Robust implementations of real time algorithms for maintaining the convex hull of a dynamic set in the plane

SUMEET SHIRGURE , University of Southern California, USA

This paper documents the implementation details and the interface of a C++ software package that allows for maintaining the convex hull of a dynamic set of points in the 2D plane. By maintaining the convex hull we mean the following things (a) point insertion and point deletion requests are handled by updating the data structures in memory, and (b) said structures allow for effeciently querying standard questions like determining whether another point lies inside the resulting convex polygon, finding tangents from an external point, and finding the farthest points along a given direction. By real time we mean that the individual requests are handled quickly and the time complexity analysis is *not* amortized over multiple queries. The presented software package is *robust* and *exact*, in the sense that if geometric errors arise because of floating point imprecision, they will not be due to code written within our scope. In particular if all numerals are exact to arbitrary precision, the code provided will always find the exact convex hull of a given set of points, and answer queries exactly to given precision. This feature is possible due to the use of C++ template meta programming.

CCS Concepts: • **Mathematics and computing**; • **Computational geometry**;

Additional Key Words and Phrases: Dynamic convex hulls, C++ implementation, Computational geometry

ACM Reference Format:

Sumeet Shirgure . 2023. Robust implementations of real time algorithms for maintaining the convex hull of a dynamic set in the plane. 1, 1 (September 2023), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The set of dynamic convex hull problems is a class of dynamic problems in computational geometry. It consists of the maintenance of the convex hull for input data undergoing a sequence of discrete changes, i.e. when input data elements may be inserted or deleted. It shouldn't be confused with the kinetic convex hull, which studies similar problems for continuously moving points.

We will consider two settings for the dynamic hull. One where points are merely inserted into a dynamic set and not deleted. We call this the "online" setting. The other is where points are both requested to be inserted and removed from a dynamic set. We call this the (fully) "dynamic" setting.

Note that in the online setting once a point falls inside the interior of the hull it is no longer under consideration. This allows us to discard parts of the old hull that are in the interior of the new hull. The same is not true for the dynamic case, where points in the interior must be preserved for potential future computation.

There has been plenty of prior theoretical work : e.g the online version was solved optimally in [9] taking $O(log(h))$ time for all requests; while the dynamic version was solved in [8] with a running time of $O(log^2(n))$ per point insertion/deletion and $O(log(h))$ query time for point in polygon/tangent/farthest point queries. n and h here being the size of the set and the size of the hull currently respectively. It's these algorithms that have been implemented along

Author's address: Sumeet Shirgure , sshirgur@usc.edu, University of Southern California, Los Angeles, California, USA.

with this paper. The papers in [1], [2], [7] give improvements over the classical literature, but are unwieldy in practice due to high constant factors.

The papers in [3] and [5] give various applications for the dynamic convex hulls and semi-dynamic (deletions only) hulls and respectively. The paper in [4] explores a problem in query sheduling in the context of database management systems that uses an online hull structure. We also note that a possible application of a dynamic hull structure is the "monotone windowed convex-hull" problem where we are to maintain the convex hull of a fixed-size window from a stream of points, possibly ordered in one of the coordinates. Such problems might be of use in time series analysis and forecasting. The hope is that the existence of a software library like the one presented will alleviate the need to rewrite new code every time a new application is found.

The rest of the paper is structured as follows : Section 2 describes the C++ software interface for the online and dynamic hull libraries. Section 3 discusses theory behind the implemented data structures and algorithms. Section 4 concludes the report by summarizing the contributions of this paper.

2 SOFTWARE INTERFACE

The presented software provides two headers, one for the online hull, and one for the dynamic hull. Both of them share most of the C++ interface as shown below :

```
template <typename Field> class DynamicHull {
public:
  void add_point(Point<Field> const&);
  bool remove_point(Point<Field> const&); // not present in Online hull
  std::optional< std::pair< Point<Field>, Point<Field> > > get_tangents (Point<Field> const&);
  std::pair< Point<Field>, Point<Field> > get_extremal_points(Point<Field> const&);
private:
  // rest of the data structures and algorithms
};
```

The add and remove points return wether the operation was successful. Here the `std::optional` signifies that the return value may or may not be present. In this implementation it's absent exactly when the point input to the function is strictly inside the hull. The `get_extremal_points` function returns the point or points on the line segment that maximizes the dot product along the provided direction.

The template argument `typename Field` is key to the robustness mentioned earlier. The implementation doesn't assume an upper limit on the precision of any arithmetic by declaring `int` or `double` in its internals. All that is assumed about the `Field` type is that it is a total order like the integers or the reals, that has "equality" and the "less than" operator defined, and is capable of forming a 2D vector space.

3 THEORY AND ALGORITHMS

This section will describe some theory and notations used in the rest of the paper to describe convex hulls *exactly*. The notations and definitions here directly translate to code in our implementation.

An ordered set is a set with some intrinsic order. It is denoted here as $(e_0, e_1, ...e_{n-1})$ This set has n elements and there is some total order among its elements. A proper prefix of it is some set $(e_0, e_1, ..., e_i)$ for $0 \leq i < n$. A suffix is the complement of a proper prefix. Prefixes and proper suffixes are defined analogously.

Lowercase alphabets like p denote a point (p_x, p_y) in the 2D vector space formed by the given field F^2. Let $\prec$ denote the lexicographical ordering $p \prec q \iff p_x < q_x \vee p_x = q_x \wedge p_y < q_y$. Exactly one of $p \prec q, p = q, q \prec p$ is always true. $p \preceq q \iff \neg(q \prec p)$. We also denote other common vector operations as follows : addition $p \pm q \equiv (p_x \pm q_x, p_y \pm q_y)$, scalar product $p \cdot q \equiv p_x q_x + p_y q_y$, vector product $p \times q = p_x q_y - q_x p_y$ (just magnitude). The commonly used orientation test is performed using cross products : $ccw(u, v, p)$ tests whether point p lies strictly to the left of the line passing through u and v, when the observer is facing v standing upright at u. $ccw(u, v, p) \equiv (v - u) \times (p - u) \geq 0$ with equality if and only if all three points are collinear. Let $CH(S)$ denote the convex hull region of a point set S.

We define the lower and upper hulls as follows :

$$LH(S) \equiv \{p \in S : (p_x, y) \notin CH(S) \forall y < p_y)\} \cup max_{\prec}\{S\}$$
$$UH(S) \equiv \{p \in S : (p_x, y) \notin CH(S) \forall y > p_y)\} \cup min_{\prec}\{S\}$$

Note that $CH(S)$ spans the complete area of the hull whearas $LH(S)$ and $UH(S)$ are subsets of vertices in $CH(S) \cap S$. $LH(S)$ and $UH(S)$ intersect at exactly two points : $min_{\prec}\{S\}$ and $max_{\prec}\{S\}$. They are defined so because they partition the line segments on the boundary of $CH(S)$ into two symmetric chains. If we denote by $-T$ the point set $\{-p : p \in T\}$, where each point undergoes a centrosymmetric transform $(p_x, p_y) \rightarrow (-p_x, -p_y)$, then notice that the following relation holds : $-UH(S) = LH(-S)$. This allows us to treat lower and upper hulls separately, and combine their results wherever necessary. The rest of the paper focuses on maintaining just the lower hulls under point addition and deletions.

3.1 Online hull

We now consider the task of maintaining $LH(S)$ of an incremental set S subject to point additions $S \leftarrow S \cup \{q\}$. The ideas here are mostly taken from [9], with a few improvements. $LH(S)$ is stored in memory as an ordered set of $n - 1$ point pairs $((p_0, p_1), (p_1, p_2)...(p_{n-2}, p_{n-1}))$, where each pair (p_i, p_{i+1}) corresponds to a line segment $\overline{p_i p_{i+1}}$ on $LH(S)$. The points at all times satisfy the inequalities $p_0 \prec p_1 \prec p_2 ... \prec p_{n-1}$. Furthermore $p_0 = min_{\prec}\{S\}$ and $p_{n-1} = max_{\prec}\{S\}$. We also invariably maintain that no three consecutive points are collinear in the sequence.

For adding a new point q, we only consider three mutually disjoint cases :

1. $q \prec p_0$: In this case, some prefix $((p_0, p_1), ..., (p_{i-1} p_i))$ is removed, and is replaced by a single element $((q, p_i))$ corresponding to the tangent $\overline{q p_i}$. This removed prefix corresponds to the points that will be in the interior of the newly formed $LH(S \cup \{q\})$. Note that p_i will be the first line segment with $ccw(p_i, p_{i+1}, q)$ in the sequence. And if no segments in the sequence pass the orientation test, then the entire lower hull is removed.
2. $p_{n-1} \prec q$ This case is similar to the first case, only now we delete some suffix.
3. $p_0 \preceq q \preceq p_{n-1} \rightarrow \exists j : p_j \preceq q \prec p_{j+1}$ Now we check if q lies inside $CH(S)$ by the orientation test $ccw(p_j, p_{j+1}, q)$. If the point lies outside, note that both p_j and p_{j+1} lie in the interior of $CH(S \cup \{q\})$. So we split the ordered list into two segments $((p_0, p_1), ...(p_{j-1}, p_j))$ and $((p_{j+1}, p_{j+2}), ...)$. We then follow the procedure in case 1 for the right subsegments and that in case 2 for the left subsegments.

With the process described above, we handle point insertion requests while also discarding points in the new interior. To get the tangents without adding q to S, we simply retrieve the indices without adding/deleting any segments. For a point q to lie in the interior, $p_0 \preceq q \preceq p_{n-1}$ must be applicable and the orientation test must pass for both the upper and lower hulls. Cases 1 and 2 give us one tangent, and case 3 gives us both of them depending on which orientation test passes. So in cases 1 and 2, the other tangent comes from the same procedure applied to the upper hull.

Finding these indices, and splicing the lists quickly are performed using standard available balanced binary search tree data structures and binary search algorithms on trees. We use treaps (randomized binary search trees) for these purposes [10] because of their easy implementation and fast performance.

We formulate and solve the problem of finding the farthest point along a direction as follows : Given a set S, and a direction v, find the subset of points $argmax_{p \in CH(S)}\{v \cdot x\}$ Note that the result is either a single point or a line segment on the boundary of the hull. We can assume that $v_y < 0 \vee v_y = 0 \wedge v_x > 0$. If v lies in the other half, we symmetrically solve the same problem using the upper hull. In our representation $((p_0, p_1), ...)$ of $LH(S)$ we find the first segment (p_i, p_{i+1}) satisfying $p_{i+1} \cdot v \leq p_i \cdot v$. If the inequality is strict, p_i is the solution, and if it's an equality, the line segment (p_i, p_{i+1}) is the solution. If no line segment satisfies the condition, then $max_{\prec}\{S\}$ is the solution.

3.2 Dynamic hull

We now consider the task of maintaining $LH(S)$ of a fully dynamic set S under point addition $S \leftarrow S \cup \{q\}$ and point removal requests $S \leftarrow S \setminus \{q\}$. The ideas here are mostly taken from [8], with a few minor improvements.

The big picture is to simulate the standard split and merge approach to compute the lower hull dynamically. We start with a tree-like structure T for maintaining the dynamic set S. T should have the following characteristics :

1. T should be approximately height balanced, so that the height is roughly $O(log(|T|))$.
2. The leaves store the data regarding the individual points of S.
3. The internal nodes have exactly two children, and will represent the "merge" step of the split and merge.
4. T supports addition and deletion of leaves in $O(log(|T|))$ time.

Such trees arise in Huffman coding [6], [11], and can be easily modified to satisfy all of these criteria. However, we don't go that route. Rather, we again implement such a tree using randomization ideas taken from [10] for ease of implementation and performance. Randomized search trees like these have an additional random "priority" associated with each tree node, and is used to keep the tree approximately height balanced. They are binary search trees by the lexicographical ordering of points in S, and binary max heaps by priorities. For our purposes, it's enough to push down all of the leaves (points of S) by setting their priorities to $-\infty$. We easily add the constraint that no node with a priority of $-\infty$ is a parent of another node.

Each leaf contains the information regarding the point in S it corresponds to. And each internal node contains the lower hull of all of the points in the leaves descended from said node. The lower hull is again represented as an ordered sequence of line segments like in section 3.1.

Whenever we ascend up the tree node T_s, with children T_l and T_r, we compute the convex hull of all leaves descended from T_s by merging the convex hull information present in T_l and T_r. If the sequence $((l_0, l_1), ...(l_{m-2}, l_{m-1}))$ and $((r_0, r_1), ...(r_{n-2}, r_{n-1}))$ are the lower hulls in T_l and T_r respectively, then there exists a bridge segment (l_i, r_j) such that the sequence $((l_0, l_1)...(l_{i-1}, l_i), (l_i, r_j), (r_j, r_{j+1})...(r_{n-2}, r_{n-1})$ is the convex hull of points in T_s. If one or both of the children of T_s is a leaf, we again revert to finding tangents using the case analysis presented in section 3.1. The bridge is found given the two sequence using a simultaneous binary search as described in [8] in time $O(log(m) + log(n))$, provided the two sequences are represented in memory using binary search trees. Some of the case analysis in finding the bridge is simplified and the reader is referred to the code for more details. Also note that the residual suffix of the hull of T_l and the residual prefix of the hull of T_r are not discarded. Instead, the above operations are reversed whenever we descend down the tree. The residuals are used to reconstruct the hulls of T_l and T_r from the hull of T_s after cutting the merged sequence along the bridge.

Because of the way these internal nodes handle sequences of hulls, at any point in time, either an internal node has its own consistent view of the hull of the leaves it's responsible for, or it's data is irrelevant. Once the computation reaches the root, the hull sequence at the root is consistent with hull of the entire set S. Local operations on nodes of T must therefore make the data at said node consistent before proceeding with said operations. E.g rotating a node of T (which we don't do, but mention for possible future reference) must first make the data at that node and its relevant relatives consistent before proceeding with the rotation.

The tangent and farthest point requests are handled the same way as described before in section 3.1, on the sequence at the root of T once it becomes consistent after point addition and/or deletions.

4 SUMMARY AND FUTURE WORK

We described an implementation of an online and a dynamic convex hull data structure, that maintains the sequence of convex hulls quickly and answers standard requests like point in polygon, tangent and extreme point queries in logarithmic time. The online case handles point insertions in $O(log(n))$ time while the fully dynamic case handles point insertions and deletions in $O(log(n)^2)$ time.

For future work, it's possible to try to parallelize parts of the data structures. E.g in the online setting, the part of code that deallocates the memory required to store parts of the hull that aren't needed anymore can be delegated to a separate process. This means that point addition and other requests aren't blocked by memory deallocation. Another area of optimization could be the batch processing of point addition queries in the fully dynamic setting. If the point being added is in the interior, we can save some time on answering the request by not recomputing the tree but rather "saving it for later" so to speak. This saves time because point addition in the dynamic setting is $O(log(n)^2)$ as of now, but point in polygon query is only $O(log(n))$. It's possible that the newer works take these ideas into account, but turning those into code is a challenge.

REFERENCES

[1] Gerth Stølting Brodal and Riko Jacob. 2000. Dynamic Planar Convex Hull with Optimal Query Time and O(log n · log log n) Update Time. In *Algorithm Theory - SWAT 2000*. Springer Berlin Heidelberg, Berlin, Heidelberg, 57–70.

[2] Timothy M. Chan. 2001. Dynamic Planar Convex Hull Operations in Near-Logarithmic Amortized Time. *J. ACM* 48, 1 (jan 2001), 1–12. https://doi.org/10.1145/363647.363652

[3] Timothy M. Chan. 2011. Three Problems about Dynamic Convex Hulls. In *Proceedings of the Twenty-Seventh Annual Symposium on Computational Geometry* (Paris, France) *(SoCG '11)*. Association for Computing Machinery, New York, NY, USA, 27–36. https://doi.org/10.1145/1998196.1998201

[4] Yun Chi, Hakan Hacígümüş, Wang-Pin Hsiung, and Jeffrey F. Naughton. 2013. Distribution-Based Query Scheduling. *Proc. VLDB Endow.* 6, 9 (jul 2013), 673–684. https://doi.org/10.14778/2536360.2536367

[5] John Hershberger and Subhash Suri. 1992. Applications of a semi-dynamic convex hull algorithm. *BIT Numerical Mathematics* 32, 2 (1992), 249–267.

[6] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

[7] Riko Jacob and Gerth Stølting Brodal. 2019. Dynamic Planar Convex Hull. arXiv:1902.11169 [cs.CG]

[8] Mark H. Overmars and Jan van Leeuwen. 1981. Maintenance of configurations in the plane. *J. Comput. System Sci.* 23, 2 (1981), 166–204. https://doi.org/10.1016/0022-0000(81)90012-X

[9] F. P. Preparata. 1979. An Optimal Real-Time Algorithm for Planar Convex Hulls. *Commun. ACM* 22, 7 (jul 1979), 402–405. https://doi.org/10.1145/359131.359132

[10] Raimund Seidel and Cecilia R Aragon. 1996. Randomized search trees. *Algorithmica* 16, 4-5 (1996), 464–497.

[11] Jan Van Leeuwen. 1976. On the Construction of Huffman Trees.. In *ICALP*. 382–410.