SPACE-EFFICIENT PLANAR CONVEX HULL ALGORITHMS*

Hervé Brönnimann† John Iacono† Jyrki Katajainen‡ Pat Morin§ Jason Morrison§
Godfried Toussaint¶

ABSTRACT. A space-efficient algorithm is one in which the output is given in the same location as the input and only a small amount of additional memory is used by the algorithm. We describe four space-efficient algorithms for computing the convex hull of a planar point set.

1 Introduction

Let $S = \{S[0], \ldots, S[n-1]\}$ be a set of n distinct points in the Euclidean plane. The *convex hull* of S is the minimal convex region that contains every point of S. From this definition, it follows that the convex hull of S is a convex polygon whose vertices are points of S. For convenience, we say that a point p is "on the convex hull of S" if p is a vertex of the convex hull of S.

As early as 1972, Graham [13] gave a convex hull algorithm with $O(n \log n)$ worst-case running time in which all branching is done based on the results of comparisons between quadratic polynomials. Shamos [33] later showed that, in any model of computation where sorting has an $\Omega(n \log n)$ lower bound, every convex hull algorithm must require $\Omega(n \log n)$ time for some inputs. Despite these matching upper and lower bounds, and probably because of the many applications of convex hulls, a number of other planar convex hull algorithms have been published since Graham's algorithm [1, 2, 4, 6, 11, 17, 21, 28, 29, 36].

Of particular note is the "Ultimate(?)" algorithm of Kirkpatrick and Seidel [21] that computes the convex hull of a set of n points in the plane in $O(n \log h)$ time, where h is the number of vertices of the convex hull. (Later, the same result was obtained by Chan using a much simpler algorithm [3].) The same authors show that, on algebraic decision trees of any fixed order, $\Omega(n \log h)$ is a lower bound for computing convex hulls of sets of n points having convex hulls with h vertices.

Because of the importance of planar convex hulls, it is natural to try and improve the running time and storage requirements of planar convex hull algorithms. In this paper, we focus on reducing the intermediate storage used in the computation of planar convex hulls. In particular, we describe *in-place* and *in situ* algorithms for computing convex hulls. These algorithms take the input points as an array

*This research was partly funded by the National Science Foundation, the Natural Sciences and Engineering Research Council of Canada and the Danish Natural Science Research Council under contract 9801749 (project Performance Engineering).

†CIS, Polytechnic University, Six Metrotech, Brooklyn, New York, 11201. {hbr,jiacono}@poly.edu

‡Department of Computing, University of Copenhagen, jyrki@diku.dk

§School of Computer Science, Carleton University, 1125 Colonel By Dr., Ottawa, Ontario, CANADA, K1S 5B6. {morin,morrison}@cs.carleton.ca

¶School of Computer Science, McGill University, godfried@cgm.cs.mcgill.ca

and output the vertices of the convex hull in clockwise order, in the same array. During the execution of the algorithm, additional working storage is kept to a minimum. In the case of in-place algorithms, the extra storage is kept in $O(1)$ while *in situ* algorithms allow an extra memory of size $O(\log n)$. After execution of the algorithm, the array contains exactly the same points, but in a different order. For convenience, we refer to in-place and *in situ* algorithms as *space-efficient*.

Space-efficient algorithms have several practical advantages over traditional algorithms. Primarily, space-efficient algorithms allow for the processing of larger data sets. Any algorithm that uses separate input and output arrays will, by necessity, require enough memory to store $2n$ points. In contrast, a space-efficient algorithm needs only enough memory to store n points plus $O(\log n)$ or $O(1)$ working space. Related to this is the fact that space-efficient algorithms usually exhibit greater locality of reference, which makes them very practical for implementation on modern computer architectures with memory hierarchies. A final advantage of space-efficient algorithms, especially in mission critical applications, is that they are less prone to failure since they do not require the allocation of large amounts of memory that may not be available at run time.

We describe four space-efficient planar convex hull algorithms. The first is in-place, uses Graham's scan in combination with an in-place sorting algorithm, and runs in $O(n \log n)$ time. The second and third algorithms run in $O(n \log h)$ time, are *in situ* and are based on algorithms of Chan *et al.* [4] and Kirkpatrick and Seidel [21], respectively. The fourth ("More Ultimate?") algorithm is based on an algorithm of Chan [3], runs in $O(n \log h)$ time and is in-place. The first two algorithms are simple, implementable, and efficient in practice. To justify this claim, we have implemented both algorithms and made the source code freely available [25].

To the best of our knowledge, this paper is the first to study the problem of computing convex hulls using space-efficient algorithms. This seems surprising, given the close relation between planar convex hulls and sorting, and the large body of literature on space-efficient sorting and merging algorithms [7–10, 12, 15, 16, 18–20, 23, 26, 31, 34, 35, 37]. The main reason for this is probably that the scan portion of Graham's original algorithm [13] is inherently in-place, so in-place sorting algorithms already provide an $O(n \log n)$ time in-place convex hull algorithm.

The remainder of the paper is organized as follows: In Sections 2, 3 and 4 the four algorithms are described, and in Section 5 the results are summarized and some open problems are presented.

2 An $O(n \log n)$ Time Algorithm

In this section, we present a simple in-place implementation of Graham's convex hull algorithm [13] or, more precisely, Andrew's modification of Graham's algorithm [1]. The algorithm requires the use of an in-place sorting algorithm. This can be any efficient in-place sorting algorithm (see, e.g., [19, 37]), so we refer to this algorithm simply as InPlace-Sort.

Because this is probably the most practically relevant algorithm given in this paper, we begin by describing the most conceptually simple version of the algorithm, and then describe a slightly more involved version that improves the constants in the running time.

2.1 The Basic Algorithm

Let S be a set of $n > 1$ points and Let l be the line through the bottommost-leftmost point of S and the topmost-rightmost point of S. The *upper convex hull* of S is the convex hull of all points in S that are above, or on, l and the lower convex hull of S is the convex hull of all points of S that are below, or on, l. It is well-known that the convex hull of a point set is the union of its upper and lower convex hulls (cf. [30]).

Graham's scan computes the upper (or lower) hull of an x-monotone chain incrementally, storing the partially computed hull on a stack. The addition of each new point involves removing zero or more points from the top of the stack and then pushing the new point onto the top of the stack.

The following pseudo-code uses the InPlace-Sort algorithm and Graham's scan to compute the upper or lower hull of the point set S. The parameter d is used to determine whether the upper or lower hull is being computed. If $d = 1$, then InPlace-Sort sorts the points by increasing order of lexicographic (x, y)-values and the upper hull is computed. If $d = -1$, then InPlace-Sort sorts the points by decreasing order and the lower hull is computed. The value of h corresponds to the number of elements on the stack.

In the following, and in all remaining pseudo-code, $S = S[0], \ldots, S[n-1]$ is an array containing the input points.

```
Graham-InPlace-Scan(S, n, d)
1: InPlace-Sort(S, n, d)
2: h ← 1
3: for i ← 1 ... n − 1 do
4:    while h ≥ 2 and not right_turn(S[h − 2], S[h − 1], S[i]) do
5:       h ← h − 1 { pop top element from the stack }
6:    swap S[i] ↔ S[h]
7:    h ← h + 1
8: return h
```

It is not hard to verify that when the algorithm returns in Line 8, the elements of S that appear on the upper (or lower) convex hull are stored in $S[0], \ldots, S[h-1]$. In the case of an upper hull computation ($d = 1$), the hull vertices are sorted left-to-right (clockwise), while in the case of a lower hull computation ($d = -1$), the hull vertices are sorted right-to-left (also clockwise).

To compute the convex hull of the point set S, we proceed as follows (refer to Fig. 1): First we make a call to Graham-InPlace-Scan to compute the vertices of the upper hull of S and store them in clockwise order at positions $S[0], \ldots, S[h-1]$. It follows that $S[0]$ is the bottommost-leftmost point of S and that $S[h-1]$ is the topmost-rightmost point of S. We then use $h-1$ swaps to bring $S[0]$ to position $S[h-1]$ while keeping the relative ordering of $S[1], \ldots S[h-1]$ unchanged. Finally, we make a call to Graham-InPlace-Scan to compute the lower convex hull of $S[h-2], \ldots, S[n-1]$ (which is also the lower convex hull of S). This stores the vertices of the lower convex hull in $S[h-2], \ldots, S[h+h'-2]$ in clockwise order. The end result is that the convex hull of S is stored in $S[0], \ldots, S[h+h'-2]$ in clockwise order.

The following pseudo-code gives a more precise description of the algorithm. We use the C

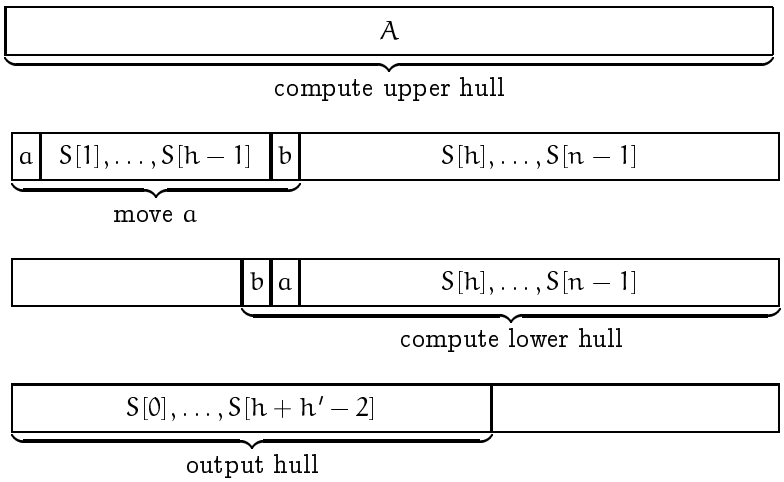

Figure 1: The execution of the GRAHAM-INPLACE-HULL algorithm.

pointer notation $S + i$ to denote (the starting position of) the of array $S[i], \ldots, S[n-1]$.

GRAHAM-INPLACE-HULL(S, n)
1: $h \leftarrow$ GRAHAM-INPLACE-SCAN$(S, n, 1)$
2: **for** $i \leftarrow 0 \ldots h-2$ **do**
3: **swap** $S[i] \leftrightarrow S[i+1]$
4: $h' \leftarrow$ GRAHAM-INPLACE-SCAN$(S + h - 2, n - h + 2, -1)$
5: **return** $h + h' - 2$

Each call to GRAHAM-INPLACE-SCAN executes in $O(n \log n)$ time, and the loop in lines 2–3 takes $O(h)$ time. Therefore, the total running time of the algorithm is $O(n \log n)$. The amount of extra storage used by INPLACE-SORT is $O(1)$, as is the storage used by both our procedures.

Theorem 1 *Algorithm* GRAHAM-INPLACE-HULL *computes the convex hull of a set of* n *points in* $O(n \log n)$ *time using* $O(1)$ *additional memory.*

The algorithm of Section 4 makes use of GRAHAM-INPLACE-SCAN. However, the algorithm requires that the resulting convex hull be stored in clockwise order beginning with the leftmost vertex. We note that this output format can easily be achieved in an $O(n)$ time postprocessing step.

2.2 The Optimized Algorithm

The constants in the running time of GRAHAM-INPLACE-HULL can be improved by first finding the extreme points a and b and using these points to partition the array into two parts, one that contains vertices that can only appear on the upper hull and one that contains vertices that can only appear on the lower hull. Fig. 2 gives a graphical description of this. In this way, each point (except a and b) takes part in only one call to GRAHAM-INPLACE-SCAN.

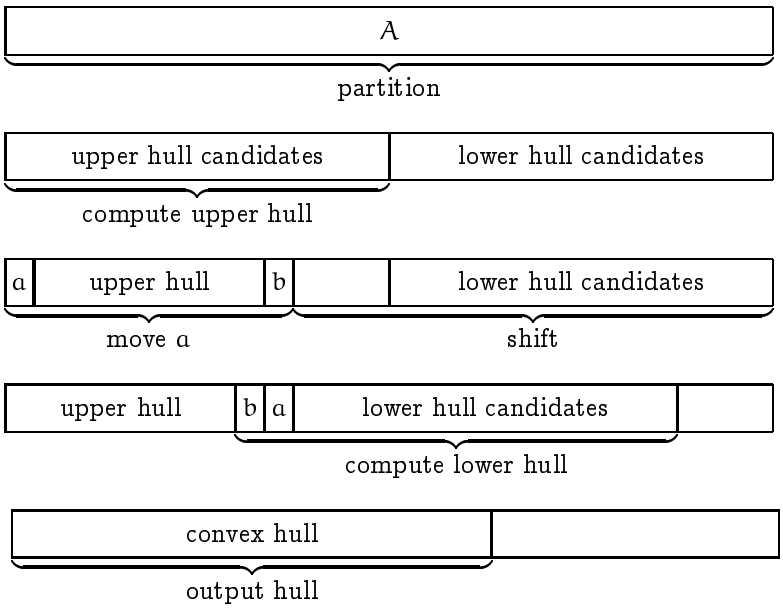

Figure 2: A faster implementation of Graham-InPlace-Hull.

To further reduce the constants in the algorithm, one can implement InPlace-Sort with the in-place merge-sort algorithm of Katajainen *et al.* [19]. This algorithm requires only $n \log_2 n + O(n)$ comparisons and $\frac{3}{2} n \log_2 n + O(n)$ swaps to sort n elements. Since Graham's scan performs only $2n - h$ right-turn tests when computing the upper hull of n points having h points on the upper hull, the resulting algorithm performs at most $3n - h$ right-turn tests (the extra n comes from the initial partitioning step). We call this algorithm Opt-Graham-InPlace-Hull.

Theorem 2 Opt-Graham-InPlace-Hull *computes the convex hull of* n *points in* $O(n \log n)$ *time using at most* $3n - h$ *right turn tests,* $\frac{3}{2} n \log_2 n + O(n)$ *swaps,* $n \log_2 n + O(n)$ *lexicographic comparisons and* $O(1)$ *additional memory, where* h *is the number of vertices of the convex hull.*

Finally, we note that if the array A is already sorted in lexicographic order then no lexicographic comparisons are necessary. One can use an in-place stable partitioning algorithm to partition A into the set of upper hull candidates and the set of lower hull candidates while preserving the sorted order within each set. There exists such a stable partitioning algorithm that runs in $O(n)$ time and performs $O(n)$ comparisons [18]. (In this context, each comparison is actually a right turn test.) Since the algorithm is stable, the original sorted order of the input is preserved and no additional sorting step is necessary. We call the resulting algorithm Sorted-Graham-InPlace-Hull.

Theorem 3 Sorted-Graham-InPlace-Hull *computes the convex hull of* n *points given in lexicographic order in* $O(n)$ *time using* $O(n)$ *right turn tests,* $O(n)$ *swaps, no lexicographic comparisons and* $O(1)$ *additional memory.*

A final option for an in-place implementation of Graham's is to sort the points in S radially about some point p in the interior of the convex hull. Once this is done, one call to Graham-InPlace-

SCAN will compute the entire convex hull. Unfortunately, this method requires $O(n \log n)$ right turn tests, so it will likely be slower than methods that use only $O(n)$ right turn tests.

3 Two $O(n \log h)$ Time In-Situ Algorithms

In this section, we show how to compute the upper (and symmetrically, lower) hull of S in $O(n \log h)$ time *in situ*, where h is the number of points of S that are on the upper (respectively, lower) hull of S. We discuss two algorithms, due to Kirkpatrick and Seidel [21], and Chan, Snoeyink and Yap [4]. Both algorithms are recursive and partition the problem into two roughly equal-sized subproblems. They use different strategies for this purpose, however.

3.1 Chan, Snoeyink and Yap's Algorithm

We first show how to transform the $O(n \log h)$ time algorithm of Chan *et al.* into an *in situ* algorithm. The algorithm begins by arbitrarily grouping the elements of S into $\lfloor n/2 \rfloor$ pairs. From these pairs, the pair with median slope s is found using a linear time median-finding algorithm.[1] We then find a point $p \in S$ such that the line through p with slope s has all points of S below it. Naturally, p is a vertex of the convex hull of S.

Let $q.x$ denote the x coordinate of the point q and let $\pi(i)$ denote the index of the element that is paired with $S[i]$. We now use p, and our grouping to partition the elements of S into three groups S^0, S^1, and S^2 as follows (see Fig. 3):

$$S[i] \in \begin{cases} S^0 & \text{if } S[i].x \leq p.x \text{ and } (S[\pi(i)], p) \text{ is not above } S[i], \\ S^1 & \text{if } S[i].x > p.x \text{ and } (S[\pi(i)], p) \text{ is not above } S[i], \text{ and} \\ S^2 & \text{otherwise.} \end{cases}$$

where (a, b) denotes the line segment with endpoints a and b. The algorithm then recursively computes the upper hull of $S^0 \cup \{p\}$ and $S^1 \cup \{p\}$ and outputs the concatenation of the two. For a discussion of correctness and a proof that this algorithm runs in $O(n \log h)$ time, see the original paper [4].

Now we turn to the problem of making this an *in situ* algorithm. The choice of median slope s ensures that $S^0 \leq 3n/4$ and $S^1 \leq 3n/4$, so the algorithm uses only $O(\log n)$ levels of recursion. Our strategy is to implement each level using $O(1)$ local variables and one call to a median-finding routine that uses $O(\log n)$ additional memory.

For simplicity, assume n is odd. The case when n is even is easily handled by processing an extra unpaired element after all the paired elements have been processed. To pair off elements, we pair consecutive elements of S, so that $\pi(i) = i + 1$ if i is even or $\pi(i) = i - 1$ if i is odd. Several *in situ* linear time median-finding algorithms exist (see, e.g., Horowitz *et al.* [14, Section 3.6] or Lai and Wood [22]) that can be used to find the pair $(S[i], S[i+1])$ with median slope.

The tricky part of the implementation is the partitioning of S into sets S^0, S^1 and S^2. The difficulty lies in the fact that the elements are grouped into pairs, but the two elements of the same pair

[1] Bhattacharya and Sen [2] and Wenger [36] have both noted that median-finding can be replaced by choosing a random pair of elements. The expected running time of the resulting algorithm is $O(n \log h)$.

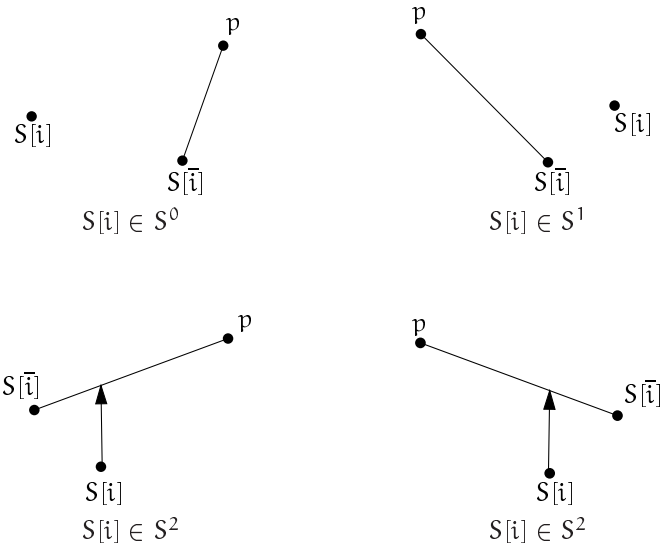

Figure 3: Partitioning S into S^0, S^1 and S^2.

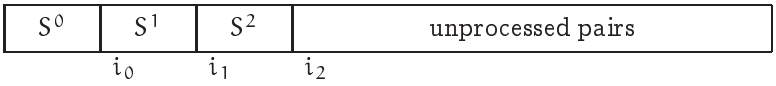

Figure 4: Paritioning into sets S^0, S^1 and S^2.

may belong to different sets S^i and S^j. To do this partitioning, we process the pairs from left-to-right and maintain the sets S^0, S^1 and S^2 in the leftmost part of the array (see Fig. 4). More precisely, we maintain three indices i_0, i_1 and i_2, where $i_j - 1$ is the index of the last element in S^j. In this way, i_2 is the index of the first element in the next unprocessed pair. At each step, we examine the next unprocessed pair, classify each of the two points as belonging to S^0, S^1 or S^2 and add them to the appropriate sets. While adding the points to these sets, we may have to shift each of the S_i by up to two locations. However, we are not required to preserve the order within each set S^i, so this shifting is easily done in $O(1)$ time by moving at most two of the leftmost elements in each set.

Fig. 5 recaps the algorithm for computing the upper hull of S. First the algorithm partitions S into the sets S^0, S^1 and S^2. It then recurses on the set S^0. After the recursive call, the convex hull of S^0 is stored at the beginning of the array S, and the last element of this hull is the point p that was used for partitioning. The algorithm then shifts S^1 leftward so that it is adjacent to p and recurses on $S^1 \cup \{p\}$. The end result is the upper hull of S being stored consecutively and in clockwise order at the beginning of the array S. Using the technique from Section 2 (Figures 1 and 2), this upper hull algorithm can be made into a convex hull algorithm with the same running time and memory requirements.

Theorem 4 *Algorithm* CSY-InSitu-Hull *computes the convex hull of* n *points in* $O(n \log h)$ *time using* $O(\log n)$ *additional storage, where* h *is the number of vertices of the convex hull.*

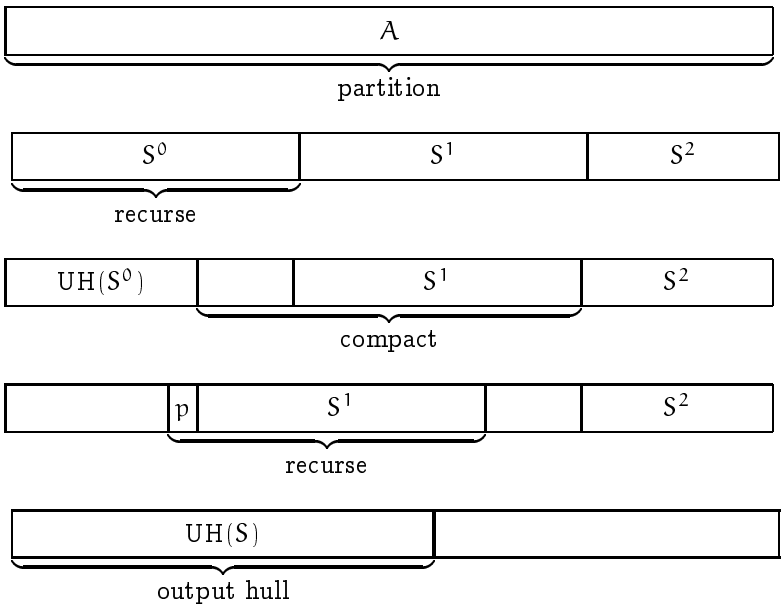

Figure 5: Overview of the CSY-InSitu-Hull algorithm.

3.2 Kirkpatrick and Seidel's Algorithm

The previous algorithm solves the partitioning problem by finding a point p on the convex hull that leaves roughly the same number of vertices on each side. Kirkpatrick and Seidel's original solution to the partitioning problem is to first find an *edge* of the upper hull (the *upper bridge*) that leaves approximately the same number of points on each side.

Suppose that we can find such an edge pq with $p.x < q.x$, such that S^0 consists of the points left of p, S^1 the points right of q, and S^2 the points below pq, and furthermore such that $|S^0| \leq 3n/4$ and $|S^1| \leq 3n/4$. The algorithm recursively computes the upper hulls of $S^0 \cup \{p\}$ and $S^1 \cup \{q\}$, and outputs the concatenation of the two, in $O(n \log h)$ total time. Clearly, if pq is an edge of the convex hull, the result is the upper hull of S. For a proof of the running time, see the original paper [21].

Unlike the previous algorithm, partitioning S in-place into S^0, S^1 and S^2 once p and q are known is trivial, since it is not necessary to maintain a pairing of the edges. Since $|S^0| \leq 3n/4$ and $|S^1| \leq 3n/4$, there are $O(\log n)$ levels of recursion. Therefore, if we can find the upper bridge in linear time in-place, the algorithm will thus be performed *in situ*.

The **upper bridge** problem asks: *Given two sets S^0 and S^1 of points separated by a vertical line $y = x_0$, which are the two endpoints $p \in S^0$ and $q \in S^1$ such that the edge pq is on the upper hull of $S^0 \cup S^1$?* This problem is dual to the **separated 2D linear programming** problem which can be phrased as: *Given two sets L^0 and L^1 of lines with positive and negative slopes respectively, compute the point with smallest y-coordinate that is above all the lines.* This linear program is always feasible and the solution is always the intersection of a pair of lines of opposite slopes.

Denoting the point of coordinates x and y by $[x, y]$, and the line of equation $ax + by + c = 0$ by

$[a, b, c]$, the duality given by $\varphi([x, y]) = [x_0 - x, -1, y - x_0(x_0 - x)]$ and $\varphi([a, b, c]) = (x_0 + \frac{a}{b}, -\frac{c+ax_0}{b})$ has the property that if p is below l, then $\varphi(l)$ is above $\varphi(p)$. Moreover, p is to the left (resp. right) of $y = x_0$ if and only if $\varphi(p)$ has positive (resp. negative) slope. In turn, this implies that the solution to the separated 2D linear programming problem given by $L = \varphi(S)$ is dual to the solution of the upper bridge problem. This is the intuition behind the original algorithm [21].

Note that the duality does not really have to be computed: the 2D linear programming problem can be solved directly with the points of S, only the geometric predicates involving the points are transformed into predicates on lines via the transformation φ. Thus if we can answer the 2D linear programming in-place, we can also answer the upper bridge problem in-place.

As in the original algorithm, we first compute the median abscissa x_0 of S *in situ* and partition S into two roughly equal-sized subsets around x_0. This enforces that $|S^0| \leq 3n/4$ and $|S^1| \leq 3n/4$.

There is an algorithm due to Seidel [32] which solves the 2D linear programming problem in expected linear time and is very simple. It assumes that the order of the lines is random (we could always enforce this by shuffling the set S randomly in linear time prior to each linear programming query). Upon close examination, the algorithm does not need to reorder the input and in fact works in-place, maintaining only two indices to scan both sets of lines, and two indices to remember the two lines making up the current optimal solution.

Megiddo [24] gave a worst-case linear-time algorithm. We adapt this algorithm to run in-place, and explain it for lines in the dual setting. Megiddo's algorithm assumes that there are at least 8 lines, otherwise a brute force method can be used. The lines in L are paired up and ordered by slope within each pair: in the in-place implementation, $L[i]$ is paired with $L[\pi(i)]$. Again using *in situ* median-finding, the pair whose point of intersection has median abscissa x_0 can be found in linear time (and those pairs intersecting to the left of x_0 are placed in the first half, while the pairs intersecting to the right of x_0 in the second half). We only have to take care that when exchanging two pairs, each line in the first pair is exchanged with the corresponding line in the second pair. Next, the line $l \in L$ which intersects the vertical line $x = x_0$ at the highest ordinate is found. Recall that the solution to the linear programming problem is the lowest point which is above all lines. Therefore, if the slope of l is negative, then the solution to the linear programming problem is to the right of x_0, otherwise the solution is to the left of x_0.

In the first case, we scan the pairs in the first half: the line of smallest slope in each pair of the first half can be discarded since to the right of x_0 it is always below its paired line and hence cannot define the solution. In the second case, the line of largest slope in each pair of the second half can be discarded. Discarded lines can be put at the end of the array by swapping with the last as yet undiscarded line. This works in the second case as well if the pairs in the second half are examined in the *reverse order* (beginning at the end and moving towards the middle of the array) since the discarded zone grows twice as slowly as the lines in the examined pairs.

The choice of medians ensure that $n/4$ lines have been discarded in any case. At the end of this process, we are left with a set L' of at most $\lceil 3n/4 \rceil$ lines, such that the solution to the original problem is defined by two of these lines. Care must be taken to include the last line in the $3n/4$ if the original number of lines was odd. Hence, the solution of the linear programming problem on L' is the same as that of L. The algorithm is run again on L' instead of L, until the size of L' falls below 8 at which point a brute-force method is used. (In practice, Seidel's algorithm can be used under a certain fixed size determined during the fine-tuning.)

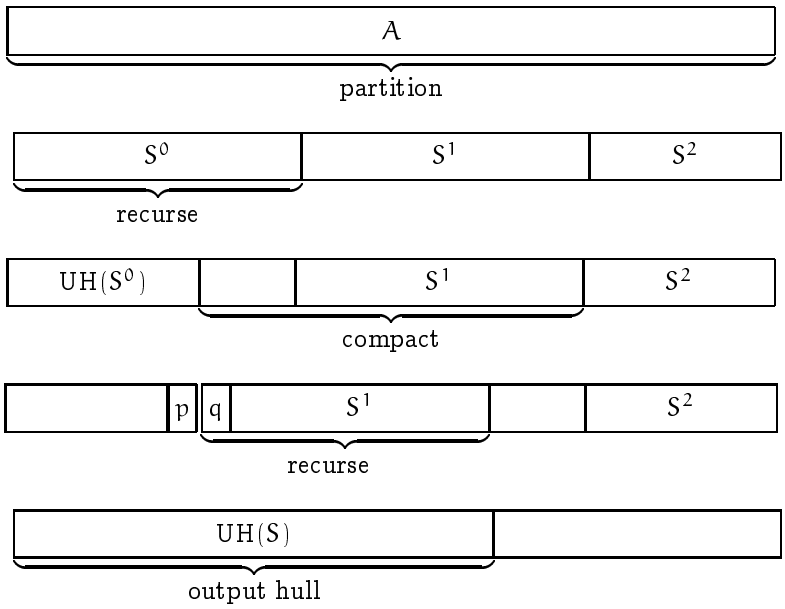

Figure 6: Overview of the KS-InSitu-Hull algorithm.

Theorem 5 *The above algorithm,* Megiddo-Inplace-LP-2D, *solves a separated 2D linear programming problem in-place in linear time.*

Figure 6 recaps the algorithm for computing the upper hull of S. First the algorithm computes the median abscissa x_0 of S, and the upper bridge pq by using the dual of the algorithm Megiddo-InPlace-LP-2D. The bridge is used to partition S into the sets S^0, S^1 and S^2. The algorithm then recurses on the set S^0. After the recursive call, the convex hull of S^0 is stored at the beginning of the array S, and the last element of this hull is the first endpoint p of the upper bridge. The algorithm then shifts S^1 leftward so that it is adjacent to pq and recurses on $S^1 \cup \{q\}$. The end result is the upper hull of S being stored consecutively and in clockwise order at the beginning of the array S.

Theorem 6 *The above algorithm,* KS-InSitu-Hull, *computes the convex hull of* S *in* $O(n \log h)$ *time using* $O(\log n)$ *additional storage, where* h *is the number of vertices of the convex hull.*

4 An $O(n \log h)$ Time In-Place Algorithm

Next, we give an $O(n \log h)$ time in-place planar convex hull algorithm. Our algorithm is a modification of Chan's $O(n \log h)$ time algorithm, which is essentially a speedup of Jarvis' March [17]. We begin with a review of Chan's algorithm, and thereafter we describe the modifications needed for making it in-place.

Chan's algorithm runs in rounds. During i^{th} round the algorithm finds the first $g_i = 2^{2^i}$ points on the convex hull. Once $g_i \geq h$ the rounds end as the algorithm detects that it has found all points

on the convex hull. During round i, the algorithm partitions the input points into n/g_i groups of size g_i and computes the convex hull of each group. The vertices on the convex hull are output in clockwise order beginning with the leftmost vertex. Each successive vertex is obtained by finding tangents from the previous vertex to each of the n/g_i convex hulls. The next vertex is determined, as in Jarvis' March, by choosing the vertex having largest polar angle with respect to the previously found vertex as origin. In the case where the largest polar angle is not unique, ties are broken by taking the farthest vertex from the previously found vertex.

Finding a tangent to an individual convex hull can be done in $O(\log g_i)$ time if the vertices of the convex hull are stored in an array in clockwise order [5, 27, 30]. There are n/g_i tangent finding operations per iteration and g_i iterations in round i. Therefore, round i takes $O(n \log g_i) = O(n2^i)$ time. Since there are at most $\lceil \log \log h \rceil$ rounds, the total cost of Chan's algorithm is $\sum_{i=1}^{\lceil \log \log h \rceil} O(n2^i) = O(n \log h)$.

Next we show how to implement each round using only $O(1)$ additional storage. Assume for the sake of simplicity that n is a multiple of g_i. For the grouping step, we build n/g_i groups of size g_i by taking groups of consecutive elements in S and computing their convex hulls using Graham-InPlace-Hull. Two questions now arise: (1) Once we start the tangent-finding steps, where do we put the convex hull vertices as we find them? (2) In order to find a tangent from a point to a group in $O(\log g_i)$ time we need to know the size of the convex hull of the group. How can we keep track of all these sizes using only $O(1)$ extra memory?

To answer the first question, we store convex hull vertices at the beginning of the array S in the order that we find them. That is, when we find the k^{th} vertex on the convex hull, we swap it with $S[k-1]$. At this point, the convex hull of the first group and the group containing the newly found convex hull vertex have changed. Therefore, we recompute both of these convex hulls at a cost of $O(g_i \log g_i)$.

To keep track of the size of the convex hull of each group without storing the size explicity we use a reordering trick. Let $G[0], \ldots, G[g_i - 1]$ denote the elements of a group G and let $<$ denote lexicographic comparison of (x, y) values. We say that the *sign* of $G[j]$ is $+$ if $G[j] < G[j+1]$, and $-$ otherwise. If the convex hull of G contains h vertices, then it follows that the first elements $G[0], \ldots, G[h-2]$ have signs that form a sequence of 1 or more $+$'s followed by 0 or more $-$'s. Furthermore, the elements $G[h], \ldots, G[g_i - 1]$ can be reordered so that the remainder of the signs form an alternating sequence.

To test if a point $G[i]$ is on the convex hull of G for $i = 0, 1, 2$ we simply observe that all three such vertices must be on the convex hull of G unless they are collinear, in which case only $G[0]$ and $G[1]$ are on the convex hull of G.

To test if a point $G[i]$, $i \geq 3$ is on the convex hull of G, we examine the sequence of signs formed by $G[i]$, $G[i-1]$, $G[i-2]$, and $G[i-3]$. If this sequence does not contain two consecutive $+$'s or two consecutive $-$'s then a simple case analysis will convince the reader that $G[i]$ is not on the convex hull of G. Otherwise, at least one of $G[i]$, $G[i-1]$, $G[i-2]$, or $G[i-3]$ is on the convex hull of G. To determine which of these vertices is the last such vertex, we perform tests of the form right_turn$(G[j], G[j+1], G[0])$, for $j = i-3, \ldots, i-1$ (see Fig. 7). The first value for which this test returns false is the index j of the final element of the convex hull of G. If no such test returns false then i is on the convex hull of G.

We have now provided all the tools for an in-place implementation of Chan's algorithm. Except for the cost of recomputing convex hulls of groups after modifying them, the running time of this implementation is asymptotically the same as that of the original algorithm. Therefore, we need

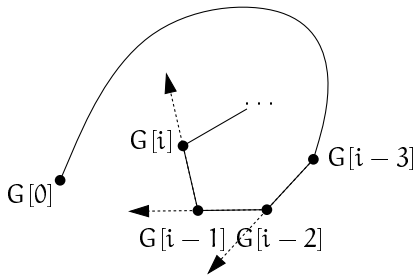

Figure 7: The first vertex to fail the right_turn test is the last vertex on the convex hull of G.

only bound this extra cost. During one step of round i, we find one convex hull vertex and recompute the convex hull of two groups. The cost of recomputing these convex hulls is $O(g_i \log g_i)$ and there are at most g_i steps in round i. Therefore, the total cost of recomputing convex hull vertices at round i is $O({g_i}^2 \log g_i) \subseteq O(n)$ for all $g_i \leq (n/\log n)^{1/2}$. Hence, the total cost of round i is $O(g_i^2 \log g_i + n \log g_i) \subseteq O(n \log g_i)$ for any $g_i < (n/\log n)^{1/2}$. Since we can abort the algorithm when $g_i \geq (n/\log n)^{1/2}$ and use Graham-InPlace-Hull, the overall running time of the algorithm is again $O(n \log h)$.

Theorem 7 *The above algorithm,* Chan-InPlace-Hull*, computes the convex hull of* n *points in* $O(n \log h)$ *time using* $O(1)$ *additional storage, where* h *is the number of vertices of the convex hull.*

The constants in Chan-InPlace-Hull can be improved using the following trick that is mentioned by Chan [3]. When round i terminates without finding the entire convex hull, the g_i convex hull points that were computed should not be discarded. Instead, the grouping in round $i+1$ is done on the remaining $n - g_i$ points, thus eliminating the need to recompute the first g_i convex hull vertices. This optimization works perfectly when applied to Chan-InPlace-Hull since the first g_i convex hull points are already stored at locations $S[0], \ldots, S[g_i - 1]$.

5 Conclusions

We have given four space-efficient algorithms for computing the convex hull of a planar point set. The first algorithm is in-place and runs in $O(n \log n)$ time. The second and third algorithms are *in situ* and run in $O(n \log h)$ time. The fourth algorithm is in-place and and runs in $O(n \log h)$ time.

The first two algorithms are reasonably simple and implementable, and their running times compare favourably with those of convex hull algorithms that use additional storage. In order to facilitate comparisons with other convex hull implementations, our source code is available for download [25].

Although we have assumed throughout the paper that all of the input points are distinct, the algorithms in this paper can be modified to handle the case in which the input is a multiset. These modifications are technical, but relatively straightforward. In particular, care must be taken with respect to "side of line" tests and the size encoding scheme used in Section 4 needs to make use of a third symbol, 0, used for consecutive identical elements.

The ideas presented in this paper also apply to other problems. The *maximal elements* problem is that of determining all elements $S[i]$ such that $S[j].x \leq S[i].x$ and $S[j].y \leq S[i].y$ for all $0 \leq j < n$. An algorithm almost identical to Graham's scan can be used to solve the maximal elements problems in $O(n \log n)$ time, and this can easily be implemented in-place. Furthermore, an in-place algorithm almost identical to that in Section 4 can be used to solve the maximal elements problem in $O(n \log h)$ time, where h is the number of maximal elements.

The question of *in situ* and in-place algorithms for maximal elements and convex hulls in dimensions $d \geq 3$ is still open. In order for this question to make sense, we ask only that the algorithm identify which input points are maximal or on the convex hull. Testing whether a given point is maximal can be done in $O(dn)$ time using the definition of maximality. Testing whether a single point is on the convex hull is a $d-1$ dimensional linear programming problem that can be solved in-place in $O(d!n)$ expected time using Seidel's algorithm [32]. Thus, the maximal elements problem can be solved in $O(dn^2)$ time and the convex hull problem can be solved in $O(d!n^2)$ time using in-place algorithms. Are there algorithms with reduced dependence on n?

More generally, one might ask what other computational geometry problems admit space-efficient algorithms. Some problems that immediately come to mind are those of computing k-piercings of sets, finding maximum cliques in intersection graphs, computing largest empty disks, and finding ham-sandwich cuts.

Acknowledgements

The authors are grateful to an anonymous referee for making an observation about the algorithm in Section 3.1 that allowed us to greatly simplify the partitioning step of the algorithmxs.

References

[1] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979. Corrigendum, *Information Processing Letters*, 10:168, 1980.

[2] B. K. Bhattacharya and S. Sen. On a simple, practical, optimal, output-sensitive randomized planar convex hull algorithm. *Journal of Algorithms*, 25(1):177–193, 1997.

[3] T. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996.

[4] T. Chan, J. Snoeyink, and C. K. Yap. Primal dividing and dual pruning: Output-sensitive construction of four-dimensional polytopes and three-dimensional Voronoi diagrams. *Discrete & Computational Geometry*, 18:433–454, 1997.

[5] B. Chazelle and D. P. Dobkin. Intersection of convex objects in 2 and 3 dimensions. *Journal of the ACM*, 34:1–27, 1987.

[6] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(1):387–421, 1988.

[7] E. W. Dijkstra. Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, 1(3):223–233, 1982.

[8] E. W. Dijkstra and A. J. M. van Gasteren. An introduction to three algorithms for sorting in situ. *Information Processing Letters*, 15(3):129–134, 1982.

[9] S. Dvořak and B. Ďurian. Stable linear time sublinear space merging. *The Computer Journal*, 30(4):372–375, 1987.

[10] S. Dvořak and B. Ďurian. Unstable linear time O(1) space merging. *The Computer Journal*, 31(3):279–282, 1988.

[11] W. Eddy. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, 3(4):398–403, 1977.

[12] R. W. Floyd. Algorithm 245, Treesort 3. *Communications of the ACM*, 7:401, 1964.

[13] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[14] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998.

[15] B.-C. Huang and M. A. Langston. Practical in-place merging. *Communications of the ACM*, 31(3):348–352, 1988.

[16] B.-C. Huang and M. A. Langston. Fast stable merging and sorting in constant extra space. *The Computer Journal*, 35(6):643–650, 1992.

[17] A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2:18–21, 1973.

[18] J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32(4):580–585, 1992.

[19] J. Katajainen, T. Pasanen, and J. Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3:27–40, 1996.

[20] J. Katajainen and T. A. Pasanen. In-place sorting with fewer moves. *Information Processing Letters*, 70(1):31–37, 1999.

[21] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986.

[22] T. W. Lai and D. Wood. Implicit selection. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 14–23. Springer-Verlag, 1988.

[23] H. Mannila and E. Ukkonen. A simple linear-time algorithm for in situ merging. *Information Processing Letters*, 18(4):203–208, 1984.

[24] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31(1):114–127, 1984.

[25] P. Morin. insitu.tgz. Available online at `http://www.cs.carleton.ca/~morin/`, 2002.

[26] J. I. Munro, V. Raman, and J. S. Salowe. Stable in situ sorting and minimum data movement. *BIT*, 30(2):220–234, 1990.

[27] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[28] F. P. Preparata. An optimal real time algorithm for planar convex hulls. *Communications of the ACM*, 22:402–405, 1979.

[29] F. P. Preparata and S. J. Hong. Convex hulls of finite point sets in two and three dimensions. *Communications of the ACM*, 2(20):87–93, 1977.

[30] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[31] J. Salowe and W. Steiger. Simplified stable merging tasks. *Journal of Algorithms*, 8(4):557–571, 1987.

[32] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6:423–434, 1991.

[33] M. I. Shamos. *Computational Geometry*. PhD thesis, Yale University, 1978.

[34] H. W. Six and L. Wegner. Sorting a random access file in situ. *The Computer Journal*, 27(3):270–275, 1984.

[35] A. Symvonis. Optimal stable merging. *The Computer Journal*, 38(8):681–690, 1995.

[36] R. Wenger. Randomized quick hull. *Algorithmica*, 17:322–329, 1997.

[37] J. W. J. Williams. Algorithm 232, Heapsort. *Communications of the ACM*, 7:347–348, 1964.