

Simple and Robust Dynamic Two-Dimensional Convex Hull

Emil Toftegaard Gæde* Inge Li Gørtz * Ivor van der Hoog * Christoffer Krogh*
Eva Rotenberg *

Abstract

The *convex hull* of a data set P is the smallest convex set that contains P . A *dynamic* data set is one where points are inserted and deleted. In this work, we present a new data structure for convex hull, that allows for efficient dynamic updates, in theory and practice.

In a dynamic convex hull implementation, the following traits are desirable: (1) algorithms for efficiently answering queries as to whether a specified point is inside or outside the hull, (2) adhering to geometric robustness, and (3) algorithmic simplicity.

Furthermore, a specific but well-motivated type of two-dimensional data is *rank-based* data. Here, the input is a set of real-valued numbers Y where for any number $y \in Y$ its rank is its index in Y 's sorted order. Each value in Y can be mapped to a point (RANK, VALUE) to obtain a two-dimensional point set. Note that for a single update, a linear number of (RANK, VALUE)-pairs may change; posing a challenge for dynamic algorithms. It is desirable for a dynamic convex hull implementation to also (4) accommodate rank-based data.

In this work, we give an efficient, *geometrically robust*, dynamic convex hull algorithm, that *facilitates queries* to whether a point is internal. Furthermore, our construction can be used to efficiently update the convex hull of *rank-ordered data*, when the real-valued point set is subject to insertions and deletions. Our improved solution is based on an *algorithmic simplification* of the classical convex hull data structure by Overmars and van Leeuwen [STOC'80], combined with new algorithmic insights.

Our theoretical guarantees on the update time match those of Overmars and van Leeuwen, namely $O(\log^2 |P|)$, while we allow a wider range of functionalities (including rank-based data). Our algorithmic simplification includes simplifying an 11-case check down to a 3-case check that can be written in 20 lines of easily readable C-code. We extend our solution to provide a trade-off between theoretical guarantees and the practical performance of our algorithm. We test and compare our solutions extensively on inputs that were

generated randomly or adversarially, including benchmarking datasets from the literature.

Acknowledgements. This research was supported by Independent Research Fund Denmark grant 2020-2023 (9131-00044B) “Dynamic Network Analysis” and Eva Rotenberg’s Carlsberg Foundation Young Researcher Fellowship CF21-0302 - “Graph Algorithms with Geometric Applications”. This project has additionally received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987.

1 Introduction

In analysis of spatial data, computing the convex hull yields one of the most fundamental characteristics of the data set. The convex hull itself serves as a computational stepping stone towards promptly answering relevant queries as they arrive online. Seen from the *static* perspective, the problem of computing the convex hull has received much attention. In this paper, we study the *dynamic* setting, where data points may arbitrarily be added, deleted, or altered. Convex hulls have many applications for evolving or changing data sets, illustrating the need for an efficient, practical, and publicly available dynamic convex hull implementation.

Problem statement. The convex hull $CH(P)$ of a point set P is the minimal convex shape that contains all points in P . Convex hulls are one of the most prominent studied objects in computational geometry [9] due to their many applications which include top-queries [28, 37], clustering [33, 25, 44], road network analysis [34, 47, 25], and data pruning/dimension reduction [26, 38, 30, 35]. A convex hull data structure can store $CH(P)$ either *explicitly* or *implicitly*. An explicit data structure stores the vertices (or edges) of the hull, in their cyclical ordering, as a balanced binary tree.

An *implicit* convex hull data structure supports (a subset of) common convex hull queries without maintaining the edges of the convex hull. The points of P that lie on the convex hull of P are called the *extreme* points of P . An implicit data structure stores a data structure on P that can answer queries on the convex hull such as [13]:

*Technical University of Denmark, Kongens Lyngby, Denmark

1. Finding the extreme point of P in a query direction,
2. Deciding whether a line intersects $CH(P)$,
3. Finding the two hull vertices tangent to a query,
4. Deciding whether a query point q lies in $CH(P)$,
5. Finding the intersection with a query line,
6. Finding the intersection between two hulls.

We briefly note that with any explicit convex hull (storing h points) one can answer each of these queries in $O(\log h)$ time. We wish to dynamically maintain a convex hull supporting these queries subject to point insertions and deletions.

Related work. In the static setting, where the data set does not change, computing an explicit convex hull of a set of n points can be done in optimal $O(n \log n)$ time, e.g., with Graham’s scan [27] or Andrew’s vertical sweep line [2], or with the QuickHull $O(n \log n)$ expected time algorithm [5]. Crucially, Graham’s scan takes $O(n)$ time if the point set is already sorted by their radial ordering. There also exist static, theoretically optimal, output-sensitive algorithms, due to Kirkpatrick and Seidel [31] and later Chan [11]: They obtain $O(n \log h)$ running time where h denotes the number of points of P that lie on the convex hull. In a dynamic setting, the only data structure to explicitly maintain the convex hull is the $O(\log^2 n)$ algorithm by Overmars and van Leeuwen [40]. This is also the best explicit dynamic convex hull algorithm with worst-case updates.

The first dynamic implicit data structure for the convex hull is by Chan [12], who achieves a linear-size data structure with $O(\log^{1+\varepsilon} n)$ amortized update time, and supports queries 1–3 in $O(\log n)$ time (here, $\varepsilon > 0$ is some arbitrarily small constant). The update time was improved by Brodal and Jacob [9] to $O(\log n \log \log n)$ amortized. The original work by Chan [12] can answer queries 4–6 in $O(\log^{3/2} n)$ worst case time. This result was later improved by Chan [13] to support these queries in $O(\log^{1+\varepsilon} n)$ expected time.

Convex hull implementations. Despite its numerous practical applications, implementations of dynamic convex hulls are scarce. Many approaches rely upon static algorithms for the convex hull (e.g., [7, 45, 8, 23]), and thus have a linear update time. The CGAL library for computational geometry algorithms has no implementation of dynamic convex hull algorithms. Instead, they use a dynamic algorithm for maintaining the Delaunay triangulation of a point set, which itself contains the convex hull. However, dynamic Delaunay triangulations have near-linear update time and linear recourse, rendering them useless for polylogarithmic dynamic updates. Chi, Hacıgümüs, Hsiung, and Naughton [16] provide a Java implementation of the dynamic algorithm by Overmars and van Leeuwen, which they use for an application in scheduling. Compared

to ours, their algorithm lacks complete geometric robustness, it does not facilitate queries, and it cannot be extended to handle rank-ordered data. Independently, Cisneros [19] presents a C implementation of the algorithm by Overmars and van Leeuwen. Unfortunately, this implementation is insertion-only and has memory leaks. The algorithms for implicit convex hulls by Chan [12] and Brodal and Jacobs [9] are not implemented and do not support the operations required for our applications. The newer algorithm by Chan [13] does support these operations. It is our impression that the three latter papers [12, 9, 13], present algorithms that are sufficiently complex that further simplifications and new ideas are needed if they should be implemented in a way that is efficient in practice.

Applications of (dynamic) convex hull. Convex hulls, or convex hull queries, have countless applications. Here, we list some broad areas of application of dynamic convex hull queries. In Appendix C we elaborate on recent applications.

Query 4 can be used for constraint satisfaction problems. Indeed, consider a set of constraints given by a collection of halfplanes. Each line supporting these halfplanes can be mapped to a point (i.e. its *dual*). The feasible region is subsequently given by the convex hulls of these points. Using Query 4, we can efficiently test if a candidate value satisfies all constraints.

Query 5 can, given a preference vector specifying the weight of the x and y coordinates, return the preferred element of P . Repeated application of the convex hull can give the top k preferred elements [37, 28]. Top-direction queries are often used as an intermediary pipeline step in database algorithms [18, 16, 42].

Query 6 can be used to determine a line that minimizes the maximal distance to P . This has a direct application in (linear) regression, where we want to replace P by a line that for any x -coordinate, can predict the corresponding value in P [14, 46, 23].

The explicit convex hull itself (represented as a subset of point $P' \subset P$) has applications as a pruning step for classification algorithms. Intuitively, the convex hull P' is the subset of ‘extremal’ points in P . Many classification algorithms are most sensitive to these extremal values. By maintaining the convex hull, one can subsequently run classification algorithms on smaller-complexity input [44, 48, 26, 29]. Similarly, sometimes only extremal points are interesting for learning algorithms (pruning the number of points in the training data). Examples include support vector machines (SVM) algorithms that use convex hull computations in their underlying algorithms [7, 45, 15, 20, 36]. Existing SVM algorithms use static convex hulls algorithms.

We want to briefly note one special application of

convex hull algorithms. Suppose that we are storing a (large) set of values $Y \subset \mathbb{R}$. The rank of each $y \in Y$, is its index in the sorted order of Y and we want to support rank queries that for query values q report the corresponding rank. If Y is a set subject to insertions and deletions then this is the *dynamic indexing* problem and it is well-studied [3, 6, 10, 41, 43]. Recently, a new family of indexing data structures, called learned indexes, has been introduced [32, 22, 23]. Consider a set of values Y and the corresponding two-dimensional point set P_Y that maps every value in x to the coordinate (RANK, VALUE). By maintaining the convex hull of P_Y , recent work [32, 22, 23] uses ML techniques that, for some given parameter ε , predict the rank of each element with some additive error ε .

Our contribution. Via new theoretical insights, we provide a new practical algorithm for the dynamic convex hull problem, and provide an efficient, geometrically robust implementation. Our implementation makes use of the CGAL CORE library, to facilitate usage in geometric computations. More specifically:

- We establish a characterization for where to insert edges into a dynamic convex hull based on edge-to-edge comparison. This allows for storing the convex hull as a collection of edges (as opposed to points) which simplifies update logic compared to [40, 16].
- We apply this characterization to the algorithm by Overmars and van Leeuwen [40] to get an implementation of a linear-size data structure to explicitly maintain a convex hull of size h in a balanced binary tree with $O(\log^2 n)$ update time that facilitates queries in $O(\log h)$ time.
- We observe that OvL [40], even when simplified using our streamlined edge-based logic, comes with considerable overhead by maintaining what is called concatenable queues. We propose a new definition of explicit convex hull, where we do not store the edges of the convex hull in their cyclical ordering. Instead, we only require that we may (contrary to previous implicit convex hulls) answer queries 1–6 and that we can report the h points on the convex hull in $O(h \log n)$ time.
- With our edge-based logic, we construct a new algorithm, **Eilice**, for maintaining this hull in $O(\log^2 n)$ worst-case update time with $O(\log n)$ queries.
- We provide both a geometrically robust version of Eilice, as well as an approximate version since both versions have applications in practice.
- For rank-based convex hulls, we present and implement new algorithms with the guarantees.

Note: These are the first dynamic algorithms for rank-based hulls with polylogarithmic update time.

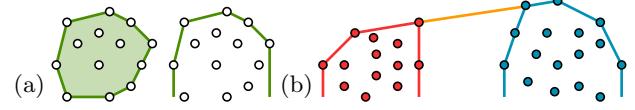


Figure 1: (a) The hulls $CH(P)$ and $CH^+(P)$. (b) Given two upper convex hulls $CH^+(R)$ and $CH^+(B)$, we construct their bridge.

- We compare our implementations (the simplified OvL and the new Eilice) to the preexisting dynamic convex hull implementations on a variety of generated and adversarial outputs. We conclude that:
 - geometric robustness comes at an expense of computation time,
 - as soon as the access pattern has less than a few hundred queries per update, dynamic algorithms have a clear advantage,
 - we obtain improved performance compared to the state-of-the-art (including [18, 16]).

2 Preliminaries

Let P be a set of n points in \mathbb{R}^2 . For any two points (α, β) we denote by $\overline{\alpha\beta}$ the segment between them. We say that a line ℓ separates P whenever it contains at least one point of P in the interior of either halfplane bounded by the line. The convex hull $CH(P)$ is the edge-set of the minimal convex shape that contains all points in P . Equivalently, the convex hull consists of all points P' such that each line through $(p_1, p_2) \in P' \times P'$ does not separate P .

We say a point a *precedes* a convex hull edge α if it has an x -coordinate that is lesser or equal than the left endpoint of α . The *upper convex hull* $CH^+(P)$ is defined as the convex hull of the set $P \cup \{0, -\infty\}$ ((Figure 1 (a)). The *lower convex hull* $CH^-(P)$ is defined as the convex hull of $P \cup \{0, \infty\}$, and when we view convex hulls as an area in \mathbb{R}^2 , $CH(P) = CH^+(P) \cap CH^-(P)$.

Consider two point sets A and B that are separated by a vertical line. The convex hull $CH(A \cup B)$ is formed by two segments of $CH(A)$ and $CH(B)$, together with up to two edges called *bridges*. Commonly, a bridge is defined as a *minimal* segment \overline{ab} for points $(a, b) \in A \times B$ where the line extending \overline{ab} does not separate $A \cup B$, see Figure 1(b). One of the bridges coincides with $CH^+(A \cup B)$, and one with $CH^-(A \cup B)$.

Partial Hull Trees Overmars and van Leeuwen [40] maintain the convex hull of a two-dimensional point set P by maintaining $CH^+(P)$ and $CH^-(P)$ in two separate linear-size data structures

that can answer each of the queries (1)–(6) in $O(\log h)$ time where h is the size of the convex hull. By computing the points of intersection between $CH^+(P)$ and $CH^-(P)$ they can identify all edges of $CH(P)$.

They store $CH^+(P)$ and $CH^-(P)$ in a data structure that we will refer to as a Partial Hull Tree (PHT). We recall the data structure for storing the upper convex hull $CH^+(P)$ as these data structures are symmetrical.

DEFINITION 1. Given a two-dimensional point set P , the Partial Hull Tree stores P in a leaf-based balanced binary tree T^+ (sorted by x -coordinates). For each interior node $v \in T^+$, denote by $\pi(v)$ the points in the leaves at the subtree rooted at v . For each node $v \in T^+$ with children (x, y) and parent v' , the Partial Hull Tree stores:

- The unique bridge $e^+(v)$ between the upper hulls $CH^+(\pi(x))$ and $CH^+(\pi(y))$.
- A concatenable queue $\mathbb{E}^*(v)$. This is a balanced binary tree of the vertices of $CH^+(\pi(v))$ that are not in $CH^+(\pi(v'))$ (where v' is the parent of v).
- For the root r of T^* , $\mathbb{E}^*(r)$ equals $CH^+(P)$.

Dynamically maintaining a Partial Hull Tree.

Let p be a point added to or removed from P and denote by ρ the root-to-leaf path to p . In [40], they restore the Partial Hull Tree by computing for every node $v \in \rho$ the bridge $e^+(v)$. Let x and y be the children of v . We define a pivot $(\alpha_x, \beta_x, \gamma_x)$ as any triple of consecutive points on $CH^+(\pi(x))$. Overmars and van Leeuwen consider a pair of pivots $(\alpha_x, \beta_x, \gamma_x)$ and $(\alpha_y, \beta_y, \gamma_y)$ of x and y . Through a case distinction of 11 cases (Figure 2) they can conclude if $e^+(v)$ has its left endpoint preceding or succeeding β_x . This gives the following dynamic update algorithm [40] that has two main steps:

Traversing down. First, they traverse the path from the root to the leaf containing p . For every node v they encounter, they consider the concatenable queues $\mathbb{E}^*(v), \mathbb{E}^*(x), \mathbb{E}^*(y)$. Here x and y denote the left and right child of v , respectively. Through the split and join operations of balanced binary trees, they obtain the convex hulls $CH^+(\pi(x) \setminus p)$ and $CH^+(\pi(y) \setminus p)$ in $O(\log n)$ time as balanced binary trees $\mathbb{E}(x)$ and $\mathbb{E}(y)$.

Bubbling up. Next, they traverse the path from the leaf containing p to the root bottom-up and compute for every vertex v on the path the bridge $e^+(v)$. Let v have children x and y :

1. Each inner node of $\mathbb{E}(x)$ represents a vertex β_x . In $O(1)$ time they obtain the predecessor or successor of β_x on $CH^+(\pi(x) \setminus p)$. The endpoints of these edges give a pivot $(\alpha_x, \beta_x, \gamma_x)$.
2. Similarly, each inner node of $\mathbb{E}(y)$ comes with a pivot. Given two pivots, they decide in $O(1)$ time if

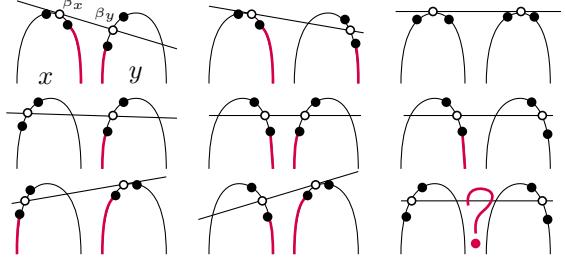


Figure 2: The 9 main cases for a line through the pivots (β_x, β_y) . For each case, [40] can exclude at least $\frac{1}{4}$ of the remaining convex hull points as an endpoint of $e^+(v)$. The 9'th case is further split into three others.

the endpoints of $e^+(v)$ succeed, precede or coincide with the pivots (Figure 2). Depending on the case, they recurse down $\mathbb{E}(x)$ and/or $\mathbb{E}(y)$.

3. Since there are at most $O(n)$ points on $CH^+(\pi(x))$ and $CH^+(\pi(y))$, they recurse at most $O(\log n)$ times before they find the bridge $e^+(v)$ between $CH^+(\pi(x))$ and $CH^+(\pi(y))$.

Given the bridge $e^+(v)$, they update $\mathbb{E}^*(v)$. After having updated $\mathbb{E}^*(r)$ for the root r , they have computed the convex hull $CH^+(P)$.

Runtime analysis. There are $O(\log n)$ nodes v on the root-to-leaf path ρ in T^+ . They iterate over ρ to obtain for all $v \in \rho$ a binary tree $\mathbb{E}(v)$ in $O(\log^2 n)$ total time. Then, they traverse ρ from p towards the root. For each $v \in \rho$ that the three steps are performed in $O(\log n)$ time. Thus the total running time is $O(\log^2 n)$.

The logarithmic method We briefly mention the logarithmic method, since Ferragina and Vinciguerra [23] used it to maintain convex hulls to predict the rank of elements in a fully dynamic set. The logarithmic method transforms (decomposable) static data structures into insertion-only dynamic structures (see e.g. [39]). It uses a bucketing scheme of buckets that increases in size exponentially. In Appendix D we describe the logarithmic technique applied to the convex hull, and why doing so leads to incorrect query algorithms and hulls.

3 Simplified Algorithm for Convex Hulls

In this section, we establish a new characterization for where to insert new edges into a dynamic convex hull. In this analysis, we use edges of the current convex hull as pivots, as opposed to the analysis in [40] that uses points (Figure 2). We then use it to provide a more straightforward implementation of the algorithm in [40]. Our analysis makes the algorithm in [40] not only simpler but also more efficient as we may store and

traverse fewer pointers. To this end, we denote by T^+ the Partial Hull Tree of P where we **redefine** bridges and concatenable queues:

DEFINITION 2. Let $v \in T^+$ have children x and y . We define its bridge $e^+(v)$ as a **maximal** segment ab for points $(a, b) \in CH^+(\pi(x)) \times CH^+(\pi(y))$ where the line extending ab does not separate $\pi(x) \cup \pi(y)$ (Figure 3).

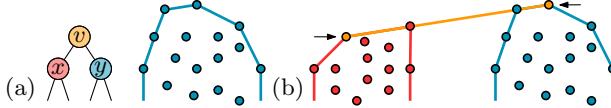


Figure 3: (a) $v \in T^+$ with children x and y . We show the upper convex hull $CH^+(\pi(y))$. (b) We redefine the bridge $e^+(v)$ as the **maximal** non-separating segment between $CH^+(\pi(x))$ and $CH^+(\pi(y))$.

DEFINITION 3. Let $v \in T^+$ have children x and y . Denote by $\mathbb{E}(v)$ the edges of $CH^+(\pi(v))$ in their cyclical ordering. We define $\mathbb{E}^*(x) := \mathbb{E}(x) \setminus \mathbb{E}(v)$ (Figure 4). The concatenable queue (or c-queue) is $\mathbb{E}^*(x)$ stored in its cyclical ordering as a balanced binary tree.

LEMMA 3.1. Let $v \in T^+$ have children x and y . Given $\mathbb{E}(v)$, $\mathbb{E}^*(x)$, and $\mathbb{E}^*(y)$, we may obtain $\mathbb{E}(x)$ in $O(\log n)$ time using the split and join operations on binary trees.

Proof. The edges of $\mathbb{E}^*(x)$ are a contiguous interval in $\mathbb{E}(v)$. Moreover, an edge is in $\mathbb{E}(x) \cap \mathbb{E}(y)$ if and only if it precedes the bridge $e^+(v)$. Thus, we achieve the lemma through Algorithm 1. \square

Whenever we invoke Algorithm 1, we store the tree \mathbb{E}_R so that we may invert the algorithm to obtain $\mathbb{E}(v)$ from $\mathbb{E}^*(x)$ in $O(\log n)$ time.

The Partial Hull Tree and its implementation. The Partial Hull Tree T^+ is a balanced binary tree on P sorted by x -coordinate. Each node $v \in T^+$ with children x and y stores the bridge $e^+(v)$ between $CH^+(\pi(x))$ and $CH^+(\pi(y))$ and the c-queue $\mathbb{E}^*(v)$. The Partial Hull Tree T^- is defined analogously, using lower convex hulls.

We note that we can avoid storing P twice: by maintaining only one tree T where every node v maintains both $e^+(v)$ and $e^-(v)$. We choose to implement the Partial Hull Tree and all c-queues as AVL trees because we want to have worst-case guarantees. To simplify our internal algorithm's logic, we define the leaves of our c-queue to be points that are the endpoints of their parent edges. With slight abuse of notation, we

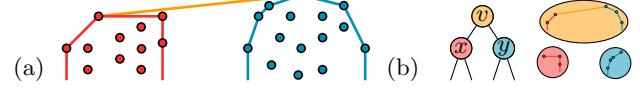


Figure 4: (a) Hulls $\mathbb{E}(x)$ and $\mathbb{E}(y)$. (b) The root v stores $\mathbb{E}^*(v) = \mathbb{E}(v)$. Its left child x stores $\mathbb{E}^*(x) = \mathbb{E}(x) \setminus \mathbb{E}(v)$.

refer indistinguishably between edges $\sigma \in \mathbb{E}(v)$ and their c -queue node. That is, for each node $w \in T$ there is a unique ancestor v of w where the bridge $e^+(w) = \sigma \in \mathbb{E}^*(v)$.

Algorithm 1 splitHull(left child x , balanced tree $\mathbb{E}(v)$)

```
( $\mathbb{E}_L, \mathbb{E}_R \leftarrow \mathbb{E}(v).Split(e^+(v))$ )
 $\mathbb{E}(x) \leftarrow \mathbb{E}_L(v).Join(\mathbb{E}^*(x))$ 
return ( $\mathbb{E}(x), \mathbb{E}_R$ )
```

3.1 Recomputing bridges using only edges

Let $v \in T$ have children x and y . Our new definition of bridges and c -queues allows for new logic to recompute the bridge $e^+(v)$ from $\mathbb{E}(x)$ and $\mathbb{E}(y)$ in $O(\log n)$ time. We show this through two lemmas:

In Lemma 3.2, we consider two edges $(\alpha, \beta) \in \mathbb{E}(x) \times \mathbb{E}(y)$. In our data structure, these are both roots of a subtree in their respective trees. We can decide in $O(1)$ time whether $e^+(v)$'s endpoints succeed or precede these edges: discarding at least one child of either α or β until we find one endpoint of $e^+(v)$.

In Lemma 3.3, let $e^+(v) = (a, b)$. Given a and $\beta \in \mathbb{E}(y)$, we can decide in $O(1)$ time whether b precedes or succeeds β . The result of these two lemmas is an $O(\log n)$ algorithm to find $e^+(v)$ (Algorithm 2).

LEMMA 3.2. Let $v \in T$ have children x and y . Given edges $\alpha \in \mathbb{E}(x)$ and $\beta \in \mathbb{E}(y)$ with center points l and r , respectively, we can decide whether $e^+(v)$'s endpoints succeed/precede α and β in $O(1)$ time using only arithmetic and comparisons.

Proof. The proof is a case distinction, depending on the slope of the segment \overline{lr} . Denote by (a, b) the (unknown) endpoints of $e^+(v)$. We consider three cases (where the first two are not mutually exclusive) and their consequence:

Slope(α) \leq Slope(\overline{lr}): a must precede α . Denote by (α_1, α_2) the endpoints of α . Consider the triangle $\Delta = (\alpha_1, l, r)$. Since $\text{slope}(\alpha) \leq \text{slope}(\overline{lr})$, and r per construction has an x -coordinate greater or equal to that of α_2 , the point α_2 must be on or under the edge \overline{lr} of this triangle (Figure 5). Per definition of convexity, the bridge $e^+(v)$ bounds a halfplane that contains all

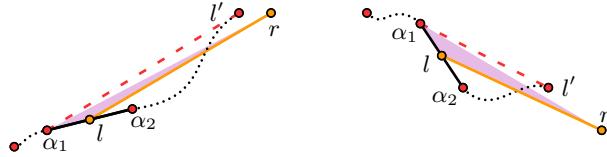


Figure 5: Lemma 3.2, case 1, for upward and downward slopes. The purple triangle denotes Δ .

points in $CH^+(\pi(v))$. Thus, the bridge $e^+(v)$ bounds a halfplane that must contain Δ .

We claim that any point l' in $\pi(x)$ right of α_1 cannot be an endpoint of $e^+(v)$. Indeed, any such point l' must lie on or above the line through $\overline{\alpha_1 r}$ (since the halfplane bounded by $e^+(v)$ must contain both α_1 and r). This implies that we have found a $l' \in \pi(x)$ such that the segment $\overline{\alpha_1 l'}$ has a greater slope than α : which contradicts the assumption that α was part of the upper convex hull of $\pi(x)$.

Slope(\overline{lr}) \leq Slope(β): b must succeed β . This proof is symmetric to the previous case.

Slope(α) $>$ Slope(\overline{lr}) $>$ Slope(β). Denote by m a vertical line separating $\pi(x)$ and $\pi(y)$ (we can compute such an m in constant time). Denote by γ the intersection point between the supporting lines of α and β . We prove that:

- if γ lies on or left of m , a must succeed α , and
- if γ lies on or right of m , b must precede β .

We prove the first subcase (Figure 6), as the second subcase is symmetric. Denote by Γ the area that can contain b , the area bounded by the halfplane supporting β and right of m . Because γ lies left of m , Γ is entirely contained in the halfplane bounded by the supporting line of β . Consider any vertex $l' \in \pi(x)$ preceding α , the segment $\overline{l'b}$ must be contained in the halfplane bounded by α . However, this implies that the segment $\overline{l'b}$ bounds a halfplane that excludes α_2 . Thus, a must succeed α . Note that if a succeeds α (and, a lies in the green area of the figure) then b may succeed or precede β . \square

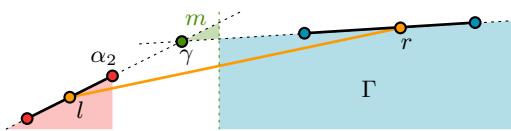


Figure 6: Lemma 3.2, case 3.

LEMMA 3.3. Let $v \in T$ be an internal node with children x and y . Given the left endpoint a of $e^+(v)$ and any edge $\beta \in \mathbb{E}(y)$, we can decide whether the right endpoint

b of $e^+(v)$ succeeds or precedes β in $O(1)$ time using only arithmetic and comparisons.

Proof. Denote by r the center point of β . The sequence of edges on the upper convex hull of $\pi(v)$ per definition has decreasing slope. It immediately follows that if $\text{Slope}(\overline{ar}) > \text{Slope}(\beta)$ then b must precede β . Similarly, if $\text{Slope}(\overline{ar}) < \text{Slope}(\beta)$ then b must succeed β . Finally, if $\text{Slope}(\overline{ar}) = \text{Slope}(\beta)$ then the points a, r and the endpoints of β are collinear. Since our bridge is the maximal segment bounding a halfplane that contains $\pi(v)$, b must succeed β . \square

COROLLARY 3.1. Let $v \in T$ have children x and y . Given $\mathbb{E}(x)$ and $\mathbb{E}(y)$ as balanced trees, we may compute the bridge $e^+(v)$ in $O(\log n)$ time (Algorithm 2).

Algorithm 2 FindBridge(node v , $\alpha \in \mathbb{E}(x)$, $\beta \in \mathbb{E}(y)$)

```

1: while ! ( $\alpha$ .isLeaf and  $\beta$ .isLeaf) do
2:    $(l, r) \leftarrow (\alpha, \beta)$ .getCenter()
3:   if Slope( $\alpha$ )  $\leq$  Slope( $\overline{lr}$ ) and ! $\alpha$ .isLeaf then
4:      $\alpha \leftarrow \alpha$ .leftChild
5:   if Slope( $\overline{lr}$ )  $\leq$  Slope( $\beta$ ) and ! $\beta$ .isLeaf then
6:      $\beta \leftarrow \beta$ .rightChild
7:   if neither first two cases apply then
8:     if  $\beta$ .isLeaf then
9:        $\alpha \leftarrow \alpha$ .rightChild
10:    else if  $\alpha$ .isLeaf then
11:       $\beta \leftarrow \beta$ .leftChild
12:    else
13:       $\gamma \leftarrow \text{intersectLines}(\alpha, \beta)$ 
14:       $m \leftarrow \text{verticalLineSeparating}(\pi(x), \pi(y))$ 
15:      if  $\gamma$  left of  $m$  then
16:         $\alpha \leftarrow \alpha$ .rightChild
17:      else
18:         $\beta \leftarrow \beta$ .leftChild
19: return  $(\alpha, \beta)$ 

```

3.2 The update algorithm using c-queues Now, we are ready to present our implementation of the dynamic algorithm in [40], using our edge-based logic. We call this algorithm OvL. Let us insert or delete a point $p \in P$ and denote by ρ the root-to-leaf path to p in T . We compute for all $v \in \rho$ the bridge $e^+(v)$. This allows us to restore all c-queues, and store $CH^+(P)$ in $\mathbb{E}^*(r)$ (where r is the root of T). Our algorithm works in two steps, that both take $O(\log^2 n)$ time:

Traversing down. We start at the root r where $\mathbb{E}^*(r) = \mathbb{E}(r)$. We traverse ρ from the root to p . For each node v , we assume we have $\mathbb{E}(v)$ as a balanced binary tree. Let p lie in the left child x of v ($p \in$

$\pi(x)$). We invoke $\text{splitHull}(x, \mathbb{E}(v))$: destroying $\mathbb{E}(v)$ and storing $\mathbb{E}_R(v)$ in v . The splitHull function gives us $\mathbb{E}(x)$ in $O(\log n)$ time, and with $\mathbb{E}(x)$ we recurse on x .

Bubbling up. For the leaf l that contains p , the bridge $e^+(l)$ equals (p, p) . We create or delete this bridge accordingly. From the downwards traversal in the previous step, we have the tree $\mathbb{E}(l)$ at l . We traverse ρ from l to the root. For each node v , we compute $\mathbb{E}(v)$ as follows. Let x and y be the children of v . If x is the child on the path, we computed $\mathbb{E}(x)$ before moving up to v and we have $\mathbb{E}(y)$ from our downwards traversal. We first invoke Algorithm 2 to compute $e^+(v)$ in $O(\log n)$ time (Corollary 3.1). We then compute $\mathbb{E}(v)$ joining $\mathbb{E}(x)$ and $\mathbb{E}(y)$ around the bridge $e^+(v)$.

THEOREM 3.1. *Let P be a two-dimensional point set. We can maintain the edges of $CH^+(P)$ subject to insertions and deletions in P as a balanced binary tree $\mathbb{E}(r)$ in $O(\log^2 n)$ worst-case time per update.*

The balanced binary tree $\mathbb{E}^*(r)$ at the root stores the h edges of $CH^+(P)$. Queries 1–6 can subsequently be answered in $O(\log h)$ time using the standard search algorithms over $\mathbb{E}^*(r)$ [13].

4 Improving the Solution

The update algorithm presented in [40] differs from our new approach in the following choices:

- It traverses the path ρ in T twice: once top-down to ensure that we can compute for all $v \in \rho$ the tree $\mathbb{E}(v)$, and once bottom-up to compute the bridges.
- It stores points in P multiple times (both in T and in the c-queues) requiring either double the space or pointers that point to non-contiguous data.
- It uses the split and join operations on binary trees. Although these have $O(\log n)$ theoretical running time, they are inefficient in practice.

In Appendix A, we resolve these issues. Our algorithm no longer maintains the convex hull $CH^+(P)$ in a balanced binary tree. Instead, we maintain a new structure that we call the Partial Bridge Tree (PBT).

DEFINITION 4. *The Partial Bridge Tree (PBT) stores a point set P in a leaf-based balanced binary tree T (sorted by x -coordinate). Each node $v \in T$ stores $e^+(v)$ and $e^-(v)$.*

We can cleverly navigate our Partial Bridge Tree to get two properties:

1. $\forall v \in T$, given an edge $e \in CH^+(\pi(v))$, we can find its successor on $CH^+(\pi(v))$ in $O(\log n)$ time.
2. $\forall v \in T$, given an edge $e \in CH^+(\pi(v))$, we can find the median edge of all edges preceding e on $CH^+(\pi(v))$, in $O(\log n)$ time.

The first property allows us to report $CH^+(P)$ in $O(h \log n)$ time. The second property naively allows us to answer Queries 1–6 in $O(\log^2 n)$ time. Hence, we obtain an output that is somewhere in between the implicit and explicit convex hull.

Maintaining the Partial Bridge Tree. The second property naively also gives us an $O(\log^3 n)$ update time to maintain the Partial Bridge Tree. Indeed, consider after inserting or deleting point p in P the path ρ from p to the root of T . To restore the Partial Bridge Tree, we need to compute for all $v \in \rho$ (with children x and y) the bridge $e^+(v)$. Algorithm 2 requires as input v plus two edges $(\alpha, \beta) \in \mathbb{E}(x) \times \mathbb{E}(y)$. We then recurse by replacing α with α' (the median successor, or predecessor, on $\mathbb{E}(x)$). In Section 3.2, we obtained α' in $O(1)$ time because we had $\mathbb{E}(x)$ as a balanced binary tree. Using Property 2, we may instead get α' in $O(\log n)$ time. Thus, we obtain an update algorithm with $O(\log^3 n)$ update time and $O(\log^2 n)$ query time just by replacing the functions ‘ $\alpha.\text{leftChild}$ ’ and ‘ $\alpha.\text{rightChild}$ ’ with the algorithm from Property 2.

We can, however, do better. We show that invoking Property 2 $O(\log n)$ times takes $O(\log n)$ worst-case total time. This creates a new algorithm for dynamic convex hull which we call **Eilice**. Eilice has a worst-case update time of $O(\log^2 n)$, matching OvL [40], and a query time of $O(\log n)$. This update algorithm no longer requires us to maintain c-queues. In addition, we no longer require the initialization step of OvL: ensuring that we traverse T exactly once. Thus, it circumvents the downsides of OvL at the cost of query time efficiency.

5 Rank-based Convex Hulls

Finally, we consider rank-based convex hulls. Let Y be a set of values, where the rank of $y \in Y$ is its index in the sorted order. We denote by P_Y the two-dimensional point set that is obtained by mapping each value in Y to (RANK, VALUE) and wish to dynamically maintain $CH(P_Y)$. The problem in this setting is that after inserting into or deleting from Y , the x -coordinate of linearly many points in P_Y changes. Changing a value y may change $CH(P_Y)$ by $\Theta(n)$ edges, even if y itself was not on the convex hull (Figure 14). The key observation to maintaining the convex hull in this setting is the following. After updating an element $y \in Y$ a bridge $e^+(v) = (a, b)$ in the Partial Hull Tree is updated if and only if y is in the subtree rooted at v . That is, if y is not in the subtree rooted at v then the x -coordinates of (a, b) may both increase or decrease by one, but the bridge $e^+(v)$ remains a segment between the same two values. Since the convex hull is implied by the set of all bridges in T , we may still maintain the Partial Hull Tree with the previous root-to-leaf update strategy.

Implicit bridges In a Partial Hull Tree T , the leaves store the values in P_Y , sorted by x -coordinate. I.e., we store the values of Y in the leaves of T in their stored order. For a node v with children x and y , the bridge $e^+(v)$ is the bridge between the convex hulls $CH^+(\pi(x))$ and $CH^+(\pi(y))$. For a bridge $e^+(v)$, we can no longer store the endpoints of the bridge explicitly: as the x -coordinate of all bridges may radically change after an update in Y . We define the implicit bridge $\varepsilon^+(v)$ which stores only the two values (y_1, y_2) corresponding to the endpoints of $e^+(v)$. At this point, we wish to note that we can easily maintain the Partial Hull Tree using implicit bridges with a factor $O(\log n)$ overhead. Indeed, we may run any of the two proposed algorithms. Whenever we need to consider a bridge $\alpha = e^+(x)$, we can get the corresponding values (y_1, y_2) from the implicit bridge. Then, we may perform a binary search over Y to obtain their corresponding ranks. Thus, at $O(\log n)$ overhead, we always have explicit access to the endpoints of α .

In Appendix B we show that we can cleverly navigate T to avoid this overhead. Our key contribution is that we show that we can perform a similar trick as for Eilice: performing $O(\log n)$ such rank queries in $O(\log n)$ total time. This gives the first dynamic algorithm for rank-based convex hulls with worst-case $O(\log^2 n)$ update time. Our solution has $O(\log h)$ query time (used as an extension on OVL) or $O(\log n)$ query time (when applied as an extension of Eilice).

6 Experiments

In order to examine the empirical performance of our data structures, we perform a series of tests across several types of input, comparing the performance of our structure variants with each other, and to state of the art implementations of static convex hull algorithms.

Test environment and implementations. For experiments we consider the following implementations:

Simplified OvL our simplified Partial Hull Tree structure using concatenable queues.

Eilice our simplified Partial Hull Tree structure without concatenable queues.

CHHN a Java implementation of the original Overmars and van Leeuwen structure [17].

Eddy CGAL implementation of Eddy's algorithm [21] ($O(nh)$ construction time).

GA CGAL implementation of Andrew's variant of the Graham scan algorithm [1] ($O(n \log n)$ construction time).

The implementations of our data structures are available at [4]. As a robustness check, we have compared the points we have found on the hull to those

found by static convex hull algorithms implemented in CGAL, and verified that they are in agreement.

We note that comparing implementations across programming languages is inherently a dubious task. Not only do the languages differ in their approach to things like memory management, which can greatly influence practical performance, but in this case, the comparisons are further troubled by the fact that geometric robustness has not been implemented for CHHN. The comparisons between the Java and C++ implementations should therefore not be seen as direct performance comparisons, but rather the Java implementation acts as an indicator of existing solutions.

The experiments are run on a cluster node with an Intel Xeon Gold 6226R CPU on a single core at 2.8GHz.

Test data. We generated four distinct data sets of 2^{20} points each, following Gamby and Katajainen [24] who perform experimental analysis on static convex hull algorithms. The first three categories are from [24], and the last is tailored to measure behaviour when all points lie on the convex hull. Recall that h denotes the number of points in $CH(P)$. We consider the following data sets:

Uniformly random data each point of P is drawn uniformly at random from a square; thus, the expected number of extremal points is $\Theta(\log n)$.

Disk-truncated data where each point is drawn uniformly at random, but only added to P when it lies in a pre-specified disk. Here, the expected number of extremal points is $\Theta(n^{1/3})$.

Bell data where data is generated according to a normal distribution (and thus, we expect $\Theta(\sqrt{\log n})$ extremal points).

Circular data where points are sampled from the boundary of a pre-specified disk; i.e. every data point belongs to the hull.

For the uniformly random and Bell data points, we follow the specifications of [24]. For the disk-truncated and circular points, we consider disks of radius 1000 centered at $(0, 0)$. We note that in all cases, contrary to Gamby and Katajainen [24], we do not restrict the points to have integer coordinates. This restriction served to avoid correctness issues that can arise in practice when comparing floating point values of finite precision. Integration with the CGAL geometry kernel allows us to circumvent these geometric robustness issues, at some cost of performance, opening up for use in applications that do not allow for such a restricted input.

Experiments. The contribution of this work is an updatable, queriable convex hull implementation for two-dimensional and rank-based datasets. We thus test

for the following categories:

Dynamic construction a simulated dynamic scenario in which we construct the hull of a point set by repeated extension.

Extension the time it takes to extend the point set P with an additional point set and obtain the convex hull of their union.

Queries the time it takes to perform a sequence of point-containment queries on an existing hull.

Updates the time it takes to perform an interspersed sequence of insertions and deletions on an existing hull.

The dynamic construction test serves to emulate behaviour one might encounter in a typical dynamic setting. The hull is repeatedly extended, so that queries can be answered often on the current state of the hull. This brings a clear disadvantage to the static algorithms, and thus we examine how various rates of extension affect their performance.

For extensions, the static algorithms simply rebuild the hull. This is to determine if the dynamic variations can compare to simply rebuilding the hull using a static algorithm, eliminating the need for a dynamic structure in cases where queries are needed on an irregular basis. Each measurement of extension adds 50,000 new points, while each set of updates consists of 1000 updates split evenly between insertions and deletions. There is no overlap between inserted and deleted points, and the order of updates is random, emulating the updates pattern one might encounter in practical applications.

Due to the speed at which queries can be answered, we for each measurement perform 2^{20} queries for points chosen according to a uniform distribution over a bounding box of the area in which the full test data resides.

Results and discussion. For ease of readability, we highlight only tests on specific data sets. The remainder can be found in Appendix E, and the raw results can be found in [4].

In Figure 8 we show the dynamic construction test using 500 point extensions. At roughly a million points and forward we see that exact Eilice beats the static variations, while the inexact implementations take over sooner.

In practical applications, one's access pattern might require access to the hull more often than once every 500 updates. For a scenario with access every 50 updates we get the results seen in Figure 9 where the clear benefit of a dynamic structure can be seen.

To explicitly showcase the difference in performance when comparing exact and inexact implementations, we show in Figure 7 the extension time on uniform data.

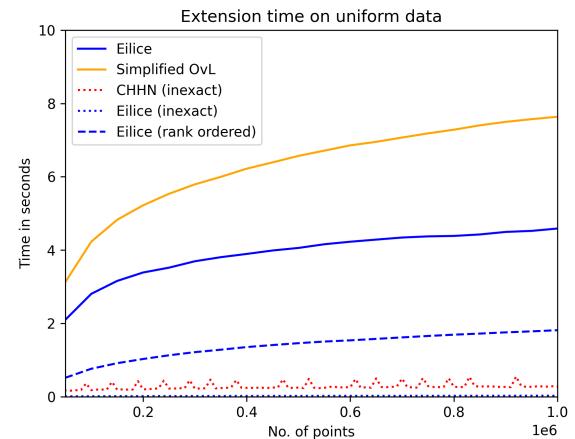


Figure 7: Extension time on uniform data.

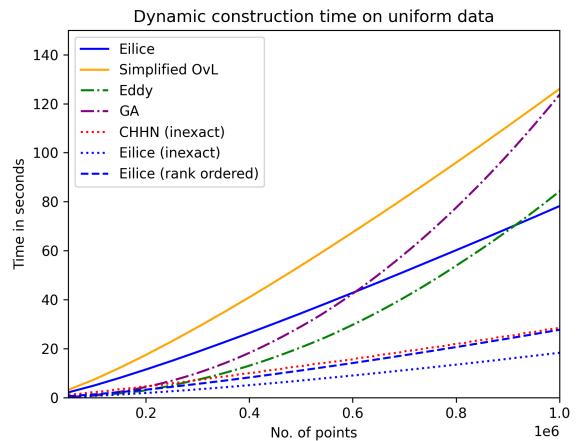


Figure 8: Dynamic construction time using 500 point extensions on uniform data.

What should be noted here is the gap between the inexact and exact data structures. Replacing simple floating point comparisons with more robust and involved operations takes a large toll on performance. This is perhaps not surprising, but it is necessary if one wants to guarantee correct comparisons, and thus correct behaviour. We also show the time spent extending the rank-ordered implementation. This is because rank-ordered integers avoid some of the pitfalls of geometric robustness, and thus fall into a similar category as the inexact implementations. Of note is that the rank-ordered implementation has to do additional bookkeeping, placing it somewhat in the middle of the extremes. We also note the logarithmic nature of the exact dynamic implementations. The jagged curve of CHHN does not stem from

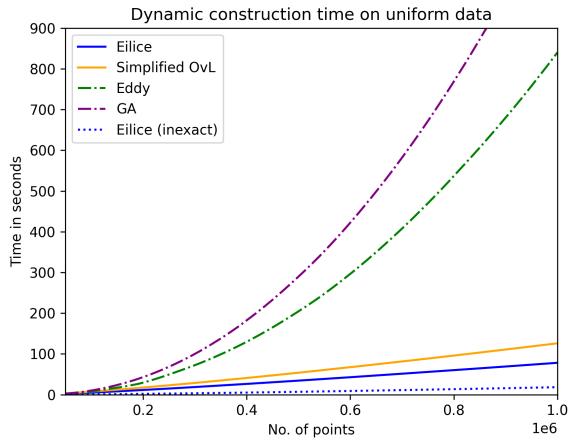


Figure 9: Dynamic construction time using 50 point extensions on uniform data.

the usage of an amortized structure, but rather the Java language’s built-in garbage collector periodically performing memory cleaning. The static algorithms have a large advantage here, requiring a single reconstruction for every 50,000 insertions, placing them far ahead of most of the competition.

To clarify the behaviour of extension we also show in Figure 10 the behaviour excluding the exact dynamic implementations. Although tempting to simply claim that our simplification causes a speedup of an order of magnitude compared to the regular Overmars and van Leeuwen structure, the difference here between Java and C++ muddles any conclusions. Here it is also clearer that the static algorithms have super-logarithmic growth, however, the constants associated with the exact implementations mean that data has to grow much larger for the exact dynamic implementations to extend faster.

For the queries, we specifically consider the circular and uniform random data. Recall that the expected number of points on the hull of the uniform data, h , is $O(\log n)$ for this point set. The benefit of maintaining concatenable queues is that they provide the ability to answer queries in $O(\log h)$ time rather than worst-case $O(\log n)$. In Figure 11, however, we see that the Eilice algorithm obtains similar performance to the simplified OvL solution, even whilst not having concatenable queues. This behaviour can be explained as follows: the Eilice query algorithm considers an edge $e^+(v) = \alpha$ on the upper convex hull. Then, Eilice either outputs the correct answer or considers the left (or right) child x of v . If the Eilice algorithm does not output the correct answer, there exist two cases. Either the bridge $e^+(x)$ is also on the convex hull, or

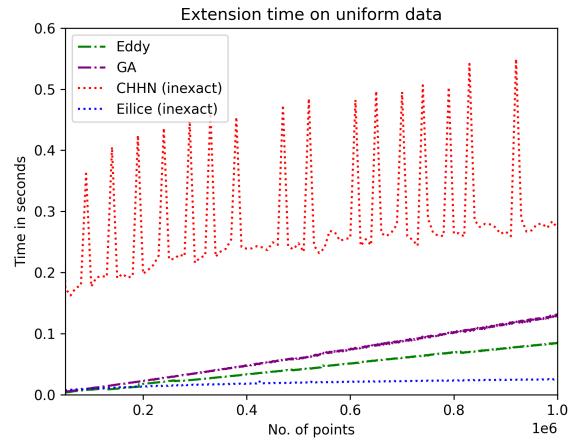


Figure 10: Extension time on uniform data of select implementations.

the x -coordinate of $e^+(x)$ succeeds the left endpoint of α . The Eilice algorithm makes a comparison that the simplified OvL does not make, *only* in the latter case. Due to how the tree is defined and constructed, that case is rare. Thus, in practice, the Eilice and simplified OvL algorithms oftentimes execute the same decision tree before reaching the desired output (even though Eilice has a worse worst-case guarantee).

For the update measurements, shown in Figure 12, we again see the logarithmic behaviour for both dynamic algorithms. As expected, Eilice beats simplified OvL by virtue of not having the overhead associated with maintaining concatenable queues.

Our experiments highlight several key points when deciding what algorithms to employ for dynamic convex hulls in practice. If geometric robustness is not necessary, one can achieve much greater performance. This might be the case in applications that accept erroneous queries, in which data is inherently already unreliable, or where data happens to be distributed enough that errors become unlikely. As an example of the latter, the uniform random data set was correctly handled by the inexact data structures, while the circular data was much more prone to precision errors.

If geometric robustness is necessary, but queries can be relegated to infrequent batches, static recomputations might be preferred, if the queries are infrequent enough or the data sizes are large enough.

If both geometric robustness and dynamic behaviour is required, or data is rank-based, Eilice is the clear winner. While leaving out the concatenable queues gives worse worst-case guarantees, it requires an adversarial query pattern for the overhead of Eilice to become worse.

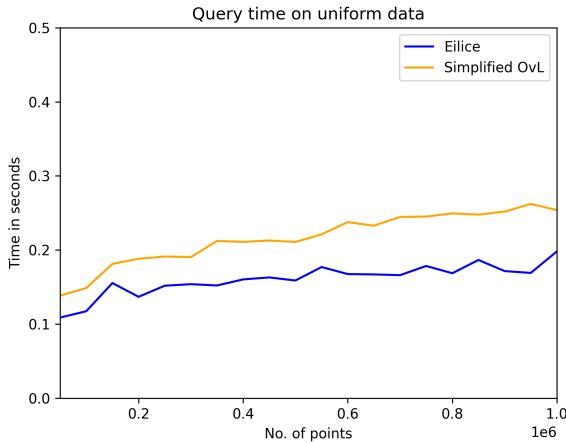
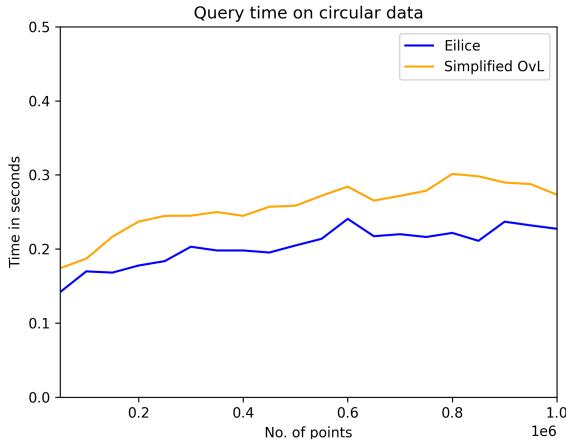


Figure 11: Query time on circular and uniform data.

To conclude: We simplify the bridge-finding procedure of Overmars and van Leeuwen, by a clever change from vertex based to edge-based computations. This simplification leads to much less error-prone implementations, at no practical cost. We implement and test our simplified data structure, along with an additional trade-off between queries and updates, by leaving out concatenable queues. The implementations are publicly available at [4]. We examine the practical implications of these simplifications, providing insight into not only how our structure performs, but also what factors one should consider when employing a dynamic convex hull data structure, with guidance for various use cases.

Finally, the simplified algorithm also easily extends to rank-ordered data, accommodating updates and queries in polylogarithmic time, thus giving the first data structure for dynamic maintenance of convex hulls of rank-ordered data with non-trivial update bounds.

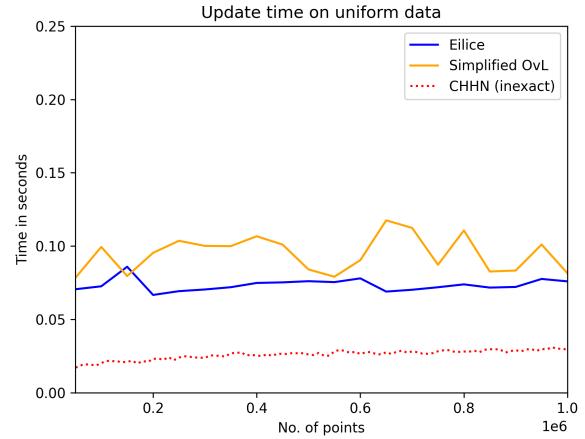


Figure 12: Time spent on an interspersed sequence of 500 insertions and 500 deletions on an existing convex hull on uniform data.

References

- [1] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inf. Process. Lett.*, 9(5):216–219, 1979.
- [2] Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 1979.
- [3] Manos Athanassoulis and Anastasia Ailamaki. Bf-tree: approximate tree indexing. *Proceedings of the 40th International Conference on Very Large Databases*, 2014.
- [4] Anonymous authors. Dynamic Convex Hull implementations. <https://anonymous.4open.science/r/DynamicConvexHull-1E62/>, 2023.
- [5] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hull. Technical report, Technical Report GCG53, The Geometry Center, MN, 1993.
- [6] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000.
- [7] Kristin P Bennett and Erin J Bredensteiner. Geometry in learning. *MAA NOTES*, 2000.
- [8] Antoine Bordes and Léon Bottou. The huller: a simple and efficient online svm. In *Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16*, pages 505–512. Springer, 2005.
- [9] Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002. Proceedings., 2002.

- [10] Chee-Yong Chan and Yannis E Ioannidis. Bitmap index design and evaluation. *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998.
- [11] Timothy M Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 1996.
- [12] Timothy M Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *Journal of the ACM (JACM)*, 2001.
- [13] Timothy M Chan. Three problems about dynamic convex hulls. *Proceedings of the twenty-seventh annual symposium on Computational geometry*, 2011.
- [14] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R Smith. The onion technique: Indexing for linear optimization queries. *ACM SIGMOD International Conference on Management of data*, 2000.
- [15] Asdrúbal López Chau, Xiaou Li, and Wen Yu. Convex and concave hulls for classification with support vector machine. *Neurocomputing*, 2013.
- [16] Yun Chi, Hakan Hacigümüş, Wang-Pin Hsiung, and Jeffrey F Naughton. Distribution-based query scheduling. *Proceedings of the VLDB Endowment*, 2013.
- [17] Yun Chi, Hakan Hacigümüş, Wang-Pin Hsiung, and Jeffrey F Naughton. Java implementation convex hull. https://github.com/yunchi/Dynamic_Convex_Hull, 2013.
- [18] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. icbs: incremental cost-based scheduling under piecewise linear slas. *Proceedings of the VLDB Endowment*, 4(9):563–574, 2011.
- [19] Jose Alberto Cisneros. Maintenance of the convex hull of a dynamic set. Master’s thesis, 2007.
- [20] David Crisp and Christopher J Burges. A geometric interpretation of v-svm classifiers. *Advances in neural information processing systems*, 1999.
- [21] William F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3(4):398–403, 1977.
- [22] Paolo Ferragina and Giorgio Vinciguerra. Learned data structures. *Recent Trends in Learning From Data: Tutorials from the INNS Big Data and Deep Learning Conference (INNSBDDL2019)*, 2020.
- [23] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 2020.
- [24] Ask Neve Gamby and Jyrki Katajainen. Convex-hull algorithms: Implementation, testing, and experimentation. *Algorithms*, 2018.
- [25] Xu Gao and Fusheng Yu. Trajectory clustering using a new distance based on minimum convex hull. *2017 Joint 17th World Congress of International Fuzzy Systems Association and 9th International Conference on Soft Computing and Intelligent Systems (IFSA-SCIS)*, 2017.
- [26] Thomas Giorginis, Stefanos Ougiaroglou, Georgios Evangelidis, and Dimitris A Dervos. Fast data reduction by space partitioning via convex hull and mbr computation. *Pattern Recognition*, 2022.
- [27] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters (IPL)*, 1972.
- [28] Sun-Young Ihm, Ki-Eun Lee, Aziz Nasridinov, Jun-Seok Heo, and Young-Ho Park. Approximate convex skyline: a partitioned layer-based index for efficient processing top-k queries. *Knowledge-Based Systems*, 2014.
- [29] MA Jayaram and Hasan Fleyeh. Convex hulls in image processing: a scoping review. *American Journal of Intelligent Systems*, 2016.
- [30] Hamid Reza Khosravani, AE Ruano, and Pedro M Ferreira. A convex hull-based data selection method for data driven models. *Applied Soft Computing*, 2016.
- [31] David G Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM journal on computing*, 1986.
- [32] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. *Proceedings of the 2018 international conference on management of data*, 2018.
- [33] Luca Liparulo, Andrea Proietti, and Massimo Panella. Fuzzy clustering using the convex hull as geometrical model. *Advances in Fuzzy Systems*, 2015.
- [34] Kaiqi Liu and Jianqiang Wang. Fast dynamic vehicle detection in road scenarios based on pose estimation with convex-hull model. *Sensors*, 2019.
- [35] Dragos D Margineantu and Thomas G Dietterich. Pruning adaptive boosting. *ICML*, 1997.
- [36] Michael E Mavroforakis, Margaritis Sdralis, and Sergios Theodoridis. A novel svm geometric algorithm based on reduced convex hulls. *18th International Conference on Pattern Recognition (ICPR’06)*, 2006.
- [37] Kyriakos Mouratidis. Geometric approaches for top-k queries [tutorial]. 2017.
- [38] George Ostroumov and Nagiza F Samatova. On fastmap and the convex hull of multivariate data: toward fast and robust dimension reduction. *IEEE transactions on pattern analysis and machine intelligence*, 2005.
- [39] Mark H Overmars. *The design of dynamic data structures*. Springer Science & Business Media, 1983.
- [40] Mark H Overmars and Jan van Leeuwen. Dynamically maintaining configurations in the plane (detailed abstract). *Proceedings of the twelfth annual ACM Symposium on Theory of Computing*, 1980.
- [41] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.
- [42] Hongchao Qin, Rong-Hua Li, Ye Yuan, Guoren Wang, Lu Qin, and Zhiwei Zhang. Mining bursting core in large temporal graphs. *Proceedings of the VLDB Endowment*, 2022.
- [43] Jun Rao and Ross. Kenneth. Cache conscious indexing for decision-support in main memory. *Conference on Very Large Data Bases (VLDB)*, 1999.

- [44] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Density-based clustering in spatial databases: The algorithm gdbcscan and its applications. *Data mining and knowledge discovery*, 1998.
- [45] Di Wang, Hong Qiao, Bo Zhang, and Min Wang. Online support vector machine based on convex hull vertices selection. *IEEE transactions on neural networks and learning systems*, 2013.
- [46] Tom F Wilderjans, Eva Ceulemans, and Kristof Meers. Chull: A generic convex-hull-based model selection method. *Behavior research methods*, 2013.
- [47] Da Yan, Zhou Zhao, and Wilfred Ng. Efficient algorithms for finding optimal meeting point on road networks. *Proceedings of the VLDB Endowment*, 2011.
- [48] Bo Yuan and Chew Lim Tan. Convex hull based skew estimation. *Pattern Recognition*, 2007.

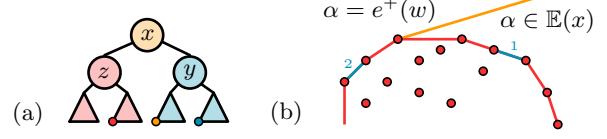


Figure 13: (a) A node v with children x and y . We show the medians as vertices with the same colour as the corresponding node.

A Eilice: updates without c-queues

The update algorithm OvL has three components that make it practically less efficient:

- It traverses the path ρ in T twice: once top-down to ensure that we can compute for all $v \in \rho$ the tree $E(v)$, and once bottom-up to compute the bridges.
- It stores points in P multiple times (both in T and in the c-queues) requiring either double the space or pointers that point to non-contiguous data.
- It uses the split and join operations on binary trees. Although these have $O(\log n)$ theoretical running time, are very inefficient in practice.

In this subsection, we show that we may avoid maintaining c-queues and these downsides by navigating only T . As a result, we obtain different outputs than the OvL algorithm. We no longer are able to store $CH(P)$ in a balanced binary tree. Instead, we ensure that at all times we can report the h points on $CH(P)$ in $O(h \log n)$ time. In addition, we show that we facilitate convex hull queries in $O(\log n)$ time as opposed to $O(\log h)$ time.

Key definitions. Definition 4 gives the definition of our data structure: the Partial Bridge Tree T (PBT). Let $v \in T$ be an internal node, then per definition, v has two children. The *median* $med(v)$ is the leftmost point in the subtree of the right child of v (see Figure 13 (a)). Our key observation is that for every node v : the bridge $e^+(v)$ (when projected onto the x -axis) is an x -interval that contains $med(v)$:

LEMMA A.1. *Let v be a node in T and e be the corresponding bridge. Then $e^+(v)$ either has $med(v)$ as its right endpoint or has two endpoints that precede and succeed $med(v)$, respectively.*

Proof. Let v have children x and y . The lemma follows from the fact that $med(v)$ is the leftmost point in $\pi(y)$, all points in $\pi(x)$ precede $med(v)$, and $e^+(v) = (a, b) \in \pi(x) \times \pi(y)$. \square

We may use this lemma, together with the following observation to obtain an equivalent to Corollary 3.1:

OBSERVATION 1. Let $w \in T$ and denote by $\alpha = e^+(w) = (a, b)$. For any vertex l in the left subtree under w with bridge $\gamma = e^+(l)$, there are two cases (see Figure 13(b)):

1. The left endpoint of γ succeeds a . Then all bridges in the right subtree of l are either not on $\mathbb{E}(x)$ (as they are ‘covered’ by α) or also succeed a .
2. The right endpoint of γ precedes a . Then $\gamma \in \mathbb{E}(x)$.

LEMMA A.2. Let $v \in T$ have children x and y . Recall that $\mathbb{E}(x)$ and $\mathbb{E}(y)$ are the convex hull edges of $CH^+(\pi(x))$ and $CH^+(\pi(y))$ in their cyclical ordering. Given edges $\alpha = e^+(x)$ and $\beta = e^+(y)$, we may compute the bridge $e^+(v)$ in $O(\log n)$ time (Algorithm 2+3).

Proof. We no longer have access to $\mathbb{E}(x)$ and $\mathbb{E}(y)$ as balanced binary trees. In Algorithm 2, we consider edges $(\alpha, \beta) \in \mathbb{E}(x) \times \mathbb{E}(y)$ and we use Lemma 3.2 to decide whether the bridge endpoints $e^+(v)$ precede or succeed α . Suppose that the left endpoint of $e^+(v)$ precedes α (all other scenarios are symmetrical). Then, we replace α by an edge $\alpha' \in \mathbb{E}(x)$ that precedes α . In Section 3, we obtain α' by simply replacing α with its left child in the balanced binary tree on $\mathbb{E}(x)$.

Since we no longer have this binary tree, for any edge $\alpha \in \mathbb{E}(x)$ its left child (or right child) is no longer well-defined. Thus all code lines that say $\alpha.\text{leftChild}$ (or $\alpha.\text{rightChild}$) were no longer well-defined. However, for all $\alpha \in \mathbb{E}(x)$ there exists a node $w \in T$ where $\alpha = e^+(w)$. We now note that Algorithm 3 gives us an edge α' preceding α on $\mathbb{E}(x)$.

This leads to a simple alteration of Algorithm 2. Note that the function $\alpha.\text{rightChild}()$ can be defined analogously. We replace all calls:

- $\alpha.\text{leftChild}$ by $\alpha.\text{leftChild}()$,
- $\beta.\text{leftChild}$ by $\beta.\text{leftChild}()$,
- $\alpha.\text{rightChild}$ by $\alpha.\text{rightChild}()$,
- $\beta.\text{rightChild}$ by $\beta.\text{rightChild}()$.

In each iteration of the while loop, either α is a leaf in T or an edge on $\mathbb{E}(x)$. We may call invoke the function $\text{leftChild}()$ at most $O(\log n)$ times until we reach a leaf of T . Thus, Algorithm 2 terminates in $O(\log n)$ time and computes the bridge $e^+(v)$. \square

Algorithm 3 $\alpha.\text{leftChild}()$ for $\alpha = e^+(w)$ and $\alpha \in \mathbb{E}(x)$

```

1:  $l \leftarrow$  the left child of  $w$  in  $T$ 
2:  $\gamma \leftarrow e^+(l)$ 
3: if  $\gamma.x$  succeeds  $\alpha.x$  then
4:   return  $l.\text{leftChild}()$ 
5: return  $(e^+(l), l)$ 

```

LEMMA A.3. Given the Partial Bridge Tree T , we can report the h edges on $CH^+(P)$ in $O(h \log n)$ time.

Proof. This lemma follows almost immediately from Observation 1 and the $\text{leftChild}()$ and $\text{rightChild}()$ functions. Indeed, let r be the root of T , then $e^+(r) = \alpha \in \mathbb{E}(r) = CH^+(P)$. We recursively invoke $e^+(r).\text{leftChild}()$ to obtain an edge $\beta \in CH^+(P)$. All edges of $CH^+(P)$ in between α and $\beta = e^+(w)$, per definition succeed α and precede β . Thus, we may recurse into the right subtree of w to obtain vertices $x \in T$. We may again apply Observation 1 to note that if $e^+(x) \in CH^+(P)$ if and only if it precedes β and succeeds α . This way, each time we explore a new subtree in T we find at least one edge of $CH^+(P)$. Since our function recurses at most $O(\log n)$ times before it reaches a leaf of T and thus we output $CH^+(P)$ in $O(h \log n)$ time. \square

Update algorithm. Let p be a point that is inserted or deleted in P . We restore the Partial Bridge Tree as follows: we do a root-to-leaf traversal in T to the leaf l that contains p : spending $O(1)$ time per node. For deletions, we may avoid this by following a pointer to the leaf in T . Denote by ρ the path from l to the root of T . For all $v \in \rho$, bottom-up, we use Lemma A.2 to compute $e^+(v)$ and this restores the Partial Bridge Tree. Thus, we have an $O(\log^2 n)$ update algorithm to maintain our Partial Bridge Tree.

A.1 Supporting queries What remains is to show that our data structure can answer Queries (1)-(6) in $O(\log n)$ time. The standard algorithms to answer these queries assume access to a balanced binary tree $\mathbb{E}(r)$ that stores the upper convex hull edges in their sorted order. Each of these algorithms subsequently does the following: they consider an edge $\alpha \in \mathbb{E}(r)$. They can either answer the query immediately using α or discard all edges of the convex hull preceding or succeeding α . Doing this for both the upper and lower convex hull allows queries can be answered in $O(\log h)$ time where h is the number of edges in $\mathbb{E}(r)$. Using Algorithm 3 to navigate T instead of $\mathbb{E}(r)$ allows our approach to immediately answer queries in $O(\log n)$ time instead. To illustrate this, we show how to answer Query (4): deciding whether a query point q lies in $CH(P)$.

LEMMA A.4. Let q be an arbitrary query point in \mathbb{R}^2 . Using our data structure we can test if $q \in CH(P)$ using our Partial Hull Tree in $O(\log n)$ time.

Proof. Note that q lies in $CH(P)$ if and only if it lies in both $CH^+(P)$ and $CH^-(P)$. The point q lies in $CH^+(P)$ if and only if there exists a unique edge

$\gamma^* \in \mathbb{E}(r)$ where: q succeeds the left endpoint of γ^* , q precedes the right endpoint of γ^* . Moreover, q must lie below the line through γ^* . Denote by r the root of the Partial Hull Tree.

Per definition, $e^+(r) \in \mathbb{E}(r)$. Set $\alpha \leftarrow e^+(r)$. We test in $O(1)$ time whether q succeeds the left endpoint of α and precedes the right endpoint of α . If both conditions are true then $\gamma^* = \alpha$, we output whether q is below the line through α . If q precedes the left endpoint of α then γ^* must precede α on the upper convex hull. Thus, we may step left from α using Algorithm 3. It follows that in $O(\log n)$ steps, we find γ and answer the query accordingly. \square

Maintaining our data structure, with the above query algorithms, implies the following theorem:

THEOREM A.1. *Let P be a two-dimensional point set. We can store P in an $O(n)$ size data structure with $O(\log^2 n)$ worst-case time per update such that we may report the h edges on $CH(P)$ in $O(h \log n)$ worst-case time. Moreover, we support all convex hull queries in $O(\log n)$ worst-case time.*

B Ranked-based convex hulls

In this section, we consider rank-based convex hulls. Let Y be a set of values, where the rank of $y \in Y$ is its index in the sorted order. We denote by P_Y the two-dimensional point set that is obtained by mapping each value in Y to (RANK, VALUE) and wish to dynamically maintain $CH(P_Y)$. The problem in this setting is that after inserting into or deleting from Y , the x -coordinate of $O(n)$ points in P_Y changes. Changing a value y may change $CH(P_Y)$ by $O(n)$ edges, even if y itself was not on the convex hull (Figure 14). The key observation to maintaining the convex hull in this setting is the following. After updating an element $y \in Y$ a bridge $e^+(v) = (a, b)$ in the Partial Hull Tree is updated if and only if y is in the subtree rooted at v . That is, if y is not in the subtree rooted at v then the x -coordinates of (a, b) may both increase or decrease by one, but the bridge $e^+(v)$ remains a segment between the same two values. Since the convex hull is implied by the set of all bridges in T , we may still maintain the Partial Hull Tree with the previous root-to-leaf update strategy.

Implicit bridges In a Partial Hull Tree T , the leaves store the values in P_Y , sorted by x -coordinate. I.e., we store the values of Y in the leaves of T in their stored order. For a node v with children x and y , the bridge $e^+(v)$ is the bridge between the convex hulls $CH^+(\pi(x))$ and $CH^+(\pi(y))$. For a bridge $e^+(v)$, we can no longer store the endpoints of the bridge explicitly: as the x -coordinate of all bridges may radically change after an update in Y . We define the

implicit bridge $\varepsilon^+(v)$ which stores only the two values (y_1, y_2) corresponding to the endpoints of $e^+(v)$. At this point, we wish to note that we can easily maintain the Partial Hull Tree using implicit bridges with a factor $O(\log n)$ overhead. Indeed, we may run any of the two proposed algorithms. Whenever we need to consider a bridge $\alpha = e^+(x)$, we can get the corresponding values (y_1, y_2) from the implicit bridge. Then, we may perform a binary search over Y to obtain their corresponding ranks. Thus, at $O(\log n)$ overhead, we always have explicit access to the endpoints of α .

In the remainder of this section we show that we can cleverly navigate T to avoid this overhead. To this end, we recall that for a vertex v its median $med(v)$ was the leftmost child in the right subtree rooted at v . We define the widths $\omega_1^+(v)$ (and $\omega_2^+(v)$) to be the rank-difference between y_1 and $med(v)$ (and y_2 and $med(v)$). Finally, we denote $\omega(e^+(x)) = \omega_1^+(v) + \omega_2^+(v)$. See Figure 14(b) for an example. In our Partial Hull Tree, we now require that each $v \in T$ stores the size of its subtree, a pointer to the median of v $med(v)$, and two implicit bridges $\varepsilon^+(v), \varepsilon^-(v)$ plus their widths $\omega_1^+(v), \omega_2^+(v), \omega_1^-(v), \omega_2^-(v)$. We note:

LEMMA B.1. *Let $v \in T$, and denote by r the rank of $med(v)$. Given r and v , we can compute the bridge $e^+(v)$ in $O(1)$ time.*

Proof. The x -coordinate of the left endpoint of $e^+(v)$ is simply r minus $\omega_1^+(v)$. The x -coordinate of the right endpoint of $e^+(v)$ is r plus $\omega_1^+(v)$. The y -coordinates of both endpoints are stored in the implicit bridge $\varepsilon^+(v)$. \square

B.1 Updates using c-queues Recall that for any $v \in T$, $\mathbb{E}(v)$ was a balanced binary tree on the convex hull edges $CH^+(\pi(v))$ in their cyclical ordering. If v had a child x , then the concatenable queue $\mathbb{E}^*(x)$ was defined as the balanced binary tree $\mathbb{E}(X) \setminus \mathbb{E}(v)$. We now define $\mathcal{E}(v)$ and $\mathcal{E}^*(x)$ analogously, where the inner nodes of the balanced binary trees store implicit edges bridges instead. Moreover, we demand that every $\alpha \in \mathcal{E}(x)$ stores the sum over all its descendants β of $\omega(\beta)$. Note that these sums can be maintained at

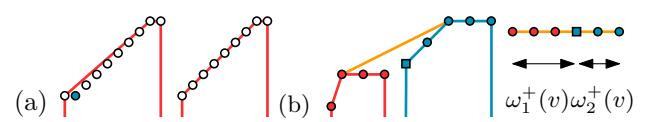


Figure 14: (a) A set Y of values, mapped to P_Y . Deleting the blue value adds $O(n)$ edges to $CH(P_Y)$. (b) The bridge $e^+(v)$, the vertex $med(v)$ as a square and the corresponding widths.

$O(\log n)$ time per insertion, deletion, split or join in the tree. Moreover, for each tree rotation, these sums can be updated in $O(1)$ time. When storing the convex hull for rank-ordered data, we demand that each node $v \in T$ stores $\mathcal{E}^*(v)$; with the aforementioned sums in each node.

Let us insert or delete a value $y \in Y$. The update algorithm consists of three components which we will explain in order:

- an initialization over the root-to-leaf traversal in T to y ,
- a bubble-up update strategy, where we compute for each node v on the leaf-to-root path the new bridge $e^+(v)$. We compute the bridge with its exact coordinates; from which we will derive $\varepsilon^+(v)$ and the widths in $O(1)$ time. Finally,
- a search algorithm to recompute for v the bridge $e^+(v)$.

Traversing down. We start at the root r where $\mathcal{E}^*(r) = \mathcal{E}(r)$. We do a root-to-leaf traversal to y . For each node v that we encounter, we store the rank of $med(v)$. We assume that as we arrive at a node v , we have $\mathcal{E}(v)$ as a balanced binary tree. Let $y \in \pi(x)$ for the left child x of v . We invoke $\text{splitHull}(x, \mathcal{E}(v))$, store $\mathcal{E}_R(v)$ in v . We compute $\mathcal{E}(x)$ from the result of splitHull and recurse.

Bubbling up. In the leaf l that contains y , the bridge $e^+(l)$ equals (y, y) and $med(l) = y$. We then consider each node v on the leaf-to-root path from l in T . Note that we have stored the rank r of $med(v)$. Using r , $\mathcal{E}(x)$, and $\mathcal{E}_R(v)$, we recompute $e^+(v)$. From $e^+(v)$ and r , we compute $\varepsilon^+(v)$ and the corresponding widths. This allows us to compute $\mathcal{E}(v)$ and continue upwards in T . If our update requires us to rotate v in T , we recompute the bridge of all rotating nodes.

Recomputing bridges. Given a node $v \in T$, the rank r of $med(v)$, $\mathcal{E}(x)$, and $\mathcal{E}_R(v)$ we do as follows. Denote by x the left child of v and by z the right child of v . From r , we can compute the ranks of $med(x)$ and $med(z)$ in $O(1)$ time. Given $\mathcal{E}_R(v)$ we can compute $\mathcal{E}(z)$ from $\mathcal{E}^*(z)$ using a single join operation. The root of $\mathcal{E}(x)$ is $\alpha' = \varepsilon^+(x)$. We apply Lemma B.1 to compute $\alpha = e^+(x)$ in $O(1)$ time. Similarly, we obtain $\beta = e^+(z)$ in $O(1)$ time. It follows that we may apply Lemma 3.2.

Suppose that the Lemma indicates that we may discard the subtree of $\mathbb{E}(x)$ right of α . Let $\gamma = e^+(w)$ be the left child of α (for some $w \in T$). Denote by $\gamma' = \varepsilon^+(w)$ the left child of γ' . The rank of the right endpoint of γ is equal to the rank of the left endpoint of α , minus the width of all edges between γ and α . Thus, we may compute the edge γ from γ' in $O(1)$ time by subtracting the width sum stored in γ . We can perform

a symmetrical procedure to compute an edge $\zeta \in \mathbb{E}(z)$ and recurse to compute $e^+(v)$. Doing this for all $v \in T$ on the leaf-to-root path from the update implies the following theorem:

THEOREM B.1. *Let Y be a dynamic set of values. We can maintain the edges of $CH^+(P_Y)$ (stored as implicit edges $\varepsilon(w)$) subject to insertions and deletions in P as a balanced binary tree $\mathcal{E}(r)$ in $O(\log^2 n)$ worst-case time per update.*

What remains is to show that we can support queries in $O(\log h)$ time. Let r be the root of the Partial Hull Tree T and denote by $\mathcal{E}(r)$ the corresponding concatenable queue. The root of $\mathcal{E}(r)$ is the implicit bridge $\varepsilon^+(r)$. We may maintain the rank of $med(r)$ at no additional overhead. Thus by Lemma B.1, we may compute the edge α at the root of $\mathbb{E}(r)$ in $O(1)$ time. From here, we can perform queries in the same way that we find bridges: deciding whether to go into the left or right subtree of our current edge in $\mathcal{E}(r)$, and computing the explicit edge at the root of the new subtree in $O(1)$ time. It follows that we may answer Queries (1)-(6) in $O(\log h)$ time (where h is the number of edges on the convex hull of P_Y).

B.2 The update algorithm without c-queues
 Let $v \in T$ and r be the rank of $med(v)$. Finally let x be any child of v . In the previous section we showed that we may compute the rank r' of $med(x)$ from r in $O(1)$ time by subtracting from (or adding to) r the size of the other subtree of v . By Lemma B.1, we may use r' to compute $e^+(x)$ from $\varepsilon^+(x)$ in $O(1)$ time. This implies that by maintaining our new concatenable queues $\mathcal{E}^*(x)$, we may run the algorithm of Section A, without any additional overhead (other than maintaining for every node $v \in T$ the size of its subtree).

Our repository also includes implementations for maintaining rank-based convex hulls.

C Applying Dynamic Convex Hull

We review three applications of a dynamic convex hull in VDLB.

Query scheduling. For the query scheduling problem we consider incoming database queries, which each come with associated costs and gains. Efficient queries may be prioritized, whereas expensive queries may even be dropped if they bring too little gain compared to the other current queries. The goal is to prioritize queries to maximize the gains. At VDLB 2011, Chi, Moon, and Hacıgümüş [18] study this problem, assuming that the cost/gain function of queries follows a piecewise linear function. I.e., the gains of answering a query diminish piecewise linearly with the response

time. They present a heuristic optimizer that relies upon a dynamic convex hull algorithm. This convex hull algorithm maps (dualizes) the piecewise linear functions to two-dimensional points. Because they work with dualized points, they need to maintain the convex hull explicitly. They maintain the convex hull using the Overmars and van Leeuwen algorithm. They subsequently can use the convex hull to deduce which queries to prioritize. At VDLB 2013, Chi, Hacigümüş, Hsing, and Naughton [16] study the same problem, under the assumption that there exists some underlying cost distribution on query time per query type. On a high level, they design an equation that maps every query q to a two-dimensional point which they call their Shepard score. They subsequently want to prioritize queries with critical Shepard scores. To this end, they maintain the Shepard scores (dualized) in the convex hull data structure by Overmars and van Leeuwen. The authors present their own Java implementation to maintain this hull, which can be found at [17].

Mining bursty subgraphs In temporal graphs, each edge can be represented as a triple (u, v, t) , where u, v are two end nodes of one edge and t denotes the interaction time between u and v [42]. A time-frame is *bursty* whenever a large number of events occurs in a short time. In VDLB 2022, Qin, Li, Yuan, Wang, Qin and Zhang [42] study the problem of ‘mining’ bursty patterns in a temporal graph. They present a dynamic program, that for every vertex u in the graph computes the cumulative density of a ‘front’ from u in the graph G over time. To compute this front, they repeatedly recompute the (lower) convex hull of their function from u . This step may be replaced by a dynamic convex hull algorithm.

PGM index Ferragina and Vinciguerra [23] study a dynamic sorted set of values Y subject to rank queries. They map Y to the point set P_Y and choose some $\varepsilon > 0$. A segment s ε -covers a consecutive set of values (y_1, \dots, y_k) whenever s is within distance ε of all corresponding points (p_1, \dots, p_k) . An ε -cover is a set of segments S that together ε -cover all points in P_Y . Denote by m_ε the minimal size for an ε -cover. The authors present an amortized $O(\log n)$ algorithm to maintain an ε -cover. They claim to support rank queries in $O(\log n(\log m_\varepsilon + \log \varepsilon))$ time. Their approach (under the hood) maintains the convex hull of P_Y through the logarithmic method: splitting Y over buckets B , and maintaining for the points P_B a minimal ε -cover (through indirectly maintaining $CH(P_B)$). Their approach has the same pitfall as hull queries under the logarithmic method: when Y is split over buckets, the complexity $CH(P_B)$ (and thereby the complexity of the ε -cover) may significantly exceed

that of $CH(P_Y)$. Without an algorithm to maintain $CH(P_Y)$ explicitly, their query time is $O(\log n(\log n + \log \varepsilon))$.

D The Logarithmic Method and Convex Hull

Here we describe the logarithmic technique applied to the convex hull, and why doing so leads to incorrect query algorithms and hulls.

Consider P under insertions and Graham’s scan algorithm (which uses $O(n)$ time for a sorted input set). We store P in buckets B_i of 2^i elements, where each B_i is either full or empty. Denote by $\pi(B) \subset P$ the points stored in a bucket B . For each bucket B we store the convex hull $CH(\pi(B))$. Whenever we insert an element, we insert it in B_2 . Suppose that an insertion causes B_2 to be full and let the first k buckets be full. We then merge buckets B_2, \dots, B_k into the new bucket B_{k+1} . During the merge, we ensure that the new bucket B_{k+1} contains all values in sorted order in $O(2^{k+1})$ time by the same procedure as merge sort. Then, we construct the convex hull in $O(2^{k+1})$ time using Graham’s scan. This procedure has $O(\log n)$ amortized update time. However it does not maintain the convex hull of P , nor does it answer convex hull queries correctly:

Consider (Figure 15) a set P of red and blue points where the convex hull of P contains three points. When we split the convex hull over two buckets (red and blue) both convex hulls contain many points. Moreover, suppose that we query whether a point q lies in $CH(P)$ (the black square). It may be that q is in none of the convex hulls $CH(B_i)$, even though it does lie in $CH(P)$. This inconsistency causes [23] to store something slightly different from advertised (Appendix C).

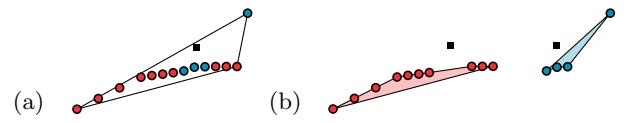


Figure 15: (a) A point set P and its convex hull. (b) When we split P over two buckets B, B' , the hulls $CH(\pi(B))$ and $CH(\pi(B'))$ have considerably higher complexity, and don’t contain the black query square.

E Additional Results

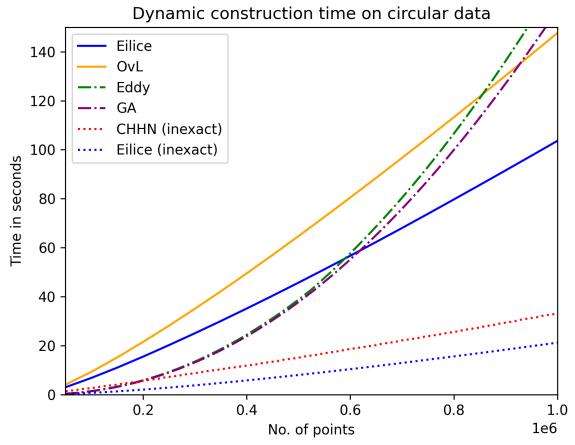
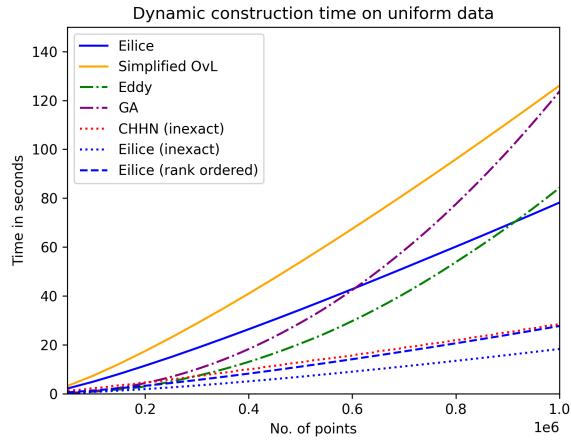
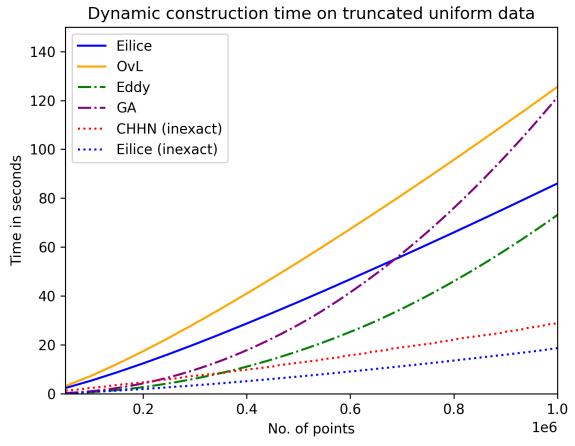
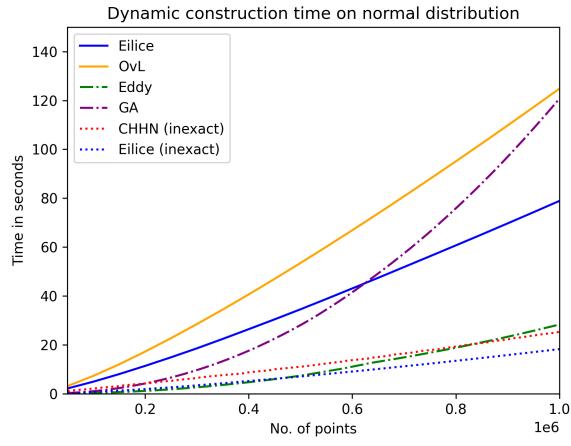


Figure 16: Dynamic construction time using 500 point extensions on various distributions.

Figure 17: Dynamic construction time using 500 point extensions on various distributions.

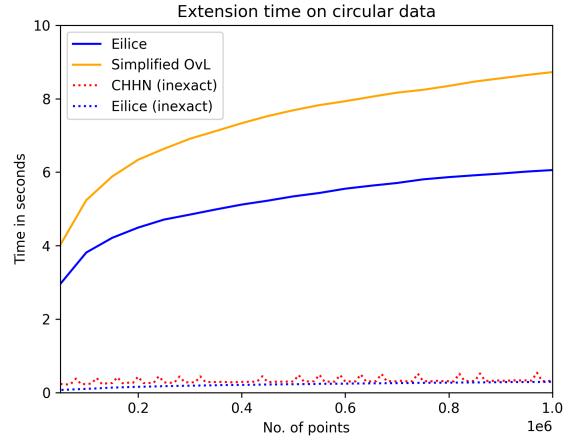
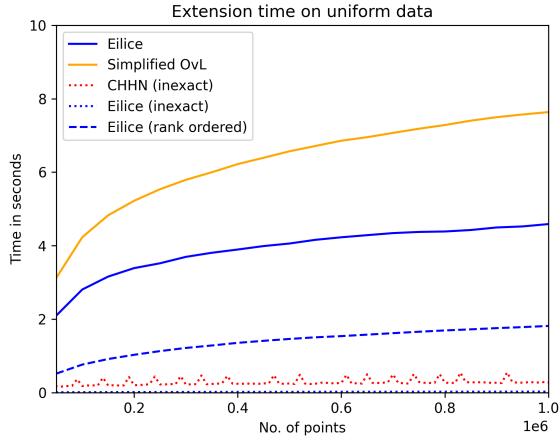
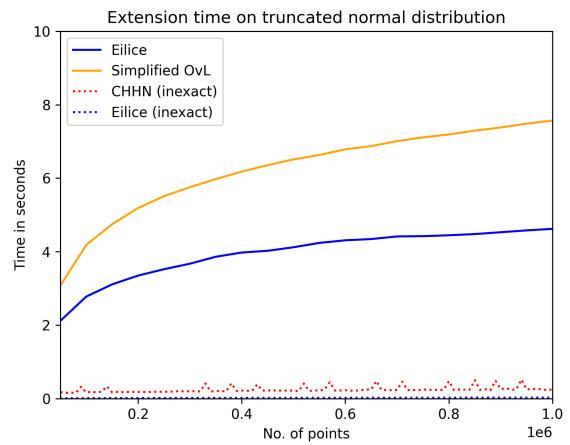
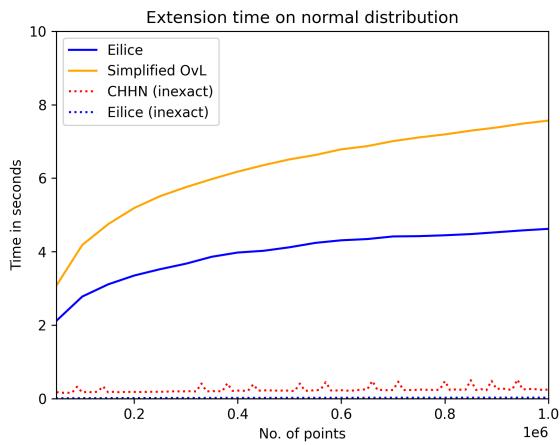


Figure 18: Extension time on various distributions.

Figure 19: Extension time on various distributions.

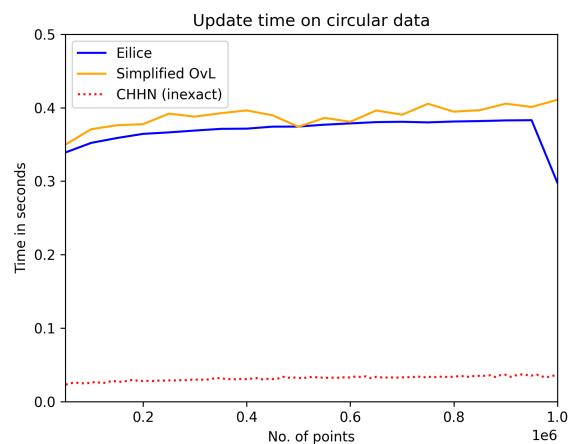
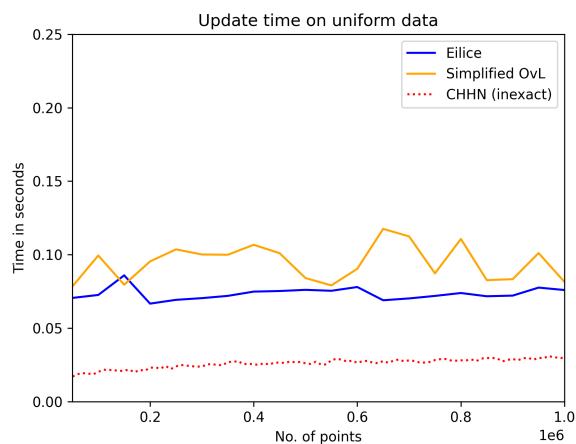
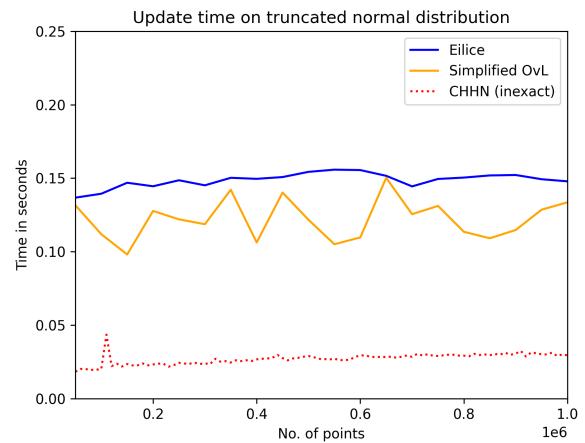
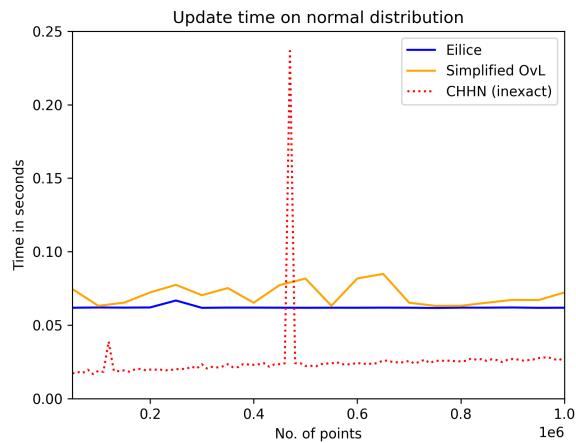


Figure 20: Update time on various distributions.

Figure 21: Update time on various distributions.

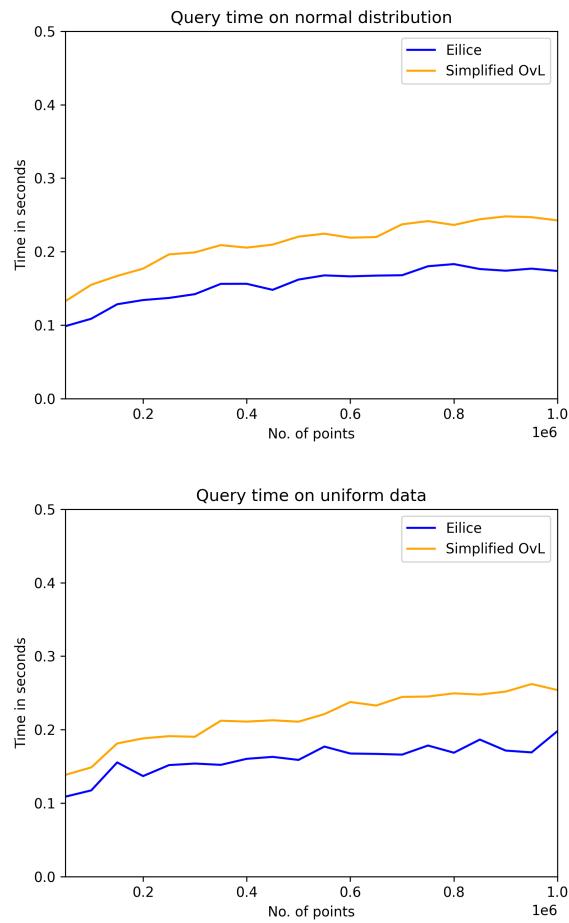


Figure 22: Query time on various distributions.

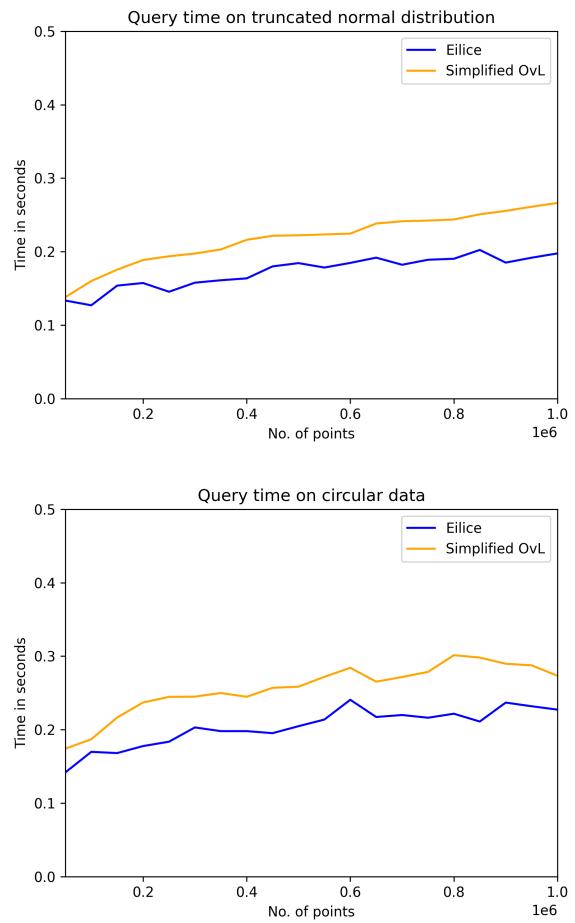


Figure 23: Query time on various distributions.