In-Place Algorithms for Computing (Layers of) Maxima

Henrik Blunck[*] Jan Vahrenhold[†]

Abstract

We describe space-efficient algorithms for solving problems related to finding maxima among points in two and three dimensions. Our algorithms run in optimal $\mathcal{O}(n \log_2 n)$ time and require $\mathcal{O}(1)$ space in addition to the representation of the input.

1 Introduction

In this paper, we consider the fundamental geometric problems of computing the maxima of point sets in two and three dimensions and of computing the layers of maxima in two dimensions. Given two points p and q, the point p is said to *dominate* the point q iff the coordinates of p are larger than the coordinates of q in all dimensions. A point p is said to be a *maximal point* (or: a *maximum*) of $\mathcal{P}$ iff it is not dominated by any other point in $\mathcal{P}$. The union $\texttt{MAX}(\mathcal{P})$ of all points in $\mathcal{P}$ that are maximal is called the *set of maxima* of $\mathcal{P}$. This notion can be extended in a natural way to compute *layers* of maxima [7]: After $\texttt{MAX}(\mathcal{P})$ has been identified, the computation is iterated for $\mathcal{P} := \mathcal{P} \setminus \texttt{MAX}(\mathcal{P})$, i.e., the next layer of maxima is computed until $\mathcal{P}$ becomes empty.

Related Work The problem of finding maxima of a set of n points has a variety of applications in statistics, economics, and operations research (as noted by Preparata and Shamos [14]), and thus was among the first problems having been studied in Computational Geometry: In $\mathbb{R}^2$ and $\mathbb{R}^3$, the best known algorithm, Kung, Luccio, and Preparata's algorithm [11], identifies the set of maxima in $\mathcal{O}(n \log_2 n)$ time which is optimal since the problem exhibits a sorting lower bound [11, 14]. For constant dimensionality $d \geq 4$, their divide-and-conquer approach yields an algorithm with $\mathcal{O}(n \log_2^{d-2} n)$ running time, and Matoušek [12] gave an $\mathcal{O}(n^{2.688})$ algorithm for the case $d = n$. The problem has also been studied for dynamically changing point sets in two dimensions [10] and under assumptions about the distribution of the input points in higher dimensions [1, 8]. Buchsbaum and Goodrich [7] presented an $\mathcal{O}(n \log_2 n)$ algorithm for computing the layers of maxima for point sets in three dimensions. Their approach is based on the plane-sweeping paradigm and relies on dynamic fractional cascading to maintain a point-location structure for dynamically changing two-dimensional layers of maxima. The maxima problem has also been actively investigated in the database community following the definition of the SQL "`skyline`" operator [2] that is used to compute the set of maxima. Spatial index structures have been used to produce the "skyline" practically efficient and/or in a progressive way, that is outputting results while the algorithm is running—see [13] and the references therein. For none of these approaches, non-trivial upper bounds are known.

The Model The goal of investigating space-efficient algorithms is to design algorithms that use very little extra space in addition to the space used for representing the input. The input is assumed to be stored in an array `A` of size n, thereby allowing random access. We assume that a constant size memory can hold a constant number of words. Each word can hold one pointer, or an $\mathcal{O}(\log_2 n)$ bit integer, and a constant number of words can hold one element of the input array. The extra memory used by an algorithm is measured in terms of the number of extra words; an *in-place* algorithm uses $\mathcal{O}(1)$ extra words of memory. It has been shown that some fundamental geometric problems such as 2D convex hulls and closest pairs can be solved *in-place* and in optimal time [3, 4, 6]. More involved problems (range searching, line-segment intersection) can be (currently) solved *in-place* only if one is willing to accept near-optimal running time [5, 15], and 3D convex hulls and related problems seem to require both (poly-)logarithmic extra space and time [5].

Our Contribution The main issue in designing in-place algorithms is that most powerful algorithmic tools (unbalanced recursion, sweeping, multi-level data structures, fractional cascading) require $\Omega(\log_2 n)$ or even $\Omega(n)$ extra space, e.g., for the recursion stack or pointers. This raises the question of whether there exists a time-space trade-off for geometric algorithms besides range-searching. We make a further step towards a negative answer to this question by demonstrating that $\mathcal{O}(1)$ extra space is suf-

[*]Westfälische Wilhelms-Universität Münster, Institut für Informatik, Einsteinstr. 62, 48149 Münster, Germany. E-mail: `blunck@math.uni-muenster.de`

[†]Westfälische Wilhelms-Universität Münster, Institut für Informatik, Einsteinstr. 62, 48149 Münster, Germany. E-mail: `jan@math.uni-muenster.de`

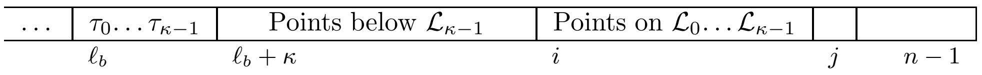
$\ldots$
$\tau_0 \ldots \tau_{\kappa-1}$
Points below $\mathcal{L}_{\kappa-1}$
Points on $\mathcal{L}_0 \ldots \mathcal{L}_{\kappa-1}$
ℓ_b
$\ell_b + \kappa$
i
j
$n-1$

Figure 1: Data layout for processing the topmost κ layers.

ficient to obtain optimal $\mathcal{O}(n \log_2 n)$ algorithms for computing skylines in two and three dimensions and two-dimensional layers of maxima. The solution to the latter problem is of particular interest since it is the first optimal in-place algorithm for a geometric problem that is not amenable to a solution based on balanced divide-and-conquer or Graham's scan.

2 Computing the Skyline in $\mathbb{R}^2$ and $\mathbb{R}^3$

A point p from a point set $\mathcal{P}$ is said to be *maximal* if no other point from $\mathcal{P}$ has larger coordinates in *all* dimensions; ties are broken using a standard shearing technique. This definition has been transferred by Kung *et al.* [11] into a plane-sweeping algorithm for the two-dimensional case and a divide-and-conquer approach for the higher-dimensional case. The output of the algorithm will consist of a permutation of the input array A and an index k such that k points constituting the set of maxima are stored sorted by decreasing y-coordinates in $\texttt{A}[0, \ldots, k-1]$.

Lemma 1 *The* skyline, *i.e., the set of maxima of a set $\mathcal{P}$ of n points in two dimensions can be computed in-place and in optimal time $\mathcal{O}(n \log_2 n)$. If $\mathcal{P}$ is sorted according to $<_y$, the running time is in $\mathcal{O}(n)$.*

For the case of a three-dimensional input, we implement Kung *et al.*'s [11] divide-and-conquer algorithm using an in-place divide-and-conquer scheme we have proposed earlier [3]; this scheme is based on in-place routines for median-finding, partitioning, and merging. Since we cannot explicitly keep track of the number of maxima in each subproblem, we have to recover them algorithmically during each merging step.

Theorem 2 *The* skyline, *i.e., the set of maxima of an n-element point set in three dimensions can be computed in-place and in optimal time $\mathcal{O}(n \log_2 n)$.*

3 Computing the Layers of Maxima in $\mathbb{R}^2$

A naïve approach to computing all layers of maxima would be to iteratively use the in-place algorithm described in Section 2. Since a point set may exhibit a linear number of layers, this will lead to $\mathcal{O}(n^2 \log_2 n)$ worst-case running time. In this section, we will show that we can simultaneously peel off multiple layers such that the resulting algorithm runs in optimal $\mathcal{O}(n \log_2 n)$ time; its goal is to rearrange the input such that the points are grouped by layers and each layer is sorted by decreasing y-coordinate.

3.1 Computing the Topmost κ Layers

Our algorithm imitates counting-sort, i.e., prior to actually partitioning the points into layers, it first computes the number of points for each of the layers. This can be done efficiently:

Lemma 3 *The number of layers of maxima exhibited by an n-element point set in two dimensions can be computed in-place and in $\mathcal{O}(n \log_2 n)$ time.*

Counting the points on the topmost κ layers In this section, we prove the following lemma:

Lemma 4 *The cardinality c_i of each of the topmost κ layers of $\texttt{A}[0, \ldots, n-1]$ can be computed in $\mathcal{O}(n \log_2 n)$ time. If the points are presorted, the complexity is $\mathcal{O}(n + \xi \log_2 \kappa)$ where $\xi = \sum_{i=0}^{\kappa-1} c_i$.*

To illustrate the algorithm, let us assume that we have already peeled off some layers and stored the result in $\texttt{A}[0, \ldots, \ell_b - 1]$. Inductively, we maintain the following invariant which, prior to the first iteration, can be established by sorting and by setting $\ell_b := 0$:

Invariant (SORT): The points that have not yet been assigned to a layer are stored in $\texttt{A}[\ell_b, \ldots, n-1]$ and are sorted by decreasing y-coordinate.

Let us further assume that the total number of points on the topmost κ layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$ of the *remaining* points stored in $\texttt{A}[\ell_b, \ldots, n-1]$ is ξ and that $\ell_b + 2\xi \le n$. The first step then is to stably extract the ξ points on the topmost κ layers and move them to $\texttt{A}[\ell_b, \ldots, \ell_b + \xi - 1]$ while maintaining the sorted y-order in $\texttt{A}[\ell_b + \xi, \ldots, n-1]$. The algorithm processes the points as they are stored in the array, i.e., in decreasing y-order. It maintains the invariant that, when processing point $\texttt{A}[j]$, all points that already have been identified as "below $\mathcal{L}_{\kappa-1}$" are stored in decreasing y-order in $\texttt{A}[\ell_b + \kappa, \ldots, i-1]$ for some $i \in [\ell_b + \kappa, \ldots, j]$—see Figure 1. Since the points are sorted according to $<_y$, we can efficiently sweep the plane top-down. For each $h \in [0, \ldots, \kappa-1]$, we maintain the *tail* τ_h, the lowest point seen so far that is known to lie on $\mathcal{L}_h$.

The point $\texttt{A}[j]$ now either is classified as "below $\mathcal{L}_{\kappa-1}$" (p_{i_1} in Figure 2) or replaces the tail τ_h of some layer $\mathcal{L}_h$ (p_{i_2} in in Figure 2). In the former case, $\texttt{A}[j]$ is stably moved directly behind $\texttt{A}[\ell_b + \kappa, \ldots, i-1]$, i.e., it is swapped with $\texttt{A}[i]$ and i is then incremented by one. In the latter case, $\texttt{A}[j]$ is swapped with τ_h, i.e., with $\texttt{A}[\ell_b + h]$, and we increment the counter c_h

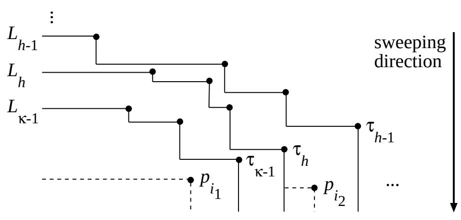

Figure 2: Maintaining the lowest point of each layer.

by one. When we have reached the end of the array, we inductively see that $\mathtt{A}[\ell_b+\kappa,\ldots,i-1]$ contains the points below $\mathcal{L}_{\kappa-1}$ in sorted order. Furthermore, by the definition of ξ, we know that the two subarrays $\mathtt{A}[\ell_b,\ldots,\ell_b+\kappa-1]$ (containing the tails) and $\mathtt{A}[i,\ldots,n-1]$ (containing the remaining points on the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$) together consist of exactly ξ points. We then swap (in linear time) $\mathtt{A}[\ell_b+\kappa,\ldots,i-1]$ (containing the elements below $\mathcal{L}_{\kappa-1}$) and $\mathtt{A}[i,\ldots,n-1]$ such that the ξ elements on the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$ (tails and non-tails) are blocked in $\mathtt{A}[\ell_b,\ldots,\ell_b+\xi-1]$. To re-establish the y-order of these ξ points, we sort them in $\mathcal{O}(\xi\log_2\xi)\subset\mathcal{O}(\xi\log_2 n)$ time, that is, we establish Invariant (SORT) for $\mathtt{A}[\ell_b,\ldots,\xi-1]$. Thus, the overall running time for counting the number of points on the topmost κ layers and for re-establishing Invariant (SORT) is $\mathcal{O}(n+\xi\log_2 n)$ where $\xi=\sum_{i=0}^{\kappa-1}c_i$.

Sorting the points by layer Using the counters c_i computed during the previous step, we now run a variant of counting sort to extract the layers $\mathcal{L}_0$ through $\mathcal{L}_{\kappa-1}$ in sorted y-order. To do this in-place, we use the subarray $\mathtt{A}[\ell_b+\xi,\ldots,\ell_b+2\xi-1]$ as scratch space that will hold the layers to be constructed (note that we assume $\ell_b+2\xi\leq n$ and that $\xi=\sum_{i=0}^{\kappa-1}c_i$ holds by definition). To re-establish Invariant (SORT), we finally sort $\mathtt{A}[\ell_b+\xi,\ldots,\ell_b+2\xi-1]$ (note that the points $\mathtt{A}[\ell_b+2\xi,\ldots,n-1]$ have not been touched and thus still are sorted) and update $\ell_b:=\ell_b+\xi$. The running time for sorted extraction of the ξ points on the topmost κ layers and for re-establishing Invariant (SORT) for $\mathtt{A}[\ell_b+\xi,\ldots,n-1]$ is $\mathcal{O}(n+\xi\log_2 n)$.

3.2 Extracting all Layers in Sorted Order

The above algorithm is built on two major assumptions that need to be maintained in an in-place setting: (1) we have to have access to κ counters and (2) the subarray $\mathtt{A}[\ell_b,\ldots,n-1]$ has to be large enough to accommodate two subarrays of size ξ. The first issue to be resolved is how to maintain a non-constant number κ of counters without using $\Theta(\kappa)$ extra space. Each such counter c_i is required to represent values up to n, and thus has to consist of $\log_2 n$ bits.

We will resort to a standard technique in the design of space-efficient algorithms, namely to encode a single bit by a permutation of two distinct (but comparable) elements q and r: assuming $q<r$, the permutation rq encodes a binary zero, and the permutation qr encodes a binary one. As the elements in our case are two-dimensional points, we will use the y-order for deciding whether two points encode a binary zero or a binary one.[1] Using a block of $\frac{1}{3}n$ elements, we can encode $\frac{1}{6}n$ bits, i.e., $\frac{1}{6}n/\log_2 n$ counters, and this implies that the maximum number of layers for which we can run the algorithm of Lemma 4 is $\kappa=\frac{1}{6}n/\log_2 n$. Lemma 4 gives an $\mathcal{O}(n+\xi\log_2 n)$ bound for each run, and thus we have to make sure that maintaining the counters does not interfere with keeping the overall number of iterations in $\mathcal{O}(\log_2 n)$.

3.2.1 The case $\ell_b<\frac{1}{3}n$

If, prior to the current iteration, $\ell_b<\frac{1}{3}n$ holds, we maintain the counters in $\mathtt{A}[\frac{2}{3}n,\ldots,n-1]$.

Counting the points on the topmost κ layers By Invariant (SORT), $\mathtt{A}[\ell_b,\ldots,n-1]$ is sorted by decreasing y-coordinate, so all counters encode the value zero. We set $\kappa:=\frac{1}{6}n/\log_2 n$ and count the elements on each of the topmost κ layers. Note that, since the algorithm will process *all* points in $\mathtt{A}[\ell_b,\ldots,n-1]$, any point q in $\mathtt{A}[\frac{2}{3}n,\ldots,n-1]$ may be swapped to the front of the array since it may become the tail τ_i of some layer $\mathcal{L}_i$. Using a more careful implementation of the approach given in Section 3.1, we can compute all counters and re-establish Invariant (SORT) in $\mathcal{O}(n+\xi\log_2 n)$ time. After we have computed the values of all counters c_i—but prior to re-establishing Invariant (SORT)—we compute the prefix sums of c_0 through $c_{\kappa-1}$, i.e., we replace c_j by $\hat{c}_j:=\sum_{i=0}^{j}c_j$. This can be done in-place spending $\mathcal{O}(\log_2 n)$ time per counter, i.e., in $\mathcal{O}(n)$ overall time. We also maintain the maximal index κ' such that $\ell_b+2\hat{c}_{\kappa'}<\frac{2}{3}n$.

Extracting and sorting the points on the topmost κ layers If the index κ' described above exists, we run (a slightly modified implementation of) the algorithm for extracting the $\xi':=\hat{c}_{\kappa'}$ points on the κ' topmost layers as described in Section 3.1. Because of the way κ' was chosen, we can guarantee that the scratch space of size $\xi'=\hat{c}_{\kappa'}$ needed for the counting-sort-like partitioning will not interfere with the space $\mathtt{A}[\frac{2}{3}n,\ldots,n-1]$ reserved for representing the counters. The total cost for extracting ξ' points is $\mathcal{O}(n+\xi'\log_2 n)$; this also includes the cost for sorting the scratch space $\mathtt{A}[\ell_b+\xi',\ldots,\ell_b+2\xi'-1]$ and re-establishing Invariant (SORT) (see Section 3.1).

[1]A *set* cannot contain duplicates; hence the relative order of two points is unique. Furthermore, the set of maxima of a *multiset* M consists of the same points as the set of maxima of the *set* obtained by removing the duplicates from M. Duplicate removal can be done in-place and in $\mathcal{O}(n\log_2 n)$ time.

If $\kappa' < \kappa$, i.e., if we extract some but not all $\kappa = \frac{1}{6}n/\log_2 n$ layers, we will additionally run the $\mathcal{O}(n \log_2 n)$ skyline computation algorithm described in Section 2 as a post-processing step to also extract the points on the next topmost layer, regardless of its size. Similarly, if the index κ' does not exist at all, we extract the topmost layer $\mathcal{L}_0$ using the $\mathcal{O}(n \log_2 n)$ skyline computation algorithm on $\mathtt{A}[\ell_b, \ldots, n-1]$—note that in this case the topmost layer $\mathcal{L}_0$ contains $c_0 > \frac{1}{2}\left(\frac{2}{3}n - \ell_b\right) > \frac{1}{2}\left(\frac{2}{3}n - \frac{1}{3}n\right) \in \Theta(n)$ points. In any case, we spend another $\mathcal{O}(n \log_2 n)$ time to re-establish Invariant (SORT) by sorting.

Analysis Our analysis classifies each iteration according to whether or not all ξ points on the topmost $\kappa = \frac{1}{6}n/\log_2 n$ layers are moved to their final position in the array. If all ξ points are moved, we know that $\xi \geq \frac{1}{6}n/\log_2 n$, and thus only a logarithmic number of such iterations can exist. Also, we can distribute the $\mathcal{O}(n + \xi \log_2 n)$ time spent per iteration such that each iteration gets charged $\mathcal{O}(n)$ time and that each of the ξ points moved to its final position gets charged $\mathcal{O}(\log_2 n)$ time, so the overall cost for all such iterations is $\mathcal{O}(n \log_2 n)$. If less than κ layers can be processed in the iteration in question (this also includes the case that κ' does not exist), the $\mathcal{O}(n + \xi \log_2 n)$ cost for counting the ξ points on the topmost κ layers and the $\mathcal{O}(n + \xi' \log_2 n)$ cost for extracting ξ' points on the topmost κ' layers is dominated by the $\mathcal{O}(n \log_2 n)$ cost for the successive skyline computation. The definition of κ' guarantees that, after we have performed the skyline computation, we have advanced the index ℓ_b by at least $\frac{1}{2}\left(\frac{2}{3}n - \ell_b\right)$ steps. Since $\ell_b < \frac{1}{3}n$, there is only a constant number of such iterations, hence their overall cost is $\mathcal{O}(n \log_2 n)$.

3.2.2 The case $\ell_b \geq \frac{1}{3}n$

If, prior to the current iteration, $\ell_b \geq \frac{1}{3}n$ holds, we maintain the counters in $\mathtt{A}[0, \ldots, \frac{1}{3}n - 1]$. Note that this subarray contains (part of) the layers that have been computed already. Since maintaining a counter will involve swapping some of the elements in $\mathtt{A}[0, \ldots, \frac{1}{3}n - 1]$, this will disturb the y-order of (some of) the layers already computed, and we have to make sure that we can reconstruct the layer order.

As for the case $\ell_b < \frac{1}{3}n$ we either extract all ξ points on the topmost κ layers in $\mathcal{O}(n + \xi \log_2 n)$ time or extract less than κ layers followed by a skyline computation in $\mathcal{O}(\nu \log_2 \nu)$ time where $\nu := n - \ell_b$. In both cases, the complexity given also includes the cost for re-establishing Invariant (SORT). Summing up, the cost for all iterations in which $\ell_b < \frac{1}{3}n$ and for all iterations in which $\ell_b \geq \frac{1}{3}n$ is $\mathcal{O}(n \log_2 n)$. Restoring the "counter space" to the original order can be done in the same time complexity, and in the full version, we prove that a simple linear-time scan is sufficient to detect all "layer boundaries". Combining this with the fact that each point gets charged $\mathcal{O}(\log_2 n)$ cost for the iteration in which it is moved to its final location, we obtain our main result:

Theorem 5 *All layers of maxima of an n-element point set in two dimensions can be computed in-place and in optimal time $\mathcal{O}(n \log_2 n)$ such that the points in each layer are sorted by decreasing y-coordinate.*

References

[1] J. Bentley, K. Clarkson, and D. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica*, 9(2):168–183, 1993.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. 17th Intl. Conf. Data Engineering*, pp. 421–430. 2001.

[3] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry: Theory & Applications*, 2006.

[4] H. Brönnimann and T. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. In *Proc. Latin American Theoretical Informatics*, LNCS 2976, pp. 162–171. 2004.

[5] H. Brönnimann, T. Chan, and E. Chen. Towards in-place geometric algorithms. In *Proc. 20th Symp. Computational Geometry*, pp. 239–246. 2004.

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

[7] A. Buchsbaum and M. Goodrich. Three-dimensional layers of maxima. *Algorithmica*, 39(4):275–286, 2004.

[8] H. Dai and X. Zhang. Improved linear expected-time algorithms for computing maxima. In *Proc. Latin American Theoretical Informatics*, LNCS 2976, pp. 181–192. 2004.

[9] V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1–2):159–181, 2000.

[10] S. Kapoor. Dynamic maintenance of maxima of 2-D point sets. *SIAM J. Computing*, 29(6):1858–1877, 2000.

[11] H. Kung, F. Luccio, and F. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[12] J. Matoušek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.

[13] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Systems*, 30(1):41–82, 2005.

[14] F. Preparata and M. Shamos. *Computational Geometry. An Introduction.* Springer, 1988.

[15] J. Vahrenhold. Line-segment intersection made in-place. In *Proc. 9th Intl. Workshop Algorithms and Data Structures*, LNCS 3608, pp. 146–157, 2005.