On the Space-Efficiency of the "Ultimate Planar Convex Hull Algorithm"

Jan Vahrenhold*

Abstract

The output-sensitive "ultimate planar convex hull algorithm" of Kirkpatrick and Seidel [16] recently has been shown by Afshani *et al.* [1] to be *instance-optimal*. We revisit this algorithm with a focus on space-efficiency and prove that it can be implemented as an in-place algorithm, i.e., using $\mathcal{O}(1)$ working space.

1 Introduction

How much working space does an algorithm require? This question may be asked on its own accord, but there are important practical implications as well. In so-called *resource-constrained* systems, at least one of the resources needed for working on a problem instance is limited by practical constraints and thus scarce relative to the size of the problem instance. Examples include sensors or smartphones where memory and energy are limited, but also workstations where the main memory is scarce relative to terabyte-sized or larger data sets.

In this note, we are working in the space-efficient model of computation, where the primary objective is to analyze the space an algorithm needs in addition to representing the input. An algorithm is said to be *in situ* if it requires $\mathcal{O}(\log N)$ extra words of memory, and it is said to be *in-place* if the extra space requirement is $\mathcal{O}(1)$ words. Since the input elements must not be destroyed or modified, an in-place algorithm can be seen as permuting the input such that it represents the output. As usual, we assume that a word of memory is capable of representing an input element or an index, thus, when measured in bits, these space bounds translate to $\mathcal{O}(\log^2 N)$ bits and $\mathcal{O}(\log N)$ bits.

The study of space-efficient algorithm goes back to fundamental one-dimensional problems such as sorting, merging, and selecting [12, 17, 21]; in the past decade, it has been extended to problems for point sets in two and three dimensions [4, 5, 6, 7, 8, 10, 13, 14, 19, 20]. Recently, also problems for polygons and special classes of graphs have been investigated in the space-efficient model of computation [2, 3].

*Department of Computer Science, Westfälische Wilhelms-Universität Münster, jan.vahrenhold@uni-muenster.de. Part of this work has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis" (http://sfb876.tu-dortmund.de), project A2.

Among the above results, two are of particular importance for our work: Brönnimann *et al.* [8] investigated the space-efficiency of planar convex hull algorithms, i.e., the complexity of computing the H-element convex hull of a set of N points in two dimensions. The authors proved that both the "ultimate" algorithm by Kirkpatrick and Seidel [16] and its simplified variant by Chan, Snoeyink, and Yap [11] can be implemented as *in situ* algorithms, i.e., using $\mathcal{O}(\log N)$ extra words of space, while maintaining an output-sensitive, optimal running time of $\mathcal{O}(N \log H)$. Since, however, not only Graham's optimal algorithm [15] but also Chan's optimal, output-sensitive algorithm [9] was shown be implementable as an in-place algorithm, Brönnimann *et al.* [8] conjectured Chan's algorithm to be the "more ultimate" planar convex hull algorithm. The second relevant result was proved by Bose *et al.* [6]; we developed a framework for simulating a balanced divide-and-conquer scheme using only $\mathcal{O}(1)$ working space. In particular, we showed how such a recursive scheme can be implemented using a constant-sized (in terms of words) stack and, among other results, developed linear-time, in-place algorithms for selecting, un-selecting, and k-selection in sorted arrays.

The optimal, output-sensitive algorithms by Kirkpatrick and Seidel [16] and by Chan, Snoeyink, and Yap [11] recently have been shown by Afshani *et al.* [1] to be *instance-optimal* in the sense that the maximum running time (over all possible permutations of the input) of the algorithm does not differ by more than a constant factor from the minimum of the average running times (again, over all possible permutations of the input) over all algorithms that solve this problem. Afshani *et al.* pointed out that such an instance-optimal algorithm is "immediately also competitive against *randomized* (Las Vegas) algorithms" [1, p. 130]. In this note, we prove:

Theorem 1 *The deterministic, time-optimal, output-sensitive, and instance-optimal planar convex hull algorithm by Kirkpatrick and Seidel can be realized as an in-place algorithm, i.e., using $\mathcal{O}(1)$ working space.*

Since Chan's algorithm is known to be time-optimal, output-sensitive, and implementable as an in-place algorithm but not known to be instance-optimal, the above theorem improves the state-of-the-art of characterizing the "ultimate planar convex hull algorithm" in favor of Kirkpatrick and Seidel's algorithm.

2 The Algorithm by Kirkpatrick and Seidel

The convex hull algorithm by Kirkpatrick and Seidel [16] uses a divide-and-conquer approach where, just like in the quicksort algorithm, the bulk of the work is done prior to recursion. To compute the upper hull of a given point set, the algorithm first finds a "bridge", i.e., a hull edge crossing the median x-coordinate of the point set, removes all points in the slab spanned by the endpoints of the bridge, and recurses on the remaining non-trivial point sets. The lower hull is computed analogously. Assuming that the bridge can be found in linear time, Kirkpatrick and Seidel upper-bound the running time $f(N, H)$ for computing the H-element convex hull of an N-element point set as follows [16, p. 290]:

$$f(N,H) \leq \begin{cases} cn & \text{if } H = 2 \\ cn + \max\limits_{H_\ell + H_r = H} \{f\left(\frac{N}{2}, H_\ell\right) + f\left(\frac{N}{2}, H_r\right)\} & \text{if } H > 2 \end{cases}$$

Here, H_ℓ and H_r denote the number of hull points left and not to the left of the median x-coordinate. Solving this equation yields an upper bound of $\mathcal{O}(N \log H)$.

To determine the bridge (or, rather, its endpoints) in linear time, the authors first determine the median x-coordinate using a k-selection algorithm. They then consider pairs of points and again use a k-selection algorithm to find a pair of points inducing a line with median slope. If this line cannot be shown to induce the bridge, its slope is used to prune away a constant fraction of the points, and the algorithm recurses on the remaining points.[1] Each invocation of this algorithm runs in linear time as long as both k-selection and the pruning procedure can be executed in linear time.

Special care must be taken to handle pairs of points inducing lines with infinite slope (in this case, the pair is excluded from the k-selection algorithm, and the lower point is pruned immediately). Also, to ensure instance-optimality, Afshani *et al.* [1] require that all points strictly below the line through the leftmost and rightmost point of the point set need to be pruned prior to determining the median x-coordinate.

3 Space-Efficient Building Blocks

As we will see, there are three building blocks that need to be made in-place: selecting a subset of the input, finding the k-th element according to some order, and running a divide-and-conquer algorithm. Previously, we gave linear-time algorithms for the first two tasks [6]. For the sake of self-containedness, we present the pseudocode for the selection task:

[1] There is a simplified version of this algorithm using randomized 2D linear programming [8, 18]. It is unclear, however, if instance-optimality can be shown for the resulting convex hull algorithm and, more important, whether such a characterization is meaningful for a randomized algorithm.

Algorithm 1 SUBSETSELECTION(A, b, e, f): selecting a subset from an array $A[b, \ldots, e-1]$ using a $(0,1)$-valued function f that can be evaluated in constant time [6]. If A is sorted, there is a linear-time inverse oblivious of f.

Ensure: $A[b, \ldots, i-1]$ contains all elements of $A[b, \ldots, e-1]$ for which f evaluates to one.

```
1: i ← b, j ← b and m ← b + 1.
2: while i < e and j < e do
3:     while i < e and f(A[i]) = 1 do
4:         i ← i + 1.   ▷ Move i such that f(A[i]) = 0.
5:     j ← max{i + 1, j + 1};
6:     while j < e and f(A[j]) = 0 do
7:         j ← j + 1.   ▷ Move j such that f(A[j]) = 1.
8:     if j < e then
9:         swap A[i] ↔ A[j].
10: Return i.
```

Also, we discussed how to use a bit stack of $\mathcal{O}(\log n)$ bits, i.e., $\mathcal{O}(1)$ words, to implement the following template for the case of an almost perfectly balanced divide-and-conquer, i.e., for the case that the size of the "left" part of the recursion always is a power of two.

Algorithm 2 RECURSIVE(A, b, e): Standard template for recursive divide-and-conquer algorithms [6].

```
1: if e − b ≤ 2^{h_0} (=size of the recursion base) then
2:     BaseCode(A, b, e)          ▷ Solve small instances
3: else
4:     PreCode(A, b, e)
       ▷ Setup Subproblem 1 in A[b, ..., ⌊(b + e)/2⌋ − 1]
5:     Recursive(A, b, ⌊(b + e)/2⌋)          ▷ Recurse left
6:     MidCode(A, b, e)
       ▷ Setup Subproblem 2 in A[⌊(b + e)/2⌋, ..., e − 1]
7:     Recursive(A, ⌊(b + e)/2⌋, e)          ▷ Recurse right
8:     PostCode(A, b, e)
       ▷ Merge Subproblems 1 and 2 in A[b, ..., e − 1]
```

While, in most situations, requiring an almost perfectly balanced partition is not a constraint for divide-and-conquer algorithms, such a partition cannot be guaranteed for Kirkpatrick and Seidel's algorithm which, due to several pruning steps, does not balance the sizes of the subproblems effectively handled in the recursive calls. Thus, the central algorithmic problem we need to address is how to recover the original values of b and e after returning from a call to RECURSIVE, i.e., prior to calling MIDCODE and POSTCODE, using globally no more than $\mathcal{O}(1)$ working space.

4 An Implementation With $\mathcal{O}(1)$ Working Space

In this section, we show how to implement the algorithm by Kirkpatrick and Seidel using $\mathcal{O}(1)$ working space. To

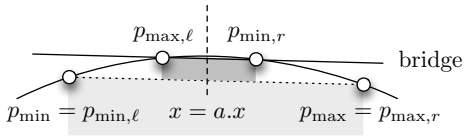

Figure 1: Extremal hull points.

facilitate the exposition, we first describe how to adapt the general divide-and-conquer template (Algorithm 2) to deal with unbalanced recursive calls (Section 4.1). We then show how to represent and combine the output of the recursive calls (Section 4.2). Finally, we present a linear-time, in-place algorithm for finding the bridge needed for the divide-step (Section 4.3).

4.1 Adapting the Divide-and-Conquer Template

Using the terminology of the preceding section, the algorithm by Kirkpatrick and Seidel will be run on an array $A[0, \ldots, n-1]$ where in each recursive step, i.e., for parameters b and e, we first invoke PreCode to do the following (see Figure 1):

1. Identify the extremal hull points $p_{\min}$ and $p_{\max}$.
2. Prune away all points strictly below $\overline{p_{\min}p_{\max}}$, i.e., in the light gray area in Figure 1 (this guarantees instance-optimality [1, Section 3.1]).
3. Determine the point a with median x-coordinate among the remaining points.
4. Compute the bridge, i.e., the segment induced by the maximal[2] hull point $p_{\max,\ell}$ to the left of a and the minimal hull point $p_{\min,r}$ not to the left of a.
5. Prune away all points below the bridge, i.e., in the dark gray area in Figure 1.
6. Adjust the indices b and e such that $A[b, \ldots, e-1]$ contains all unpruned points not to right of $p_{\max,\ell}$.

After returning from the recursive call processing the unpruned points, we need to call MidCode with the *original* values of b and e, i.e., with the same values that PreCode had been called with. Similarly, after returning from the second recursive call, these original values need to be passed to PostCode. Unlike in the standard divide-and-conquer template (Algorithm 2) we cannot simply assume that number of points passed to a recursive call is exactly half the number of points originally passed to the invoking method; in fact, the central point that allows for proving the optimal $\mathcal{O}(N \log H)$ time complexity is that this is *not* the case.

It turns out, however, that three simple invariants allow for reconstructing these values in time $\mathcal{O}(e-b)$:

Invariant (A): Prior to executing and after having executed Recursive(A, b, e), the two vertices of the upper hull that have extremal coordinates in $A[b, \ldots, e-1]$ are stored in $A[b]$ and $A[b+1]$.

Invariant (B): The points passed to Recursive(A, b, e) are exactly the points in $A[0, \ldots, n-1]$ that lie between $A[b]$ and $A[b+1]$ as characterized in Invariant (A).

Invariant (C): After having executed Recursive(A, b, e), the values of b and e have been restored.

Using one linear scan and two swap operations, Invariant (A) can be trivially established prior to invoking Recursive(A, 0, n). Since the points with extremal coordinates in $A[0, \ldots, n-1]$ are also the extremal hull vertices, Invariant (B) holds as well. It is also easy to realize that all invariants can be guaranteed to be maintained using $\mathcal{O}(1)$ time and working space for constant-sized recursion base cases.

Lemma 2 *If Invariant (A) holds prior to invoking* PreCode*, this method can be implemented as a linear-time method with $\mathcal{O}(1)$ working space such that Invariant (A) also holds prior to invoking* Recursive *for the "left" recursion.*

Proof. We consider each step of PreCode (as described above) in turn. Since Invariant (A) holds prior to invoking PreCode, Step 1 (identification of $p_{\min}$ and $p_{\max}$) trivially is a constant-time operation. Step 2 (pruning) can be implemented using SubsetSelection(A, b, e, f) where $f(p) = 0$ holds iff p is strictly below $\overline{A[b]A[b+1]}$. Since this algorithm moves the pruned elements to $A[e', \ldots, e-1]$, the resulting subarray looks as follows.

	$p_{\min}$	$p_{\max}$	unpruned	pruned	
	b			e'	e

Using $\mathcal{O}(1)$ working space (among other things, to keep track of the location of $p_{\min}$ and $p_{\max}$ to eventually restore them to $A[b]$ and $A[b+1]$), we run a linear-time, in-place k-selection algorithm [6, 17] to find the point a with median x-coordinate in $A[b, \ldots, e'-1]$. We then run an in-place version of the bridge-finding algorithm (see Section 4.3) to determine the two hull vertices $p_{\max,\ell}$ and $p_{\min,r}$ defining the bridge.

To prepare for the next recursive call, we move $p_{\max,\ell}$ to $A[b+2]$ and swap the two points in $A[b]$, i.e., $p_{\min} = p_{\min,\ell}$, and $A[b+1]$, i.e., $p_{\max}$. We then increment b by one and run SubsetSelection(A, b, e', f) where $f(p) = 1$ holds iff either $p.x < p_{\max,\ell}.x$ or $p = p_{\max,\ell}$ to prune all points not to the right of $p_{\max,\ell}$.

	$p_{\max}$	$p_{\min,\ell}$	$p_{\max,\ell}$	unpruned	pruned	
		b			e''	e'

Finally, we set $e := e''$ (the index returned by SubsetSelection) and are ready to recurse on $A[b, \ldots, e-1]$. Since, by construction, Invariants (A) and (B) hold for this call and since all steps take linear time and require $\mathcal{O}(1)$ working space, the lemma follows. □

[2] As usual, points lying on a hull edge, i.e., points in degenerate position, are not considered to be hull points.

Storing $p_{\max} = p_{\max,r}$ in front of the subarray passed to the first recursive call allows us to easily recover the indices b and e needed for invoking MidCode; in a slight abuse of notation, this method is invoked with the indices b and e passed to Recursive (these indices are available by Invariant (C)) and does the following:

1. Recover the value of e passed to PreCode.
2. Recover the right endpoint $p_{\min,r}$ of the bridge.
3. Select all points in $A[b, \ldots, e-1]$ that lie between $p_{\min,r}$ and $p_{\max,r}$.
4. Establish Invariant (A).

Lemma 3 *If Invariants (A) and (C) hold prior to invoking* MidCode *and if Invariant (B) held prior to the preceding call to* Recursive*, the* MidCode *method can be implemented as a linear-time method with $\mathcal{O}(1)$ working space such that Invariants (A), (B), and (C) hold prior to invoking* Recursive *for the "right" recursion.*

Proof. To recover the (original) value of e that was passed to PreCode, we scan forward from $A[e]$, i.e., the first item not passed to the preceding recursive call. By Invariant (C), we know that the index of the first element in $A[e, \ldots, n-1]$ with an x-coordinate larger than $A[b-1] = p_{\max}$ (or n if no such element exists) is the value of e we are looking for. This scan takes linear time and uses $\mathcal{O}(1)$ working space. We then decrement b by one to include the element $p_{\max} = p_{\max,r}$.

To recover the right endpoint $p_{\min,r}$ of the bridge, we exploit the fact that the bridge lies on the unique line passing through $p_{\max,\ell}$ and a point to the right of $p_{\max,\ell}$ such that no point in $A[b, \ldots, e-1]$ lies above this line.[3] A simple proof by contradiction shows that this other bridge point indeed lies to the right of the point a used for originally splitting the point set.

Now that $p_{\min,r}$ and $p_{\max,r}$ have been recovered, we use the same techniques as in PreCode to save the point $p_{\min}$, to select the points to be passed to the recursive call, to move $p_{\min,r}$ and $p_{\max,r}$ to the front of the subarray, and to adjust the index b accordingly. The resulting array then looks as follows:

	$p_{\min}$	$p_{\min,r}$	$p_{\max,r}$	unpruned	pruned	
		b			e''	e'

Finally, we set $e := e''$ (the index returned by SubsetSelection) and are ready to recurse on $A[b, \ldots, e-1]$. Since, by construction, Invariants (A) and (B) hold for this call and since all steps take linear time and require $\mathcal{O}(1)$ working space, the lemma follows. □

After the second call to Recursive, the call to PostCode is used to establish Invariants (A) and (C) for the invoking call to Recursive. In the light of this, PostCode needs to perform the following steps:

1. Recover the value of e passed to MidCode.
2. Recover the left endpoint $p_{\max,\ell}$ of the bridge.
3. Establish Invariant (A).

Lemma 4 *If Invariants (A) and (C) hold prior invoking* PostCode *and if Invariant (B) held prior to the preceding call to* Recursive*, the* PostCode *method can be implemented as a linear-time method with $\mathcal{O}(1)$ working space such that Invariants (A) and (C) hold.*

Proof. We proceed as in the proof of Lemma 3, i.e., we first recover the value of e that was passed to MidCode by checking boundary conditions w.r.t. $p_{\max,r} = A[b+2]$, adjusting b, and recovering the left bridge point. Since the value of b remained the same over the invocations of all relevant methods and since the value of e was recovered after each such invocation, Invariant (C) is established. Using a linear scan and two swap operations, Invariant (A) can be established as well. All algorithms run in linear time and use $\mathcal{O}(1)$ working space. □

4.2 Representing the Output

The description of the algorithm so far only focused on ensuring that the "boundaries" of the recursive calls can be recovered efficiently. In this subsection, we discuss how to represent the output from a call to Recursive. For this, we establish a fourth invariant:

Invariant (D): After a call to Recursive(A, b, e), the upper convex hull vertices (if any) between $A[b]$ and $A[b+1]$ (see Invariants (A) and (B)) are stored in increasing x-order starting at $A[b+2]$.

Obviously, this invariant can be established trivially for constant-sized recursion base cases.

Lemma 5 *If Invariants (A) and (D) hold after a call to* Recursive*(A, b, e), and if Invariant (B) held prior to the preceding call to* Recursive*, the upper convex hull computed during this recursive call can be recovered based upon the knowledge of b only.*

Proof. This proof exploits the fact that vertices on the upper convex hull form right turns when traversed in increasing x-order. The recovery algorithm first checks whether $A[b+2]$[4] lies strictly above the segment $\overline{A[b]A[b+1]}$. If this is not the case, the upper convex hull consists of $A[b]$ and $A[b+1]$ only, and we are done. Otherwise, the algorithm scans forward from $i = b+3$ until $A[i]$ lies right of $A[b+1]$ (in this case $i = e$, and we are done), $A[i]$ does not lie right of $A[i-1]$, or $A[i-1]$, $A[i]$, and $A[b+1]$ do not form a right turn (in the last two cases, $A[i-1]$ is the last hull vertex). □

[3] This line may not be determined by a unique point, namely if more than three points are allowed to be collinear. In this case, the rightmost of these points is the recovered bridge point since this choice minimizes the number of points passed to the next recursive call (see also Footnote 2 and Section 4.3).

[4] For the sake of simplicity, we assume that the size of the recursion base case is larger than two.

Lemma 5 implies that the upper convex hull can be reconstructed in linear time and $\mathcal{O}(1)$ working space after having returned from RECURSIVE(A, 0, n): Scan forward from $A[0]$ to recover the index H, i.e., the number of points on the upper hull, and stably exchange $A[1](= p_{\max})$ and $A[2, \ldots, H-1]$.

We now show how to maintain Invariant (D) throughout the algorithm. Inductively assume that, after the "left" call to RECURSIVE, we have computed the hull points (denoted by "$\frown_\ell$") between $p_{\min,\ell}$ and $p_{\max,\ell}$. Also, by Invariant (C), we have restored b and e to its original values. Then, the subarray looks as follows:

	$p_{\max}$	$p_{\min,\ell}$	$p_{\max,\ell}$	$\frown_\ell$	$\cdots$	
	b				c	e

By Lemma 5, we know that we can identify the index c such that $A[b{+}3, \ldots, c{-}1]$ stores the "$\frown_\ell$"-points. Using the (folklore) linear-time, in-place algorithm for swapping two adjacent blocks, we then move this subarray to the end of $A[b, \ldots, e-1]$.

	$p_{\max}$	$p_{\min,\ell}$	$p_{\max,\ell}$	$\cdots$	$\frown_\ell$	
	b				e'	e

We then continue as in the proof of Lemma 3. Similarly, after the "right" call to RECURSIVE, the subarray looks as follows (b and e are available by Invariant (C), c and e' are recovered as implied by Lemma 5):

	$p_{\min}$	$p_{\min,r}$	$p_{\max,r}$	$\frown_r$	$\cdots$	$\frown_\ell$	
	b				c	e'	e

Using linear-time, in-place swapping, we rearrange the contents of the subarray such that Invariants (A) and (D) hold, and proceed as in the proof of Lemma 4.

	$p_{\min,\ell}$	$p_{\max,r}$	$\frown_\ell$	$p_{\max,\ell}$	$p_{\min,r}$	$\frown_r$	$\cdots$	
	b							e

Since Invariant (D) can be established for constant-sized recursion base cases in a straightforward way, the above discussion implies that we can maintain Invariant (D) during each execution of RECURSIVE with linear time and $\mathcal{O}(1)$ working space as claimed.

4.3 Finding a Bridge

In the proof of Lemma 2, we assumed that there is a linear-time, in-place algorithm for finding the two endpoints $p_{\max,\ell}$ and $p_{\min,r}$ of the upper hull edge crossing the vertical line $x = a.x$. While we could simply refer to the in-place implementation proposed by Brönnimann *et al.* [8, Theorem 5], we give the details for the sake of self-containedness and to show that this computation does not interfere with maintaining the invariants.

By the discussion in the preceding subsections, we know that in this situation the subarray looks as follows:

	$p_{\min}$	$p_{\max}$	unpruned	pruned	
	b			$e'' := e'$	e

Following Kirkpatrick and Seidel, we form pairs of points and order them by increasing x-coordinate. If the number of points is odd, we use $\mathcal{O}(1)$ space to store the remaining point $\tilde{p}$. Using SUBSETSELECTION we then move all pairs where the two points have the same x-coordinate ("↑") to the end of the array.

	$p_{\min}$	$p_{\max}$	↗	$\cdots$	↗	↑	$\cdots$	↑	
	b					e'''			e''

From now on, we use $\mathcal{O}(1)$ space to keep track of the position of the points $p_{\max}$ and $p_{\min}$ such that they can be restored to $A[b, b+1]$ (hence establishing Invariant (A)) after the bridge has been found.

To find the bridge, we run a linear-time, in-place k-selection algorithm [6, 17] to determine the pair of points in $A[b, \ldots, e''' - 1]$ inducing the line with median slope K. Based upon this slope, we twice run SUBSETSELECTION to partition $A[b, \ldots, e''' - 1]$ into three sets of pairs: SMALL (slope less than K), EQUAL (slope K), and LARGE (slope larger than K).

	SMALL	EQUAL	LARGE	↑	$\cdots$	↑	
	b	c	c'	e'''			e''

Just as in the original algorithm, the two endpoints of the edge with slope K are found by scanning over $A[b, \ldots, e''' - 1]$ to find, among all points p maximizing $p.y - K \cdot p.x$, the points with minimum and maximum x-coordinate. This step is easily seen to both take linear time and use $\mathcal{O}(1)$ working space.

If the edge constructed this way crosses the vertical line $x = a.x$, we have found the hull vertices $p_{\max,\ell}$ and $p_{\min,r}$. We record these vertices using $\mathcal{O}(1)$ space and then spend linear time to move $p_{\max}$ and $p_{\min}$ back to $A[b, b+1]$. Finally, we discard the indices c, c', e'', and e''', keep the indices e and e', and resume the algorithm described in the proof of Lemma 2.

If, however, both endpoints of the edge lie to the right of the vertical line $x = a.x$, we need to recurse on all points in LARGE and the left points of all pairs in SMALL and EQUAL *plus* the upper points of all pairs with the same x-coordinate (the case that no endpoint lies to the right of the vertical line is handled symmetrically). To prepare for this (tail) recursion, we first swap LARGE to the beginning of $A[b, \ldots, e'' - 1]$ and then use SUBSETSELECTION to select the appropriate point from each remaining pair. If the number of points we started with was odd, we also add the point $\tilde{p}$ to the points to be processed next. To prepare for the next iteration, we again group pairs of points as described above and update the indices e''' and e'' accordingly.

	↗	$\cdots$	↗	↑	$\cdots$	↑	$\cdots$	
	b			e'''			e''	

The correctness of this algorithm and its linear runtime follow from the original proofs presented by Kirkpatrick and Seidel. With respect to the space requirement, we observe that, in addition to the constant number of indices used in k-selection algorithms and the calls to SubsetSelection, the iterative algorithm outlined above requires only to maintain a constant number of "global" indices: the original indices b and e, two indices to keep track of $p_{\max}$ and $p_{\min}$, and the index e' denoting the end of the current working set. The other indices, i.e., e'', e''', c, c', and possibly the index to keep track of the "excess" element $\tilde{p}$ are indices local to each iteration and can be discarded at the end of this iteration. In summary, the above discussion implies that we can indeed find a bridge in linear time and using $\mathcal{O}(1)$ working space while maintaining the invariants.

Putting everything together, this establishes a proof of Theorem 1, i.e., we have shown that we can realize the deterministic, time-optimal, output-sensitive, and instance-optimal planar convex hull algorithm by Kirkpatrick and Seidel using $\mathcal{O}(1)$ working space. Thus, we have established one more optimality criterion to hold for the "ultimate planar convex hull algorithm".

Note added in proof An alternative in-place algorithm has been suggested by Raimund Seidel [personal communication]: Viewed holistically, the "marriage-before-conquest-algorithm" maintains an ordered sequence of upper-hull edges and gaps and proceeds always by finding a bridge in the leftmost gap until no gap is left. The suggested alternative approach realizes this using an iterative, non-recursive algorithm which requires each points to be labeled either "extreme", "dead", or "alive". Assuming that the input does not contain duplicates, these labels can be stored implicitly by locally rearranging the input points: consider blocks of consecutive 7 points in the input array; there is a canonical lexicographic order of those points; storing the points in any one of the 7! permutations allows to encode any one of the 3^7 labellings of those points, since $7! > 3^7$. We leave the details to the reader.

References

[1] P. Afshani, J. Barbay, and T. M.-Y. Chan. Instance-optimal geometric algorithms. In *Proc. IEEE Symp. Foundations of Computer Science*, pp. 129–138. 2009.

[2] T. Asano and B. Doerr. Memory-constrained algorithms for shortest path problems. In *Proc. Canadian Conf. Computational Geometry*, pp. 315–318, 2011.

[3] T. Asano, W. Mulzer, G. Rote, and Y. Wang. Constant-work-space algorithms for geometric problems. *Journal of Computational Geometry*, 2(1):46–68, 2011.

[4] H. Blunck and J. Vahrenhold. In-place randomized slope selection. In *Proc. Intl. Conf. on Algorithms and Complexity*, LNCS 3998, pp. 31–40, 2006.

[5] H. Blunck and J. Vahrenhold. In-place algorithms for computing (layers of) maxima. *Algorithmica*, 57(1):1–21, May 2010.

[6] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry: Theory & Applications*, 37(3):209–227, Aug. 2007.

[7] H. Brönnimann, T. M.-Y. Chan, and E. Y. Chen. Towards in-place geometric algorithms. In *Proc. Symp. Computational Geometry*, pp. 239–246. 2004.

[8] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321(1):25–40, June 2004.

[9] T. M.-Y. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Computational Geometry: Theory and Applications*, 16(14):361–368, Apr. 1996.

[10] T. M.-Y. Chan and E. Y. Chen. Optimal in-place and cache-oblivious algorithms for 3-d convex hulls and 2-d segment intersection. *Computational Geometry: Theory and Applications*, 43(8):636–646, Oct. 2010.

[11] T. M.-Y. Chan, J. S. 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(4):433–454, Dec. 1997.

[12] J.-C. Chen. Optimizing stable in-place merging. *Theoretical Computer Science*, 302(1–3):191–210, June 2003.

[13] M. De, A. Maheshwari, S. Nandy, and M. Smid. An in-place priority search tree. In *Proc. Canadian Conf. Computational Geometry*, pp. 331–336, 2011.

[14] M. De and S. Nandy. Space-efficient algorithms for empty space recognition among a point set in 2D and 3D. In *Proc. Canadian Conf. Computational Geometry*, pp. 347–353, 2011.

[15] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, June 1972.

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

[17] T. W. Lai and D. Wood. Implicit selection. In *Proc. Scand. Workshop on Algorithm Theory*, LNCS 318, pp. 14–23, 1988.

[18] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6(4):423–434, Dec. 1991.

[19] J. Vahrenhold. An in-place algorithm for Klee's measure problem in two dimensions. *Information Processing Letters*, 102:169–174, May 2007.

[20] J. Vahrenhold. Line-segment intersection made in-place. *Computational Geometry: Theory & Applications*, 38(3):213–230, Oct. 2007.

[21] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964.