Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time

Hervé Brönnimann[1][*] and Timothy M. Chan[2][**]

[1] Computer and Information Science, Polytechnic University, Six Metrotech Center, Brooklyn, NY 11201, USA.
[2] School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada.

Abstract. We present space-efficient algorithms for computing the convex hull of a simple polygonal line in-place, in linear time. It turns out that the problem is as hard as stable partition, i.e., if there were a truly simple solution then stable partition would also have a truly simple solution, and vice versa. Nevertheless, we present a simple self-contained solution that uses $O(\log n)$ space, and indicate how to improve it to $O(1)$ space with the same techniques used for stable partition. If the points inside the convex hull can be discarded, then there is a truly simple solution that uses a single call to stable partition, and even that call can be spared if only extreme points are desired (and not their order). If the polygonal line is closed, then the problem admits a very simple solution which does not call for stable partitioning at all.

1 Introduction

An algorithm is *space-efficient*, if its implementation requires little or no extra memory beyond that which is needed to store the input. In-place algorithms are tricky to devise due to the limited memory considerations. For the classical sorting problem, both quicksort and heapsort are in-place algorithms (it is well-known that the first can be implemented with logarithmic expected amount of extra memory, and the second with a constant amount [4]). It turns out that devising in-place merge and mergesort is a challenge [6, 8, 9]. Many other classical problems have been considered when space is dear.

Recently, several classical problems of computational geometry have been revisited with space requirements in mind. Two-dimensional convex hull of points is one of them that has been solved in almost every respect in the past twenty years: there are optimal, output-sensitive solutions which compute the smallest convex polygon enclosing a set of points in the plane. In [3], Brönnimann et al. give optimal in-place algorithms for computing two-dimensional convex hulls. For this problem, the points on the convex hull can be reordered at the beginning

of the array, so that the output merely consists of a permutation of the input (encoded in the input array itself), and the number of points on the hull.

Space-efficient algorithms have many advantages over their classical counterparts. Mostly, they necessitate little memory beyond the input itself, so they typically avoid virtual memory / paging and external I/O bottlenecks (unless the input itself is too large to fit in primary memory, in which case I/O-efficient algorithms can be used).

Convex hull of a simple polygonal line. Computing the convex hull of a simple polygonal (either open or closed) is another of the classical problems of computational geometry, and was long suspected to be solvable in linear time. Unfortunately, correct algorithms are outnumbered by the number of algorithms proposed in the literature that have turned out to be flawed [1]. Two algorithms that are correct are Lee's algorithm [10] (a variant of Graham's scan for closed polygonal chains) and Melkman's [11] (works for open polygonal chains as well, in an online fashion).

The problem is two-fold: the polygonal line can be either closed or open. For closed chains, we can use Lee's stack-based algorithm and implement the stack implicitly in the array, using the fact that the vertices on the convex hull are sorted. This leads to a linear-time constant-space solution presented in Section 2.1.

For open chains, the problem is complicated by the fact that either endpoint may lie inside the convex hull. The only solutions known use a deque and we show how to encode a deque implicitly with logarithmic extra storage in Section 3.1. We then improve the storage required to constant size using known techniques. Another solution can be obtained by using the relationship mentioned in [3] between convex hulls and stable partition and by reusing space from points that have been discovered to be non-extreme.

Other related work. There seems to be little point for in-place algorithms when the output requires linear space simply to write down, so one may assume that the output is a permutation of the input or can otherwise be represented in a small amount of memory (e.g., the answer to many geometric optimization problems typically consist of identifying a constant-sized subset of the input). Recently, Chen and Chan [5] proposed another model in which the output is written to an output stream and never read again; only a limited of extra memory is available for workspace. They gave a solution for the problem of counting or enumerating the intersections among a set of n line segments in the plane, with $O(\log^2 n)$ extra memory. There could be up to $\Theta(n^2)$ intersections, but they are written to the output stream and never needed again beyond what is stored in the logarithmic working memory. A similar model holds for other classical problems such as Voronoi diagrams and 3D convex hulls.

Equivalence with stable partition. There are linear-time algorithms for performing stable partition in-place (i.e., how to sort an array of 0's and 1's in-place, respecting the orders of the 0's and 1's); see papers by Munro, Raman, and

Salowe [12] and Katajainen and Pasanen [7]. These algorithms are not simple, however. Nevertheless, efficient implementations are provided as a routine in modern algorithm libraries, e.g. the C++ STL. A truly practical implementation may use available extra storage to speed up the computation, and only resort to the more involved algorithms mentioned above if no extra storage can be spared. Hence it makes sense to try and obtain simple algorithm that use stable partition as subroutine.

The partitioning problem itself is linear-time reducible to convex hull in the following way: Given an array A of 0's and 1's, compute the convex hull of the polygonal line defined by $B[i] = \left(i, i(n-i)(A[i] - 0.5)\right)$. These points lie on two parabolas $y = \pm\frac{1}{2}x(n-x)$, and therefore all appear on the boundary of the convex hull, first the points for which $A[i] = 0$ in order on the lower parabola and those for which $A[i] = 1$ in reverse order on the upper parabola. Thus the stable partition can be constructed in linear time by reversing the 1's in the final array. It thus appears difficult to have a truly simple linear-time algorithm for computing the convex hull of a simple polygonal line, given that no truly simple linear-time algorithm exists for stable partition.

It turns out that by using stable partition as an oracle, we obtain a very simple algorithm. If we are not interested in the order of the points on the convex hull, then we may even forego stable partition altogether, and therefore obtain a truly simple algorithm given in Section 3.3.

2   Closed chains

For closed simple polygons, the solution turns out to be very simple: the input is represented in an array $A[1] \ldots A[n]$ of vertices, which can be cyclically permuted at will. One may therefore assume that $A[1]$ is a vertex of the convex hull (e.g. the vertex of minimum abscissa). There is an algorithm due to Lee which closely resembles Graham's scan and uses only one stack [10]. We give a brief outline of the algorithm first, then show how to implement it in a space-efficient manner.

2.1   Overview of Lee's algorithm

In this description, we assume that the polygon is oriented counterclockwise and that $A[1]$ is minimal in some direction. Fortunately, the orientation of the polygon is given by the order type of $(A[n], A[1], A[2])$ and can be computed in $O(1)$ time. Should it turn otherwise, the whole array can be reversed. The invariant is the vertices on the stack form a convex polygon (when viewed as a circular sequence). That is, the vertices in the stack form a convex polygonal line, and the point at the bottom of the stack (i.e. $A[1]$) is always to the left of the line joining two consecutive points in the stack.

Lee's algorithm starts by pushing the first two vertices on the stack, and maintains the line L that connects the top two vertices on the stack, as well as the line L' joining the bottom and top vertices of the stack. When a point is

processed, it may fall into several regions as depicted in figure 1(left). There are several cases:

- If current vertex is not to the left of L, restore the invariant by backtracking/deleting vertices from the top of the stack until a convex turn is encountered, or there is only one vertex left on the stack. Then push the current vertex on the stack and recompute L and L'.
- If current vertex is to the left of L but not to the left of the line L' (i.e. falls into the pink or yellow region), then ignore it and process next vertex.
- If current vertex is to the left of both L and L', then push the current vertex on the stack and recompute L and L'.

Note that in the third case, both lines L and L' always rotate counterclockwise, but in the first case, after popping vertices from the stack they may rotate either way. In particular, some previously processed vertices may end up on the left side of L'. The algorithm therefore does *not* maintain the invariant that the stack contains the convex hull of the vertices seen so far. It *does* however maintain the invariant that the stack contains the prefix of the convex hull ending at the vertex on top of the stack, and that the subsequent points (from the top of the stack to the current point) are to the right of L'. In particular, if the last point has maximal abscissa, the stack contains the lower convex hull of the points, a fact that will be useful later. And more importantly, Lee proved that if the chain is closed, then after all the points have been processed the stack contains the entire convex hull.
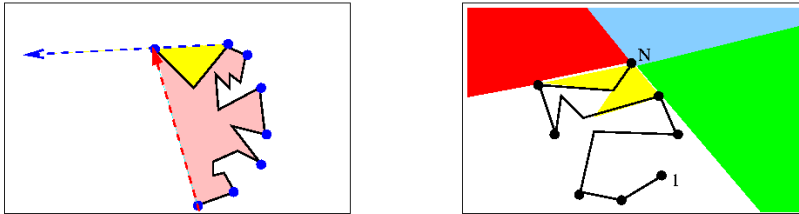

Fig. 1. (left) The two types of regions for processing a point in Lee's algorithm. The points on the stack are in blue, the blue line is L and the red line is L'. (right) The four types of regions for processing a point in Melkman's algorithm. (Both diagrams courtesy of Greg Aloupis).

2.2   A space-efficient implementation

Implementing the previous algorithm in place is trivial since the stack can be stored in the prefix of the array. The minimal-abscissa vertex can be found and

the entire array permuted, both in linear time and with a constant amount of extra memory. Moreover, the points inside the convex hull can be kept in the array, if we are careful to perform swaps instead of assignments when popping from and pushing into the stack. The algorithm then produces a permutation of the input and an index h such that $A[1]..A[h]$ form a convex polygon and the points in $A[h+1]..A[n]$ are inside the convex hull. Note that this algorithm is not much different than the in-place Graham-Andrew scan proposed by Brönnimann et al. [3], when the points have already been sorted by abscissa; the only modification is the use of the line L', and the fact that the points haven't been sorted but instead that they lie on a simple polygonal line. The runtime of the algorithm is clearly linear, and it uses $O(1)$ extra memory.

2.3 Open chains: a special case

Although this algorithm only works for closed chains (see the next section for open chains), it also works for open chains in the special case where both endpoints are extreme vertices. For simplicity of the discussion, we assume that $A[1]$ has minimal abscissa, and $A[n]$ is maximal. This lets us use the fact that the convex hull is a concatenation of the lower hull (from $A[1]$ to $A[n]$) and the reverse of the upper hull (from $A[n]$ to $A[1]$).

As observed above, running the one-stack algorithm of from $A[1]$ to $A[n]$ would produce the lower hull, and from $A[n]$ back to $A[1]$ the upper hull. Unfortunately, we cannot run both, but we can separate the points above and below the line L joining $A[1]$ to $A[n]$ by using stable partition, and reversing the second half. This gives us a polygonal line $\mathcal{L}_1$ joining $A[1]$ to $A[n]$ that stays below the straight line L, followed by another polygonal line $\mathcal{L}_2$ starting at $A[n]$ that stays above L.

Unfortunately, the resulting polygonal lines are not necessarily simple, but they have structure: in particular the lower hull of $\mathcal{L}_1$ is the same as that for $\mathcal{L}$, and the vertices of $\mathcal{L}_1$ occur in the same order they occur on $\mathcal{L}$ (this is a consequence of Jordan's curve theorem, proved in [2]). This is sufficient to ensure that the one-stack algorithm still works as intended on $\mathcal{L}_1$ and produces the lower hull of $\mathcal{L}$. Similarly, running the algorithm on $\mathcal{L}_2$ produces the upper hull of $\mathcal{L}$. The two hulls can be concatenated in place to form the whole convex hull of $\mathcal{L}$.

3 Open chains

While the former algorithm works easily for closed chains, it does not work for open polygonal chains, due to the fact that some vertices might be removed from the stack but appear to the left of the line L' and therefore contribute to the convex hull. Melkman [11] showed how to use a deque instead of a stack to cope with this problem. We give a brief outline of his algorithm first, then show how to adapt the implementation to make it space-efficient.

3.1 Overview of Melkman's algorithm

The points are processed in the order in which they appear in the input array. Melkman's algorithm maintains their CH as a deque (insertion and deletion from both front and back), which is initially the triangle formed by the first three points. For simplicity, we describe a version that uses a deque and one extra point, stored in a special register, which contains the last point added to the convex hull.

When a point is processed, it can fall into four types of regions, depicted in figure 1(right); note that it cannot fall into any other region without violating the simplicity of the polygonal line. These regions are determined solely by the point in the special register and the front and back vertices of the queue. The invariant of the algorithm is that the special point and the vertices in the deque form a convex polygon, when viewed as a circular sequence . The simplicity of the polygonal line, together with Jordan's curve theorem, imply that when a point comes out of the yellow region, it always does so through one of the two edges of this polygon that join the special point to the top or the bottom vertex of the deque.

o

- If yellow, ignore this and all following vertices until one emerges into the other regions.
- If red, push the point in the special register onto the front of the deque, then insert the current point into the special register. To restore the invariant, backtrack/delete vertices from the front of the deque until a convex turn is encountered.
- If green, push the point in the special register onto the back of the deque, then insert the current point into the special register. Restore the invariant by backtracking/deleting vertices from the back of the deque until a convex turn is encountered.
- If blue, simply replace the point in the special register by the current point, and restore the invariants as in both cases red and green.

This process is repeated for every point in turn. Note that the algorithm is completely symmetric and therefore does not assume the any orientation of the polygonal line. In fact, the first point of the array does *not* need to appear on the final convex hull, nor does the chain need to be closed. The algorithm is online, meaning that points can be added in a streaming fashion.

3.2 A space-efficient implementation using implicit pointers

The main problem is how to implement a deque of n elements in place, i.e., using only the first n cells of the array when n points have been processed. This is a non-trivial task, at least as hard as stable partitioning. We show that techniques developed for stable partitioning can actually be adapted to solve our problem.

If we represent the deque as a doubly linked list, then each deque operation can trivially be accomplished in constant time. The problem with this approach

is of course the extra space needed for the pointers. The key idea is that pointers need not be stored explicitly but can be encoded implicitly via permutations of the input elements: since the points in the deque form a convex polygon, they are sorted by, e.g., angular order. One way to do that is to fix the origin inside the convex hull (e.g., the barycenter of the first three points) and pick a direction (e.g., the horizontal). In the sequel, when we say that a is less than b, we mean that its principal polar angle is less than that of b.

In more details, we store the first few and last few elements in two separate small deques (the *front deque* and *back deque*). The rest of the elements are stored inside the given array, which is divided into blocks of size $s = 4\lceil \log_2 n \rceil$. The blocks are linked together, in order, by the following scheme: Within each block, we encode $2\lceil \log_2 n \rceil$ bits by pairing consecutive elements and permuting each pair (a, b) so that having a less than b means the corresponding bit is a 0 and vice versa. These bits form the two pointer fields (the successor and predecessor) of the doubly linked list.

Insertions/deletions to the front/back are done directly within the two small deques, whose sizes are kept between 0 and $2s$. When the size of the front/back deque reaches $2s$ (a *full event*), we extract s elements from it, form a new block, and update two pointers from and to the new block. When the size of the front/back deque reaches 0 (an *empty event*), we take out the first/last block of the linked list, and insert its s elements into the small deque; furthermore, to ensure that the used blocks occupy a prefix of the array, we swap the deleted block with a block at the end of the array and readjust a constant number of pointers. After a full event, the corresponding small deque has exactly s elements and hence the next event will not occur for another s operation. Each such event processing requires $O(s)$ time, but two events are separated by at least s insertions/deletions, so the amortized cost per insertion/deletion in the deque is $O(1)$.

The extra space used is $O(\log n)$, for the two small deques. By more theoretical tricks, the space complexity can be made even smaller. One option is to handle the small deques recursively, by dividing their elements into tinier blocks in the same manner. Similar to Munro, Raman, and Salowe's stable partition method [12], this should result in an $O(\log^* n)$ space bound. Another option is to recurse for just two levels until the deque size is small enough ($O(\log \log n)$) so that all pointers can be packed into a single word, and pointer manipulations can be done in $O(1)$ RAM operations by table lookup. This is analogous to Katajainen and Pasanen's stable partition method [7] and should yield $O(1)$ space. Since either option is probably too complicated for actual implementation, we will not elaborate further on these refinements.

At the end, to produce the convex hull vertices in order in a prefix of the array, we can simply perform repeated deletions from one end of the deque. Although consecutive pairs have been permuted by the above process, we can permute the pairs back, knowing that they should form a convex polygon. As before, by being careful, we can ensure that points not on the hull boundary remain in a suffix of the array.

3.3   A simpler, "destructive" implementation

If points not on the hull boundary need not be in the final array and can be destroyed, we can give a simple algorithm that directly reduces the problem to stable partitioning. In fact, if the convex hull vertices need not be ordered in the final array (i.e., we just want to identify the set of extreme points), we can avoid the stable partitioning subroutine altogether and thus obtain a truly simple algorithm.

The problem is again how to implement the deque in-place. The key idea is this: if there are no deletions (i.e., all the points are on the boundary of the convex hull), then nothing needs to be done: all the vertices are extreme; in fact, in this case simply stably partitioning the points with respect to the line joining the first and the last point and reversing the second portion produces the convex hull. But if there are many deletions, cells of deleted elements can be used to store other information (pointers, for example).

We describe one approach based on this idea. For simplicity's sake, we assume that each cell can hold two extra bits for marking purposes (*live* or *dead*, and $-$ or $+$); later we will discuss how this assumption can be removed. The deque has to be stored within the first n cells of the array, where n is the current number of insertions (*not* the number of elements currently in the deque).

Basically, the deque is decomposed into two stacks: elements of sign $-$ in the array form the front part reversed, and elements of sign $+$ form the back part. Insertion is straightforward: just put the new element at the end of the array, and mark it $-$ or $+$ depending on whether we are inserting to the front or back of the deque.

An element is deleted by marking it as dead. To speed up computation, we use dead cells to store pointers as follows: Consider the elements of the array of one sign, in left-to-right order. They form a sequence of alternating blocks of live elements and dead elements. The invariant is that the rightmost element of each dead block should hold a pointer (i.e., the index) to the rightmost element of the preceding live block. See figure 2 for an illustration, where the O's represent live elements and the X's represent dead elements.
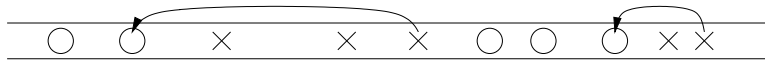

Fig. 2. Representing a deque (or two stacks) in a single array. Live and dead elements of one sign are shown.

It is not difficult to maintain this invariant after a deletion: just imagine when the rightmost O is changed to an X in figure 2; several cases may arise, but only a constant number of pointers need to be updated. To demonstrate the simplicity of the approach, we provide complete pseudocode of the insertion and deletion procedure below. Here, ℓ_σ points to the rightmost live element of sign $\sigma \in \{-, +\}$, and d_σ points to the rightmost dead element of sign σ.

$\text{Insert}_\sigma(x)$:
 1. $\ell_\sigma = k = k + 1$, $A[k] = x$
 2. mark $A[k]$ live and of sign σ

$\text{Delete}_\sigma()$:
 1. mark $A[\ell_\sigma]$ dead
 2. if $\ell_\sigma > d_\sigma$ then $d_\sigma = \ell_\sigma$
 3. i = predecessor of ℓ_σ among elements of sign σ
 4. if predecessor exists then
 5.   if $A[i]$ is live then $\ell_\sigma = A[d_\sigma] = i$ else $\ell_\sigma = A[d_\sigma] = A[i]$
 6. else {
 7.   compress array by keeping only live elements
 8. k = size of compressed array
 9.   reverse first half of array and switch the sign of these elements
 10. }

Searching for the predecessor of a live element (line 3) is a non-constant-time operation and requires time proportional to the distance between the element and its predecessor. However, this search is done at most once for each element (when its status changes from live to dead), so the total time is still linear in the size of the array.

One scenario has not yet been addressed: what if we run out of elements of one sign? This can be fixed by a standard amortization trick (used in the well-known two-stack simulation of a deque): we just re-divide the deque in the middle and start a new phase, as described in lines 7–10. (Notice that lines 7 and 9 can be done in-place easily.) If the i-th phase initially has k_i elements and ends after m_i insertion/deletion operations, then the phase requires $O(k_i + m_i)$ total time. Because the above strategy ensures that $m_i \geq k_i/2$, the running time of the i-th phase is $O(m_i)$ and the overall running time is $O(n)$, i.e., the amortized cost per update remains $O(1)$.

At the end, we can compress the array to remove all dead elements and thus have the convex hull vertices stored in a prefix of the array. If the vertices are required to be ordered, we can invoke a stable partition subroutine to put all $-$'s before all $+$'s and reverse the $-$ elements; otherwise, our algorithm is completely self-contained.

Finally, if it is not possible to steal two extra bits per cell, we can insert/delete to the deque only when we have gathered a pair of elements. We can permute the pair (a, b) so that having a left of b means the sign is $-$ and vice versa. A dead cell can be signaled by a pair (a, b) with either a or b a point at infinity.

4   Conclusion

The problem of computing the convex hull of a simple polygonal line is well-known to be solvable in linear time. In this paper, we have shown that it can be solved in linear time in-place, and that the in-place problem is as hard as stable partition in the following sense: any linear-time algorithm for one implies a not

too convoluted linear-time algorithm for the other. Given that the algorithms for stable partition are rather involved, we do not expect an easy solution for this problem either. Nevertheless, we have given a simple $O(\log n)$-space solution which can be extended to an $O(1)$-space solution at the expense of the complexity of the implementation. If the chain is closed, the problem admits of a very simple in-place linear-time solution, which does not call for stable partitioning at all. If the chain is open but both endpoints are extreme, then a single call to stable partition and two calls to the same very simple in-place linear-time algorithm solve the problem.

References

1. G. Aloupis. A History of Linear-time Convex Hull Algorithms for Simple Polygons. `http://cgm.cs.mcgill.ca/~athens/cs601/`
2. J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry.* Cambridge University Press, 1998.
3. H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G.T. Toussaint. Space-efficient planar convex hull algorithms. To appear in *Theoretical Computer Science.* Special issue of selected papers from Latin American Theoretical INformatics (LATIN 2002).
4. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* 2nd edition, MIT Press, 2001.
5. E. Y. Chen and T. M. Chan. A Space-Efficient Algorithm for Segment Intersection. In *Proc. 15th Canadian Conference on Computational Geometry*, pp. 68-71, 2003.
6. V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science* 237:159-181, 2000.
7. J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT* 32:580-585, 1992.
8. J. Katajainen and T. Pasanen. Sorting multiset stably in minimum space. *Acta Informatica* 31:410-421, 1994.
9. J. Katajainen, T. Pasanen and J. Teuhola. Practical in-place mergesort. *Nordic Journal of Computing* 3:27-40, 1996.
10. D.T. Lee. On finding the convex hull of a simple polygon. *International Journal of Computing & Information Sciences* 12(2):87-98, 1983.
11. A. Melkman. On-line construction of the convex hull of a simple polygon. *Information Processing Letters* 25:11-12, 1987.
12. J. I. Munro, V. Raman, and J. S. Salowe. Stable in Situ Sorting and Minimum Data Movement. *BIT* 30(2): 220-234, 1990.