In-Place Randomized Slope Selection

Henrik Blunck* Jan Vahrenhold†

Abstract

Slope selection, i.e. selecting the slope with rank k among all $\binom{n}{2}$ lines induced by a collection $\mathcal{P}$ of points, results in a widely used robust estimator for line-fitting. In this paper, we demonstrate that it is possible to perform slope selection in expected $\mathcal{O}(n \cdot \log_2 n)$ time using only constant extra space in addition to the space needed for representing the input.

1 Introduction

Computing a *line estimator*, i.e., fitting a line to a collection $\mathcal{P}$ of n data points $\{p_1, \ldots, p_n\}$ in the plane is a frequent task in statistical analysis. A frequently used robust line estimator is the so-called *Theil-Sen* estimator (see [13] and the references therein) which considers all $\binom{n}{2}$ lines induced by the points in $\mathcal{P}$ and selects the line with median slope. This problem is also known as the (median) *slope selection* problem and has been shown to exhibit an $\Omega(n \cdot \log_2 n)$ lower bound [6]. Several deterministic algorithms for solving this problem in optimal $\mathcal{O}(n \cdot \log_2 n)$ running time have been presented [5, 6, 10], however, as noted by Matoušek *et al.* [13], they are based on relatively complicated concepts such as parametric search, sorting network, expander graphs, or cuttings. More practical approaches have resulted in randomized algorithms with expected $\mathcal{O}(n \cdot \log_2 n)$ running time [7, 12, 15].

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. An *in-place* algorithms uses $\mathcal{O}(1)$ extra words of memory. Recently, a number of in-place algorithms have been designed for solving geometric problems—see, e.g., [1, 2, 3, 4, 16].

*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`

In addition to theoretical considerations, one reason for investigating space-efficient algorithms is that they have the potential of using the different stages of hierarchical memory, e.g., caches, to a much higher degree of efficiency. Another motivation, especially for designing algorithms for statistical data analysis, comes from the recently increased interest in sensor networks where small-scale computing devices are used to collect large amounts of data. Since the memory of such sensor devices usually is very limited, and since transmitting data is much more costly than local computation, it is desirable to process as much data as possible locally before transmitting (intermediate) results.

Our Results In this paper we show how to solve the slope selection problem in expected optimal $\mathcal{O}(n \log_2 n)$ time while at the same time using only constant extra space. Our algorithm follows the approach of Matoušek [12], and, in the course of implementing his algorithm in-place, we also devise an in-place variant of the so-called *randomized interpolation search* technique. This variant, together with an algorithmic subroutine for efficiently constructing and storing a set of randomly sampled intersections, is of independent interest, since it can be used as a substitute for Megiddo's *parametric search* technique [14].

2 Randomized Interpolation Search

In the following, the slope selection problem is studied in the dual setting where each point (x, y) is identified with the line $\{(\xi, \upsilon) \mid \upsilon = x \cdot \xi - y\}$ and vice versa. Selecting the k-th smallest slope is dual to the following problem: Given a set of n lines in the plane, find the k-th leftmost intersection point induced by the arrangement of the lines. Since the duality transform can be performed in an implicit way, we will assume that our input is given as a set $\mathcal{P}$ of lines in the plane.

Since it is infeasible to compute all $\Theta(n^2)$ intersections induced by $\mathcal{P}$, the algorithm of Matoušek [12] maintains a vertical strip $\langle b, e\rangle := [b, e] \times \mathbb{R} \subset \mathbb{R}^2$ that is guaranteed to contain the k-th smallest intersection point. For a parameter r (to be defined later), the algorithm first constructs a sample $\mathcal{R}$ of size r drawn from the intersections inside $\langle b, e\rangle$. It then selects two intersections from $\mathcal{R}$ whose x-coordinates are used to construct a (narrower) candidate strip $\langle b', e'\rangle$. The algorithm then checks whether $\langle b', e'\rangle$ indeed contains the k-th smallest intersection point. If this is not the

Dagstuhl Seminar Proceedings 06091
Data Structures
http://drops.dagstuhl.de/opus/volltexte/2006/839

case, the process is repeated for $\langle b, e\rangle$ but using a new sample $\mathcal{R}$, otherwise the algorithm iterates with the refined strip $\langle b', e'\rangle$. The iteration terminates when the number $|\mathcal{I}(b, e)|$ of intersections within $\langle b, e\rangle$ is no larger than r: in this case, the k-th leftmost intersection point can be computed directly by enumerating all intersections in $\langle b, e\rangle$ and selecting the appropriate one. This refinement strategy is referred to as *randomized interpolation search*—see [12]. The efficiency of the resulting algorithm for slope selection is based upon the following lemma which (applied iteratively) implies that the number $|\mathcal{I}(b, e)|$ of intersections that lie inside $\langle b, e\rangle$ can be reduced to $\mathcal{O}(r)$ using an expected constant number of iterations:

Lemma 1 (Lemma 2.1 in [13]) *Given a set of numbers $\Theta = \{\theta_1, \theta_2, \ldots, \theta_N\}$, an index k ($1 \leq k \leq N$), and an integer $r > 0$, we can compute in $\mathcal{O}(r)$ time an interval $[\theta_{lo}, \theta_{hi}]$, such that, with probability $1-1/\Omega(\sqrt{r})$, the k-th smallest element of Θ lies within this interval, and the number of elements in Θ that lie within the interval is at most $N/\Omega(\sqrt{r})$.*

The above lemma will be applied for $N \in \mathcal{O}(n^2)$. Furthermore, Matoušek *et al.* [13] proved that we may choose $r := \lceil n^\beta \rceil$ for any $0 < \beta < 1$ without affecting the asymptotic efficiency of the resulting algorithm, and thus we will set $r := \lceil \sqrt{n} \rceil$. As we will see below, our in-place algorithm will be working with binary encoded numbers, and thus accessing a single number θ_i will take $\mathcal{O}(\log_2 n)$ time. Thus, the in-place version of the algorithm implied by Lemma 1 runs in $\mathcal{O}(r \cdot \log_2 n)$ time.

It remains to describe how to construct (and store!) the sampled set $\mathcal{R}$ of $r = \lceil \sqrt{n} \rceil$ intersections in an in-place setting, i.e., using only constant extra space. Furthermore, we need to discuss how to compute $|\mathcal{I}(b, e)|$. To this effect, we describe an algorithm for the first task, which also provides a solution for the second one. Anticipating the results presented in the next section, we combine them with the above lemma and the original analyses of Matoušek *et al.* [12, 13]:

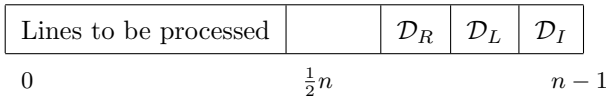
Theorem 2 *The slope selection problem for a set of n input points in the plane can be solved in-place and in expected optimal $\mathcal{O}(n \cdot \log_2 n)$ running time.*

3 Constructing the Random Sample $\mathcal{R}$

Following the approach of Matoušek [12, Lemma 1], we first draw (with replacement) a set of r random integers from $\{0, \ldots, |\mathcal{I}(b, e)| - 1\}$ where $|\mathcal{I}(b, e)|$ is the number of intersections in $\langle b, e\rangle$; these numbers give the ranks of the intersections that will be part of $\mathcal{R}$ with respect to the order in which they are found.

The main ingredient used for efficiently processing intersections in $\langle b, e\rangle$ is the following well-known observation: the number of intersections inside $\langle e, b\rangle$ is exactly the number of inversions between the permutation of $\mathcal{P}$ that arranges the lines in sorted $<_b$-order (the vertical order at $x = b$) and the permutation that arranges the lines in sorted $<_e$-order (the vertical order at $x = b$). Thus, to efficiently compute $|\mathcal{I}(b, e)|$, we can run the classic divide-and-conquer algorithm for inversion counting—see, e.g., [11]. While doing so, we keep track of the total number of inversions/intersections seen so far and "record" an intersection if its rank matches one of the r given ranks. If the ranks are sorted, we can process them in constant extra time per inversion counting operation. We process the "recursion tree" of (our adaption of) the inversion counting algorithm (see Algorithm 1) in a bottom-up, level-by-level traversal, i.e., without the need of maintaining a recursion stack. Since we need to maintain the r ranks and the intersections computed so far in an in-place setting, the algorithm is divided into three phases: During the first phase, we process the lines stored in $\mathtt{A}[0, \ldots, n/2 - 1]$ and use $\mathtt{A}[n/2, \ldots, n - 1]$ to encode the ranks and the intersections found so far. We then reverse the roles of both subarrays, and finalize the algorithm with a third phase that processes the intersections induced by lines stored in different halves of the array.

Three In-Place Data Structures For maintaining more than $\mathcal{O}(1)$ numbers or indices, we resort to a standard technique in the design of space-efficient algorithms, namely to encode a single bit by a permutation of two objects q and r: For lines q, r with $q <_b r$, the permutation qr encodes a binary zero, and the permutation rq encodes a binary one. We use the subarray of size $n/2$ that does not contain the lines to be processed in the current phase to represent three (implicit) data structures $\mathcal{D}_R$, $\mathcal{D}_L$, and $\mathcal{D}_I$ that occupy a subarray of size $4 \cdot r \cdot \log_2 n$ each:



Storing Ranks The randomly generated ranks in the range $[0, \ldots, n^2 - 1]$ are encoded in a "sorted-list" data structure $\mathcal{D}_R$. At initialization of $\mathcal{D}_R$, these ranks are sorted using *heapsort* [17], which performs $\mathcal{O}(r \cdot \log_2 r)$ operations each of which requires decoding a binary-encoded integer or swapping the values of two "rank elements".[1] This results in $\mathcal{O}(r \cdot \log_2 r \cdot \log_2 n)$ time spent for sorting. Having sorted the ranks, our algorithm will be able to traverse the list and report each rank to be processed in $\mathcal{O}(\log_2 n)$ time.

[1] Note, that swapping two encoded values a, b (as done by heapsort) does not require swapping the *blocks* of input elements used for encoding a and b—it merely involves updating the *permutations* used to represent bits. Therefore, each input element will be at most one position off its correct position.

Storing Lines Involved in Intersections We use a "sorted-list" data structure $\mathcal{D}_L$ to record (references to) lines involved in all of the sampled intersections found so far. These (references to) lines are maintained in sorted $<_b$-order. Every reference to a line is inserted into $\mathcal{D}_L$ using insertion sort (ignoring duplicates), and this leads to $\mathcal{O}(r^2 \cdot \log_2 n)$ global cost for maintaining $\mathcal{D}_L$.

Storing Intersections The "linked-list" data structure $\mathcal{D}_I$ records the intersections found so far by indexing into $\mathcal{D}_L$. To add an intersection induced by two lines ℓ_1 and ℓ_2 to $\mathcal{D}_I$, we first insert references to ℓ_1 and ℓ_2 in sorted $<_b$-order into $\mathcal{D}_L$ and then append the pair (i, j) referencing the references in $\mathcal{D}_L$ to these two lines at the end of $\mathcal{D}_I$. The cost for performing a single insert to $\mathcal{D}_I$ is $\mathcal{O}(\log_2 n)$, and thus we have a global update cost of $\mathcal{O}(r \cdot \log_2 n)$.

3.1 Processing one Half of the Subarray

The algorithms for processing the two halves of $\mathtt{A}[0, \ldots, n-1]$ are symmetric, and thus we present the algorithm for processing the subarray $\mathtt{A}[0, \ldots, n/2-1]$.

Counting Inversions The algorithm for counting *all* inversions in $\langle b, e\rangle$ is an extension of the iterative *mergesort* algorithm: starting from the set of lines in sorted $<_b$-order, the algorithm iteratively merges the lines into $<_e$-order while counting inversions. During each *merge*-step of the algorithm, two subarrays already in sorted $<_e$-order are merged into a single $<_e$-sorted subarray. Each of these subarrays has been processed during the previous iteration, and thus all inversions involving lines from only one of these subarrays have been processed. An obvious, yet crucial, fact guaranteeing the correctness of the inversion counting algorithm is that any two subarrays $\mathtt{A}_1$ and $\mathtt{A}_2$ that are merged in the j-th iteration of processing the m-th bottom-most level of the recursion tree are of the form $\mathtt{A}_1 := \mathtt{A}[j \cdot 2^m, \ldots, (j+1) \cdot 2^m - 1]$ and $\mathtt{A}_2 := \mathtt{A}[(j+1) \cdot 2^m, \ldots, (j+2) \cdot 2^m - 1]$[2]. Since in our case all lines are initially sorted in $<_b$-order, for every invocation of the algorithm CountAndRecord (Algorithm 1) depicted below the following holds: Each line in $\mathtt{A}_1$ precedes all lines in $\mathtt{A}_2$ with respect to the $<_b$-order and the union $\mathtt{A}_1 \cup \mathtt{A}_2$ forms a complete interval of the input in $<_b$-order.

Algorithm 1 can be implemented using constant extra space, and, excluding the time needed for recording the relevant intersections, its running time is linear in the size of the union of the two subarrays to be merged. Thus, excluding the time needed for updating $\mathcal{D}_R$, $\mathcal{D}_L$, and $\mathcal{D}_I$, its time complexity is in $\mathcal{O}(n \log_2 n)$. Also, leaving out the code in Lines 6–9, we may use Algorithm 1 to compute $|\mathcal{I}(b, e)|$.

[2]The algorithm can be easily modified to handle instances where n is not a power of two [1].

Algorithm 1 CountAndRecord$(\mathtt{A}_1, \mathtt{A}_2, \langle b, e\rangle)$ increments the global count c of intersections (falling inside $\langle b, e\rangle$) by the number of intersections induced by lines in $\mathtt{A}_1$ and $\mathtt{A}_2$ while recording intersections to be sampled in $\mathcal{D}_I$.

Require: $\mathtt{A}_1$ and $\mathtt{A}_2$ are sorted according to $<_e$.
Ensure: $\mathtt{A}_1$ and $\mathtt{A}_2$ are sorted according to $<_e$.

```
1:  Let i_1 := 0 and i_2 := 0.
2:  for i = 0 to length(A_1 ∪ A_2) − 1 do
3:    Let ℓ_1 := A_1[i_1] and ℓ_2 := A_2[i_2].
        {The i-th element in sorted order is ℓ_1 or ℓ_2.}
4:    if ℓ_1 <_b ℓ_2 then
5:      Let c_{i_1} := i_2.   {♯(inversions induced by ℓ_1)
                                = ♯(elements in A_2 preceding ℓ_1)}
6:      for each rank ρ in D_R ∩ [c, ..., c + c_{i_1}] do
7:        Let ℓ be the line stored at A_2[ρ − c].
8:        Update D_I to record the pair (ℓ_1, ℓ) as the
          intersection with rank ρ.
9:      end for
10:     Let c := c + c_{i_1}.        {Count intersections.}
11:     i_1 := i_1 + 1.                {Advance i_1.}
12:   else
13:     i_2 := i_2 + 1.                {Advance i_2.}
14:   end if
15: end for
```

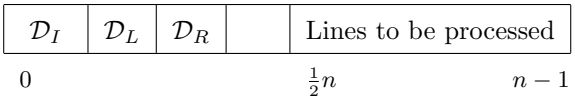
Merging Two Subarrays Into Sorted $<_e$-Order It remains to discuss the actual merging process that is required to merge $\mathtt{A}_1$ and $\mathtt{A}_2$ into sorted $<_e$-order. Since each of the subarrays is sorted according to $<_e$, a simple application of the linear-time merging algorithm of Geffert *et al.* [8] will produce the desired result. However, we also need to update the values stored in $\mathcal{D}_L$, since they reference the lines involved in the intersections found so far by directly indexing into $\mathtt{A}$. Merging $\mathtt{A}_1$ and $\mathtt{A}_2$ seems to corrupt the information recorded in $\mathcal{D}_L$, but fortunately the information of what goes where can be computed on the fly while running CountAndRecord: During the i-th iteration, this algorithm computes the element with rank i in the final sorted order (Line 3 of Algorithm 1), and thus we simply check whether the line ℓ that will be the i-th element in sorted order is involved in an intersection. In this case, we simply update the reference to ℓ to point to ℓ's position after the merge step: the i-th position in the union of $\mathtt{A}_1$ and $\mathtt{A}_2$.

We point out that we have to be very careful when maintaining (in $\mathcal{D}_I$) the references to elements stored in $\mathcal{D}_L$: with every new intersection recorded, some references may need to change their position in order to maintain the $<_b$-order. Due to space constraints, we omit the description of the update algorithm, which is summarized in the following lemma:

Lemma 3 *The global extra cost incurred by updating the references stored in the data structure $\mathcal{D}_L$ while merging subarrays is in $\mathcal{O}(r \cdot \log_2 r \cdot \log_2^2 n)$.*

3.2 Finishing Up

After we have processed the first half of the input array (using the second half to maintain the data structures $\mathcal{D}_R$, $\mathcal{D}_L$, and $\mathcal{D}_I$), we reverse the roles of the two subarrays. To do so, we first need to copy the contents of the data structures to the first half of the array. The important detail to keep in mind is that, as a result of the inversion-counting algorithm, the first half of the array is sorted according to $<_e$. Thus, when copying the contents of the data structures to the first half of the array, the order according to which we have to decide whether two lines encode a binary zero or a binary one, is the $<_e$-order.



Having run our subroutine on $\mathtt{A}[n/2, \ldots, n-1]$, we need to finalize the algorithm by processing all intersections induced by lines stored in different halves of the array. As it turns out, we do not need to actually *merge* the lines in $\mathtt{A}[0, \ldots, n/2-1]$ and $\mathtt{A}[n/2, \ldots, n-1]$—it is sufficient to count the inversions and construct the needed intersections. This means that we can simply run Algorithm 1 without the modifications needed to record the "what-goes-where" information. Since the binary encoding scheme permutes only adjacent elements, the lines used to encode the data structures $\mathcal{D}_R$, $\mathcal{D}_L$, and $\mathcal{D}_I$ are at most one position off their correct position (in sorted $<_e$-order). Thus, we can process them with constant extra (look-ahead) space.

As a result of this final invocation of CountAndRecord, the data structure $\mathcal{D}_I$ will reference r pairs of entries in $\mathcal{D}_L$ which in turn reference pairs of lines in $\mathtt{A}$. To select the two intersection points whose x-coordinates delimit the candidate strip $\langle b', e' \rangle$ that might be used during the next iteration (see [12]), we could run a selection algorithm. However, since we have chosen r small enough, we can simply sort the pairs in $\mathcal{D}_I$ according to the x-coordinate of their intersection. The running time for the sorting (including the time for resolving one level of indirection) is $\mathcal{O}(r \cdot \log_2 r \cdot \log_2 n)$.

Combining the above, Lemma 3, and the fact that $r \in \mathcal{O}(\sqrt{n})$, we obtain the following lemma:

Lemma 4 *A random sample $\mathcal{R}$ of $r = \lceil \sqrt{n} \rceil$ intersections inside a strip $\langle b, e \rangle$ can be constructed in-place and in $\mathcal{O}(n \cdot \log_2 n)$ time.*

The same algorithm can be used to explicitly construct *all* of the at most r intersection points falling inside $\langle b', e' \rangle$ in the final iteration of the slope selection algorithm. This finishes the proof of Theorem 2.

References

[1] 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. To appear.

[2] H. Brönnimann and T. M.-Y. 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.

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

[4] 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, 2004.

[5] H. Brönnimann and B. M. Chazelle. Optimal slope selection via cuttings. *Computational Geometry: Theory and Applications*, 10(1):23–29, 1998.

[6] R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Computing*, 18(4):792–810, Aug. 1989.

[7] M. B. Dillencourt, D. M. Mount, and N. S. Nethanyahu. A randomized algorithm for slope selection. *Intl. J. Computational Geometry and Applications*, 2(1):1–27, 1992.

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

[9] P. Huber. *Robust Statistics*. Wiley, 1981.

[10] M. J. Katz and M. Sharir. Optimal slope selection via expanders. *Information Processing Letters*, 47(3):115–122, 1993.

[11] J. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[12] J. Matoušek. Randomized optimal algorithm for slope selection. *Information Processing Letters*, 39(4):183–187, 1991.

[13] J. Matoušek, D. M. Mount, and N. S. Nethanyahu. Efficient randomized algorithms for the repeated median line estimator. *Algorithmica*, 20(2):136–150,1998.

[14] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.

[15] L. Shafer and W. L. Steiger. Randomizing optimal geometric algorithms. In *Proc. 5th Canadian Conf. Computational Geometry*, pp. 133–138, 1993.

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

[17] J. W. J. Williams. Algorithm 232: Heapsort. *Comm. ACM*, 7(6):347–348, 1964.