Dynamic Convex Hull for Simple Polygonal Chains in Constant Amortized Time per Update

Norbert Bus* Lilian Buzer*

Abstract

We present a new algorithm to construct a dynamic convex hull in the Euclidean plane, supporting insertion and deletion of points. Both operations require amortized constant time. At each step the vertices of the convex hull are accessible in constant time. The algorithm is on-line, does not require prior knowledge of all the points. The only assumptions are that the points have to be located on a simple polygonal chain and that the insertions and deletions must be carried out in the order induced by the polygonal chain.

1 Introduction

The convex hull of a set of n points in a Euclidean space is the smallest convex set containing all the points. Constructing the convex hull of a point set is a fundamental problem in computational geometry. It has applications in, e.g., pattern recognition, image processing and micro-magnetic analysis [2, 6]. Many problems can be reduced to determining the convex hull of a point set, such as Delaunay triangulation and half space intersection [5]. Therefore developing robust and efficient algorithms for the core problem has received much attention. If one considers the real-RAM model, an optimal output sensitive algorithm to construct the convex hull of n points in a plane was published in [7] having $O(n \log h)$ time complexity where h is the output size. If the point set is a simple polygonal chain, the best algorithm, a result of Melkman, runs in linear time [1]. If one requires the data structure to be dynamic, namely to handle insertions and deletions of arbitrary points an optimal algorithm requiring $O(\log n)$ time for both operations was proposed in [3]. Changing the computational model to the word-RAM model and using Graham's scan [8] to construct a convex hull the running time is essentially the time to sort the points, taking, e.g., $O(n \log \log n)$ time [9]. Dynamic data structures supporting deletion and insertion in the word-RAM model require an optimal $O(\frac{\log n}{\log \log n})$ time for both operations assuming that word length is $\Theta(\log n)$, see [4].

*Université Paris-Est, LIGM, A3SI-ESIEE, France, {busn, buzerl}@esiee.fr

Our contribution In this paper we give an on-line algorithm to construct the dynamic convex hull of a simple polygonal chain in the Euclidean plane supporting deletion of points from the back of the chain and insertion of points in the front of the chain. Both operations require amortized constant time considering the real-RAM model. The main idea of the algorithm is to maintain two convex hulls, one for efficiently handling insertions and one for deletions. These two hulls constitute the convex hull of the polygonal chain.

2 Overview of our algorithm

Our algorithm works in phases. For a precise formulation let us first define some necessary notations. A polygonal chain S in the Euclidean plane, with n vertices, is defined as an ordered list of vertices $S = (p_1, p_2, \ldots, p_n)$ such that any two consecutive vertices, p_i and p_{i+1} are connected by a line segment. A polygonal chain is called simple when it is not self-intersecting. For simplicity, we assume that the points are in general position. Our algorithm handles insertion and deletion of points into the current convex hull in the order induced by S. This results in the fact that the current convex hull always contains a contiguous subchain of S, let us denote it by $S_i^j = (p_i, \cdots, p_j)$ and the points are effectively inserted/deleted in a FIFO manner. Let us denote the convex hull of S_i^j with C_i^j. Therefore, given a convex hull C_i^j, inserting a point results in C_i^{j+1} while removing the first point results in C_{i+1}^j.

At the beginning of each phase, we initialize a simple data structure called the *phase convex hull* that maintains the representation of the convex hull of a subrange of the polygonal chain. Each phase handles an arbitrary number of insertions and handles deleting the points that were present when the phase started. Assuming that the phase convex hull first covered S_a^b this means we can delete the points $p_a \ldots p_b$. A phase ends, when we first delete a point that was not covered by the initial convex hull. After that, a new phase starts and we initialize a new phase convex hull. See Figure 1.

We state the main result of our algorithm in Theorem 1.

This is an extended abstract of a presentation given at EuroCG 2015. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear in a conference with formal proceedings and/or in a journal.

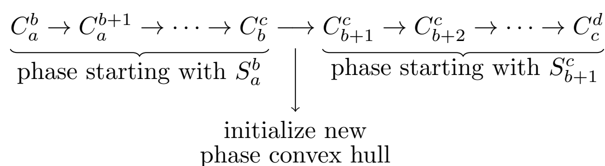

Figure 1: Example of two phases

Theorem 1 *The amortized time complexity of insertion and deletion of points in a convex hull of a simple polygonal chain is constant.*

Proof. Assume that each point has been inserted and later removed. In Section 6 we show that the $i-th$ phase runs in $O(k_i + l_i)$ time where k_i is the number of insertions in the phase and l_i is the number of deletions. During the whole algorithm each point has been inserted and deleted exactly once hence there have been n insertions and n deletions overall. Therefore the overall running time of the phases is $O(n)$, yielding the desired result. □

3 Convex hulls

In this section we introduce the phase convex hull representing the convex hull of the polygonal chain. Furthermore, we describe two convex hulls that are maintained during the phase. These two hulls' purpose is to enable that insertion and deletion of points for the phase convex hull run in constant amortized time. We suggest that the reader is familiar with the Melkman algorithm [1] as our method builds heavily on it. Briefly, the method builds the convex hull of a polygonal chain by iteratively (in the proper order) adding the points to the convex hull and modifying it as necessary. At the beginning of the phase let S_a^b be the current polygonal chain while at the end let it be S_b^c. Let us denote by S_i^j the polygonal chain at an arbitrary step during the phase and C_i^j the corresponding convex hull.

Phase convex hull The *phase convex hull* denoted by C^* is the data structure representing the convex hull C_i^j, containing all of its points in two dequeues. Every point is contained in exactly one of the two dequeues except for two: the front of both dequeues refer to the same point of the subchain, the one contained in C_i^j with highest index and similarly, the back of both dequeues refer to the same point of the subchain, the one contained in C_i^j with lowest index. We refer to the front of both dequeues as front opening and to the back as back opening. Connecting the two dequeues gives the ordered circular list of points in C_i^j. See Figure 2. Clearly, if one considers the back opening to be *closed* (i.e., as if being glued together, removing the duplicate copy of the back point), one has the data structure used in the Melkman algorithm. At the beginning of the phase C^* is built for S_a^b using the Melkman algorithm.

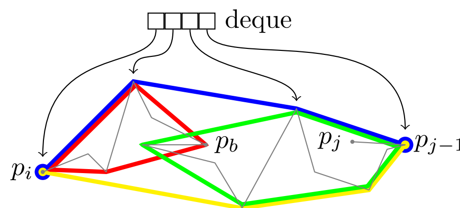

Figure 2: The two dequeues (blue and yellow) constituting C^*. For clarity we include one schematically. In green and red we depict C^+ and C^- respectively.

The following data structures aid to handle insertion or more importantly deletion of points (since the phase convex hull itself could handle insertions).

Incremental convex hull The *incremental convex hull* is a convex hull incrementally built with the Melkman algorithm for the points in the polygonal chain added after the initialization of the phase. Let us denote it by C^+. At any step S_i^j, formally, C^+ is the same as C_{b+1}^j.

Decremental convex hull The *decremental convex hull* is a convex hull built with the Melkman algorithm for the points present at the beginning of the phase but according to the reverse order of the points. Let us denote it by C^-. At each deletion a point is removed from C^-. At any step S_i^j, formally, C^- is the convex hull of the points S_b^i (note the reverse order). This data structure has to maintain additional information that will be used for efficient deletion of points. First, while creating C^-, for each point in S_b^a, the list of points that were removed from the previous convex hull should be kept. Let us call these points the *history* of a point. This enables that the algorithm can be 'rewound', i.e, the points of C^- can be deleted in a FILO order. As C^- was built in the reverse order this is exactly the deletion order we need. Second, the polygonal regions defined by $C_k^j \backslash C_{k+1}^j$ for $a \leq k \leq b$, in other words the difference between consecutive convex hulls in the Melkman algorithm for building C^-, have to be kept. At the beginnning of each phase we build the decremental convex hull for S_b^a. In Figure 3 we show the regions where the red polygonal chain is S_a^b. The green line corresponds to the polygonal chain of the points inserted during the phase.

We will need a simple property of the regions, namely that they form an ordered partitioning of the plane that enables a certain operation. See Lemma 2.

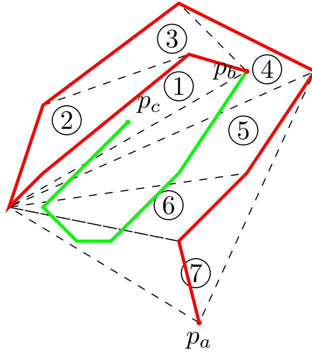

Figure 3: Regions.

Lemma 2 *While constructing C^- for S_a^b one can create an ordered list of regions in $O(b-a)$ time. Such a partitioning enables the maintenance of the highest ordered region that contains any point inserted during the phase. This operation takes $O(c-b)$ time for a phase.*

In Figure 3 the highest ordered region containing a point inserted in the phase is region no.7. The lemma is a straightforward consequence of the Melkman algorithm as given the current highest ordered region containing an inserted point one only has to check whether a newly inserted point is contained in the region following (in order) the current one. One has to be careful when adding the point p_{b+1} as there was no highest ordered region before, but this case is trivial.

4 Insertion

In this section we describe the method to handle the insertion of p_{j+1} into the convex hull of S_i^j.

In order to insert the point p_{j+1} into C^* it is sufficient to do one step of the Melkman algorithm at the front opening of the dequeues. For that, consider the back opening of C^* to be closed, i.e., the two dequeues behave like one. One has to be cautious when the Melkman algorithm deletes the point being the back opening as the new back opening should be by our definition the point in C_i^j with least index. C^+ has to to be updated with p_{j+1} as well, which is done using the Melkman algorithm. Moreover, as long as the front opening is one of the points in C^- one has to update the highest ordered region of C^- containing any inserted point.

5 Deletion

In this section we describe the deletion of the point p_i from S_i^j. We will need a property concerning the points in the phase convex hull C^*. Note, that the points of C^* either belong to C^+ or C^-. The following lemma describes how the points' distribution (with respect to whether they belong to C^+ or C^-) changes during the execution of the operations in a phase. It states that there cannot be arbitrary distributions, e.g., points from C^+ and C^- in an alternating order.

Lemma 3 *The points in C^* are partitioned into contiguous ranges according to whether they belonging to C^- or C^+. At any step in a phase there are at most two such partitions, one containing points of C^+ and one containing points of C^-. The partitioning changes in a specific pattern during a phase: at the beginning there are only points from C^- in C^*; then two partitions; finally only points from C^+ are located in C^*.*

The fact that C^* is partitioned into at most two parts is a consequence of the simplicity of the polygonal chain. The strict ordering also follows easily since C^+ is monotonically growing while C^- is monotonically shrinking.

To delete the point p_i there are several scenarios that have to be handled differently. As a common point in all cases p_i has to be removed from C^- and its history has to be added to C^- ('rewinding' the Melkman algorithm). Let us first group the different cases according to Lemma 3.

Case 1: If the phase convex hull contains only points from C^- then one can simply remove p_i from C^* and add the points of the history of p_i to C^*. This is valid as long as there shall be no point of C^+ added to C^*. The highest ordered region is maintained exactly for checking this. As long as the deleted region has no points assigned to it we can proceed as explained. If it is not empty then an expensive operation is required, namely to add all the points of C^+ to C^*. This can be done with the Melkman algorithm considering the back opening to be closed. Even though this operation might require linear time in the number of insertions it can happen only once for each phase therefore its amortized time complexity is constant.

Case 2: If the phase convex hull contains points from both C^+ and C^-, we have the most complicated case. Obviously the point p_i to be removed is in C^-. Let us denote the neighbors of p_i in C^* by x and y. The edge between x and y would become an edge of C^* if there are no points in the triangle defined by p_i, x, y. But usually this is not the case therefore one has to create new edges of C^* that correspond to the vertices located in this triangle. Adding these points (and at the same time checking if there are points inside the triangle) is done as follows. We further categorize this case into three sub cases depending on the neighbors of p_i.

Case 2a: If both x and y belong to C^- then clearly C^* can only be modified by the points from the history of the currently deleted point p_i. In such a situation using the Melkman algorithm one can add the ordered history of p_i to C^*.

Case 2b: If one point, e.g., x belongs to C^+ then there might be points of C^+ that have to be inserted into C^*. In this case first the history of p_i should be inserted into C^* and then starting from x we shall add the vertices of C^+ in the circular order (starting with the neighbor of x not contained in C^*) using the Melkman algorithm but just as long as they create new vertices on C^*. One can show that if a point of C^+ is inside C^* then there are no other points that shall be inserted.

Case 2c: If both x and y belong to C^+ then a similar process has to be carried out namely first inserting the points of the history and then the vertices of C^+ in the proper circular order starting from x and y. Note that x and y define two different parts of C^+ that have to be inserted into C^* and to maintain a low running time one has to insert points from these two parts in an alternating order (one cannot proceed with points from the part of x after finishing the points starting from y). When a point remains inside the phase convex hull we can finish inserting points from its part. See Figure 4 for a schematic illustration of the process. In order to be able to utilize the Melkman algorithm it has to be true that the added points belong to a simple polygonal chain, otherwise using Melkman would be impossible. This can be ensured by using some auxiliary points thats purpose is only to make the path non self intersecting.

Case 3: If C^* contains only points of C^+ than $C^- \subset C^+$ therefore there is no change in C^*.

How to add auxiliary points and the proof of correctness, e.g, why is it sufficient to add only the history of p_i in Case 2 is left for the full paper.

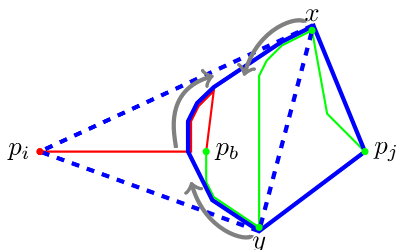

Figure 4: Deleting the point p_i. Solid blue lines denote the convex hull after deleting p_i. Gray arrows denote the points that have to be inserted into C^*.

6 Complexity

Let us now show that each phase takes time linear in the number of insertions and deletions. Let us denote them by k and l respectively.

Initialization: Clearly, initializing C^*, C^+ and C^- is linear in the number of points present at the beginning of the phase since we only utilize a modified Melkman algorithm. This indeed is the same as the number of points deleted in the phase. Therefore the complexity of initialization is $O(l)$.

Insertion: Using the Melkman algorithm for both the phase convex hull and the incremental convex hull takes $O(k)$ time. During insertion of points one has to also maintain the highest ordered region containing any point of C^+. This can be done in $O(k)$ time.

Deletion: Clearly, during executing the deletions, the number of points inserted into C^* using the Melkman algorithm is not more than the total size of history of points in C^- which is $O(l)$ and the points of C^+ added to C^* which is less than $O(k)$ as no point can reappear on C^* due to the monotonicity of the operations.

This results in the following theorem.

Theorem 4 *The running time of one phase is $O(k + l)$ given that there are k insertions and l deletions.*

7 Conclusion

We have presented an algorithm that handles insertion and deletion of points from/into the convex hull of a simple polygonal chain. Each operation takes amortized constant time. The operations are carried out in a FIFO manner, namely that points are inserted on one end of the chain while points are deleted from the other. It would be interesting to see if this constraint can be removed, namely that points can be inserted/deleted on both ends of the chain.

References

[1] A. A. Melkman. *On-line construction of the convex hull of a simple polyline* Information Processing Letters, 1987.

[2] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction* Springer-Verlag, 1985.

[3] G. S. Brodal and R. Jacob. *Dynamic Planar Convex Hull* FOCS, 2002.

[4] E. D. Demaine and M. Patrascu. *Tight Bounds for Dynamic Convex Hull Queries (Again)* SOCG, 2007.

[5] F. Aurenhammer. *Voronoi Diagrams – a Survey of a Fundamental Geometric Data Structure* ACM Computing Surveys, 1991.

[6] D. G. Porter, E. Glavinas, P. Dhagat, J. A. O'Sullivan, R. S. Indeck and M. W. Muller. *Irregular grain structure in micromagnetic simulation* Journal of Applied Physics, 1996.

[7] T. M. Chan. *Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions* Discrete and Computational Geometry, 1996.

[8] R. L. Graham. *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set* Information Processing Letters, 1972.

[9] Y. Han. *Deterministic sorting in $O(n \log \log n)$ time and linear space* Journal of Algorithms, 2004.