Lines: 500

Dynamic Convex Hulls for Simple Paths

Bruce Brewer
Kahlert School of Computing, University of Utah, Salt Lake City, UT 84112, USA

Gerth Stølting Brodal
Department of Computer Science, Aarhus University, Aabogade 34, 8200 Aarhus N, Denmark.

Haitao Wang
Kahlert School of Computing, University of Utah, Salt Lake City, UT 84112, USA

Abstract

We consider two restricted cases of the planar dynamic convex hull problem with point insertions and deletions. We assume all updates are performed on a deque (double-ended queue) of points. The first case considers the monotonic path case, where all points are sorted in a given direction, say horizontally left-to-right, and only the leftmost and rightmost points can be inserted and deleted. The second case, which is more general, assumes that the points in the deque constitute a simple path. For both cases, we present solutions supporting deque insertions and deletions in worst-case constant time and standard queries on the convex hull of the points in $O(\log n)$ time, where n is the number of points in the current point set. The convex hull of the current point set can be reported in $O(h + \log n)$ time, where h is the number of edges of the convex hull. For the 1-sided monotone path case, where updates are only allowed on one side, the reporting time can be reduced to $O(h)$, and queries on the convex hull are supported in $O(\log h)$ time. All our time bounds are worst case. In addition, we prove lower bounds that match these time bounds, and thus our results are optimal.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases Dynamic convex hull, convex hull queries, simple paths, path updates, deque

Related Version *Full Version*: https://arxiv.org/abs/2403.05697

Funding *Bruce Brewer*: Supported in part by NSF under Grant CCF-2300356.
Gerth Stølting Brodal: Supported by Independent Research Fund Denmark, grant 9131-00113B.
Haitao Wang: Supported in part by NSF under Grant CCF-2300356.

1 Introduction

Computing the convex hull of a set of n points in the plane is a classic problem in computational geometry. In the static setting, several algorithms can compute the convex hull in $O(n \log n)$ time [2, 14], or in output-sensitive $O(n \log h)$ time [7, 23]; we use h to denote the size of the convex hull throughout the paper. Linear time is also possible for certain special cases, e.g., if points are sorted [2, 14] or points are vertices of a simple path [15, 25].

Overmars and van Leeuwen [27] studied the problem in the dynamic context where points can be inserted and deleted. Their data structure can support the insertion and deletion of points in $O(\log^2 n)$ time, where n is the number of points stored. The convex hull itself can be output in $O(h)$ time and queries on the convex hull can be answered in $O(\log n)$ time. Some example convex hull queries are (see Figure 1): Determine whether a point q is outside the convex hull, and if yes, compute the tangents (i.e., find the tangent points) of the convex hull through q. Given a direction ρ, compute an extreme point on the convex hull along ρ. Given a line ℓ, determine whether ℓ intersects the convex hull, and if yes, find the two edges (bridges) on the convex hull intersected by ℓ. Tangent and extreme point queries are examples of *decomposable* queries, which are queries whose answers can be obtained in constant time from the query answers for any constant number of subsets that form a partition of the point set. In contrast, bridge queries are not decomposable.

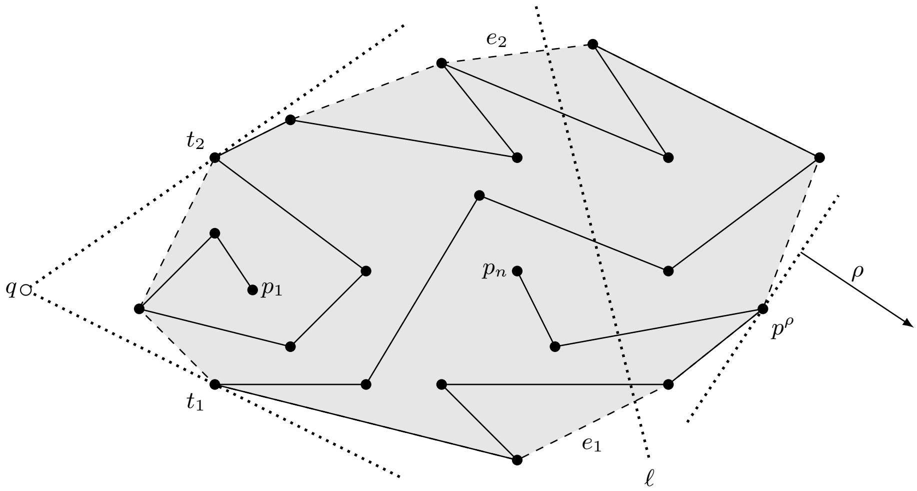

Figure 1 The convex hull (dashed) of a simple path $p_1, \ldots, p_n$ (solid). Three types of convex hull queries are shown (dotted): the tangent points t_1 and t_2 with a query point q outside the convex hull; the extreme point p^ρ in direction ρ; and the two convex hull edges e_1 and e_2 intersecting a line ℓ.

Chan [8] improved the update (insertion/deletion) time to amortized $O(\log^{1+\varepsilon} n)$, for any $\varepsilon > 0$. Tangent and extreme point queries are supported in $O(\log n)$ time, and the convex hull can be reported in $O(h \log n)$ time. The bridge query time was increased to $O(\log^{3/2} n)$. The update time was subsequently improved to amortized $O(\log n \log\log n)$ by Brodal and Jacob [3] and Kaplan, Tarjan, Tsioutsiouliklis [22], and to amortized $O(\log n)$ by Brodal and Jacob [4]. Chan [9] improved the time for bridge queries to $2^{O\left(\sqrt{\log\log n \log\log\log n}\right)} \log n$, with the same amortized update time. It is known that sub-logarithmic update time and logarithmic query time are not possible. For example, to achieve $O(\log n)$ time extreme point queries, an amortized update time $\Omega(\log n)$ is necessary [3].

In this paper, we consider the dynamic convex hull problem for restricted updates, where we can achieve worst-case constant update time and logarithmic query time. In particular, we assume that the points are inserted and deleted in a *deque* (double-ended queue) and that they are geometrically restricted. We consider two restrictions: The first is the monotone path case, where all points in the deque are sorted in a given direction, say horizontally left-to-right, and only the leftmost and rightmost points can be inserted and deleted. The second case allows the points to form a simple path, where updates are restricted to both ends of the path. The simple path problem was previously studied by Friedman, Hershberger, and Snoeyink [13], who supported deque insertions in amortized $O(\log n)$ time, deletions in amortized $O(1)$ time, and queries in $O(\log n)$ time. Bus and Buzer [6] considered a special case of the problem where insertions only happen to the "front" end of the path and deletions are only on points at the "rear" end. They achieved $O(1)$ amortized update time to support $O(h)$ time hull reporting. However, hull queries were not considered in [6]. Wang [33] recently considered a special monotone path case where updates are restricted to queue-like updates, i.e., insert a point to the right of the point set and delete the leftmost point of the point set. Wang called it *window-sliding* updates and achieved amortized constant time updates, hull queries in $O(\log h)$ time,[1] and hull reporting in $O(h)$ time.

[1] The runtime was $O(\log n)$ in the conference paper but was subsequently improved to $O(\log h)$ in the arXiv version https://arxiv.org/abs/2305.08055.

Table 1 Known and new results for dynamic convex hull on paths. O_A are amortized time bounds. – denotes operation is not supported. For an update, h denotes the maximum size of the hull before and after the update. DL = delete left, IR = insert right, etc.

Reference	DL	IL	IR	DR	Queries	Reporting
No geometric restrictions						
Preparata [28] + rollback	–	–	$O(\log h)$	$O(\log h)$	$O(\log h)$	$O(h)$
Monotone path						
Andrews' sweep [2]	–	$O_A(1)$	$O_A(1)$	–	$O(\log h)$	$O(h)$
Wang [33]	$O_A(1)$	–	$O_A(1)$	–	$O(\log h)$	$O(h)$
New (Theorem 5)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(h + \log n)$
New (Theorem 6)	–	–	$O(1)$	$O(1)$	$O(\log h)$	$O(h)$
Simple path						
Friedman et al. [13]	$O_A(1)$	$O_A(\log n)$	$O_A(\log n)$	$O_A(1)$	$O(\log n)$	–
Bus and Buzer [6]	$O_A(1)$	–	$O_A(1)$	–	–	$O(h)$
New (Theorem 7)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(h + \log n)$

1.1 Our results

We present data structures for the monotone path and the simple path variants. For both problems, we support deque insertions and deletions in worst-case constant time. We can answer extreme point, tangent, and bridge queries in $O(\log n)$ time, and we can report the convex hull in $O(h + \log n)$ time. For the *one-sided* monotone case, where updates are only allowed on one side, the reporting time can be reduced to $O(h)$, and convex hull queries are supported in $O(\log h)$ time. That is, they are only dependent on the current hull size and independent of the number of points in the set. In addition, we show that these time bounds are the best possible by proving matching lower bounds. The previous and new bounds for the various versions of the dynamic convex hull problem are summarized in Table 1.

Our results are obtained by a combination of several ideas. To support deque updates, we partition the deque into left and right parts and treat these parts as two independent stack problems. Queries then need to compose the convex hull information from both the stack problems. This strategy has previously been used by Friedman, Hershberger, and Snoeyink [13] and by Wang [33]. To support deletions in the stack structures, we store rollback information when performing insertions. When one of the stacks becomes nearly empty, we repartition the deque into two new stacks of balanced sizes. To achieve worst-case bounds, the repartition is done with incremental global rebuilding ahead of time [26]. To achieve worst-case insertion time, we perform incremental merging of convex hull structures, where we exploit that the convex hulls of two horizontally separated sets can be combined in worst-case $O(\log n)$ time [27] and that the convex hulls of a bipartition of a simple path can be combined in $O(\log^2 n)$ time [16]. To reduce the query bounds for the 1-sided monotone path problem to be dependent on h instead of n, we adopt ideas from Sundar's priority queue with attrition [30]. In particular, we partition the stack of points into four lists (possibly with some interior points removed), of which three lists are in convex position, and three lists have size $O(h)$. We believe this idea is interesting in its own right as, to our knowledge, this is the first time Sundar's approach has been used to solve a geometric problem.

1.2 Other related work

Andrew's algorithm [2] is an incremental algorithm that explicitly maintains the convex hull of the points considered so far. It can add the next point to the right and left of the convex hull in amortized $O(1)$ time. Preparata [28] presented an insertion-only solution maintaining the convex hull in an AVL tree [1] that supports the insertion of an arbitrary point in $O(\log h)$ time, queries on the convex hull in $O(\log h)$ time, and reporting queries in $O(h)$ time. For the *stack* version, where updates form a stack, a general technique to support deletions is by having a stack of rollback information, i.e., the changes performed by the insertions. The time bound for deletions will then match that for insertions, provided that insertion bounds are worst-case. Applying this idea to [28], we have a stack dynamic convex hull solution with $O(\log h)$ time updates. Note that these time bounds hold for arbitrary new points inserted without geometric restrictions. The only limitation is that updates form a stack.

Hershberger and Suri [19] considered the offline version of the dynamic convex hull problem, assuming the sequence of insertions and deletions is known in advance, supporting updates in amortized $O(\log n)$ time. Hershberger and Suri [20] also considered the semi-dynamic deletion-only version of the problem, supporting initial construction and a sequence of n deletions in $O(n \log n)$ time.

Given a simple path of n vertices, Guibas, Hershberger, and Snoeyink [16] considered the problem of building a data structure so that the convex hull of a query subpath (specified by its two ends) can be (implicitly) constructed to support queries on the convex hull. Using a compact interval tree, they gave a data structure of $O(n \log\log n)$ space with $O(\log n)$ query time. The space was recently improved to $O(n)$ by Wang [32]. There are also other problems in the literature regarding convex hulls for simple paths. For example, Hershberger and Snoeyink [18] considered the problem of maintaining convex hulls for a simple path under split operations at certain extreme points, which improves the previous work in [11].

Notation. We define some notation that will be used throughout the paper. For any compact subset R of the plane (e.g., R is a set of points or a simple path), let $\mathcal{H}(R)$ denote the convex hull of R and let $|\mathcal{H}(R)|$ denote the number of vertices of $\mathcal{H}(R)$. We also use ∂R to denote the boundary of R.

For a dynamic set P of points, we define the following operations: InsertRight: Insert a point to P that is to the right of all of the points of P; DeleteRight: Delete the rightmost point of P; InsertLeft: Insert a point to P that is to the left of all the points of P; DeleteLeft: Delete the leftmost point of P; HullReport: Report the convex hull $\mathcal{H}(P)$ (i.e., output the vertices of $\mathcal{H}(P)$ in cyclic order around $\mathcal{H}(P)$). We also use StandardQuery to refer to standard queries on $\mathcal{H}(P)$. This includes all decomposable queries like extreme point and tangent queries. It also includes certain non-decomposable queries like bridge queries. Other queries, such as deciding if a query point is inside $\mathcal{H}(P)$, can be reduced to bridge queries.

We define the operations for the dynamic simple path π similarly. For convenience, we call the two ends of π the *rear end* and the *front end*, respectively. As such, instead of "left" and "right", we use "rear" and "front" in the names of the update operations. Therefore, we have the following four updates: InsertFront, DeleteFront, InsertRear, and DeleteRear, in addition to HullReport and StandardQuery as above.

Outline. We present our algorithms for the monotone path problem in Section 2 and for the simple path problem in Section 3. Due to the space limit, many details and proofs are omitted but can be found in the full paper.

2 The monotone path problem

In this section, we study the monotone path problem where updates occur only at the extremes in a given direction, say, the horizontal direction. That is, given a set of points $P \subset \mathbb{R}^2$, we maintain the convex hull of P, denoted by $\mathcal{H}(P)$, while points to the left and right of P may be inserted to P and the rightmost and leftmost points of P may be deleted from P. Throughout this section, we let n denote the size of the current set P and $h = |\mathcal{H}(P)|$. For ease of exposition, we assume that no three points of P are collinear.

If updates are allowed at both sides (resp., at one side), we denote it the *two-sided* (resp. *one-sided*) problem. We call the structure for the two-sided problem the "deque convex hull," where we use the standard abbreviation deque to denote a double-ended queue (according to Knuth [24, Section 2.2.1], E. J. Schweppe introduced the term deque). The one-sided problem's structure is called the "stack convex hull".

In what follows, we start with describing a "stack tree" in Section 2.1, which will be used to develop a "deque tree" in Section 2.2. We will utilize the deque tree to implement the deque convex hull in Section 2.3 for the two-sided problem. The deque tree, along with ideas from Sundar's priority queues with attrition [30], will also be used for constructing the stack convex hull in Section 2.4 for the one-sided problem.

2.1 Stack tree

Suppose P is a set of n points in $\mathbb{R}^2$ sorted from left to right. Consider the following operations on P (assuming $P = \emptyset$ initially). (1) InsertRight; (2) DeleteRight; (3) TreeRetrieval: Return the root of a balanced binary search tree (BST) that stores all points of the current P in the left-to-right order. We have the following lemma.

▶ **Lemma 1.** *Let P be an initially empty set of n points in $\mathbb{R}^2$ sorted from left to right. There exists a "Stack Tree" $ST(P)$ for P supporting the following operations: (1) InsertRight: $O(1)$ time; (2) DeleteRight: $O(1)$ time; (3) TreeRetrieval: $O(\log n)$ time.*

Remark. Note that the statement of Lemma 1 is not new. Indeed, one can simply use a finger search tree [5, 17, 31] to store P to achieve the lemma (in fact, TreeRetrieval can even be done in $O(1)$ time). We propose a stack tree as a new implementation for the lemma because it can be applied to our dynamic convex hull problem. When we use the stack tree, TreeRetrieval will be used to return the root of a tree representing the convex hull of P; in contrast, simply using a finger search tree cannot achieve the goal (the difficulty is how to efficiently maintain the convex hull to achieve constant time update). Our stack tree may be considered a framework for Lemma 1 that potentially finds other applications as well.

Structure of the stack tree. The stack tree $ST(P)$ consists of a sequence of trees T_i for $i = 0, 1, \ldots, \lceil \log \log n \rceil$. Each T_i is a balanced BST storing a contiguous subsequence of P such that for any $j < i$, all points of T_i are to the left of each point of T_j. The points of all T_i's form a partition of P. We maintain the invariant that $|T_i|$ is a multiple of 2^{2^i} and $0 \le |T_i| \le 2^{2^{i+1}}$, where $|T_i|$ represents the number of points stored in T_i. (The right side of ℓ in Figure 2 is a stack tree).

To achieve worst-case constant time insertions, the process of joining two trees is performed incrementally over subsequent insertions. Specifically, we apply the *recursive slowdown* technique of Kaplan and Tarjan [21], where every 2^{i+1}-th insertion, $i \ge 1$, performs delayed incremental work toward joining T_{i-1} with T_i, if such a join is deemed necessary.

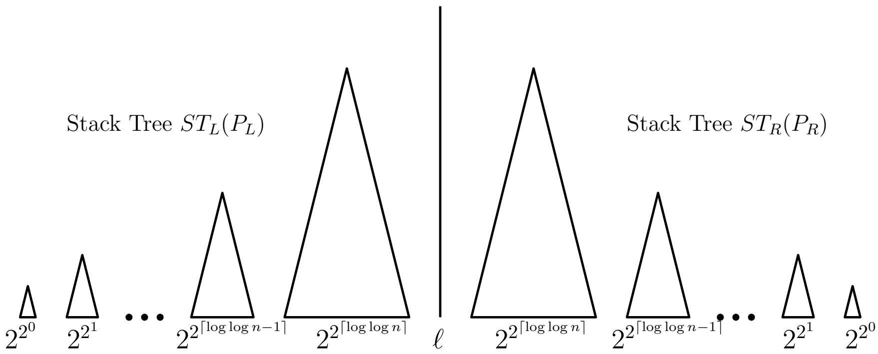

Figure 2 Illustrating a deque tree, comprising two stack trees separated by the vertical line ℓ.

Remark. The critical observation of our algorithm is that because the ranges of the trees do not overlap, we can join adjacent trees T_i and T_{i+1} to obtain (the root) of a new balanced BST that stores all points in $T_i \cup T_{i+1}$ in $O(\log(|T_i| + |T_{i+1}|))$ time. Later in the paper we generalize this idea to horizontally neighboring convex hulls which can be merged in $O(\log(|\mathcal{H}(T_i)| + |\mathcal{H}(T_{i+1})|))$ time [27] and to convex hulls over consecutive subpaths of a simple path which can be merged in $O(\log|\mathcal{H}(T_i)| \cdot \log|\mathcal{H}(T_{i+1})|)$ time [16].

InsertRight. Suppose we wish to insert into P a point p that is right of all points of P.

We start with inserting p into the tree T_0, which takes $O(1)$ time as $|T_0| = O(1)$. Next, we perform $O(1)$ delayed incremental work on a tree T_i for a particular index i. To determine i, we maintain a counter N that is a binary number. Initially, $N = 1$, and it is an invariant that $N = 1 + n$. For each insertion, we increment N by one and determine the index i of the digit which flips from 0 to 1, indexed from the right where the rightmost digit has index 0. Note that there is exactly one such digit. Then, if $i \geq 1$, we perform incremental work on T_i (i.e., joining T_{i-1} with T_i). To find the digit i in $O(1)$ time, we represent N by a sequence of ranges, where each range represents a contiguous subsequence of digits of 1's in N. For example, if N is 101100111, then the ranges are $[0,2], [5,6], [8,8]$. After N is incremented by one, N becomes 101101000, and the ranges become $[3,3], [5,6], [8,8]$. Therefore, based on the first two ranges in the range sequence, one can determine the digit that flips from 0 to 1 and update the range sequence in $O(1)$ time (note that this can be easily implemented using a linked list to store all ranges, without resorting to any bit tricks).

After i is determined, we perform incremental work on T_i as follows. We use a variable n_j to maintain the size of each tree T_j, i.e., $n_j = |T_j|$. For each tree T_j, with $j \geq 1$, we say that T_j is "blocked" if there is an incremental process for joining a previous T_{j-1} with T_j (more details to be given later) and "unblocked" otherwise (T_0 is always unblocked). If T_i is blocked, then there is an incremental process for joining a previous T_{i-1} with T_i. This process will complete within time linear in the height of T_i, which is $O(2^i)$, since $|T_i| \leq 2^{2^{i+1}}$. We perform the next c steps for the process for a sufficiently large constant c. If the joining process is completed within the c steps, we set T_i to be unblocked.

Next, if T_i is unblocked and $n_i \geq 2^{2^{i+1}}$ (in this case by Observation 2 n_i is exactly equal to $2^{2^{i+1}}$), our algorithm maintains the invariant that T_{i+1} must be unblocked by Lemma 3. In this case, we first set T_{i+1} to be blocked, and then we start an incremental process to join T_i with T_{i+1} without performing any actual steps. For reference purpose, let T_i' refer to the current T_i and let T_i start over from $\emptyset$. Using this notation, we are actually joining T_i' with T_{i+1}. Although the joining process has not been completed, we follow the convention that T_i' is now part of T_{i+1}; hence, we update $n_{i+1} = n_{i+1} + n_i$. Also, since T_i is now empty, we reset $n_i = 0$. This finishes the work due to the insertion of p. See the full paper for the proofs of Observation 2 and Lemma 3.

▶ **Observation 2.** 1. *If $n_i \geq 2^{2^{i+1}}$, then $n_i = 2^{2^{i+1}}$.*
2. *It holds that $n_i = 0$ or $2^{2^i} \leq n_i \leq 2^{2^{i+1}}$ for $i \geq 1$, and $n_0 \leq 4$.*

▶ **Lemma 3.** 1. *If $n_0 \geq 4$, then T_1 must be unblocked.*
2. *If $i \geq 1$ and $n_i \geq 2^{2^{i+1}}$ right after the process of joining T_{i-1} with T_i is completed, then T_{i+1} must be unblocked.*

As we only perform $O(1)$ incremental work, the total time for inserting p is $O(1)$.

DeleteRight. To perform DeleteRight, we maintain a stack that records the changes made on each insertion. To delete a point p, p must be the most recently inserted point, and thus all changes made due to the insertion of p are at the top of the stack. To perform the deletion, we simply pop the stack and roll back all the changes during the insertion of p.

TreeRetrieval. To perform TreeRetrieval, we start by completing all incremental joining processes. Then, we join all trees T_i's in their index order. This results in a single BST T storing all points of P. In applications, we usually need to perform binary searches on T, after which we need to continue processing insertions and deletions on P. To this end, when constructing T as above, we maintain a stack that records the changes we have made. Once we are done with queries on T, we use the stack to roll back the changes and return the stack tree to its original form right before the TreeRetrieval operation.

The runtime is $O(\log n)$ because the heights of all trees T_i form a geometric series (i.e., $\sum_{i=1}^{\lceil \log\log n \rceil} 2^i = O(\log n)$). The detailed analysis can be found in the full paper.

2.2 Deque tree

The deque tree is built upon stack trees. We have the following lemma, where TreeRetrieval is defined in the same way as in Section 2.1.

▶ **Lemma 4.** *Let P be an initially empty set of n points in $\mathbb{R}^2$ sorted from left to right. There exists a "Deque Tree" data structure $DT(P)$ for P supporting the following operations: (1) InsertRight: $O(1)$ time; (2) DeleteRight: $O(1)$ time; (3) InsertLeft: $O(1)$ time; (4) DeleteLeft: $O(1)$ time; (5) TreeRetrieval: $O(\log n)$ time.*

The statement of Lemma 4 is not new because we can also use a finger search tree [5,17] to achieve it. Here, we propose a different method for our dynamic convex hull problem.

$DT(P)$ consists of two stack trees $ST_L(P_L)$ and $ST_R(P_R)$ built from opposite directions, where P_L and P_R are the subsets of P to the left and right of a vertical dividing line ℓ, respectively (see Figure 2). To insert a point to the left of P, we insert it to $ST_L(P_L)$. To delete the leftmost point of P, we delete it from $ST_L(P_L)$. For insertion/deletion on the right side of P, we use $ST_R(P_R)$. For TreeRetrieval, we perform TreeRetrieval on both $ST_L(P_L)$ and $ST_R(P_R)$, which result in two balanced BSTs; then, we join these two trees into a single one. The time complexities of all these operations are as stated Lemma 4.

To make this idea work, we need to make sure that neither $ST_L(P_L)$ nor $ST_R(P_R)$ is empty. To this end, we apply incremental global rebuilding [26, Section 5.2.2], where we dynamically adjust the dividing line ℓ. The details are in the full paper. Note that using two stacks to form a deque structure is a natural idea and has been used elsewhere, e.g., [11,18].

2.3 Two-sided monotone path dynamic convex hull

We can tackle the 2-sided monotone path dynamic convex hull problem using the deque tree. Suppose P is a set of n points in $\mathbb{R}^2$. In addition to the operations InsertRight, DeleteRight, InsertLeft, DeleteLeft, HullReport, as defined in Section 1, we also consider the operation HullTreeRetrieval: Return the root of a BST of height $O(\log h)$ that stores all vertices of the convex hull $\mathcal{H}(P)$ (so that binary search based operations on $\mathcal{H}(P)$ can all be supported in $O(\log h)$ time). We will prove the following theorem.

▶ **Theorem 5.** *Let $P \subset \mathbb{R}^2$ be an initially empty set of points, with $n = |P|$ and $h = |\mathcal{H}(P)|$. There exists a "Deque Convex Hull" data structure $DH(P)$ of $O(n)$ space that supports the following operations: (1) InsertRight: $O(1)$ time; (2) DeleteRight: $O(1)$ time; (3) InsertLeft: $O(1)$ time; (4) DeleteLeft: $O(1)$ time; (5) HullTreeRetrieval: $O(\log n)$ time; (6) HullReport: $O(h + \log n)$ time.*

Remark. The time complexities of the four update operations in Theorem 5 are obviously optimal. The lower bound proved in the full paper establishes that the other two operations are also optimal. In particular, it is not possible to reduce the time of HullTreeRetrieval to $O(\log h)$ or reduce the time of HullReport to $O(h)$ (but this is possible for the one-sided case as shown in Section 2.4).

The deque convex hull is a direct application of the deque tree from Section 2.2. We maintain the upper hull and lower hull of $\mathcal{H}(P)$ separately. In the following, we only discuss how to maintain the upper hull, as maintaining the lower hull is similar. By slightly abusing the notation, let $\mathcal{H}(P)$ refer to the upper hull only in the following discussion.

We use a deque tree $DT(P)$ to maintain $\mathcal{H}(P)$. The $DT(P)$ consists of two stack trees ST_L and ST_R. Each stack tree is composed of a sequence of balanced search trees T_i's; each such tree T_i stores left-to-right the points of the convex hull $\mathcal{H}(P')$ for a contiguous subsequence P' of P. We follow the same algorithm as the deque tree with the following changes. During the process of joining T_{i-1} with T_i, our task here becomes merging the two hulls stored in the two trees. To perform the merge, we first compute the upper tangent of the two hulls. This can be done in $O(\log(|T_{i-1}| + |T_i|))$ time [27]. Then, we split the tree T_{i-1} into two portions at the tangent point; we do the same for T_i. Finally, we join the relevant portions of the two trees into a new tree that represents the merged hull of the two hulls. The entire procedure takes $O(\log(|T_{i-1}| + |T_i|))$ time. This time complexity is asymptotically the same as joining two trees T_{i-1} and T_i as described in Section 2.1, and thus we can still achieve the same performances for the first five operations as in Lemma 4; in particular, to perform HullTreeRetrieval, we simply call TreeRetrieval on the deque tree. Finally, for HullReport, we first perform HullTreeRetrieval to obtain a tree representing $\mathcal{H}(P)$. Then, we perform an in-order traversal on the tree, which can output $\mathcal{H}(P)$ in $O(h)$ time. Thus, the total time for HullReport is $O(h + \log n)$.

2.4 One-sided monotone path dynamic convex hull

Let P be a set of n points in $\mathbb{R}^2$. Consider the following operations on P (with $P = \emptyset$ initially): InsertRight, DeleteRight, HullTreeRetrieval, HullReport, as in Section 2.3. Applying Theorem 5, we can perform HullTreeRetrieval in $O(\log n)$ time and perform HullReport in $O(h + \log n)$ time. We have the following theorem, which reduces the HullTreeRetrieval time to $O(\log h)$ and reduces the HullReport time to $O(h)$.

▶ **Theorem 6.** *Let $P \subset \mathbb{R}^2$ be an initially empty set of points, with $n = |P|$ and $h = |\mathcal{H}(P)|$. There exists a "Stack Convex Hull" data structure $SH(P)$ of $O(n)$ space that supports the following operations: (1) InsertRight: $O(1)$ time; (2) DeleteRight: $O(1)$ time; (3) HullTreeRetrieval: $O(\log h)$ time; (4) HullReport: $O(h)$ time.*

The main idea to prove Theorem 6 is to adapt ideas from Sundar's algorithm in [30] for priority queue with attrition as well as the deque convex hull data structure from Section 2.3. As in Section 2.4, we maintain the upper and lower hulls of $\mathcal{H}(P)$ separately. By slightly abusing the notation, let $\mathcal{H}(P)$ refer to the upper hull only in the following discussion.

For any two disjoint subsets P_1 and P_2 of P, we use $P_1 \prec P_2$ to denote the case where all points of P_1 are to the left of each point of P_2. Our data structure maintains four subsets $A_1 \prec A_2 \prec A_3 \prec A_4$ of P. Each A_i, $1 \leq i \leq 3$, is a convex chain, but this may not be true for A_4. Further, the following invariants are maintained during the algorithm (which are strongly inspired by Sundar's method [30]): (1) Vertices of $\mathcal{H}(P)$ are all in $\bigcup_{i=1}^{4} A_i$; (2) A_1 is a prefix of the vertices of $\mathcal{H}(P)$ sorted from left to right; (3) $A_1 \cup A_2$ and $A_1 \cup A_3$ are both convex chains; (4) $|A_1| \geq |A_3| + 2 \cdot |A_4|$. Note that the second and fourth invariants imply that $|A_1|$, $|A_3|$, and $|A_4|$ are all bounded by $O(h)$, which helps to achieve $O(\log h)$ time for HullTreeRetrieval and $O(h)$ time for HullReport.

We omit the details, which can be found in the full paper.

3 The simple path problem

In this section, we consider the dynamic convex hull problem for a simple path. Let π be a simple path of n vertices in the plane (note that π consists of $n-1$ line segments and each segment endpoint is defined to be a *vertex* of π). Unless otherwise stated, a "point" of π always refers to a vertex of it (this is for convenience also for being consistent with the notion in Section 2). For ease of discussion, we assume that no three vertices of π are colinear.

For any subpath π' of π, let $|\pi'|$ denote the number of vertices of π, and $\mathcal{H}(\pi')$ the convex hull of π', which is also the convex hull of all vertices of π'.

We designate the two ends of π as the *front end* and the *rear end*, respectively. We consider the following operations on π: InsertFront, DeleteFront, InsertRear, DeleteRear, StandardQuery, and HullReport, as defined in Section 1. The following theorem summarizes the main result of this section.

▶ **Theorem 7.** *Let $\pi \subset \mathbb{R}^2$ be an initially empty simple path, with $n = |\pi|$ and $h = |\mathcal{H}(\pi)|$. There exists a "Deque Path Convex Hull" data structure $PH(\pi)$ of $O(n)$ space that supports the following operations: (1) InsertFront: $O(1)$ time; (2) DeleteFront: $O(1)$ time; (3) InsertRear: $O(1)$ time; (4) DeleteRear: $O(1)$ time; (5) StandardQuery: $O(\log n)$ time; (6) HullReport: $O(h + \log n)$ time.*

Remark. The lower bound in the full paper implies that all these bounds are optimal even for the "one-sided" case. In particular, it is not possible to reduce the time of HullTree-Retrieval to $O(\log h)$ or reduce the time of HullReport to $O(h)$. This is why we do not consider the one-sided simple path problem separately. For answering standard queries, our algorithm first constructs four BSTs representing convex hulls of four (consecutive) subpaths of π whose union is π and then uses these trees to answer queries. The height of the two trees for the two middle subpaths are $O(\log n)$ while the heights of the other two are $O(\log \log n)$. As such, all decomposable queries can be answered in $O(\log n)$ time. We show that certain non-decomposable queries can also be answered in $O(\log n)$ time, such as the bridge queries.

In what follows, we prove Theorem 7. One crucial property we rely on is that the convex hulls of two subpaths of a simple path intersect at most twice and thus have at most two common tangents as observed by Chazelle and Guibas [10]. Let π_1 and π_2 be two consecutive subpaths of π. Suppose we have two BSTs representing $\mathcal{H}(\pi_1)$ and $\mathcal{H}(\pi_2)$, respectively. Compared to the monotone path problem, one difficulty here (we refer to it as the "path-challenge") is that we do not have an $O(\log n)$ time algorithm to find the common tangents between $\mathcal{H}(\pi_1)$ and $\mathcal{H}(\pi_2)$ and thus merge the two hulls. The best algorithm we have takes $O(\log^2 n)$ time by a nested binary search, assuming that we have two "helper points": a point on each convex hull that is outside the other convex hull [16].

It is tempting to apply the deque hull idea of Theorem 5 (i.e., consider the points in the "path order" along π). We could get the same result as in Theorem 5 except that the HullTreeRetrieval operation now takes $O(\log^2 n)$ time and HullReport takes $O(h + \log^2 n)$ time due to the path-challenge. As such, our main effort below is to achieve $O(\log n)$ time for StandardQuery and $O(h + \log n)$ time for HullReport.

Before presenting our data structure, we introduce in Section 3.1 several basic lemmas which we will use on several occasions later on.

3.1 Basic lemmas

The following two lemmas, both from [16], will be used later.

▶ **Lemma 8.** (Guibas, Hershberger, and Snoeyink [16, Lemma 5.1]) *Let π_1 and π_2 be two consecutive subpaths of π. Suppose the convex hull $\mathcal{H}(\pi_i)$ is stored in a BST of height $O(\log |\pi_i|)$, for $i = 1, 2$. We can do the following in $O(\log(|\pi_1| + |\pi_2|))$ time: Determine whether $\mathcal{H}(\pi_2)$ is completely inside $\mathcal{H}(\pi_1)$ and if not find a "helper point" $p \in \partial\mathcal{H}(\pi_2)$ such that $p \in \partial\mathcal{H}(\pi_1 \cup \pi_2)$ and $p \notin \partial\mathcal{H}(\pi_1)$.*

▶ **Lemma 9.** (Guibas, Hershberger, and Snoeyink [16, Section 2]) *Let π_1 and π_2 be two consecutive subpaths of π. Suppose the convex hull $\mathcal{H}(\pi_i)$ is stored in a BST of height $O(\log |\pi_i|)$, $i = 1, 2$. We can compute a BST of height $O(\log(|\pi_1| + |\pi_2|))$ that stores the convex hull of $\pi_1 \cup \pi_2$ in $O(\log |\pi_1| \cdot \log |\pi_2|)$ time.*

The following lemma provides a tool for answering bridge queries, obtained with the help of the binary search algorithm of Overmars and van Leeuwen [27] for computing the common tangents of two convex polygons separated by a line. See the full paper for the detailed proof.

▶ **Lemma 10.** *Let $H_1, H_2, \ldots,$ be a collection of $O(1)$ convex polygons, each represented by a BST or an array so that binary search on each convex hull can be supported in $O(\log n)$ time. Let H be the convex hull of all these convex polygons. We can answer the following queries in $O(\log n)$ time each, where n is the total number of vertices of all these convex polygons.*

1. *Bridge queries: Given a query line ℓ, determine whether ℓ intersects H, and if yes, find the edges of H that intersect ℓ.*
2. *Given a query point p, determine whether $p \in H$, and if yes, determine whether $p \in \partial H$.*

3.2 Structure of the deque path convex hull $PH(\pi)$

We partition π into four (consecutive) subpaths π_r, π_m^r, π_m^f, and π_f from the front to the rear of π. As such, π_f and π_r contain the front and rear ends, respectively. Further, let $\pi^+ = \pi_f \cup \pi_m^f$ and $\pi^- = \pi_r \cup \pi_m^r$. Our algorithm maintains the following two invariants.

Invariants: (1) $\frac{1}{4} \leq |\pi^+|/|\pi^-| \leq 4$. (2) $|\pi_f| = O(\log^2 |\pi^+|)$ and $|\pi_r| = O(\log^2 |\pi^-|)$.

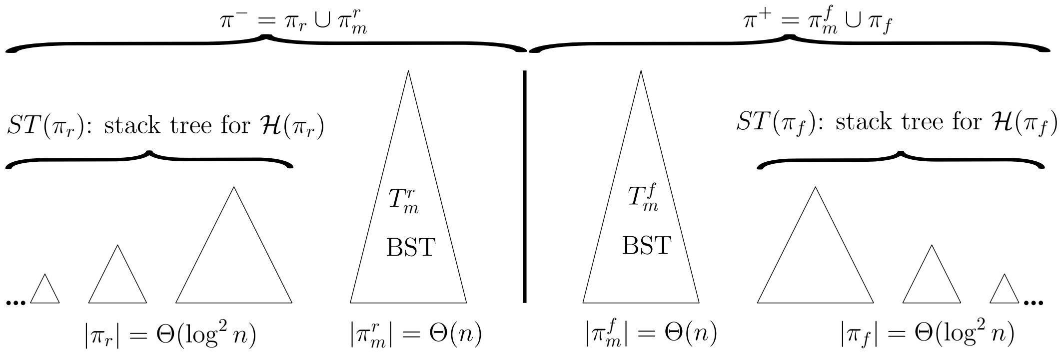

Figure 3 A schematic view of the deque path convex hull data structure $PH(\pi)$.

Note that the invariants imply $|\pi_m^f|, |\pi_m^r| = \Theta(n)$, where $n = |\pi|$. The first invariant resembles the partition of P by a dividing line ℓ in our deque tree in Section 2.2. As with the deque tree, in order to maintain the first invariant, we use the global rebuilding idea [26]. The details can be found in the full paper.

We use a stack tree $ST(\pi_f)$ to maintain the convex hull $\mathcal{H}(\pi_f)$, with the algorithm in Lemma 9 for merging two hulls of two consecutive subpaths. More specifically, we consider the vertices of π_f following their order along the path (instead of left-to-right order as in Section 2.1) with insertions and deletions only at the front end. Whenever we need to join two neighboring trees, we merge the two hulls of their subpaths by Lemma 9. Due to the second invariant, merging all trees of $ST(\pi_f)$ takes $O(\log^2 \log n)$ time, after which we obtain a single tree of height $O(\log \log n)$ that represents $\mathcal{H}(\pi_f)$. Similarly, we build a stack tree $ST(\pi_r)$ for $\mathcal{H}(\pi_r)$ but along the opposite direction of the path. See Figure 3 for an illustration.

Define $n^+ = |\pi^+|$, which is $\Theta(n)$. In order to maintain the second invariant, when π_f is too big due to insertions, we will cut a subpath of length $\Theta(\log^2 n^+)$ and concatenate it with π_m^f. When π_f becomes too small due to deletions, we will split a portion of π_m^f of length $\Theta(\log^2 n^+)$ and merge it with π_f; but this split is done implicitly using the rollback stack for deletions. As such, we need to build a data structure for maintaining π_m^f so that the above concatenate operation on π_m can be performed in $O(\log^2 n^+)$ time (this is one reason why the bound for π_f in the second invariant is set to $O(\log^2 n^+)$). We process π^- in a symmetric way. The way we handle the interaction between π_m^f and π_f (as well as their counterpart for π^-) are one main difference from our approach for the two-sided monotone path problem in Section 2.3; again this is due to the path-challenge.

Our data structure for π_m^f is simply a balanced BST T_m^f, which stores the convex hull $\mathcal{H}(\pi_m^f)$. In particular, we will use T_m^f to support the above concatenation operation (denoted by CONCATENATE) in $O(\log^2 n)$ time. For reference purpose, this is summarized in the following lemma, which is an immediate application of Lemma 9.

▶ **Lemma 11.** *Given a BST of height $O(\log |\tau|)$ representing a simple path τ of length $O(\log^2 n)$ such that the concatenation of π_f^m and τ is still a simple path, we can perform the following* CONCATENATE *operation in $O(\log^2 n)$ time: Obtain a new tree T_m^f of height $O(\log n)$ that represents the convex hull $\mathcal{H}(\pi_m^f)$, where π_m^f is the new path after concatenating with τ.*

Similarly, we use a balanced BST T_m^r to store the convex hull $\mathcal{H}(\pi_m^r)$. We have a similar lemma to the above for the CONCATENATE operation on π_m^r.

The four trees $ST(\pi_r)$, T_m^r, T_m^f, and $ST(\pi_f)$ constitute our deque path convex hull data structure $PH(\pi)$ for Theorem 7; see Figure 3. In the following, we discuss the operations.

3.3 Standard queries

For answering a decomposible query σ, we first perform a TreeRetrieval operation on $ST(\pi_f)$ to obtain a tree T_f that represents $\mathcal{H}(\pi_f)$. Since $|\pi_f| = O(\log^2 n)$, this takes $O(\log^2 \log n)$ time as discussed before. We do the same for $ST(\pi_r)$ to obtain a tree T_r for $\mathcal{H}(\pi_r)$. Recall that the tree T_m^f stores $\mathcal{H}(\pi_m^f)$ while T_m^r stores $\mathcal{H}(\pi_m^r)$. We perform query σ on each of the above four trees. Based on the answers to these trees, we can obtain the answer to the query σ for $\mathcal{H}(\pi)$ because σ is a decomposable query. Since the heights of T_f and T_r are both $O(\log\log n)$, and the heights of T_m^f and T_m^r are $O(\log n)$, the total query time is $O(\log n)$.

If σ is a bridge query, we apply Lemma 10 on the above four trees. The query time is $O(\log n)$.

3.4 Insertions and deletions

InsertFront and DeleteFront are handled by the data structure for π^+, i.e., T_m^f and $ST(\pi_f)$, while InsertRear and DeleteRear are handled by the data structure for π^-.

InsertFront. Suppose we insert a point p to the front end of π. We first perform the insertion using $ST(\pi_f)$. To maintain the second invariant, we must handle the interaction between the largest tree T_k of $ST(\pi_f)$ and the tree T_m^f. Recall that $n^+ = |\pi^+|$ and $n^+ = \Theta(n)$.

According to the second invariant and the definition of the stack tree $ST(\pi_f)$, we have $|T_k| = O(\log^2 n^+)$, and we can assume a constant c such that the total size of all trees of $ST(\pi_f)$ smaller than T_k is at most $c \cdot \log^2 n^+$. We set the size of T_k to be $(c+1)\cdot\log^2 n^+$. During the algorithm, whenever $|T_k| > (c+1)\cdot\log^2 n^+$ and there is no incremental process of joining T_{k-1} with T_k, we let $T_k' = T_k$ and let $T_k = \emptyset$, and then start to perform an incremental Concatenate operation to concatenate T_k' with T_m^f. The operation takes $O(\log^2 n^+)$ time by Lemma 11. We choose a sufficiently large constant c_1 so that each Concatenate operation can be finished within $c_1 \cdot \log^2 n^+$ steps. For each InsertFront in future, we run c_1 steps of this Concatenate algorithm. As such, within the next $\log^2 n^+$ InsertFront operations in future, the Concatenate operation will be completed. If there is an incremental Concatenate operation (that is not completed), then we say that T_m^f is *dirty*; otherwise, it is *clean*.

If T_m^f is dirty, an issue arises during a StandardQuery operation. Recall that during a StandardQuery operation, we need to perform queries on $\mathcal{H}(\pi_m^f)$ by using the tree T_m^f. However, if T_m^f is dirty, we do not have complete information for T_m^f. To address this issue, we resort to persistent data structures [12,29]. Specifically, we use a persistent tree for T_m^f so that if there is an incremental Concatenate operation, the old version of T_m^f can still be accessed (we call it the "clean version"); as such, a partially persistent tree suffices for our purpose [12,29]. After the Concatenate is completed, we designate the new version of T_m^f as clean and the old version as dirty; in this way, at any time, there is only one clean version we can refer to. During a StandardQuery operation, we can perform queries on the clean version of T_m^f. Similarly, during the query, if there is an incremental Concatenate process, T_k' is also dirty, and we need to access its clean version (i.e., the version right before T_k' started the Concatenate operation). To solve this problem, before we start Concatenate, we make another copy of T_k', denoted by T_k''. After Concatenate is completed, we make T_k'' refer to null. The above strategy causes additional $O(\log^2 n^+)$ time, i.e., update the persistent tree T_m^f and make a copy T_k''. To accommodate this additional cost, we make the constant c_1 large enough so that all these procedures can be completed within the next $\log^2 n^+$ InsertFront operations.

Recall that once we are about to start a Concatenate operation for T'_k, T_k becomes empty. We can show that Concatenate will be completed before another Concatenate operation starts. See the full paper for the detailed argument. As such, there cannot be two concurrent Concatenate operations from T_k to T^f_m.

DeleteFront. As before, we keep a stack of changes to our data structure $PH(\pi)$ due to the InsertFront operations. For each DeleteFront, we simply roll back the changes.

InsertRear and DeleteRear. Handling updates at the rear end is the same, but using T^r_m and $ST(\pi_r)$ instead. We omit the details.

3.5 Reporting the convex hull $\mathcal{H}(\pi)$

We show that the convex hull $\mathcal{H}(\pi)$ can be reported in $O(h + \log n)$ time.

As in the algorithm for StandardQuery, we first obtain in $O(\log n)$ time the four trees T_f, T_r, T^f_m, and T^r_m representing $\mathcal{H}(\pi_f)$, $\mathcal{H}(\pi_r)$, $\mathcal{H}(\pi^f_m)$, and $\mathcal{H}(\pi^r_m)$, respectively. Then, we can merge these four convex hulls using Lemma 9 in $O(\log^2 n)$ time and compute a BST $T(\pi)$ representing $\mathcal{H}(\pi)$. Finally, we can output $\mathcal{H}(\pi)$ by traversing $T(\pi)$ in additional $O(h)$ time. As such, in total $O(h + \log^2 n)$ time, $\mathcal{H}(\pi)$ can be reported. To reduce the time to $O(h + \log n)$, we first enhance our data structure $PH(\pi)$ by having it maintain the common tangents of $\mathcal{H}(\pi^f_m)$ and $\mathcal{H}(\pi^r_m)$ during updates. The details are in the full paper.

Remark. As discussed in the full paper, it is possible to achieve $O(h + \log n)$ time for HullReport without enhancing the data structure. Nevertheless, we choose to present the enhanced data structure for two reasons: (1) Enhancing the data structure will make the HullReport algorithm much simpler; (2) the enhanced data structure helps us to obtain in $O(\log n \log\log n)$ time a tree of height $O(\log n)$ to represent $\mathcal{H}(\pi)$, improving the aforementioned $O(\log^2 n)$ time algorithm.

References

1. Georgii Maksimovich Adel'son-Velskii and Evgenii Mikhailovich Landis. An algorithm for organization of information. *Doklady Akademii Nauk*, 146(2):263–266, 1962.
2. A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979. doi:10.1016/0020-0190(79)90072-3.
3. Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull with optimal query time and $O(\log n \cdot \log\log n)$ update time. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 57–70, 2000. doi:10.1007/3-540-44985-X_7.
4. Gerth Stølting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 617–626, 2002. doi:10.1109/SFCS.2002.1181985.
5. Gerth Stølting Brodal. Finger search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004. URL: https://www.cs.au.dk/~gerth/papers/finger05.pdf.
6. Norbert Bus and Lilian Buzer. Dynamic convex hull for simple polygonal chains in constant amortized time per update. In *Proceedings of the 31st European Workshop on Computational Geometry (EuroCG)*, 2015. URL: https://perso.esiee.fr/~busn/publications/2015_eurocg_dynamicConvexHull/eurocg2015_dynamicHull.pdf.

7 Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996. doi:10.1007/BF02712873.

8 Timothy M. Chan. Dynamic planar convex hull operations in near-logarithmaic amortized time. *Journal of the ACM*, 48:1–12, 2001. doi:10.1145/363647.363652.

9 Timothy M. Chan. Three problems about dynamic convex hulls. *Int. J. Comput. Geom. Appl.*, 22(4):341–364, 2012. doi:10.1142/S0218195912600096.

10 Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986. doi:10.1007/BF01840441.

11 David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. *Algorithmica*, 10:1–23, 1993. doi:10.1007/BF01908629.

12 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

13 Joseph Friedman, John Hershberger, and Jack Snoeyink. Efficiently planning compliant motion in the plane. *SIAM Journal on Computing*, 25:562–599, 1996. doi:10.1145/73833.73854.

14 Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972. doi:10.1016/0020-0190(72)90045-2.

15 Ronald L. Graham and F. Frances Yao. Finding the convex hull of a simple polygon. *Journal of Algorithms*, 4:324–331, 1983. doi:10.1016/0196-6774(83)90013-5.

16 Leonidas Guibas, John Hershberger, and Jack Snoeyink. Compact interval trees: A data structure for convex hulls. *International Journal of Computational Geometry and Applications*, 1:1–22, 1991. doi:10.1142/S0218195991000025.

17 Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC)*, pages 49–60, 1977. doi:10.1145/800105.803395.

18 John Hershberger and Jack Snoeyink. Cartographic line simplification and polygon CSG formula in $O(n \log^* n)$ time. *Computational Geometry: Theory and Applications*, 11:175–185, 1998. doi:10.1016/S0925-7721(98)00027-3.

19 John Hershberger and Subhash Suri. Offline maintenance of planar configurations. In *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 32–41, 1991. doi:10.5555/127787.127801.

20 John Hershberger and Subhash Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, 32:249–267, 1992. doi:10.1007/BF01994880.

21 Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing (STOC)*, pages 93–102, 1995. doi:10.1145/225058.225090.

22 Haim Kaplan, Robert E. Tarjan, and Kostas Tsioutsiouliklis. Faster kinetic heaps and their use in broadcast scheduling. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 836–844, 2001. doi:10.5555/365411.365793.

23 David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15:287–299, 1986. doi:10.1137/0215021.

24 Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

25 Avraham A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25(1):11–12, 1987. doi:10.1016/0020-0190(87)90086-X.

26 Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. doi:10.1007/BFB0014927.

27 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981. doi:10.1016/0022-0000(81)90012-X.

28 Franco P. Preparata. An optimal real-time algorithm for planar convex hulls. *Communications of the ACM*, 22:402–405, 1979. doi:10.1145/359131.359132.

29 Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

30 Rajamani Sundar. Worst-case data structures for the priority queue with attrition. *Information Processing Letters*, 31:69–75, 1989. doi:10.1016/0020-0190(89)90071-9.

31 Athanasios K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67:173–194, 1985. doi:10.1016/S0019-9958(85)80034-6.

32 Haitao Wang. Algorithms for subpath convex hull queries and ray-shooting among segments. In *Proceedings of the 36th International Symposium on Computational Geometry (SoCG)*, pages 69:1–69:14, 2020. doi:10.4230/LIPIcs.SoCG.2020.69.

33 Haitao Wang. Dynamic convex hulls under window-sliding updates. In *Proceedings of the 18th Algorithms and Data Structures Symposium (WADS)*, pages 689–703, 2023. doi:10.1007/978-3-031-38906-1_46.