{UV}(A_2)$ using the recursive data structure. We build a complete certificate between A_2 and B_2. This process is referred to as *re-establishing* the complete certificate, because we can re-use large portions of it.

Let $R \subseteq R_A$ be the strong rays that can have different intersections with $\mathrm{Bd}(B_1')$ and $\mathrm{Bd}(B_2')$, and $Q \subseteq Q_A$ be the selected points that have a strong ray in R. We discuss how to determine R in Section 5.2.3. We have $|R| \leq 7$ by Lemma 7. During (most of) the algorithm a point $p \in R$ is pre-selected, we do not know the precise extent of the strong certificates of p, but we are sure that it does not overlap with any other strong certificate.

Our algorithmic approach can be summarized as follows: We assume that $L = \emptyset$ and try to complete the separation certificate. If this succeeds, our assumption was true and we are done. Otherwise the failing process gives us a good starting point to identify L and establish the new complete certificate. Even though the geometric decisions to advance suspended searches and to select points where based on the wrong assumption $L = \emptyset$, they remain valid. We return to this discussion in Section 5.2.2.

We realize that there is a surfacing point by finding a point $q \in \mathrm{UV}(A_1)$ above $\overline{xy}$. Note that q itself might not be surfacing, but the point s of $\mathrm{UV}(A_1)$ with a tangent line parallel to $\overline{xy}$ does surface. We can use a lasting splitter to actually find s, and from there determine all of L with a linear scan away from s. If we happen to find s, we merely check on which side of $\overline{xy}$ it is. Then we can either conclude that $s \in L$, or that $L = \emptyset$. Our algorithm proceeds

without necessarily identifying s. Nevertheless we work with s in the sense that we can decide for a point $c \in \mathrm{UV}(A_1)$ on which side of s it is, just by comparing the slopes of the segments at c with the slope of $\overline{xy}$. Here we use the suspended search facility of a lasting splitter in geometrically different ways: As long as we assume $L = \emptyset$, we consider a light certificate, comparing strong ray intersections with light rays. As soon as we know $L \neq \emptyset$, we switch to searching for s by comparing slopes.

Let H_B be the current set of shortcut defining lines.

Fig. 15. **Algorithm "Detect surfacing":**

Assume: $r \in B_1$ gets deleted, neighbors are x and y, there is complete certificate
compute $\hat{B}'_1 = \mathrm{SC}_H(\hat{B}_1)$ (apply shortcuts) (Section 5.2.3)
determine $R \subseteq R_A$, the strong rays that might change
determine $Q \subseteq Q_A$, the selected points to R, the preselected points
classify points in Q as right of s and left of s
if (point of Q surfaces) **then goto** search surfacing point
identify the middle lasting splitter M (the one containing s)
identify $p, q \in Q$, the selected points next to M, p left of and q right of M
determine intersection of right strong ray at p with $\hat{B}'_1$ (with $\overline{xy}$)
determine intersection of left strong ray at q with $\hat{B}'_1$ (with $\overline{xy}$)
while(the light certificate of the candidate of M is not valid)
  advance suspended search of M (consider light certificate of the candidate)
  if(candidate point above $\overline{xy}$ found)
    then goto search surfacing point
  if(search on M finishes with apparently selectable point c)
    then split M and preselect c
      check if c is left or right of s
      make M the new middle suspended search around s
      instantiate suspended search on M
      determine the necessary strong ray intersection of c with $\hat{B}'_1$
determine strong ray intersections for preselected points with $\hat{B}'_1$
place appropriate pieces of $\hat{B}'_1$ in (constant size) temporary splitters
$L := \emptyset$ (no surfacing)
goto "generic algorithm"

5.2.1 Performance considerations

To achieve the aimed-at amortized performance of the data structure we should only use time $O(|L|+|J|)$ for all of the re-establishing: Every point can be once in J and once in L, it advances in its life-cycle. In particular we should only use $O(1)$ time if no point is surfacing, i.e. $L = J = \emptyset$. This is (one of) the situation(s) where we crucially use the search facility of the splitter, where the

Fig. 16. **Algorithm: "Search surfacing point"**, Process if some point o is above $\overline{xy}$
if(no preselected point is surfacing)
  then Use suspended search on M to find s by slope comparison
while (a neighbor of L on $\mathrm{UV}(A_1)$ surfaces too)
  extend L
finish suspended searches neighboring surfacing selected points (see Section 4.9)
build temporary splitter of points on $\mathrm{UV}(B_2')$, including L, up to unchanged strong ray intersection
compute J
create lasting splitter of points on $\mathrm{UV}(A_2)$, join over suspended search over J
goto "generic algorithm"

overall time spent in searches is amortized constant per element (remember that the search, including the work around the advancing, is already paid for when placing a point in the splitter). If there is a point surfacing (usually s) we scan $\mathrm{Bd}(B_1)$ in both directions away from s to determine the extent of L. We use the data structure holding A to determine J by deleting all the points of L. We use $O(|L| + |J|)$ potential when we create new splitters and extend existing ones (via the restricted JOIN operation, while we establish the new complete certificate between $\mathrm{Bd}(A_2)$ and $\mathrm{Bd}(B_2)$).

5.2.2 Geometric justification of the two stages of the algorithm

We have to argue, that none of our decisions to make a point a guard (or to (pre-)select a point, which is geometrically the same, see Section 4.1.2) becomes invalid, if it turns out that the assumption $L = \emptyset$ was wrong. It is important for this argument, that the candidate we are talking about is below $\overline{xy}$. This is always the case because we would otherwise not have made it a guard or selected it, but started the algorithm in Figure 15 searching for s.

The names in the following Lemma are illustrated in Figures 17 and 18.

Lemma 8 *Let $c \in UV(A_1)$ and c below $\overline{xy}$. Let h be the left directed canonical ray rooted at c. Let p be a selected point and f the right directed strong ray rooted at p. Let i be the intersection of $\overline{xy}$ and f (the wrongly assumed strong ray intersection). Let j be the intersection of f with $\mathrm{Bd}(B_2)$ (the real strong ray intersection).*

Then j is above h (as a line) if and only if i is above h. This is also the case for the symmetric (left and right interchanged) assumption.

PROOF. There are two cases. If $s \in L$, the point of $\mathrm{UV}(A_1)$ with a tangent line parallel to $\overline{xy}$, is between c and p, (illustrated in Figure 17) then the inter-

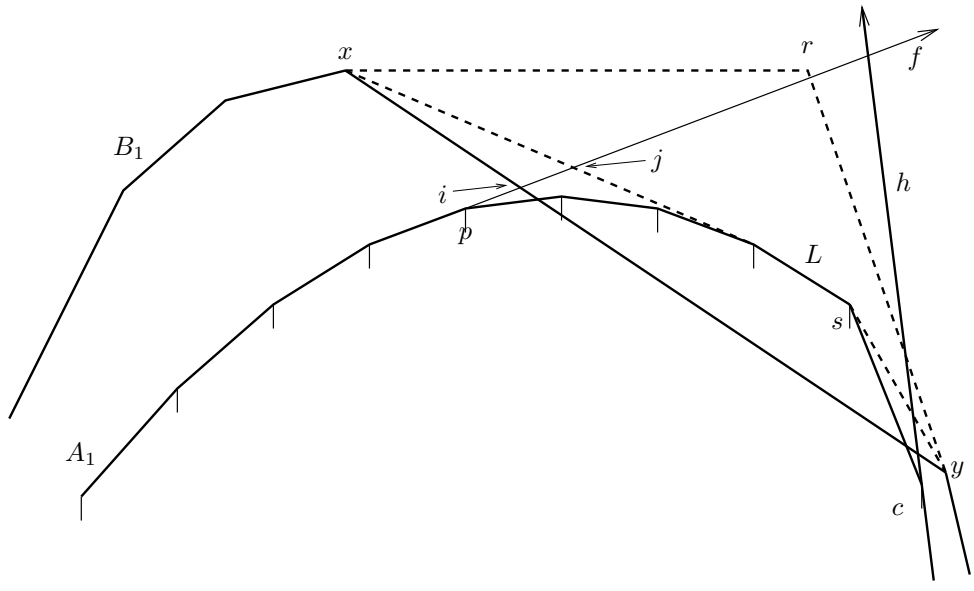

Fig. 17. Situation of the proof of Lemma 8 that all selected points obey separation, even when selecting in an extractor, different sides.

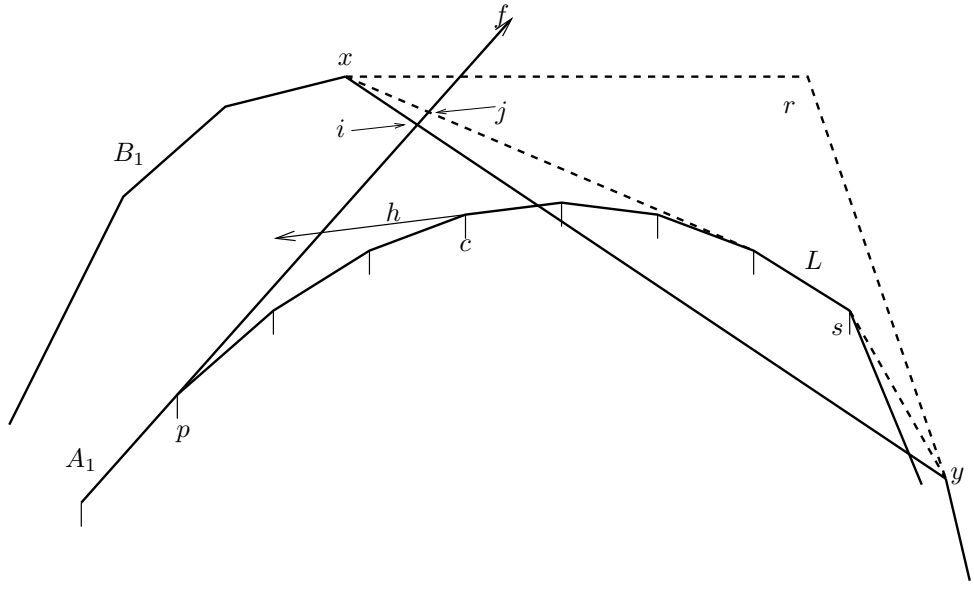

Fig. 18. Situation of the proof of Lemma 8 that all selected points obey separation, even when selecting in an extractor, same side.

section of f and h is outside $\mathrm{UC}(C_2)$, whereas both i and j are inside $\mathrm{UC}_0(C_2)$. (Monotonicity of rays and the tangent rays at s.)

If p and c are on the same side (left) of s, we have the situation of Figure 18. Because of the slope of h and because c is below $\overline{xy}$, the intersection of h with f is closer to p than i and hence also closer to p than j. □

5.2.3 Applying shortcuts and finding temporary strong ray intersections

It is possible that the triangle xry is completely above (cut away by) a shortcut $l \in H$. Then we conclude immediately that there is no surfacing point, and that the complete certificate remains valid. The flag of the data structure that marks cut-away segments allows us to check this case in constant time.

We construct the shortcut version $\hat{B}'_1 = \mathrm{SC}_H(\hat{B}_1)$, i.e., we apply the possibly relevant shortcuts to the segment $\overline{xy}$. Let $X \subseteq H$ be the set of lines that have cutting-points on $\overline{xr}$ or $\overline{ry}$. By Lemma 7 we have $|X| \leq 3$. Only shortcuts defined by X can change and intersect $\overline{xy}$. Our data structure provides direct access to X. In particular we detect the case $X = \emptyset$. In this case we continue without having to use any shortcuts.

We determine the intersections of the lines in X with the segment $\overline{xy}$. These points are taken as temporary cutting-points and we reestablish the data structure to represent the resulting $\hat{B}'_1$. These cutting-points might have to be changed later, if we find a surfacing point. If we find that a shortcut defined by line $l \in X$ has no intersection point, we can conclude that it is no longer effective and delete it from H. (An effective shortcut needs a point above it, which here cannot come from A because of $\overline{xy}$, and not from B because all shortcuts are conservative, they fit to Definition 2.

Scanning away from the representation of the segment of $\mathrm{Bd}(\hat{B}'_1)$ that lies on $\overline{xy}$, we search for the next strong ray intersection to the left and to the right. We call the set of examined segments $E \subset \mathrm{Bd}(\hat{B}'_1)$. By the discussion in Section 4.4 we know $|E| \leq 5$.

We compare slopes to identify the selected points of $\mathrm{UV}(A_1)$ that are right of s, and the ones left of s. We call the lasting splitter that contains s the *middle splitter* M. (If s itself should be selected things only get easier.) Only for M we determine the strong ray intersections with the segments of E. If such an intersection is defined by $\overline{xy}$, it is possible that this intersection might change because of a surfacing point, and is hence not final.

Alternatively it can be the case that we realize that one of the selected points is above $\overline{xy}$. This implies that there is a surfacing point and we immediately jump into the algorithm "search for a surfacing point."

If both the strong ray-intersections are final, we conclude that there is no point surfacing and that the complete certificate stays valid (none of the strong-ray intersections has changed).

5.2.4 Searching for surfacing points

Once we know that there is a surfacing point ($L \neq \emptyset$), we can just search for one of them. In particular we can search for the point s. In general M will have a suspended search. We test the left and right guard of this suspended search for being left or right of s. If they are on different sides of s we continue the search until we find s. If they are on the same side of s, say s is left of both guards, then we pre-select the left guard g. This meets the strong ray separation to the right because g is a left guard, and to the left because the surfacing point s is between g and the next selected (to the left of g) point p of $\mathrm{UV}(A_1)$, the left neighbor of M. We split M at g and use the resulting splitter between p and g to search for s.

5.2.5 Processing surfacing points

As soon as the point s (or any other surfacing point) is identified, we can extend the list L of surfacing points. Let u be the rightmost point of L, and v its right neigbor on $\mathrm{UV}(A_1)$. If v is above $\overline{uy}$, we include v into L. In terms of accounting this is no problem, we realize that the points advanced in their life-cycle, we will never process one point a second time in this way. Additionally we identified L to return to the parent data structure and we know that $B_2 = \hat{B}_1 \cup L$. We also get the shortcut version $B_2' = \mathrm{SC}_H(B_2)$ easily, because we know that none of the points of L can be above a shortcut of H_B. Hence only the up to two temporary cutting points with $\overline{xy}$ need adjustment, they are instead taken with the segment of $\mathrm{Bd}(B_2)$ connecting x with the leftmost point of L, and the a segment connecting the rightmost point of L with y.

To remove the points of L from the representation of the locally inner hull, we remove them from lasting splitters by shrinking. We first have to finish the suspended searches and deselect selected points from L, as described in Section 4.9.

We use the data structures responsible for A to delete all the points of L from A_1 in order to achieve A_2. As a result of this operations we get list J of points that replace L, i.e., a stretch of points on $\mathrm{UV}(A_2)$ that are new.

We take one temporary splitter to store all the points $L' \supset L$ and constantly many more points of $\hat{B}_1'$, such that L' is smallest possible, but begins and ends with strong ray intersections, getting us back to a situation where we can run the generic algorithm. We have the situation of a stretch of preselected points, between them two splitter S_1 and S_2 without a suspended search on A_2, between S_1 and S_2 the list J of new points on $\mathrm{UV}(A_2)$. We use the operation $\textsc{join}(S_1, J, S_2)$, resulting in the splitter S that has a suspended search with the left guard g_l (the rightmost point of S_1, i.e. the left neighbor of L on $\mathrm{UV}(A_1)$, also the left neighbor of J in $\mathrm{UV}(A_2)$) and the right guard g_r. We argue that g_r

is in fact not in the shadow of the left selected point p, and the symmetric condition. For this we consider the left directed canonical ray t at g_r and the right directed strong ray s at p, as illustrated in Figure 19. Both s and t are on tangents on $\mathrm{Bd}(A_1)$ on different sides of L. As t and the right canonical ray at g_l already intersect outside $\mathrm{UC}(B_2)$, by monotonicity the intersection of s and t is outside $\mathrm{UC}(B_2)$.

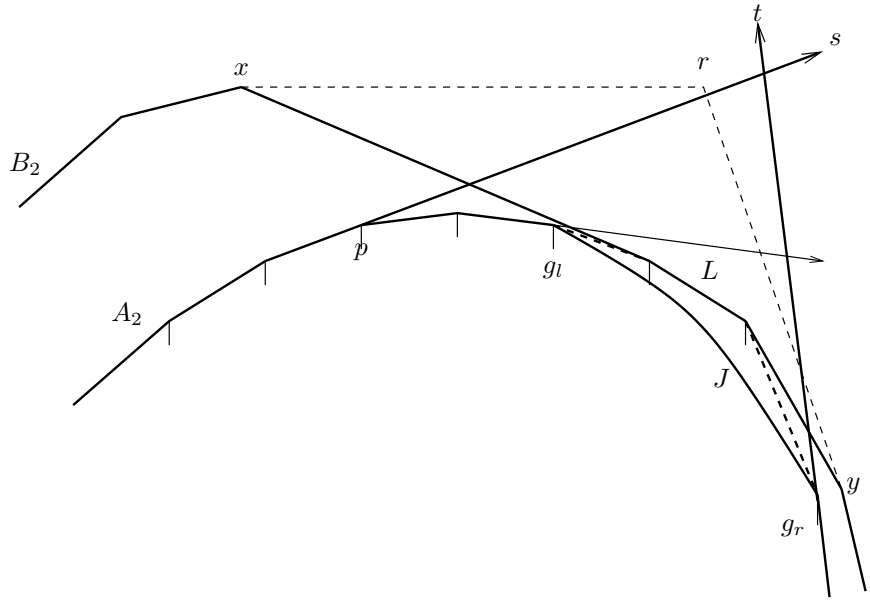

Fig. 19. The situation of joining two splitters over the new points J on $\mathrm{UV}(A_2)$.

On B_2' we have the temporary splitter. We determine all strong ray intersections using the temporary splitter (only temporary intersections with $\overline{xy}$ can still change). Using the generic algorithm, we instantiate all suspended searches and advance them to find valid light certificates, possibly selecting selectable points as we go.

6 Re-establishing the merging certificate

We use a certificate as described in Section 4 for the binary merging of two upper hulls. Re-establishing such a certificate uses several concepts that were already introduced for the extractor. It is different because the geometry is easier, points do not move from one participating set to the other. In return there are more cases to be considered, now there are different polarities, several equality points and bridges.

In analogy to the names in the extractor, we consider the case where a point $r \in B_1$ is deleted, leading to the set B_2. The neighbors of r on $\mathrm{UV}(B_1)$ are x and y, the list L consists of the new points on $\mathrm{UV}(B_2)$ between x and y, the points that replace r. Let X be the rightmost strong ray intersection,

equality point, or selected point (unchanged) on $\mathrm{Bd}(\mathrm{SC}_{H_B}(B_2))$ to the left of x, and Y the leftmost such point to the right of y. Let $L^+ \subseteq \mathrm{SC}_{H_B}(B_2))$ be the points between X and Y. The set to merge with is A, and its shortcut version $A' = \mathrm{SC}_{H_A}(A)$. Let $U \in \mathrm{Bd}(A')$ be the point aligned by the existing certificate with X, i.e., if X is an equality point then $U = X$, if X is a strong ray intersection then U is the selected point having the strong ray, if X is a selected point then U is the intersection of the right directed strong ray of X with $\mathrm{Bd}(A')$. Symmetrically we define V for Y. Let $J \subseteq \mathrm{Bd}(A')$ be the stretch between U and V. The main part of the algorithm here is to bring us in the position to run the generic algorithm on L^+ and J.

We refer to streaks of polarity B over A as normal (like the deleted point), streaks of polarity A over B are called *inverted*.

Essentially we have only two algorithms (forgetting about shortcuts, they do not really change the picture), that we both have already seen in the extractor: The first is when we have a place where some (inverted) points of L are identified to be inside $\mathrm{UC}(A)$. Then we scan to determine how far this new (part of a) inverted streak A above B extends (usually looking for a new equality point, that we know exists). This resembles the algorithm in the extractor that identifies the surfacing points. Then we create a new separation certificate for this streak, or extending an existing one, using the generic algorithm. (Both sides advance in their life-cycle.) The other one is to try to close the gaps in a certificate B above A, working under the assumption that the part of L^+ is above $\mathrm{Bd}(A)$. (using the lasting splitters of A and a temporary splitter for L^+.) If this assumption turns out to be wrong, we use the first algorithm to identify a new streak. To finish the work we also have to create new shortcuts and to find new bridges. The complete algorithm looks somewhat more complicate, mainly because we have to consider several cases in the beginning, when we take care of lost equality points. All these cases somehow fit to the above general description, but they need small adjustments to the specific situation. The different cases stem from the possibility that each of the two deleted segments $\overline{xr}$ and $\overline{ry}$ can intersect $\mathrm{Bd}(A)$ once, twice or not at all. The different possibilities are illustrated in Figure 22.

6.1 Applying shortcuts

Applying shortcuts when merging is more complicated than in the extractor, here we have to apply shortcuts to all of L. By Lemma 7 there are at most 3 shortcuts that have cutting points on $\overline{xr}$ or $\overline{ry}$.

The result of this algorithm is that the data structure represents $L^+ \subseteq B_2'$ (determining X and Y amounts to scanning over $O(1)$ points), and the cutting

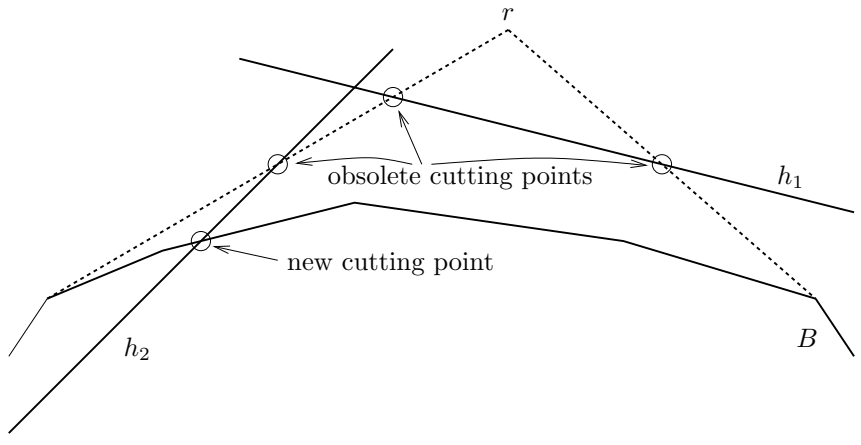

Fig. 20. Applying shortcuts to B after the deletion of r. The active shortcut defined by line h_1 becomes futile and obsolete. The active shortcut defined by h_2 shortens and changes one of its cutting points.

Fig. 21. **Algorithm "apply shortcuts"**

determine set $X \subseteq H_B$ of shortcuts cutting $\overline{xr}$ and $\overline{ry}$ ($|X| \leq 3$)
forall ($l \in X$)
  scan L and determine new cutting point(s) with l
  adjust data structure for L: shortcut flag, cutting points
  if (no point of $L \cup \{x, y\}$ above l) **then** delete l from H_B

points with $\overline{xr}$ and $\overline{ry}$ are deleted.

6.2 Scanning for an equality point

Assume we found an inversion, a vertical line h where now $\mathrm{Bd}(A)$ is above $\mathrm{Bd}(B)$, given by the intersections of h with both hulls. Assume we are in the situation that we know there is another equality point, and we want to identify it algorithmically, for example in case (b), where we know that the new streak β ends at an equality point of L^+ and J. We can afford to scan over inverted points, this is advancing the life-cycle of the scanned over points: So far we assumed that for them we still have $\mathrm{Bd}(A)$ below $\mathrm{Bd}(B)$, which we now prove to be a wrong assumption.

The geometric side of this algorithm is a simple vertical sweep line, say left-to-right. We have a sweep line σ that stops at every point of $\mathrm{UV}(A)$ and $\mathrm{UV}(B_2')$. For every such line we determine the intersection with $\mathrm{Bd}(A)$ and $\mathrm{Bd}(B)$. We identified the equality point if on σ we have $\mathrm{Bd}(B_2)$ is above $\mathrm{Bd}(A)$.

As this process extends one inverted streak, it simultaneously shrinks a normal streak. We reflect this shrinking in the data structure by deselecting points,

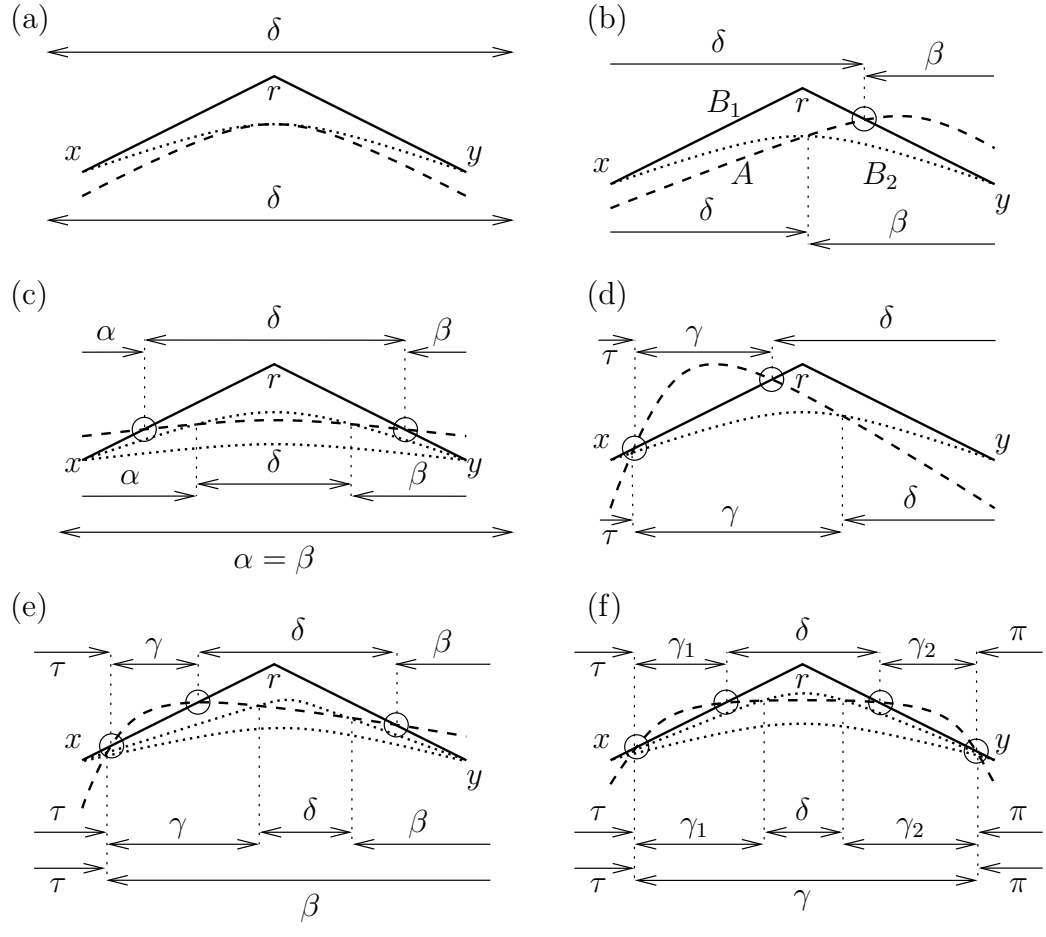

Fig. 22. Different cases of lost equality points. The streaks named γ are trivial (the locally inner hull is entirely on $\overline{xr}$ or $\overline{ry}$, there are no selected points or lasting splitter, the shortcut outer hull consists of less than 6 segments) before the deletion. The arrows above the situation indicate streaks before the deletion, arrows below the situation after the deletion. In the cases (c), (e) and (f) there are two outcomes possible, in case (c) we might have to join two non-trivial streaks. In the new streaks named δ, τ, π there can always be an even number of additional equality points. The symmetric case for (b) is reffered to as (b') (β is then called α), the same for (d) and (e).

shrinking lasting splitters, first finishing suspended searches as described in Section 4.9.

We prepare the generic algorithm to establish/complete a certificate on the extend streak by extending a lasting splitter of points on $\mathrm{UV}(B)$, and a temporary splitter of points on A'. Before running the generic algorithm we extend the temporary splitter up to the next strong ray intersection, because of shortcuts these are only constantly many points.

If we encounter a formerly trivial inverted streak, this streak gets united with the streak we are currently extending (for example in case (e) β "eats" γ).

Note that scanning over a trivial inverted streak takes $O(1)$ time because we use the shortcuts.

6.3 Modified selection

In the generic algorithm for the not-inverted streak (δ, τ, π in Figure 22), it is possible, that a preselected point $p \in \mathrm{UV}(A)$ (a point we decided to select trying to complete a certificate that $\mathrm{UV}(A)$ is locally inside $\mathrm{Bd}(B)$) is actually above $\mathrm{Bd}(B)$. We identify the situation when we determine the intersection of $\mathrm{Bd}(B')$ with the vertical line through p, using the temporary splitter, the first step of the selection algorithm. Then we found a new streak of polarity A over B that we explore as described in the Figure 23 and Section 6.2.

Fig. 23. **Algorithm "explore inverted streak" (explore A above B)**

while (the new/changed streak is not bound by equality points)
  extend streak (vertical sweep-line)
  shrink lasting splitter of A
make found equality points explicit in representation
gosub find new bridges
place scanned part of $\mathrm{Bd}(A)$ in temporary splitter
place scanned part of $\mathrm{Bd}(B) \subset \mathrm{Bd}(L)$ in lasting splitter
gosub generic algorithm for inverted streak
return (new streak has complete certificate)

6.4 Algorithm

Reacting to a deletion we run the algorithm of Figure 24 to re-establish a complete certificate. The description of the algorithm uses the names of streaks given in Figure 22. It unifies the treatment of the different cases.

6.5 Joining over dangling search

This case, and its geometric justification, is basically the same as in the extractor, when we found surfacing points.

As illustrated in Figure 25, we take all of L as new points when joining the streaks α and β (Figure 22, (c)). As r was outside $\mathrm{UC}_0(A)$, a monotonicity argument for rays shows that g_l and g_r are valid guards, i.e., g_l is not in the shadow of q, and g_r is not in the shadow of p. The part of $\mathrm{Bd}(A')$ between

Fig. 24. **Algorithm: Deletion of** $r \in \mathbf{UV}(B)$

compute $\hat{B}_2' = \mathrm{SC}_H(\hat{B}_2)$ and L^+ (apply shortcuts)
identify J, and preselect selected points of J
analyze lost equality points
place bridge-protectors for bridges using r
scan L and mark vertical lines through lost equality points
if(one lost equality point on $\overline{xr}$) (cases (b'), (c), (e'))
  then extend streak α to the right until new equality point
    if (found beginning of β instead) (case (c) "$\alpha = \beta$")
      join α and β over suspended search (Section 6.5)
      goto generic algorithm to complete certificate on $\alpha = \beta$
    gosub generic algorithm on α (inverted)
if(one lost equality point on $\overline{ry}$) (cases (b), (c) top, (e))
  then extend streak β to the left
    gosub generic algorithm to complete β
if(pair lost equality point on $\overline{xr}$) (cases (d), (e), (f))
  then extend trivial streak γ (γ_1) to both sides
    gosub generic algorithm for γ
if(pair lost equality point on $\overline{ry}$) (cases (d'), (e') top, (f') top)
  then extend trivial streak γ (γ_2) to both sides
    gosub generic algorithm for γ
gosub find new bridges above all newly identified equality points

(now the new streaks depicted below the situations in Figure 22 are identified)

place pieces of L^+ in not-inverted streaks (δ, τ, π) into one temporary splitter per streak
forall (preselected point $p \in J \subseteq A$)
  use temporary splitter to find intersection of vertical line through p with L^+
  if (p outside L^+) (we found a new inverted streak)
    then gosub "explore inverted streak"
  else (p gives rise to strong certificate)
    scan for strong ray intersections with L^+, create shortcut (establish strong certificate)
gosub generic algorithm, using "explore inverted streak"
    for inverted pre-selected points (Section 6.3)
remove all protectors

the strong ray intersections X and Y contains, by aggressive shortcuts, only constantly many points that do not advance in their life-cycle from stage (2) to (3). Hence we can put all of J ($\mathrm{Bd}(A')$ between X and Y) into a temporary splitter, and account only $O(1)$ time to the deletion.

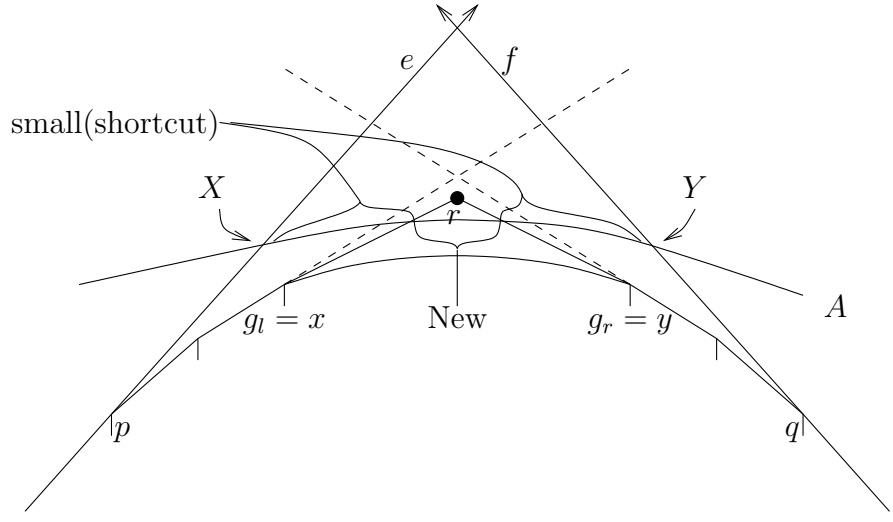

Fig. 25. Illustrating the situation of joining two strips into one dangling search.

6.6 Correctness and running time

The algorithm produces a complete certificate. First (analyzing equality points) inverted streaks are extended as much as appropriate. Whenever the algorithm finds an inversion, a (new) streak is explored and a complete certificate for the streak is established. For the not-inverted streaks the generic algorithm establishes a complete certificate, possibly finding more inverted streaks. Starting with the preselected points, the algorithm keeps track of places, where the certificate is not yet complete. It only finishes when the certificate is re-established.

The running time of the scanning algorithm is accounted for by the advanced life-cycle of the scanned-over points. The generic algorithm is already paid for when creating the lasting splitters. The remaining cost per deletion is $O(1)$.

6.7 Bridge finding

When we perform the merge-operation (creating a new data-structure), we search a bridge by a linear scan away from an equality point. We use the geometric reasoning of Overmars and van Leeuwen (5). Instead of a binary search, we perform (simultaneously/inter-twisted) two linear scans, usually away from the equality point. This algorithm considers every point only once, achieving the aimed-at performance.

If we later find a new equality point (after a deletion), we can use this algorithm as long as we did not scan over the points before. If the endpoint of a bridge gets deleted, the points between the equality point and the new bridge are

already scanned over (when we found the now-deleted bridge). We avoid this repeated scan by placing a *protector*, a double link that marks the stretches of $\mathrm{Bd}(A)$ (and also $\mathrm{Bd}(B)$) between the equality point and the deleted bridge. If our outward-scan reaches a protector, we jump to the other side of the protector and continue there. If we continue outwards, the points are fresh on the upper hull and advance in their life-cycle. Otherwise the scanned-over points are no longer below a bridge (and they will never get below a bridge again), and by this advance their life-cycle. After the certificate is complete again, we remove all protectors. The time spent in placing, using and removing protectors is charged to the deletion, there are at most 4 protectors per deletion.

Fig. 26. **Algorithm "Bridge finding"**
starts at equality point (without bridge) with placed protectors
while(bridge not correct)
adjust one endpoint, jumping to other end of protector on way out

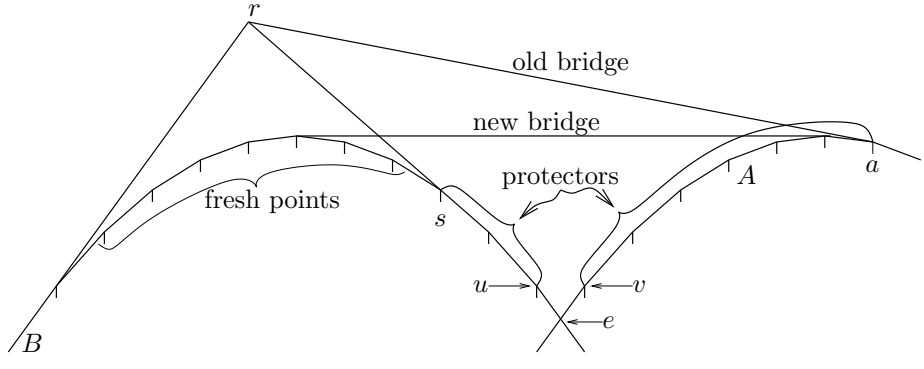

Fig. 27. The situation of placing protectors. When processing the deletion of point r we have to protect parts of the upper hulls of A and B from being scanned again, when we search for the new bridge.

7 Dealing with degeneracy

So far we used the assumption, that input points are in general position to avoid dealing with degenerate cases. This is convenient, as it lets us concentrate on the important situations instead of getting drowned by special cases. Here we summarize the situations that benefited from this assumption, and suggest how to modify the algorithms as to correctly deal with the degenerate cases.

In the data structure (and algorithm) we have to be prepared to find points, that act in two roles simultaneously, for example an input-point that is an equality point or an input point that is also a strong ray intersection.

If the same points can be both in A and B, we might see a stretche of $\mathrm{Bd}(A)$ that coincides with a stretch of $\mathrm{Bd}(B)$. We can easily handle this by treating such a stretch as an extended equality point, introducing one more stage in the life-cycle of a point there is no problem with the accounting. We have to allow deletion of points that are in such an equality stretch.

We have to decide on a policy if the strong-ray intersection and the intersection between strong and light ray are the same point. Naturally we allow these "touching" certificates to not have a gap. In return we have to catch the special case of a light certificate being based on one line (two guards, no candidate point) coinciding with a segment of the other hull, because this certificate contains an equality point (which is identified by checking for the case).

In the bridge-finding we have to break ties in a way that the resulting hull does not have two collinear segments.

8 Kinetic Heaps

A kinetic heap is a fully dynamic parametric heap with the restriction that the query line has to move to the right from query to query. This naturally leads to the notion of a "current time" time t, the argument of the last query, a kinetic query has to increase t.

Using the semidynamic geometric merging data structure of Theorem 4 we can design a kinetic heap, as explained in this section.

Theorem 9 *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 $O(n)$.*

The data structure we present here has several concepts in common with the construction we present as a proof of Theorem 1 in Section 9. It is simpler and illustrates several of the techniques nicely.

We use some ideas of the paradigm of kinetic data structures, but we do not think of it as a kinetic adaption of a static data structure.

Lemma 10 (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 logarithmic method with merging degree $\log n$. 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 from the semidynamic set.

The semidynamic data structure of Theorem 4 allows kinetic find-min queries in amortized $O(1)$ time: We maintain a pointer to the last answer, and perform a linear scan to find the answer for the next query. The time spent to perform the scan is charged to the insertion. The right endpoint e_l of the activity interval I_l of the current answer line l gives the time until which l stays the correct answer. We refer to e_l as the *(expiration) event* of l.

For the current time t we have the set L_t of lines that are answers for the find-min queries on all the $O\log^2 n)$ semidynamic sets. For all lines $l \in L_t$ we store the event e_l in a priority queue, implemented using a (2,4)-tree, giving access to the minimum value and element in $O(1)$ time, and amortized $O(\log n)$ insertions, that also pay for the deletions of events (we only delete events that are stored).

We store L_t in a kinetic heap data structure S given by the assumption in the lemma. We call S the *secondary structure*.

For a KINETIC-FIND-MIN query for time $t' > t$ we do the following: We check if t' is smaller than the minimal (first) event in the priority queue. If so we conclude $L_t = L_{t'}$, there is no further change to the data structure. Otherwise we perform delete-min operations on the priority queue until the new minimum is larger than t'. This identifies a set X of lines, that are not $L_{t'}$. We *lazily* delete the lines from X from the secondary structure by merely marking them as deleted and remove them only when the secondary data structure is rebuilt because half of the lines stored are marked as deleted. Lazy deletions cause the size of the secondary structure to increase by a factor of two, but the additional lines stored do not affect the correctness of the generated output (answers to queries).

For the semidynamic sets storing a line of X, we perform a KINETIC-FIND-MIN query for time t', leading to $L_{t'}$. We insert the result lines into the secondary structure and insert the corresponding events into the priority queue. Now the query reduces to a (kinetic) query for time t' on the secondary structure.

For a MERGE operation (stemming from the dynamization technique) we re-

move the affected events from the priority queue, and perform lazy deletions of the lines in the secondary structure. We query the merged data structure for the current time and insert the answer-line in the secondary structure, and the event into the priority queue.

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 event from the priority queue. We query the changed semidynamic data structure for the current time and insert the line into the secondary structure and the event of the answered segment into the priority queue.

The run-time analysis keeps accounts for the lines. Note that the priority queue and the secondary structure have size $O(\log^2 n)$, and that updates of the priority queue and insertions into the secondary structure take amortized $O(\log \log n)$ time. Every line pays at each of the at most $\log n / \log \log n$ merging levels one insertion into the priority queue and into the secondary structure. This includes deletions from the priority queue and lazy deletions from the secondary structure. It totals to $O(\log n)$ amortized time, charged to the insertion of the line. The kinetic queries on the semidynamic data structures is paid for by the cost for merging them, and totals to amortized $O(\log n)$ per line. A deletion pays for the forced deletion of one line in the secondary structure and for querying and reinserting one event into the priority queue, and a line into the secondary structure. □

Using Preparata's (6) semidynamic insertion only data structure, we achieve insertions in $O(\log n)$ time. 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 $O(1)$ kinetic queries. Using this data structure in Theorem 10 (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 $O(\log^2(\log^2 n) + \log n) = O(\log n)$, yielding Theorem 9.

9 General queries: Interval tree

Note that Theorem 5 and Theorem 10 are very similar, in particular the logarithmic method is applied in the very same way, i.e., with merging degree $\log n$ and the data structure of Theorem 4 for the semidynamic sets. Again we use of secondary structures, each of limited size $\log^{O(1)} n$. The difference is that we here need several such secondary structures, and each of them will be as-

sociated with the nodes of an *interval tree*. The activity interval I_l, as defined in Section 8, is used to decide in which secondary structure to store line l. Geometrically this construction is very similar to Chan's (9), in particular the reasoning for the correctness of the queries is nearly 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 $x \in I$. This data structure is described in detail in the textbook (16, page 210). It is due to Edelsbrunner (21) and McCreight (22). 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 $\mathbb{T}$. If we store the lines of the lower envelopes of the semidynamic sets at appropriate nodes of $\mathbb{T}$, as given by their activity interval, we can correctly answer FIND-MIN queries, by collecting answers along a root-to-leaf path and taking the minimum.

The tree structure of an interval tree $\mathbb{T}$ is that of a search tree storing at the leafs the endpoints of the intervals (here it will be the endpoints of intervals defined by chunks, as defined later). For every interval I exists a *canonical node* of $\mathbb{T}$, defined as the node v of $\mathbb{T}$ 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 choose the degree parameter to be $\log n$, independent of the bootstrapping. Now the height of $\mathbb{T}$ 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 (using the assumption that secondary structures allow queries in logarithmic time). Unlike Chan we allow a line to be stored anywhere on the path from the root to the canonical node of its activity interval in $\mathbb{T}$. This does not compromise the correctness of the queries, but it saves time when determining for a line the appropriate node of $\mathbb{T}$ to store it. We use this freedom 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).

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 $\mathbb{T}$ where we should insert the lines. We address this problem by partitioning the lower envelope into *chunks* of $\Theta(\log n / \log \log n)$ consecutive segments. Given that we have only $O(\log n / \log \log n)$ merging levels, and because every deletion causes only constantly many chunks that are not accounted for the mergings, the overall number of chunks invented is $O(n)$.

For every chunk c we determine its activity interval I_c as the union of the activity intervals of the lines stored in c. We search the canonical node u of I_c in $\mathbb{T}$, which takes $O(\log n)$ time. This costs per segment $O(\log \log n)$ time, the same as for inserting the line into the secondary structure at u. 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.

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 $\mathbb{T}$ (which bounds the number of chunks from one semidynamic envelope with the same canonical node), the chunk size, and the number of semidynamic sets. We simplify this bound to $O(\log^4 n)$.

The creation of a new chunk leads to the insertion of its interval-endpoints as leafs into the interval tree. This eventually requires us to split the nodes of the underlying B-tree, we destroy the associated secondary structure, and we re-insert all the lines at appropriate secondary structures. (For this we need the node to know which chunks it hosts, this is not too expensive.) Chan's argument bounding this work spent in node-split operations carries over: We charge the cost for splitting a node entirely to the newly created node. If we split the node u and create a sibling v of u, we pay $O(\log n)$ for every chunk c that is currently supposed to be stored at u (before the split). This certainly pays for finding the new canonical node of c and also to pay for inserting all lines of c into Q_u or Q_v. If we split a node on the 4 levels closest to the leafs of $\mathbb{T}$, we know that the split affects not more chunks then there are leafs below the node (every chunk has its endpoints in two leafs below its canonical node). On level 1 we have at most $n/\log n$ nodes, each having $\Theta(\log n)$ leafs below it, on level 2 at most $n/\log^2 n$ nodes, each having $\Theta(\log^2 n)$ leafs below it, and so on. As we only consider 4 levels in this way, we have $O(n)$ many chunks affected by split operations accounted for in these 4 levels. Moving a chunk costs $O(\log n)$ time, the total work is $O(n \log n)$.

There are at most $O(n/\log^4 n)$ nodes in $\mathbb{T}$ remaining to be accounted for. We can have at most $O(\log n \cdot \log^2 n)$ (degree of $\mathbb{T}$, number of semidynamic sets) many chunks stored at one node at a time. We pay $O(\log n)$ per affected chunk, so the total work we have to pay for splitting nodes is $O(\frac{n}{\log^4 n} \cdot \log^4 n) = O(n)$.

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*. It achieves that every line is stored in at most one place in the interval-tree. 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 activity 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 of $\mathbb{T}$. We do not really need to move the line, it is still stored on the path from the root to its canonical node. 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 a line $l \in S$ can enlarge activity intervals. From the semidynamic data structure we get a piece L of the lower envelope, that replaces l. This piece (possibly together with neighboring chunks) gets divided into new chunks. For every chunk we determine the canonical node of $\mathbb{T}$. We make sure that every line h of L is stored in $\mathbb{T}$ at a node above its canonical node, possibly inserting it at the canonical node of the chunk. If it is stored at node u of $\mathbb{T}$ we check that I_h is between the relevant keys of the parent node of u. If this check fails, the line h has to be stored at a different node of $\mathbb{T}$, we perform a *forced move*, inducing the additional cost of deleting h from the secondary structure at u, using the DELETE operation. As this operation is expensive (compared to the insertion), it is essential for our analysis to bound the number of forced moves. We charge this cost to deletion of line l in the following way. Every line h that was a neighbor of l on some lower envelope might be stored in $\mathbb{T}$ (when it handled by a forced move) based on an activity interval stemming from l. We distinguish two cases depending on the performance of the deletions of the secondary structure.

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.

Now we also have to analyze the space usage of the data structure. The interval tree and the chunks use $O(n)$ space. In the secondary structures every line uses $O(1)$ space as it is stored in at most one secondary structure. This totals to a space usage of $O(n)$. This finishes the proof of Theorem 5.

9.1 Tangent / arbitrary line query

If the only query we are interested in is extreme point / vertical line query, the presented data structure is sufficient. If we are interested in different queries, it is not obvious how useful the query data structure is. As the geometric principles of the interval tree are the very same as of Chan (9), queries that do not require the update-parameters to be changed can immediately used here as well. As an example we consider (in the primal setting) the following query: given a point $p \notin \mathrm{UC}(S)$ in the plane, what are the two common tangent lines of p and S? Translating the query into the dual setting, we ask for the two intersection points of an arbitrary line with the lower envelope of L_{S^*}. Given that we may perform vertical line queries, we can easily verify a hypothetic answer. Therefore it is sufficient to consider the situation under the assumption that the line intersects the lower envelope twice.

Let us focus on finding the right intersection of line l with the lower envelope $\mathrm{LE}(S)$ for a set of lines S, that is, we are in the dual setting. This is again an optimization task, we ask for the line of S with slope strictly smaller than l, that intersects l furthest to the left.

Now we need the geometric argument that we can use such queries in the secondary structure to navigate in $\mathbb{T}$ in a way that leads to the correct answer.

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

Lemma 11 *Let a and b be two vertical lines, a to the left of b. Let $S' \subseteq S$ be two sets of lines such that the lower envelope of S' at a and b coincides with the lower envelope of S. Assume that an arbitrary line query for a line ℓ on S' results in the right intersection point t. If t lies between a and b then also the right intersection T of ℓ with S (if it exists) lies between a and b.*

PROOF. By the definition of the right intersection point as the leftmost intersection of ℓ with the lines of smaller slope in S' and S we immediately have that T is not to the right of t and hence not to the right of b.

Assume that the left intersection of ℓ with $\mathrm{LE}(S')$ is also between a and b. If both intersections of ℓ with $\mathrm{LE}(S)$ exist, they are between the intersections of ℓ with $\mathrm{LE}(S')$. In this case the lemma holds.

Otherwise we know that the intersection v of ℓ with a is below the intersection u of $\mathrm{LE}(S')$ with a. Assume that T is to the left of a. Let h be a line of $S \setminus S'$ that contains T and has smaller slope than ℓ. Then the intersection of h with a is below v. But then the lower envelope of S intersects a at or below v, contradicting the statement that the two lower envelopes coincide on a. $\square$

Using this lemma, we can process an arbitrary line query for ℓ in the following way: Starting at the root, we perform the query for ℓ at the secondary structure Q at the root. If we find that there are no intersections of $\mathrm{LE}(Q)$ with ℓ, we know that there is no intersection of ℓ with S. Otherwise let u be the found right intersection point. We find the slab that is defined by the keys stored at the root that contains u. This slab identifies a child c of the root. We continue the search at c in the same way, that is, we perform another arbitrary line query to the secondary structure. Now we take the leftmost of the two results (stemming from c and the root) and use it to identify a child of c. This process we continue until we reach the leaf level. There we take the leftmost of all answers we got from secondary structures, and verify it by performing a vertical line query. It is necessary to use the currently leftmost answer as we allow lines to be stored higher up in the tree. It is necessary to verify the outcome, we cannot exclude the case that all queries to secondary structures find two intersection points, whereas there is no intersection of ℓ and $\mathrm{LE}(S)$.

For the bootstrapping we have to argue that we can perform arbitrary line queries in $O(\log n)$ time. As this kind of search is the main ingredient to perform fast inserts, this does not require an additional algorithm.

As the arbitrary line query in $\mathbb{T}$ performs precisely one query to a secondary structure at every level of $\mathbb{T}$, we get an overall time bound of $O(\log n)$.

If we run this query for a point $p \in S$, we determine whether $p \in \mathrm{UV}(S)$ and if this is the case, we find the neighbors of p in $\mathrm{UV}(S)$. We also realize if p is a point on a segment of $\mathrm{Bd}(S)$. Tangent queries allow us to report a stretch of k consecutive points on the upper hull of S in time $O(k \cdot \log n)$. This is by a $O(\log n)$ factor slower compared to an explicit representation of the convex hull.

10 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. Our method here is completely reduction based, it does not exploit the online-character of a data structure. We use the data structure to solve a parameterized decision problem, arriving at the lower bound.

The lower bounds on the decision problems hold for algebraic computation trees (23). 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 (23, 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 3 *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 $y_1 \leq y_2 \leq \cdots \leq y_k$ and for all i and j we have $x_i \neq y_j$.*

Lemma 12 *For $8 < k \leq n$ 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 $h \geq c \cdot n \log k$ for some $c > 0$.*

PROOF. Let $y_i = i$ for $i = 1, \ldots, k$. 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 (23, Theorem 3) we get $2^h 3^{n+k+h} \geq (k+1)^n$. The claimed lower bound follows with $c = (1 - \frac{\log 9}{\log 10})/(1 + \log 3)$. □

The amortized running time functions in the following theorems are to be understood that insertions (into an empty data structure) of n elements take a total of $n \cdot I(n)$ time.

Definition 4 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 13 *Let 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 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 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 DISJOINTSET$^+_{n,k}$. We check in linear time whether we have $b_1 \leq b_2 \leq \cdots \leq b_k$. If this is not the case, we reject. We insert all the values a_i into 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 DISJOINTSET$^+_{n,k}$ problem.

The reduction takes linear time. By Lemma 12 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 13 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 13 shows that the amortized insertion times of the data structure of Theorem 9 and Theorem 1 are asymptotically optimal.

Theorem 14 *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 DISJOINTSET$^+_{n,k}$. Let I be the function such that $k \cdot I(k)$ is a (tight) general upper bound on the time to insert k elements into an initially empty data structure. 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 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 12 we get for sufficiently large k and some constant c the inequality $k \cdot (1 + I(k)) + n \cdot q(k) \geq c \cdot n \cdot \log k$. Using $k \cdot (1 + I(k)) = n$ and dividing by n we get $q(k) = c \cdot \log k - 1$. □

A data structure for the semidynamic predecessor problem maintains a set S of real numbers under insertions, and allows queries for r, reporting the element $s \in S$, such that $s \leq r$, and there is no $p \in S$ with $s < p \leq r$. From Theorem 13 and Theorem 14 follows the next corollary.

Corollary 15 *Consider a data structure implementing the* SEMIDYNAMIC PREDECESSOR PROBLEM *on the real-RAM that supports* PREDECESSOR *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 $I(n) = \Omega\left(\log \frac{n}{q(n)}\right)$*

Corollary 16 *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 $y = -x^2$ at the point $(a_i, -a_i^2)$. For a member query b_i we perform KINETIC-FIND-MIN(b_i). If the query returns the tangent line through $(b_i, -b_i^2)$, we answer "$b_i \in S$". The corollary follows from Theorem 13. □

Finally we can also conclude the main theorem:

PROOF. (of Theorem 2) A semidynamic insertion-only convex-hull data structure can be used as a kinetic heap (duality), Corollary 16 provides the bound on the insertions. The bound on the queries relies on Theorem 14, with the same geometric reduction as in Corollary 16. □

If instead the convex-hull data structure provides only tangent-queries, the same lower-bounds hold. Instead of querying with the slope (the value b_i in the proof of Corollary 16, used in the primal setting) of the line that presumable exist, we query with a point on that line, for example with the point $(0, b_i^2)$. If b_i is in the set, one answer line will be $(y = 2b_i x + b_i^2)$. Only for the "neighbor on the convex hull" query we are back to the bound for the kinetic case (only bounding the INSERT operation), we can efficiently answer (one sequence of) of kinetic queries if we have a "next-neighbor" query.

11 Trade-off

The above lower bounds apply for all non-decreasing functions $q(n)$ and $I(n)$. The standard data structures for membership queries on the real-RAM are balanced search trees. This establishes a matching upper bound only for the cases where insertions are required to take $\Omega(\log n)$ time, namely for $q(n) = O(n^{1-\varepsilon})$.

We have the same situation for the dynamic planar convex hull problem. This raises the question, whether there are data structures that match the lower bound for other combinations of insertion and query times as well.

There is one simple idea for a trade-off between insertion times and query times: we simply maintain several (small) search structures and insert into one of them. In return the query operation has to query all the search structures. We will describe the predecessor problem and use balanced search trees as the underlying data structure. We focus on the insertion only case. If we want to accommodate deletions we have to perform global rebuilding following a doubling technique. This does not change the (spirit of) the result, it only makes it more complicated to describe. The argument works for worst-case and amortized complexities.

We choose a parameter function $s(n)$ that tells the data structure how many elements might be stored in one search tree. We assume that $s(n)$ is easy to evaluate (one evaluation in $O(n)$ time suffices) and non-decreasing. The data structure keeps two lists of trees, one with the trees that contain precisely $s(n)$ elements and the other with the trees containing less elements. For an INSERT(e) operation we insert e into one of the search trees that contains less than $s(n)$ elements. If no such tree exists, we create a new one. When $s(n)$ increases, we join the two lists (all trees are now smaller than $s(n)$) and create an empty list of full search trees. For a query operation we query all the search trees and combine the result.

The (amortized) insertion time is $I(n) = O(\log s(n))$, the query time is $q(n) = O(\frac{n}{s(n)} \log s(n))$. We consider the term

$$\log \frac{n}{q(n)} = \Omega\left(\log \frac{s(n)}{\log s(n)}\right) = \Omega(\log s(n) - \log\log s(n)) = \Omega(\log s(n)) \ .$$

This means that we achieve according to Theorem 14 optimal amortized insertions times.

If we are interested in a data structure for the membership, predecessor or convex hull problem that allows queries in $q(n)$ time for a smooth, easy to compute function q, then this technique allows us to have a data structure with asymptotically optimal insertion times.

12 Open problems

It remains open whether a data structure achieving worst-case $O(\log n)$ update times and fast extreme-point queries 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, or if it is possible to maintain a balanced search tree of the points currently on the convex hull. Furthermore it would be desirable to come up with a simpler data structure achieving the same running times.

References

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

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

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

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

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

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

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

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

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

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

[11] G. S. Brodal, 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, Vol. 1851 of Lecture Notes in Computer Science, Springer, 2000, pp. 57–70.

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

[13] G. S. Brodal, R. Jacob, Dynamic planar convex hull, in: Proc. 43rd Annual Symposium on Foundations of Computer Science, 2002, pp. 617–626.

[14] R. Jacob, Dynamic planar convex hull, Ph.D. thesis, BRICS, Dept. Comput. Sci., University of Aarhus (2002).

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

[16] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computa-

tional geometry: Algorithms and applications, Springer-Verlag, Berlin, 1997.

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

[18] T. M. Chan, Remarks on k-level algorithms in the plane, manuscript (1999).

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

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

[21] Edelsbrunner, Dynamic data structure for orthogonal intersection queries, Tech. Rep. F59, Inst. Informationsverarb. Tech. Univ. Graz, Graz, Austria (1980).

[22] McCreight, Efficient algorithms for enumerating intersecting intervals and rectangles, Tech. Rep. CSL-80-9, Xerox Park Palo Alto Res. Center, Palo Alto, CA (1980).

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