Available online at www.sciencedirect.com

ScienceDirect

Computational Geometry 38 (2007) 213–230

Computational Geometry Theory and Applications

www.elsevier.com/locate/comgeo

Line-segment intersection made in-place [☆]

Jan Vahrenhold

Universität Dortmund, Informatik XI, 44221 Dortmund, Germany

Received 10 October 2005; received in revised form 15 May 2006; accepted 14 September 2006

Available online 30 October 2006

Communicated by R. Klein

Abstract

We present a space-efficient algorithm for reporting all k intersections induced by a set of n line segments in the plane. Our algorithm is an in-place variant of Balaban's algorithm and, in the worst case, runs in $\mathcal{O}(n \log^2 n + k)$ time using $\mathcal{O}(1)$ extra words of memory in addition to the space used for the input to the algorithm.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Space-efficient algorithms; In-place algorithms; Line-segment intersection

1. Introduction

Researchers have studied space-efficient algorithms since the early 1970s. Examples include merging, (multiset) sorting, and partitioning problems; see [14,15,17,24,27]. Brönnimann et al. [7] were the first to consider space-efficient geometric algorithms and showed how to compute the convex hull of a planar set of n points in $\mathcal{O}(n \log h)$ time using $\mathcal{O}(1)$ extra space, where h denotes the size of the output. Recently, Brönnimann et al. [6] developed some space-efficient data structures and used them to solve a number of geometric problems such as convex hull, Delaunay triangulation, and nearest neighbor queries. Bose et al. [5] developed a general framework for geometric divide-and-conquer algorithms and derived space-efficient algorithms for the closest pair, bichromatic closest pair, and orthogonal line-segment intersection problems, and Chen and Chan [11] presented an algorithm for the general line-segment intersection problem: to report all k intersections induced by a set of n line segments in the plane.

1.1. The model

The goal is to design algorithms that use very little extra space in addition to the space used for the input to the algorithm. 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 $\mathcal{O}(\log n)$

[☆] An extended abstract has been presented at the 9th Workshop on Algorithms and Data Structures [J. Vahrenhold, Line-segment intersection made in-place, in: A. López-Ortiz, F. Dehne, J.-R. Sack (Eds.), Algorithms and Data Structures, 9th International Workshop, WADS 2005, in: Lecture Notes in Computer Science, vol. 3608, Springer, Berlin, 2005, pp. 146–157] but contains a flaw in the argumentation of Lemma 1. The main result, as shown in this paper, is correct. This work was done while at the University of Münster, Germany.

E-mail address: jan.vahrenhold@cs.uni-dortmund.de.

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. In certain cases, the output may be much larger than the size of the input. For example, given a set of n line segments, the number k of intersections may be as large as $\Theta(n^2)$. We consider the output memory to be write-only space that is usable for output but cannot be used as extra storage space by the algorithm. This model has been used by Chen and Chan [11] for variable size output, space-efficient algorithms and accurately models algorithms that have output streams with write-only buffer space. In the space-efficient model, an algorithm is said to work *in-place* iff it uses $\mathcal{O}(1)$ extra words of memory.

1.2. Related work

There is a large number of algorithms for the line-segment intersection problem that are not in-place, and we refer the reader to the recent survey by Mount [20]. We point out, however, that there are only two deterministic algorithms that solve the general line-segment intersection problem in optimal time $\mathcal{O}(n \log n + k)$: The first such algorithm, proposed by Chazelle and Edelsbrunner [9], has a space requirement of $\mathcal{O}(n + k)$ which is prohibitive in an in-place setting even if the constant hidden in the “Big-Oh”-notation is $c = 1$. The second algorithm, proposed by Balaban [1], has an $\mathcal{O}(n)$ space requirement and will be the starting point for our in-place approach. Thus, we will discuss it below in more detail.

In the space-efficient model of computation, Bose et al. [5] have presented an optimal in-place algorithm for the restricted setting when the input consists of only horizontal and vertical segments. Their algorithm runs in $\mathcal{O}(n \log n + k)$ time and uses $\mathcal{O}(1)$ words of extra memory. Chen and Chan [11] modified the well-known algorithm of Bentley and Ottmann [2] and obtained a space-efficient algorithm that runs in $\mathcal{O}((n + k) \log^2 n)$ time and uses $\mathcal{O}(\log^2 n)$ extra words of memory. If the model is changed such that the input can be destroyed irrevocably, the bounds can be improved to $\mathcal{O}((n + k) \log n)$ time and $\mathcal{O}(1)$ extra space.[1] We will improve these bounds to $\mathcal{O}(n \log^2 n + k)$ time and $\mathcal{O}(1)$ extra space thus making the algorithm in-place and establishing an optimal linear dependency on the number k of intersections reported.

1.3. In-place algorithmic tools

The philosophy of our approach is to identify building blocks of the original algorithm and to try to replace them by in-place counterparts wherever possible. We now summarize the main algorithmic ingredients that will be used (note that, since we are working in an in-place setting, the input elements always reside in an array):

Sorting: Sorting n items can be done in-place and in optimal time $\mathcal{O}(n \log n)$ using, e.g., HEAPSORT [27]. There are also in-place, asymptotically optimal algorithms for stably sorting a multiset using constant extra space [18,22].

Priority Queue: A priority queue for n items may be realized in-place using a heap data structure [27]. Each `insert`-, `min`-, and `deleteMin`-operation takes time $\mathcal{O}(\log n)$.

Interchanging adjacent blocks: Given two adjacent subarrays $\mathtt{A}[x_0, \ldots, x_1 - 1]$ and $\mathtt{A}[x_1, \ldots, x_2 - 1]$, these blocks can be interchanged in linear time. The in-place algorithm INPLACE_INTERCHANGE first uses swaps to reverse the order of the elements in each of the blocks separately and then reverses the order of $\mathtt{A}[x_0, \ldots, x_2 - 1]$ again using swaps. Katajainen and Pasanen [17] attribute this algorithm to “computer folklore”.

Merging: Given two arrays A and B that are sorted according to the same total order, an in-place algorithm INPLACE_MERGE replaces A and B by one single sorted array L using only linear time. Merging two sorted sets had been identified as a key ingredient for in-place sorting, and thus a considerable number of approaches have been taken (see [12] and the references therein). As of today, the most practical, yet asymptotically optimal, in-place approaches seem to be the linear-time, non-stable algorithm of Chen [13] and Chen’s optimization [12] of a linear-time, stable algorithm due to Geffert et al. [15].

(*Stable*) *Unmerging*: Given a sorted array $\mathtt{A}[x_0, \ldots, x_2 - 1]$ and a $\{0, 1\}$-valued predicate π, the in-place algorithm INPLACE_UNMERGE of Salowe and Steiger [24] stably partitions a sorted array $\mathtt{A}[x_0, \ldots, x_2 - 1]$ into two sorted

[1] In this model, there is no way of retrieving the original data after (or even during) the run of the algorithm. Furthermore, Chen and Chan observe that, due to the use of division, their implementation “is not guaranteed to be robust” [11].

subarrays $\mathtt{A}[x_0, \ldots, x_1 - 1]$ and $\mathtt{A}[x_1, \ldots, x_2 - 1]$ such that $\pi(x) = 0$ for all $x \in \mathtt{A}[x_0, \ldots, x_1 - 1]$ and $\pi(x) = 1$ for all $x \in \mathtt{A}[x_1, \ldots, x_2 - 1]$. This algorithm runs in linear time.[2]

Sorted Subset Selection: Given an array $\mathtt{A}[x_0, \ldots, x_2 - 1]$ that is sorted according to some total order $<_{\mathtt{A}}$ and a $\{0, 1\}$-valued predicate π, the linear-time, in-place algorithm SORTEDSUBSETSELECTION of Bose et al. [5] stably moves all $x \in \mathtt{A}[x_0, \ldots, x_2 - 1]$ for which $\pi(x) = 0$ to the front of $\mathtt{A}[x_0, \ldots, x_2 - 1]$. In contrast to the INPLACE_UNMERGE algorithm, the values of π need not to be known in advance but may be computed during the algorithm. The effects of SORTEDSUBSETSELECTION can be undone in-place and in linear time by essentially running the algorithm backwards. More specifically, the UNDOSORTEDSUBSETSELECTION uses only the information how many elements have been selected and the fact that $<_{\mathtt{A}}$ is a total order, i.e., that any two elements are directly comparable. It should be pointed out that, to reconstruct the original sorted order of $\mathtt{A}[x_0, \ldots, x_2 - 1]$, the function π is not required at all. This implies that the value $\pi(x)$ that is used during SORTEDSUBSETSELECTION may also depend on additional information (such as global variables or information about the position of x in the $<_{\mathtt{A}}$-order) without affecting the correctness of UNDOSORTEDSUBSETSELECTION.

To simplify the exposition, we will from now on refer to these in-place algorithms by their short names, e.g., HEAPSORT, INPLACE_INTERCHANGE, or SORTEDSUBSETSELECTION.

2. The algorithm

Our algorithm is an in-place version of the optimal $\mathcal{O}(n \log n + k)$ algorithm proposed by Balaban [1]. Balaban obtained this complexity by first developing an intermediate algorithm with running time $\mathcal{O}(n \log^2 n + k)$ and then applying the well-known concept of *fractional cascading* [10]. As fractional cascading relies on explicitly maintained copies of certain elements (or, in the case of Balaban's algorithm, pointers to such elements), this concept can only be applied with $\mathcal{O}(n)$ extra space which is prohibitive for an in-place algorithm. Thus, we build upon the (suboptimal) intermediate algorithm.

Balaban's intermediate algorithm is a clever combination of plane-sweeping and divide-and-conquer; the plane is subdivided into two vertical strips each containing the same number of segment endpoints, and each strip is (recursively) processed from left to right—see Fig. 1.

While doing so, the algorithm maintains the following pre- and postconditions:

Precondition Γ_{pre}: Prior to processing a strip, all segments crossing the left strip boundary are vertically ordered at the x-coordinate of the left strip boundary. The algorithm may store this set of segments in several groups; in this case, the segments are required to be ordered only within each group.

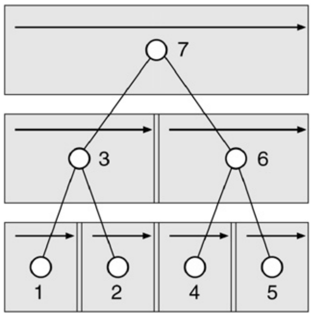


Fig. 1. Processing the recursion tree. Numbers indicate the order in which the strips are *finished*.

[2] It should be noted that this algorithm relies on the fact that each element can be classified in constant time according to the value of π. Usually, this implies that the elements *explicitly* maintain an extra bit (thus requiring extra space). For the special scenario where we will employ this unmerging algorithm, it is possible to avoid this explicit labeling. We will discuss this in more detail in Section 2.3.

Postcondition Γ_{post}: After having processed a strip, the segments crossing the right strip boundary are rearranged such that they are vertically ordered at the x-coordinate of the right strip boundary. Again, the algorithm may store this set of segments in several groups; in this case, the segments are required to be ordered only within each group.

The recursion base case is the case when a set $\mathcal{L}$ of line segments spans a vertical strip $\langle b, e\rangle := [b, e] \times \mathbb{R}$ that does not contain any endpoint of a segment. Precondition Γ_{pre} implies that this set of segments is sorted according to $<_b$, the vertical order at x-coordinate b.

In the following subsections, we will demonstrate how the recursive calls and the partitioning of the segments to be processed can be realized using only $\mathcal{O}(1)$ extra space. A main result of Balaban's analysis is that the overall number of (occurrences of all) segments participating in all recursive calls is $\mathcal{O}(n \log n + k)$. This fact, formalized in Section 2.8 as Theorem 1, implies that all algorithmic subroutines that exhibit a superlinear running time will have to be accounted for separately whereas all linear-time subroutines contribute at most $\mathcal{O}(n \log n + k)$ time to the overall complexity. We will start our discussion by presenting an important concept that allows intersection-finding by merging and binary search.

2.1. Intermezzo: Using staircases

A main algorithmic ingredient of Balaban's algorithm is to find intersections by synchronized scans over two sorted sets of segments. To formalize the terminology, Balaban defines the concept of staircases:

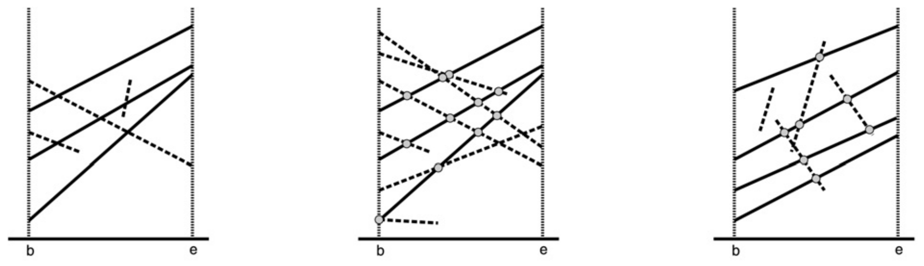
Definition 1. (Balaban [1]) Let $b \leqslant e$ be two real numbers and define $\langle b, e\rangle := [b, e] \times \mathbb{R}$ to be the vertical strip encompassing $[b, e]$. An ordered set $\mathcal{Q} = (s_1, \dots, s_m)$ of segments is called a *staircase* iff the following properties hold:

(i) Each segment $s \in \{s_1, \dots, s_m\}$ spans the strip $\langle b, e\rangle$.
(ii) No two segments $s, t \in \{s_1, \dots, s_m\}$ intersect inside the strip $\langle b, e\rangle$ unless they are identical.
(iii) The segments in $\mathcal{Q}$ are sorted according to the vertical order at $x = b$.

These properties imply that all segments are sorted according to the vertical order at any $x' \in [b, e]$.

A staircase $\mathcal{Q}$ is said to be *complete relative to a strip* $\langle b, e\rangle$ and a set $\mathcal{S}$ of segments intersecting $\langle b, e\rangle$ iff no segment from $\mathcal{S}$ can be added to $\mathcal{Q}$ without violating Properties 1 or 2. The segments of a staircase are referred to as *stairs*.

As usual, the "vertical order at $x = b$" breaks ties by ordering segments with the same y-coordinate at $x = b$ by increasing slope and (in the case of identical slope) by the x-coordinates of the right and left endpoint. Note that the above definition allows two stairs to intersect on one slab boundary, i.e., at $x = b$ or at $x = e$, but not on both *unless* they are identical. We will come back to the question of how to report these intersections later. Whenever the strip for which a staircase is defined is apparent from the context, we will not explicitly refer to it. An example of a staircase is



(a) Complete staircase. (b) Merging segments. (c) Using binary search.

Fig. 2. Examples of a staircase and its use in intersection-finding. Stairs are drawn with solid lines, and segments not belonging to a staircase are drawn with dashed lines.

```
Require: $\mathtt{A}[\ell_b, \ldots, \ell_c - 1]$ is a staircase for $\langle b, e\rangle$, $\mathtt{A}[\ell_c, \ldots, \ell_e - 1]$ is sorted in $<_\xi$-order, $\xi \in [b, e]$.
 1: $\ell' := \ell_b$;
 2: for $\ell := \ell_c$ to $\ell_e - 1$ do
 3:   while $\ell' < \ell_c$ and $\mathtt{A}[\ell'] <_\xi \mathtt{A}[\ell]$ do
 4:     $\ell' := \ell' + 1$;                                  {Find highest stair below $\mathtt{A}[\ell]$.}
 5:   if $\mathtt{A}[\ell]$ intersects $\mathtt{A}[\ell' - 1]$ inside $\langle b, e\rangle$ or at $x = b$ then
 6:     $\ell'' := \ell' - 1$;                                 {$\mathtt{A}[\ell]$ intersects stairs below.}
 7:     while $\ell'' \geqslant \ell_b$ do
 8:       if $\mathtt{A}[\ell]$ intersects $\mathtt{A}[\ell'']$ inside $\langle b, e\rangle$ or at $x = b$ then
 9:         Report intersection between $\mathtt{A}[\ell]$ and $\mathtt{A}[\ell'']$.
10:       $\ell'' := \ell'' - 1$;                              {Scan backwards in $\mathtt{A}[\ell_b, \ldots, \ell' - 1]$.}
11:   else if $\ell' < \ell_c$ and $\mathtt{A}[\ell]$ intersects $\mathtt{A}[\ell' - 1]$ inside $\langle b, e\rangle$ or at $x = b$ then
12:     $\ell'' := \ell'$;                                     {$\mathtt{A}[\ell]$ intersects stairs above.}
13:     while $\ell'' < \ell_c$ do
14:       if $\mathtt{A}[\ell]$ intersects $\mathtt{A}[\ell'']$ inside $\langle b, e\rangle$ then
15:         Report intersection between $\mathtt{A}[\ell]$ and $\mathtt{A}[\ell'']$.
16:       $\ell'' := \ell'' + 1$;                              {Scan forward in $\mathtt{A}[\ell', \ldots, \ell_c - 1]$.}
```

Algorithm 1. Algorithm MERGEINT$_\xi(\mathtt{A}, b, e, \ell_b, \ell_c, \ell_e)$.

```
Require: $\mathtt{A}[\ell_b, \ldots, \ell_b - 1]$ is a staircase for $\langle b, e\rangle$; $\mathtt{A}[\ell]$ is a segment whose endpoints lie inside $\langle b, e\rangle$.
1: $p :=$ lower endpoint of $\mathtt{A}[\ell]$.
2: $q :=$ upper endpoint of $\mathtt{A}[\ell]$.
3: Use binary search in $\mathtt{A}[\ell_b, \ldots, \ell_b - 1]$ to find the index $i$ such that $\mathtt{A}[i]$ is the lowest segment above $p$. If no such index exists, set $i := \ell_b$.
4: if $i < \ell_b$ then
5:   Use binary search in $\mathtt{A}[i, \ldots, \ell_b - 1]$ to find the index $j$ such that $\mathtt{A}[j]$ is the lowest segment above $q$. If no such index exists, set $j := \ell_b$.
6:   for $\ell' := i$ to $j - 1$ do
7:     Report intersection of $\mathtt{A}[\ell]$ and $\mathtt{A}[\ell']$.
8: else
9:   Do nothing.                                               {$s$ lies above $\mathtt{A}[\ell_c - 1]$.}
```

Algorithm 2. Algorithm SEARCHINT$(\mathtt{A}, b, e, \ell_b, \ell_e, \ell)$.

given in Fig. 2(a); the stairs are drawn with solid lines. Note that no dashed segment may be added without violating the properties of a staircase.

Figs. 2(b) and (c) show the use of staircases for intersection-finding. Assume we want to find, inside a strip $\langle b, e\rangle$, the intersections between segments in a staircase $\mathcal{Q}$ and another set of segments $\mathcal{L}'$. If, as in Fig. 2(b), the segments in $\mathcal{L}'$ cross the strip's left boundary and thus can be ordered by the y-coordinate of their intersection point with this boundary, we can find all intersections between segments in $\mathcal{Q}$ and $\mathcal{L}'$ using a synchronized merge-like scan over both ordered sets. These intersections are marked as hollow discs. The running time of such a scan while reporting κ intersections is $\mathcal{O}(|\mathcal{Q}| + |\mathcal{L}'| + \kappa)$, and we will refer to this subroutine as MERGEINT$_\xi$, where ξ the x-coordinate at which the total order is computed (in the above case, ξ would be the x-coordinate of the strip's left boundary). This routine will not only report intersections *inside* $\langle b, e\rangle$ but also intersections that coincide with the left strip boundary. The code of this subroutine, given below as Algorithm 1, is easily seen to be implementable such that the resulting algorithm runs in-place. In this implementation the staircase $\mathcal{Q}$ and the set $\mathcal{L}'$ are presented in-place as $\mathcal{Q} = \mathtt{A}[\ell_b, \ldots, \ell_c - 1]$ and $\mathcal{L}' = \mathtt{A}[\ell_c, \ldots, \ell_e - 1]$.

Let us now assume we want to find, inside a strip $\langle b, e\rangle$, the intersections between segments in a staircase $\mathcal{Q}$ and an *unordered* set $\mathcal{L}'$ of segments each of which has both endpoints inside $\langle b, e\rangle$—see Fig. 2(c). As the segments of a staircase do not induce intersections with each other and span the strip, we can use binary search to locate for each segment $s \in \mathcal{L}'$ the segment of $\mathcal{Q}$ above or below one of its endpoints. Starting from here, we can trace s through the staircase reporting all intersections with stairs of $\mathcal{Q}$. The running time of this subroutine, referred to as SEARCHINT,

```
Require: $\mathcal{L} = (s_1, \ldots, s_m)$ is ordered by $<_b$.
Ensure: $\mathcal{L}'$ and $\mathcal{Q}$ are ordered by $<_b$; $\mathcal{Q}$ is complete relative to $\langle b, e\rangle$.
1: $\mathcal{Q} := \emptyset$; $\mathcal{L}' := \emptyset$;
2: for $j = 1$ to $m$ do
3:   if $s_j$ spans $\langle b, e\rangle$ and does not intersect the last segment of $\mathcal{Q}$ within $\langle b, e\rangle$ then
4:     $\mathcal{Q} \leftarrow s_j$;
5:   else
6:     $\mathcal{L}' \leftarrow s_j$;
7: Scan though $\mathcal{Q}$ and report intersections at $x = b$.
                {Stairs may intersect at either $x = b$ or $x = e$, but intersections are only reported at $x = b$.}
8: return $(\mathcal{Q}, \mathcal{L}')$;
```

Algorithm 3. The algorithm $\text{SPLIT}_{b,e}(\mathcal{L})$ [1].

is $\mathcal{O}(|\mathcal{L}'| \log |\mathcal{Q}| + \kappa)$ where κ is the number of intersections reported.[3] Again, the code (given below as Algorithm 2) can implemented in such a way that the resulting algorithm runs in-place.

2.2. *The recursion base case*

As mentioned above, the recursion base case is the case when a set $\mathcal{L}$ of line segments spans a vertical strip $\langle b, e\rangle := [b, e] \times \mathbb{R}$ that does not contain any endpoint of a segment. Before discussing the algorithm for this case in detail, we will briefly comment on an algorithmic subroutine that is used throughout the algorithm (and also in the algorithm for the base case) to construct a staircase for a sorted set of segments.

The set $\mathcal{L}$ of segments is a group of segments crossing the left boundary of a strip, and thus, by Precondition Γ_{pre}, it is sorted according to $<_b$, the vertical order at x-coordinate b. Algorithm 3 partitions $\mathcal{L}$ into a staircase $\mathcal{Q}$ and a set $\mathcal{L}' = \mathcal{L} \setminus \mathcal{Q}$ such that both sets are sorted according to $<_b$, and that $\mathcal{Q}$ is complete relative to $\langle b, e\rangle$. (This partitioning reflects the situation mentioned in the pre- and the postcondition where the algorithm is allowed to have more than one ordered set of segments crossing a strip's boundary.) Also, this algorithm reports all stairs that intersect on the left boundary of $\langle b, e\rangle$.

The resulting staircase $\mathcal{Q}$ can be verified to be maximal using the following line of reasoning: If the staircase is not maximal, there is at least one segment $s \in \mathcal{L}'$ spanning $\langle b, e\rangle$ that could be added to $\mathcal{Q}$ without intersecting *any* segment in $\mathcal{Q}$. In particular, this segment s would intersect the segment s' that was the topmost segment in $\mathcal{Q}$ at the time the segment s had been considered for addition to $\mathcal{Q}$. This would contradict the outcome of the test in line 3 of Algorithm 3 that caused s *not* to added to $\mathcal{Q}$. As we will see below, the charging scheme used for analyzing the complexity of the global algorithm will only rely on each staircase being *maximal*, so there is no need to actually require that the staircase constructed is a *maximum* staircase.

Balaban, in his original paper [1], gave an Algorithm $\text{SEARCHINSTRIP}_{b,e}(\mathcal{L})$ that recursively uses SPLIT to partition a set $\mathcal{L}$ of segments spanning $\langle b, e\rangle$ and sorted by $<_b$ such that the set $\text{Int}_{b,e}(\mathcal{L})$ of intersections induced by $\mathcal{L}$ and falling into the strip $\langle b, e\rangle$ can be found easily using a synchronized scan over the staircase $\mathcal{Q}$ and the set $\mathcal{L}'$, both of which are ordered by $<_b$. As a side effect, this algorithm reorders the segments in $\mathcal{L}$ such that they are sorted according to $<_e$. This implies that, in the process of sweeping the plane, $\mathcal{L}$ can be used as the input for processing an adjacent strip $\langle b', e'\rangle$, i.e., a strip $\langle b', e'\rangle$ for which $b' = e$.

2.2.1. *An in-place algorithm for the recursion base case*

While this algorithm can be made in-place with relatively little effort (see the conference version of this paper [26]), we can also follow an observation due to Boissonnat and Snoeyink [3] who pointed out that, in the base case, we are computing a portion of an arrangement of lines. Exploiting the well-known correspondence between the intersections induced by a set of segments spanning $\langle b, e\rangle$ and the inversions between the permutations that give the vertical order of the segments at $x = b$ and $x = e$ [19], Algorithm $\text{INPLACE\_SEARCHINSTRIP}_{b,e}(\mathcal{L})$ thus can be implemented using

[3] It should be noted that this part of the algorithm is the sole reason for why Balaban's intermediate algorithm has a slightly suboptimal running time of $\mathcal{O}(n \log^2 n + k)$. Balaban resolves this problem by the use of fractional cascading, but (as noted before) the use of this technique is prohibitive in an in-place setting.

an insertion-sort based algorithm for constructing inversions, see, e.g. [8, Lemma 1]. This implementation is easily seen to be in-place, and its running time is linear in number $|\mathcal{L}| + |\mathrm{Int}_{b,e}(\mathcal{L})|$, that is, linear in the number of segments plus the number of intersections reported. This matches the running time of Balaban's subroutine.

2.3. Separating stairs from non-stairs in linear time

The simplified algorithm for the recursion base case given at the end of the previous section could avoid assuming the existence of a linear-time, in-place algorithm for partitioning $\mathcal{L}$ into $\mathcal{Q} + \mathcal{L}'$ while maintaining both resulting sets in sorted order. During the execution of the overall algorithm, however, we need to have access to such an algorithm, and we will present such an algorithm in this section.

In order to obtain a partition of $\mathcal{L}$ into $\mathcal{Q} + \mathcal{L}'$ where both sets are in sorted order, we could proceed as follows: we could implement an algorithm called INPLACE_SPLITANDSORT such as to use the approach of the SORTEDSUBSETSELECTION subroutine [5]. This algorithm would stably move the non-stairs, i.e., the set $\mathcal{L}'$ to the front of (the subarray holding) $\mathcal{L}$. Using the incremental SORTEDSUBSETSELECTION would be feasible as we would only need to keep track of the (position of the) topmost stair in order to decide whether the next segment in question can be added to the staircase or not. We then would sort the segments in $\mathcal{Q}$ using an optimal in-place sorting algorithm and would stably interchange the blocks holding $\mathcal{Q}$ and $\mathcal{L}'$ such that the stairs would appear in the front of the subarray. The running time of the resulting algorithm would then be summarized as follows:

Lemma 1. *The total time of Algorithm* INPLACE_SPLITANDSORT, *when run over all strips, is* $\mathcal{O}(n\log^2 n + k\log n)$.

Proof. Selecting the non-stairs can be done in-place in linear time using SORTEDSUBSETSELECTION, and each segment not added to a staircase and thus considered in another pass can be charged to (at least) one intersection with the topmost stair. Each segment appears in a exactly one staircase per invocation of INPLACE_SPLITANDSORT. However, as the overall number of segment appearances is $\mathcal{O}(n\log n + k)$ (see Theorem 2 in [1]), the global running time of sorting all staircases is in $\mathcal{O}((n\log n + k)\log(n\log n + k)) \subset \mathcal{O}(n\log^2 n + k\log n)$. □

Besides the fact that using this subroutine would introduce an undesirable extra logarithmic factor in the output-sensitive part of the algorithm, a potential drawback is that only the non-stairs would be selected stably. Both issues can be resolved if we use a linear-time, stable unmerging algorithm. However, as the INPLACE_UNMERGE algorithm of Salowe and Steiger [24] technically requires an extra bit for encoding which elements belong to which set, we need to be very carefully not to violate the requirement that no extra space in addition to the geometric description of each segment may be used.

A standard technique in the design of in-place algorithms is to encode a bit by the relative order of two comparable elements [21]. Assuming that no segment degenerates to a point, the two endpoints of a segment are distinct and comparable according to their lexicographical order. Exploiting this fact, we can implicitly encode a $\{0, 1\}$-labeling of the segments by temporarily reversing the order of the endpoints as needed, and we derive a linear-time implementation of Algorithm INPLACE_SPLIT:

Lemma 2. *If the order of the endpoints of the input segments may be reversed temporarily, Algorithm* INPLACE_SPLIT *can be realized such that it runs in time* $\mathcal{O}(|\mathcal{L}|)$.

Proof. The implementation of the subroutine INPLACE_SPLIT first runs a variant of SORTEDSUBSETSELECTION in which no segments are extracted. Instead, the algorithm keeps track of the (index of) the topmost stair and reverses the order of the segments not belonging to the staircase. Using this implicitly encoded $\{0, 1\}$-labeling, the INPLACE_UNMERGE subroutine [24] can partition $\mathcal{L}$ into $\mathcal{Q}$ and $\mathcal{L}'$. After this unmerging, all segments in $\mathcal{L}'$ are re-reversed to restore the original order of the endpoints. As all subroutines employed run in-place and in linear time, the correctness of the lemma follows. □

Even though we are not aware of any impact of this temporary modification on the intrinsic complexity of intersection-finding, one might find it irritating not to consider the input objects as atomic. Also, one might consider a setting where the segments are stored in read-only memory and where the algorithm can only access and modify an "address translation table" of pointers referencing these segments. In this case and in the presence of degenerated segments, the above approach collapses. As it turns out, we can directly modify the algorithm of Salowe and Steiger [24] to separate stairs from non-stairs, and doing so, we obtain the final version of the algorithm needed to process the base case.

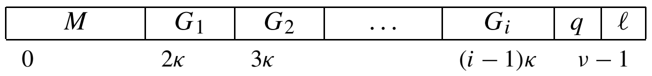
Lemma 3. *Algorithm* InPlace_Split *can be realized such that it runs in time* $\mathcal{O}(|\mathcal{L}|)$.

To obtain this result we use the algorithm implied by the proof of the following lemma:

Lemma 4. *Let $\mathcal{L}$ be a set of segments that cross the left boundary of a strip $\langle b, e\rangle$. If $\mathcal{L}$ is sorted according to $<_b$, we can unmerge $\mathcal{L}$ into a maximal set of staircases $\mathcal{Q}$ and $\mathcal{L}' := \mathcal{L} \setminus \mathcal{Q}$ such that both and $\mathcal{Q}$ and $\mathcal{L}'$ remain sorted. The algorithm runs in-place and in $\mathcal{O}(|\mathcal{L}|)$ time.*

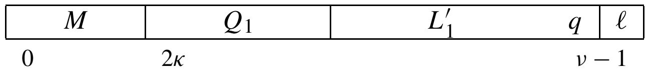
Proof. The proof consists of an explanation of how to modify the unmerging algorithm of Salowe and Steiger [24] to directly work with line segments. For the brevity of explanation, define $\nu := |\mathcal{L}|$ and $\kappa := \lceil\sqrt{\nu}\,\rceil$ and assume w.l.o.g. that the subarray in which $\mathcal{L}$ is stored is $\mathtt{A}[0, \ldots, \nu - 1]$.

The main idea of the algorithm of Salowe and Steiger is to use the first 2κ entries of $\mathtt{A}$ as a temporary buffer for rearranging the rest of the array. After the first partitioning phase the subarray looks as follows:

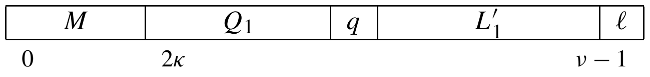
After Phase 1:


The set M stored at the beginning of $\mathtt{A}$ contains the smallest 2κ elements from $\mathcal{L}$, i.e., both stairs and non-stairs in arbitrary order. Each group G_i consists of exactly κ elements from either $\mathcal{Q}$ or $\mathcal{L}'$ in sorted order and, if G_{i_1} and G_{i_2} contain elements from the same subset ($\mathcal{Q}$ or $\mathcal{L}'$), the elements in G_{i_1} are smaller than the elements in G_{i_2} iff i_1 is less than i_2. The groups q and ℓ are potentially empty and contain the largest stairs and non-stairs in sorted order.

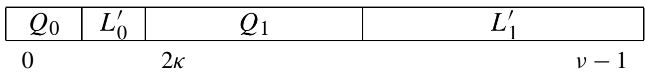
During the second phase of the algorithm, the groups $G_1, \ldots, G_i$ are rearranged such that all stairs appear consecutively and all non-stairs appear consecutively.

After Phase 2:


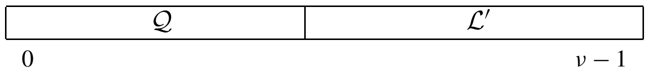
Next, the sets q and ℓ are swapped into their correct position.

After Phase 3:


During the fourth phase of the algorithm, the set M is separated into a sorted set Q_0 of stairs and a sorted set L'_0 of non-stairs.

After Phase 4:


After the fifth and final phase, these blocks have been rearranged to form the desired partition of $\mathcal{L}$ into $\mathcal{Q} := Q_0 \,\&\, Q_1$ and $\mathcal{L}' := L'_0 \,\&\, L'_1$ (the operator "&" denotes concatenation).

After Phase 5:


We will now describe the details and the running time each of these five phases separately. Clearly, the third and the fifth phase can be performed in-place and in linear time as both phases can be realized using a constant number of invocations of the InPlace_Interchange subroutine. The additional space requirement is constant, as all groups except for the groups q and ℓ contain exactly κ elements.

However, as we are working with blocks of stairs and non-stairs instead of a set of typed items from an ordered universe, we need to be a bit more careful with implementing the remaining phases. Recall that all segments in the same group are sorted according to $<_b$ and that the defining property of stairs is that a stair completely crosses the strip and does not intersect any other stair. A staircase is complete (relative to $\langle b, e\rangle$) if no segment may be added without violating these properties. Consequently, if we scan the segments in sorted order and maintain a pointer τ to the topmost stair seen so far, we can classify each element as a stair (and update τ accordingly) or as a non-stair.

The algorithm for the first phase starts at the index 2κ and scans forward until the end of the array A. The algorithm swaps all stairs into the first block of the array, i.e., into $\mathtt{A}[0,\dots,\kappa-1]$, and all non-stairs into the second block of the array, i.e., into $\mathtt{A}[\kappa,\dots,2\kappa-1]$. As soon as either of these blocks is full, say the block $\mathtt{A}[\kappa,\dots,2\kappa-1]$ of non-stairs, we increment the number g of groups constructed so far and swap this full block to the beginning of the subarray $\mathtt{A}[(2+g)\kappa,\dots,\nu-1]$. Then we continue, reusing the subarray $\mathtt{A}[\kappa,\dots,2\kappa-1]$ for the next group of non-stairs. Iterating this way until the end of the array has been reached, the algorithm always swaps elements in one group with a subset of the elements that originally constituted the subarray $\mathtt{A}[0,\dots,\kappa-1]$. Since each element is swapped into $\mathtt{A}[0,\dots,\kappa-1]$ at most once, as each block swap takes $\mathcal{O}(\kappa)$ time, and as there are at most $\mathcal{O}(\kappa)$ such block swaps (each swap moves one group into its final position), the running time of this phase is $\mathcal{O}(\kappa^2)=\mathcal{O}(\nu)$.

The above description implicitly assumes that we know the (index of the) topmost stair in $\mathtt{A}[0,\dots,2\kappa-1]$—or have access to a special, constant-size location where we store (a copy of) the topmost stair. Hence, prior to starting the first phase, we scan over the subarray $\mathtt{A}[0,\dots,2\kappa-1]$ and simulate running Balaban's SPLIT algorithm. We do so without actually permuting the stairs but instead only keeping track of the topmost stair seen so far. After we have done this, we are ready to start with the first phase. We also need to keep track of this information for running the next phase.

The second phase, i.e., regrouping the presorted groups G_i into two groups Q_1 and L_1' can be realized by performing an in-place sort of the blocks [24]. To regroup the blocks, we first perform a selection-sort type algorithm to move the groups G_i constituting the set Q_1 to the front of $\mathtt{A}[2\kappa,\dots,\nu-1]$. In the i—the iteration of this algorithm, let g_1 be the number of groups of stairs already moved to the front of $\mathtt{A}[2\kappa,\dots,\nu-1]$. Since all groups except for the groups q and ℓ contain exactly κ elements, and since q contains the largest stairs, in each iteration of the selection sort, we know that the smallest stairs identified so far are stored in $\mathtt{A}[2\kappa,\dots,(2+g_1)\kappa-1]$. During an iteration, we look at every κ-th index (starting at index $(2+g_1)\kappa$) to find the smallest segment s_{g_i} not intersecting the topmost stair seen so far (which is stored at index $(2+g_i)\kappa-1$). Since all elements inside a group are sorted and since the staircase to be constructed is complete (relative to $\langle b, e\rangle$), we know that the segment s_{g_i} is the start of the next group of stairs to be moved to the front. We then interchange (using $\mathcal{O}(\kappa)$ swaps and employing the block $\mathtt{A}[0,\dots,\kappa-2]$ as an intermediate buffer) this block with the block starting at index $(2+g_i)\kappa$.

This first part of the second phase of the algorithm runs in-place and performs $\mathcal{O}(\kappa^2)$ comparisons and $\mathcal{O}(\kappa)$ block swaps. As each of the block swaps takes $\mathcal{O}(\kappa)$ time, the overall cost for selecting the set Q_1 is $\mathcal{O}(\kappa^2)=\mathcal{O}(\nu)$. For the second part of the second phase, grouping L_1' in *sorted* order, we run the classic selection sort algorithm on the blocks where the key of a block is the $<_b$-value of its smallest segment. This part again runs in $\mathcal{O}(\kappa^2)=\mathcal{O}(\nu)$ time.

A close look at the fourth phase of the algorithm reveals that we can run exactly the same in-place algorithm again but this time exchanging single elements instead of blocks. As we are working on a set of 2κ elements, the running time again is $\mathcal{O}(\kappa^2)=\mathcal{O}(\nu)$.

Summing up the above discussion, we conclude that the algorithm of Salowe and Steiger [24] can be implemented to directly extract a complete staircase from a set of sorted segments. Following the original analysis, we see that this algorithm runs in-place and in linear time. □

2.4. The "divide" and "conquer" phases

When analyzing the complexity and the correctness of Balaban's algorithm, it is helpful to keep in mind that this algorithm reports a pair (s,t) of intersecting segments when processing the unique strip $\langle b, e\rangle$ where one of the segments, say s, is part of the staircase $\mathcal{Q}$ spanning the strip $\langle b, e\rangle$ and where the intersection point lies within $\langle b, e\rangle$ or on the left boundary of $\langle b, e\rangle$.[4] The case of stairs intersecting on the left boundary has been handled by SPLIT, so we can assume that the other segment t is not part of the staircase $\mathcal{Q}$ because segments in the same staircase do not intersect inside $\langle b, e\rangle$. There are three possible situations: (1) t crosses the left boundary of $\langle b, e\rangle$, (2) t lies

[4] Since the algorithm sweeps all slabs from left to right, the right boundary of the strip $\langle b, c\rangle$ is handled as it appears as the left boundary of the adjacent strip $\langle c, e\rangle$. See Appendix A for a description of how to handle intersections on the right boundary of the initial strip.

```
Require: $\mathcal{L}$ is ordered by $<_b$, each segment in $\mathcal{I}$ has both endpoints inside $\langle b, e\rangle$.
Ensure: $\mathcal{R}$ is ordered by $<_e$.
 1: if $e - b = 1$, i.e., if $\langle b, e\rangle$ does not contain an endpoint then
 2:   $\mathcal{R} := \textsc{SearchInStrip}_{b,e}(\mathcal{L})$;                              {Base of recursion.}
 3: else
 4:   $(\mathcal{Q}, \mathcal{L}_{left}) := \textsc{Split}_{b,e}(\mathcal{L})$;                  {Compute staircase.}
 5:   Compute $\mathrm{Int}_{b,e}(\mathcal{Q}, \mathcal{L}_{left})$ using $\textsc{MergeInt}_b$.          {Handle Situation (1).}
 6:   $c := \lfloor (b+e)/2 \rfloor$;                                     {Find endpoint with median rank.}
 7:   $\mathcal{I}_{left} := \{s \in \mathcal{I} \mid s$ lies completely inside $\langle b, c\rangle\}$;
 8:   $\mathcal{R}_{left} := \textsc{TreeSearch}(\mathcal{L}_{left}, \mathcal{I}_{left}, b, c)$;
 9:   Construct $\mathcal{L}_{right}$ from $\mathcal{R}_{left}$ by insertion/deletion of the segment whose endpoint corresponds to $c$.
10:   $\mathcal{I}_{right} := \{s \in \mathcal{I} \mid s$ lies completely inside $\langle c, e\rangle\}$;
11:   $\mathcal{R}_{right} := \textsc{TreeSearch}(\mathcal{L}_{right}, \mathcal{I}_{right}, c, e)$;
12:   Compute $\mathrm{Int}_{b,e}(\mathcal{Q}, \mathcal{R}_{right})$ using $\textsc{MergeInt}_e$.         {Handle Situation (3).}
13:   for each segment $s \in \mathcal{I}$
14:     Compute $\mathrm{Int}(\mathcal{Q}, \{s\})$ using $\textsc{SearchInt}$.                  {Handle Situation (2).}
15:   $\mathcal{R} := \textsc{Merge}_e(\mathcal{Q}, \mathcal{R}_{right})$;                     {Establish Postcondition $\Gamma_{\mathrm{post}}$.}
16: return $\mathcal{R}$;
```

Algorithm 4. The algorithm TreeSearch($\mathcal{L}, \mathcal{I}, b, e$) [1].

completely within $\langle b, e\rangle$, or (3) t crosses the right boundary of $\langle b, e\rangle$. Note that there might be segments appearing both in Situation (1) and Situation (3); we can detect (and skip) those segments when handling Situation (3) because these segments are exactly the segments crossing both strip boundaries. Thus, we make sure that no endpoint is reported twice.

Precondition Γ_{pre} implies that, when processing a strip $\langle b, e\rangle$, all segments intersecting the left boundary of the strip $\langle b, e\rangle$ are available in the form of an ordered set $\mathcal{L}$ that is sorted according to $<_b$. Similarly, Postcondition Γ_{post} requires the existence of an ordered set $\mathcal{R}$ (which is the set returned by Balaban's algorithm (Algorithm 4)) that contains the segments crossing the right strip boundary—again in sorted order. The unordered set $\mathcal{I}$ contains all segments that lie completely within the strip $\langle b, e\rangle$. Handling Situations (1)–(3) then consists of computing $\mathrm{Int}_{b,e}(\mathcal{Q}, \mathcal{L})$, $\mathrm{Int}_{b,e}(\mathcal{Q}, \mathcal{I})$, and $\mathrm{Int}_{b,e}(\mathcal{Q}, \mathcal{R})$, the sets of intersections inside $\langle b, e\rangle$ and between the segments in the staircase $\mathcal{Q}$ and in the sets $\mathcal{L}$, $\mathcal{I}$, and $\mathcal{R}$, respectively. The intersections involving segments from $\mathcal{Q}$ and segments from $\mathcal{L}$ and $\mathcal{R}$, respectively, can be found using the MergeInt subroutine presented in Section 2.1, and the intersections involving segments from $\mathcal{Q}$ and $\mathcal{I}$ can be found using the subroutine SearchInt. The intersections inside $\langle b, e\rangle$ that do not involve any $s \in \mathcal{Q}$ are found recursively.

To obtain a logarithmic depth of recursion, Balaban subdivides the set of segments that are not part of the staircase for the current strip in such a way that the same number of *endpoints* is processed in each of the recursive calls. Under the simplifying assumption that the x-coordinates of the segments are the integers $[1, \ldots, 2n]$,[5] this corresponds to subdividing with respect to the median $c := \lfloor (b+e)/2 \rfloor$, and Balaban's algorithm can be stated as Algorithm 4.

2.5. Implementing the recursive algorithm in an in-place setting

There are several issues that complicate making this algorithm in-place: First of all, like in any recursive algorithm that has to be transformed into an in-place algorithm, one has to keep track of the subarrays processed in each recursive call. It is not feasible to keep the start and end indices on a stack as this would result in using $\Omega(\log n)$ extra words of memory. The second issue to be resolved is how to partition the data prior to "going into recursion". Whereas algorithms working on point data can easily subdivide the data based upon, say, the x-coordinate by first sorting and then halving the point set, subdividing a set of segments such that the same number of endpoints appear on each side of the dividing line, seems impossible to do without splitting or copying the segments. Both splitting and copying, however, are infeasible in an in-place setting.

[5] This assumption is impossible to make in an in-place setting as one would need an extra lookup-table for translating the integer i to the x-coordinate with rank i.

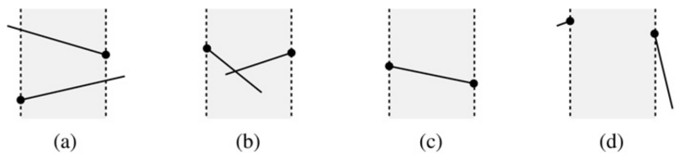


Fig. 3. Some of the configurations of segments whose endpoints define $\langle b, e\rangle$.

To guarantee both the correctness of the algorithm and the property that it uses only $\mathcal{O}(1)$ extra space, we will require the following invariants to be established at each invocation of the subroutine INPLACE_TREESEARCH($\mathtt{A}, b, e, \ell_b, \ell_e$):

Invariant A: There exists an integer $\ell_l \in \{\ell_b, \ldots, \ell_e\}$ such that all segments in $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ that cross the left boundary of $\langle b, e\rangle$ are stored in sorted $<_b$ order in $\mathtt{A}[\ell_b, \ldots, \ell_l - 1]$ (see Precondition Γ_{pre}).
Invariant B: $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ contains all segments in $\mathtt{A}$ that have at least one endpoint inside $\langle b, e\rangle$.

Additionally, we will require the following invariants to be established whenever we return from a call to INPLACE_TREESEARCH($\mathtt{A}, b, e, \ell_b, \ell_e$):

Invariant C: The strip boundaries $\langle b', e'\rangle$ of the "parent strip" are known.
Invariant D: There exists an integer $i \in \{0, \ldots, \ell_b - \ell_e\}$ such that all segments of $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ that do not cross the right strip boundary are stored in $\mathtt{A}[\ell_b, \ldots, \ell_b + i - 1]$ and that all other segments are stored in $\mathtt{A}[\ell_b + i, \ldots, \ell_e - 1]$ sorted according to $<_e$ (see Postcondition Γ_{post}).

Establishing Invariant (C) in-place is one of the most crucial steps of the algorithm. We will establish this invariant as follows: Prior to "going into recursion", we select the segments q_b and q_e whose endpoints define the strip boundary and then move it (using a linear number of swaps) to the front of the staircase $\mathcal{Q}$.[6] When moving these segments, however, it is important to keep in mind that they might be part of the staircase (Fig. 3(a)), part of $\mathcal{L}$ and/or $\mathcal{R}$ (Fig. 3(b)), identical (Fig. 3(c)), or not intersecting the interior of $\langle b, e\rangle$ at all (Fig. 3(d)), and that q_b and q_e need to be handled accordingly when looking for intersections.

In particular, this implies the following: if a segment $s \in \{q_b, q_e\}$ is part of the staircase $\mathcal{Q}$, all invocations MERGEINT and SEARCHINT for $\mathcal{Q}$ also need to check s. If a segment $s \in \{q_b, q_c, q_e\}$ is (technically) part of $\mathcal{L}$ or $\mathcal{R}$ we need to account for the fact that s is not passed to recursive calls. Since the algorithm usually reports an intersection at the unique strip where one of the segments is part of a staircase, an intersection between s and some other segment $t \in \mathcal{L} \cup \mathcal{I} \cup \mathcal{R}$ would be reported at a subslab of $\langle b, e\rangle$ where either s or t would eventually become part of a staircase. Since s is not passed down to the corresponding recursive call, we need to ensure that no such intersection will be missed. Fortunately, Invariant (B) ensures that all segments that have at least one endpoint in $\langle b, e\rangle$ are stored in $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$. This implies that we can compute all intersections involving s by simply checking each $t \in \mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ in turn and reporting intersections inside $\langle b, e\rangle$ or at $x = b$.

Any combination of the configurations for q_b and q_e is possible (see Fig. 3 for some examples), but as the overall number of combinations is constant, a constant number of bits is sufficient to encode the specific combination. Thus a "configuration" stack $\mathcal{C}$ of $\mathcal{O}(\log n)$ bits, i.e., using $\mathcal{O}(1)$ extra space, can be used to store the information necessary to recover the subset(s) into which q_b and q_e have to be reinserted when returning from the "recursive" calls.

We first use Algorithm INPLACE_SPLIT to compute the staircase $\mathcal{Q}$ and the set $\mathcal{L}_{\text{left}}$ such that the subarray looks as follows:

...	q_b	q_e	$\mathcal{Q}$	$\mathcal{L}_{\text{left}}$	$\mathcal{I} \cup \{\mathcal{R} \setminus \mathcal{L}_{\text{left}}\}$	...
			ℓ_b			ℓ_e

[6] Here and throughout the remainder of this section, we will assume that no two endpoints have the same x-coordinate. In Appendix A, we then will discuss how to get rid of this assumption.

We also maintain a "staircase" stack $\mathcal{S}$ of depth $\mathcal{O}(\log n)$ to indicate whether $\mathcal{Q}$ contains zero, one, or more segments in addition to $(q_b \cup q_e) \cap \mathcal{Q}$. This information can be encoded using $\mathcal{O}(1)$ bits per entry, i.e., using $\mathcal{O}(1)$ extra space in total. We then establish Invariant (A) by shifting $\mathcal{Q}$ in front of q_b, and prior to going into the "left recursion" we also prepare for establishing Invariant (C): We determine the segment q_c whose endpoint induces the right boundary of the left subslab (see Section 2.7 for the details of how to do this in-place) and shift the segments q_e, q_b, and q_c in front of $\mathcal{L}_{left}$. We then establish Invariant (B) by moving all elements in $\mathcal{I}_{left} \cup \{\mathcal{R}_{left} \setminus \mathcal{L}_{left}\}$ to immediately behind $\mathcal{L}_{left}$ using simple swaps (we use $\mathcal{N}$ to denote the set $(\mathcal{I} \cup \mathcal{R}) \setminus (\mathcal{I}_{left} \cup \mathcal{R}_{left})$; note that this set is not ordered in any way). We also update ℓ_b to point to the first element in $\mathcal{L}_{left}$ and update ℓ_e to point to the first segment not in $\mathcal{I}_{left} \cup \{\mathcal{R}_{left} \setminus \mathcal{L}_{left}\}$:

$\dots$	$\mathcal{Q}$	q_e	q_b	q_c	$\mathcal{L}_{left}$	$\mathcal{I}_{left} \cup \{\mathcal{R}_{left} \setminus \mathcal{L}_{left}\}$	$\mathcal{N}$	$\dots$
					ℓ_b		ℓ_e	

If we let $\mathcal{O}_{\text{left}}$ denote the set of segments whose right endpoint lies within the left subslab (i.e., $\mathcal{O}_{\text{left}}$ contains the segments that will have been swept over after we are finished with processing this subslab), Invariant (D) guarantees that upon returning from the "left recursive" call, the array has the following form:

$\dots$	$\mathcal{Q}$	q_e	q_b	q_c	$\mathcal{O}_{left}$	$\mathcal{R}_{left}$	$\mathcal{N}$	$\dots$
					ℓ_b		ℓ_e	ℓ

We can recover q_e, hence establishing Invariant (C), by simply looking at the at most three entries in front of $\mathtt{A}[\ell_b]$ (depending on the configuration encoded by the topmost element of the configuration stack $\mathcal{C}$). While doing this we also recover the old value of e; the index ℓ which corresponds to the old value of ℓ_e prior to going into the "left recursion" can be recovered by scanning forward from ℓ_e until we find the first segment not intersecting $\langle b, e\rangle$ (or reach the end of the array). We use the INPLACE_INTERCHANGE subroutine to interchange $\mathcal{R}_{left} \cup \mathcal{N}$ and $\mathcal{O}_{left}$. Note that $\mathcal{R}_{left} = \mathcal{L}_{right}$ in our setting as these sets only differ by the segment q_c which is stored separately. Also, relative to the subslab at *right*, $\mathcal{N} = \mathcal{I}_{right} \cup \{\mathcal{R}_{right} \setminus \mathcal{L}_{right}\}$. We update ℓ_e to point to the first element in $\mathcal{O}_{left}$, and shift q_e, q_b, and q_c in cyclic order.

$\dots$	$\mathcal{Q}$	q_b	q_c	q_e	$\mathcal{L}_{right}$	$\mathcal{I}_{right} \cup \{\mathcal{R}_{right} \setminus \mathcal{L}_{right}\}$	$\mathcal{O}_{left}$	$\dots$
					ℓ_b		ℓ_e	ℓ

If we let $\mathcal{O}_{\text{right}}$ denote the set of segments whose right endpoint lies within the right subslab (i.e., $\mathcal{O}_{\text{right}}$ contains the segments that will have been swept over after we are finished with processing this subslab), Invariant (D) guarantees that upon returning from the "right recursive" call, the array has the following form:

$\dots$	$\mathcal{Q}$	q_b	q_c	q_e	$\mathcal{O}_{right}$	$\mathcal{R}_{right}$	$\mathcal{O}_{left}$	$\dots$
					ℓ_b		ℓ_e	ℓ

Again, we recover the values of b and e and find the index ℓ by scanning forward from ℓ_e. Depending on whether q_c crosses the right boundary of $\langle b, e\rangle$ or not, we insert q_c into $\mathcal{R}_{right}$ or into $\mathcal{O}_{right}$. Scanning backward from ℓ_b and using the information on top of the staircase stack $\mathcal{S}$ as well as the fact that the segments in $\mathcal{Q}$ span $\langle b, e\rangle$, are non-intersecting, and are ordered by $<_b$, we determine the start of the subarray in which $\mathcal{Q}$ is stored. We then use the INPLACE_INTERCHANGE subroutine to interchange the blocks such that $\mathcal{O}_{left}$ and $\mathcal{O}_{right}$ as well as $\mathcal{Q}$ and $\mathcal{R}_{right}$ appear next to each other. Finally, we use the linear-time INPLACE_MERGE subroutine to construct $\mathcal{R}(= \mathcal{R}_{right}) \cup \mathcal{Q}$, thus establishing Invariant (D).

The resulting subarray then has the following form ($\mathcal{O} := \mathcal{O}_{\text{left}} \cup \mathcal{O}_{\text{right}}$):

$\dots$	q_b	q_e	$\mathcal{O}$	$\mathcal{Q} \cup \mathcal{R}$	$\dots$
			ℓ_b		ℓ_e

Note that, due to the use of linear-time subroutines, all interchanging, shifting, and scanning done so far takes time linear in $|\mathcal{L} \cup \mathcal{I} \cup \mathcal{R}|$. As all invariants can be established for the base case of the recursion, we conclude that the invariants can be established for each "recursive call", and thus we have established the correctness of the following algorithm:

Note that in contrast to Balaban's algorithm (Algorithm 4), the intersections between the segments from $\mathcal{Q}$ and the segments from $\mathcal{I}$ are found prior to going into the recursion. The intuition behind this is that segments that have been swept over have to be moved outside the current subarray as soon as possible such as not to interfere with reconstructing the strip boundaries.

The above description does not contain the detailed code for simulating the two "recursive" calls to IN-PLACE_TREESEARCH, since we have shown previously [5] that it is possible to handle these calls using a stack of $\mathcal{O}(\log n)$ bits, that is using $\mathcal{O}(1)$ extra space.[7] To do so we need to be able to retrieve the subset to work with upon returning from a recursive call using only $\mathcal{O}(1)$ extra space, and this is guaranteed by Invariant (C). We also did not include the code for handling the segments q_b, q_e, and q_c; for example, the step in line 19 would need to be expanded to do this. We need, however, to fill in the details of how to select the endpoint with median x-coordinate which in turn determines the segment q_c. This will be done in Section 2.7.

2.6. *Intermezzo: Practical considerations*

As discussed at the end of the previous section, we need to simulate running a recursive algorithm using only $\mathcal{O}(1)$ extra space, and this leads to a technically quite involved algorithmic scheme. Research on space-efficient algorithm, however, has not only focused on in-place algorithms but also, more generally, on algorithms with $o(n)$ extra space requirement. In a more relaxed setting, sometimes referred to as *in-situ* [7], we have $\mathcal{O}(\log n)$ extra space available. An immediate consequence is that this allows a (balanced) recursion scheme without having to use the technically involved scheme discussed by Bose et al. [5]. Since most "real" machines have a separate "activation frame" buffer for maintaining the recursion stack, the only advantage of insisting on $\mathcal{O}(1)$ extra space is that this is a theoretically "cleaner" model that takes the question of space-time dependency to the extreme. Other than that, in-situ algorithms are quite interesting since they have a much higher potential of being practical.

To realize our algorithm in an in-situ setting, we can basically remove all constructs that occurred in connection with the problem of restoring the strip boundaries (Invariant (C)). These constructs encompass all computations involving the segments q_b, q_c, and q_e (e.g., in lines 7, 10, and 28 of Algorithm 5) and are listed in Section 2.5. Since, in an in-situ setting, we can keep track of the relevant indices needed for recovering the sets of stairs and non-stairs, we can also get rid of lines 14, 15, and 18–23 of Algorithm 5 (with the appropriate changes to the rest of the code to ensure that the indices are handled correctly). Finally, we would be able to avoid implementing the flow-control mechanism needed for simulating the recursion [5].

2.7. *Selecting the median in-place*

When selecting the median of the endpoints in line 5 of Algorithm 5, we have to do so while maintaining the set $\mathcal{L}' = \mathbb{A}[\ell_b, \ldots, \ell_c - 1]$ (see line 5 of Algorithm 5) in sorted $<_b$-order. To make sure that the overall cost of median-finding does not depend on the number k of intersections reported by the algorithm, we also need to guarantee that we only process segments that have at least one endpoint inside the strip $\langle b, e\rangle$. Doing so, we enforce that each segment participates in no more than $\mathcal{O}(\log n)$ invocations of median-finding, namely in $\mathcal{O}(1)$ such invocations on each level of the recursion tree. To make the algorithm reflect this, we use SORTEDSUBSETSELECTION to stably select the segments of $\mathcal{L}'$ spanning $\langle b, e\rangle$, i.e., those segments that do not have any endpoint inside $\langle b, e\rangle$. The segments that have at least one endpoint in $\langle b, e\rangle$ are then stored consecutively in $\mathbb{A}[\ell_b + j, \ldots, \ell_e - 1]$ (for some $j \in \{0, \ldots, \ell_e - \ell_b\}$). We can count the number of endpoints inside $\langle b, e\rangle$ (and thus compute the rank of the median element) in linear time and then apply the following lemma:

Lemma 5. *Given m segments and a strip $\langle b, e\rangle$, the k-th endpoint in sorted order inside $\langle b, e\rangle$ can be found in-place and in $\mathcal{O}(m \log m)$ time.*

Proof. We simulate a plane-sweep over the set of m segments and maintain the current x-coordinate ξ as well as the number o of endpoints inside $\langle b, e\rangle$ that have already been swept over. The segments are maintained in-place in

[7] In a nutshell, we need to realize a flow-control mechanism (as in an interpreter) that keeps track of the line currently executed and also keeps track of the line at which to resume execution after returning from an "recursive" call. As there is only a constant number of possible lines where execution may resume (in our case, lines 14 and 18), we can encode this information using a constant number of bits per recursive call simulated, hence using a recursion stack of $\mathcal{O}(\log n)$ bits, or $\mathcal{O}(1)$ words.

```
1: if $\langle b,e\rangle$ does not contain any endpoint of a segment $s \in \mathtt{A}[\ell_b,\dots,\ell_e-1]$ then
2:     InPlace_SearchInStrip($\mathtt{A}, b, e, \ell_b, \ell_e$);    {Recursion Base Case.}
3: else
4:     Scan forward in $\mathtt{A}[\ell_b,\dots,\ell_e-1]$ to find the index $\ell_l$ of the first segment in $\mathtt{A}[\ell_b,\dots,\ell_e-1]$ that does not cross the left strip boundary.
       {$\mathcal{L} = \mathtt{A}[\ell_b,\dots,\ell_l-1]$}.
5:     $\ell_c :=$ InPlace_Split($\mathtt{A}, b, e, \ell_b, \ell_l$);    {$\mathcal{Q} = \mathtt{A}[\ell_b,\dots,\ell_c-1]$; $\mathcal{L}_{left} = \mathtt{A}[\ell_c,\dots,\ell_l-1]$.}
6:     {In lines 7, 10, and 28, remember to also check for $q_b$ and $q_e$}.
7:     MergeInt$_b$($\mathtt{A}, b, e, \ell_b, \ell_c, \ell_l$);    {Situation (1): Compute Int$_{b,e}(\mathcal{Q}, \mathcal{L}_{left})$.}
8:     for $\ell := \ell_l$ to $\ell_e - 1$ do
9:         if $\mathtt{A}[\ell]$ lies entirely inside $\langle b,e\rangle$ then
10:            SearchInt($\mathtt{A}, b, e, \ell_b, \ell_c, \ell$).    {Situation (2): Compute Int$_{b,e}(\mathcal{Q}, \{\mathtt{A}[\ell]\})$.}
11:    Find the index of the median of the endpoints inside the current strip. Let $c$ be the $x$-coordinate of this endpoint.
12:    Establish Invariants (A) and (B). Update $\ell_b$ and $\ell_e$ to prepare for processing left.
13:    InPlace_TreeSearch($\mathtt{A}, b, c, \ell_b, \ell_e$);
14:    Use the configuration stack to recover the old value of $e$.    {Establish Invariant (C).}
15:    Scan forward in $\mathtt{A}[\ell_b,\dots,n-1]$ to recover the old value of $\ell_e$.
16:    Establish Invariants (A) and (B). Update $\ell_b$ and $\ell_e$ to prepare for processing right.
17:    InPlace_TreeSearch($\mathtt{A}, c, e, \ell_b, \ell_e$);
18:    Use the configuration stack to recover the old value of $b$.    {Establish Invariant (C).}
19:    Scan backward in $\mathtt{A}[0,\dots,\ell_b-1]$ to find the index $\ell'$ such that $\mathtt{A}[\ell',\dots,\ell_b-1]$ stores $\mathcal{Q}$.
20:    $q := \ell_b - \ell'$;    {Number of stairs in $\mathcal{Q}$.}
21:    $\ell_{r_2} := \ell_e$;
22:    Scan forward in $\mathtt{A}[\ell_b,\dots,\ell_{r_2}-1]$ to find the index $\ell_{r_1}$ such that $\mathtt{A}[\ell_{r_1},\dots,\ell_{r_2}-1]$ stores $\mathcal{R}$.
23:    Scan forward in $\mathtt{A}[\ell_{r_2},\dots,n-1]$ to find the index $\ell$ such that $\mathtt{A}[\ell_{r_2},\dots,\ell-1]$ stores $\mathcal{O}_{left}$.
24:    $\ell_e := \ell$;    {$\ell_e$ has been recovered.}
25:    InPlace_Interchange($\mathtt{A}, \ell', \ell_b, \ell_{r_1}$);    {Interchange $\mathcal{Q}$ and $\mathcal{O}_{right}$.}
26:    InPlace_Interchange($\mathtt{A}, \ell_b, \ell_{r_2}, \ell_e$);    {Interchange $\mathcal{Q} \cup \mathcal{R}_{right}$ and $\mathcal{O}_{left}$.}
27:    $\ell := \ell_b + (\ell_e - \ell_{r_2})$;    {$\mathtt{A}[\ell_b,\dots,\ell-1]$ stores $\mathcal{O}$.}
28:    MergeInt$_e$($\mathtt{A}, b, e, \ell, \ell+q, \ell_e$);    {Situation (3): Compute Int$_{b,e}(\mathcal{Q}, \mathcal{R}_{right})$.}
29:    InPlace_Merge$_e$($\mathtt{A}, b, e, \ell, \ell+q, \ell_e$);    {Establish Invariant (D).}
```

Algorithm 5. Algorithm InPlace_TreeSearch($\mathtt{A}, b, e, \ell_b, \ell_e$).

a heap-based priority queue $\mathcal{H}$, the priority of a segment s being the smallest x-coordinate of s's endpoints that still is at least ξ. Let $p(s,\xi)$ denote the endpoint of s that realizes the priority of s at time ξ. When deleting the minimal element s from $\mathcal{H}$ we set ξ to the x-coordinate given by $p(s,\xi)$ and distinguish two cases:

Case 1: The endpoint $p(s,\xi)$ lies inside the strip $\langle b,e\rangle$. We increment the number o of endpoints inside $\langle b,e\rangle$ that have already been swept over. If $o = k$, we report s and the endpoint $p(s,\xi)$.

Case 1.1: The endpoint $p(s,\xi)$ is the left endpoint of s. Let q be the right endpoint. We reinsert s into the priority queue using the x-coordinate of q as its priority.

Case 1.2: The endpoint $p(s,\xi)$ is the right endpoint of s. We delete s from the priority queue.

Case 2: The endpoint $p(s,\xi)$ lies outside the strip. We can abort the algorithm because all elements still in the priority queue will have their unswept endpoints outside the strip $\langle b,e\rangle$.

Note that, using a heap-based priority queue [27], we can implement the priority queue in-place as all priorities are implicitly given by ξ and the segments' endpoints. As there are $\mathcal{O}(m)$ priority queue operations, the algorithm runs in time $\mathcal{O}(m\log m)$. □

After we have found the median using the algorithm implied by Lemma 5, we need to restore $\mathcal{L}'$ in sorted $<_b$ order. To this end, we then select the elements from $\mathtt{A}[\ell_b+i,\dots,\ell_e-1]$ that cross the left strip boundary, sort them in-place by $<_b$, and then merge them in-place with the segments in $\mathtt{A}[\ell_b,\dots,\ell_b+i-1]$ using the linear-time InPlace_Merge subroutine.

2.8. Analysis of the running time

For the main part of the analysis, Balaban's results carry over. Using the notation $\mathcal{S}_v = \mathcal{L}_v \cup \mathcal{I}_v \cup \mathcal{R}_v$ where the subscripts indicate that this set is "active" when processing the strip corresponding to a node v in the recursion tree $\mathcal{T}$, the following theorem holds:

Theorem 1. (Theorem 2 in [1]) $\sum_{v \in \mathcal{T}} |\mathcal{S}_v| \leqslant n\lceil 4\log n + 5\rceil + 2k$.

To make the algorithm in-place, we had to resort to some algorithmic techniques not captured in Balaban's analysis. The global cost for running the in-place version of the algorithm SPLIT is $\mathcal{O}(n\log n + k)$ (see Lemma 3 and Theorem 1). From Theorem 1, it follows that the overall extra cost for establishing the invariants is $\mathcal{O}(n\log n + k)$ as all operations performed at a node $v \in \mathcal{T}$ take time linear in $|\mathcal{S}_v|$.

Finally, we had to realize the median-finding in-place while maintaining the original order of the elements. By Lemma 6, the overall cost for this is in $\mathcal{O}(n\log^2 n)$.

Lemma 6. *The global cost incurred by median-finding is $\mathcal{O}(n\log^2 n)$.*

Proof. The median-finding algorithm considers only those segments that have at least one endpoint in the current strip. Hence, on each level of recursion, each segment is considered at most twice, so we can charge each segment s $\mathcal{O}(\log n)$ cost per level for median-finding (see Lemma 5). We charge s an additional $\mathcal{O}(\log n)$ cost per level for the at most one sorting step it participates in (when restoring $\mathcal{L}'$) and for the at most four priority queue operations (`insert` and `deleteMin`) that operate on its endpoints. As all other operations (including building the heap) require only linear time per level, the global cost incurred by median-finding is $\mathcal{O}(n\log^2 n)$ as claimed. □

The last component of the analysis is the `for`-loop in line 8 of Algorithm 5 which resembles the SEARCHINT subroutine: Each iteration of this loop takes $\mathcal{O}(\log|\mathcal{Q}_v|) \subseteq \mathcal{O}(\log n)$ time in addition to the time needed for reporting the intersections found, and each of the n segments can be part of $\mathcal{I}_w$ for $\mathcal{O}(\log n)$ nodes $w \in \mathcal{T}$. Thus, we have to account for an additional global $\mathcal{O}(n\log^2 n)$ time complexity, and combining this with Balaban's original analysis, we obtain our main result:

Theorem 2. *All k intersections induced by a set of n segments in the plane can be computed in $\mathcal{O}(n\log^2 n + k)$ time using $\mathcal{O}(1)$ extra words of memory.*

We conclude with the obvious open problem: Is it possible to compute all k intersections induced by a set of n segments in the plane in-place *and* in optimal time $\mathcal{O}(n\log n + k)$?

Acknowledgement

The author thanks Henrik Blunck who suggested to directly modify Salowe and Steiger's algorithm and the anonymous reviewers whose comments greatly helped improving the presentation. In particular, the simplified version of the in-place algorithm for the recursion base case is due to an anonymous reviewer's comment.

Appendix A. Degenerate configurations

In the discussion of Lemma 2, we noted that segments that degenerate to points pose a potential problem (which then was resolved by Lemma 3). In this appendix, we discuss how to handle such segments and other "degenerate" configurations. For a detailed discussion of how to address robustness and precision issues, we refer the reader to the survey by Schirra [25] and to the more specialized article by Boissonnat and Vigneron [4].

A.1. Vertical segments

Using standard (implicit) shearing techniques, it is possible to *implicitly* rotate or shear the coordinate system such no segment is vertical and such that the output of the algorithm is not affected—see the survey by Schirra [25] and the references therein.

A.2. Zero-length segments

Zero-length segments may pose a problem in several situations, e.g., when using the version of InPlace_Split that uses the relative order of the endpoints to encode a bit (Lemma 2). To handle input instances that contain zero-length segments (points), we proceed as follows: In a preprocessing step, we partition the input such that $\mathtt{A}[0,\ldots,n'-1]$ contains all segments that do not have length zero and such that $\mathtt{A}[n',\ldots,n-1]$ contains all points. Additionally, we sort $\mathtt{A}[n',\ldots,n-1]$ according to the lexicographic $<_x$-order. A simple scan over $\mathtt{A}[n',\ldots,n-1]$ then allows us to identify all "point-point" intersections in $\mathcal{O}(n+k)$ time.

To find all "point-segment" intersections, we then run the algorithm for finding "segment-segment" intersections on $\mathtt{A}[0,\ldots,n'-1]$ and maintain two indices ℓ'_b and ℓ'_e with $n' \leqslant \ell'_b \leqslant \ell'_e \leqslant n$ such that $\mathtt{A}[\ell'_b,\ldots,\ell'_e-1]$ contains all points inside $\langle b,e\rangle$ or at $x=b$ (note that this subarray may be empty). Whenever we invoke MergeInt on a strip's left boundary $x=b$, we additionally scan forward in $\mathtt{A}[\ell'_b,\ldots,\ell'_e-1]$ to find the subarray $\mathtt{A}[\ell'_b,\ldots,\ell''_b-1]$ the contains the points at $x=b$. Since the points are sorted according to the *lexicographic* $<_x$-order, we know that $\mathtt{A}[\ell'_b,\ldots,\ell''_b-1]$ is sorted according to the vertical order at $x=b$ (all points have the same x-coordinate). This allows us to identify all "stair-point" intersections at $x=b$ by invoking MergeInt. Similarly, for each invocation of SearchInt on a strip $\langle b,e\rangle$, we also invoke SearchInt for all points in $\mathtt{A}[\ell''_b,\ldots,\ell'_e]$ (note that invoking SearchInt for $\mathtt{A}[\ell'_b,\ldots,\ell'_e-1]$ would also identify intersections on the left boundary which have been reported by a previous call to MergeInt). The cost for processing these points can be easily seen to be accounted for by the cost for the non-degenerate case—imagine all points to be infinitesimally short horizontal segments that belong to some set $\mathcal{I}$. It is also easy to verify that the indices ℓ'_b and ℓ'_e can be updated in linear time after the boundaries of the next strip to be processed have been computed; the global cost for updating these pointers is $\mathcal{O}(n\log n)$ (for the proof of a similar statement, see Theorem 1 in Balaban's paper [1]). Thus, the overall running time is not affected by separately processing zero-length segments.

A.3. Overlapping segments

As noted before, our algorithm reports a pair (s,t) of intersecting segments when processing the unique strip $\langle b,e\rangle$ where one of the segments, say s, is part of the staircase $\mathcal{Q}$ spanning the strip $\langle b,e\rangle$ and where the intersection point lies within $\langle b,e\rangle$ or on the left boundary of $\langle b,e\rangle$. In the case of overlapping segments, we define the "intersection point" (as referred to in the previous statement) to be the rightmost left endpoint of the two segments. This definition implies that for overlapping segments starting at the same left endpoint p, this "intersection point" is p; this is consistent with our convention that intersections on the boundary of a strip are reported only for the left boundary of the strip. To avoid intersections reported multiple times, the algorithm MergeInt (merging a staircase and a sorted set $\mathcal{L}$ or $\mathcal{R}$) needs to check whether this "intersection point" of overlapping segments actually falls within the current strip. The algorithm SearchInt does not need to be modified since both endpoints are guaranteed to lie within the current strip.

A.4. Identical segments

The main idea of handling identical segments is to handle them as one group. To do so, we first sort the segments such that identical segments appear next to each other. Depending on whether or not identical segments are considered to be intersecting, we can scan through the sorted set and report all self-intersections in $\mathcal{O}(n+k)$ time.

We then run the intersection-finding algorithm. Throughout the algorithm, we maintain the invariant that, whenever segments appear in some sorted order, all copies of an element are stored consecutively. The reason for this is that the relevant operations for finding intersections always access elements in sorted order. The only exception is SearchInt where the segments in $\mathcal{I}$ are not required to be presented in sorted order. If this invariant is maintained, we can perform each constant-time operation that would access a single segment (in the case of non-degenerate input) on the k copies of a segment in $\mathcal{O}(1+k)$ time (simply scan forward to see how many identical copies exist).

Let us now consider in turn each subroutine that moves elements: Algorithms INPLACE_MERGE, INPLACE_UNMERGE, and INPLACE_INTERCHANGE can be implemented to work stably which implies that they maintain the above invariant. As a consequence, Algorithms INPLACE_SPLIT and INPLACE_SEARCHINSTRIP have the same property. Also, the SORTEDSUBSETSELECTION maintains the order in the selected set and UNDOSORTEDSUBSETSELECTION is guaranteed to re-establish the original order as long as identical segments have been treated in a consistent way by the selection algorithm, i.e., if identical segments have been selected or not selected as a group. Note that this is not the same as requiring that the selection and restoring process is stable, i.e., that the relative order of identical segments is maintained.

Finally, for the median-finding algorithm (Section 2.7), we need to take into account that multiple copies of a segment also induce multiple copies on the endpoints. Since we want to treat identical segments as a group, we need to modify the algorithm for computing the rank of the median as well as the algorithm for actually finding the element with median rank. To do so, we run the heap-based sweep-algorithm twice. In the first run, we simply compute the number of endpoints induced by different segments: whenever we extract the next segment from the heap, we increment a counter by one iff the sweep-line has moved since the last extraction or this segment is different from the segment extracted before. In the second run, we again sweep left to right, and whenever we extract a segment identical to the segment we have extracted before and have not moved the sweep-line, we simply ignore it.

A.5. Multiple occurrences of x-coordinates

In Section 2.5, we had assumed that no two endpoints share the same x-coordinate. As a consequence, the median of the endpoint alway gave a balanced partitioning. If multiple endpoints share the same x-coordinate it is not always obvious how to split according to the vertical line through the median.

We will handle such situations similar to the approach described in the previous paragraph. Using a small modification of the sweep-line algorithm, we determine the median $\hat{c}$ of the *set* of x-coordinates realized by endpoints inside the strip $\langle b, e\rangle$. We then partition the segments such that the set $\mathcal{E}$ of segments that end at $x=\hat{c}$ is considered to be inside $\langle b, \hat{c}\rangle$ whereas the set $\mathcal{B}$ of segments that start at $x=\hat{c}$ is considered to be inside $\langle \hat{c}, e\rangle$. This also allows us to uniquely reconstruct the right boundary when returning from a "recursive" call. While the recursion may not be balanced, the important point is that the recursion depth is given by the logarithm of the number of distinct x-coordinate, and thus still is in $\mathcal{O}(\log n)$. In fact, multiple endpoints sharing the same coordinate will lead to an earlier end of the recursion process, since the condition for terminating the recursion is whether or not endpoints exist *inside* a strip.

It remains to discuss how to report intersections that are induced by segments that meet at $x=\hat{c}$. These segments can be identified (and their intersection reported) by sorting both $\mathcal{B}$ and $\mathcal{E}$ according to the vertical order at $x=\hat{c}$. Intersections induced by $\mathcal{E}$ can be found by a single scan and intersections between $\mathcal{E}$ and $\mathcal{B}$ as well as intersections between $\mathcal{E}$ and the set $\mathcal{R}$ of segments crossing $x=\hat{c}$ are found by invoking (a slightly modified version of) MERGEINT$_{\hat{c}}$. Note that intersections induced by $\mathcal{B}$ are found when processing $\langle \hat{c}, e\rangle$. The cost for performing these steps is dominated by the cost for the heap-based sweep and the cost for reporting the intersections, so the global complexity of the algorithm is not affected.

A.6. Intersections on the rightmost boundary

In Section 2.4, we noted that the algorithm finds all intersections inside a strip $\langle b, e\rangle$ and on the left boundary of the strip $\langle b, e\rangle$. With the exception of the initial strip, each strip $\langle b, e\rangle$ has an adjacent strip $\langle e, f\rangle$, so intersections at $x=e$ will be found while processing $\langle e, f\rangle$ (this also prevents intersections from being reported more than once). To identify intersections on the right boundary of the initial strip $\langle b, e\rangle$, we first note that the value of e is given by the rightmost x-coordinate of any input segment. Thus, an intersection at $x=e$ corresponds to coinciding endpoints at $x=e$. To prevent this from happening, we simply add a horizontal dummy segment to the right of the bounding box containing all segments such that the right boundary of the initial strip is strictly to the right of $x=e$. Since this segment does not introducing new intersections, it does not affect the output and does not asymptotically increase the time and space complexity of the algorithm. Alternatively, we could also identify the segments ending at $x=e$ and use a sorting-based approach to identify all intersections in a post- or pre-processing step.

References

[1] I.J. Balaban, An optimal algorithm for finding segments [*sic!*] intersections, in: Proceedings of the Eleventh Annual Symposium on Computational Geometry, ACM Press, New York, 1995, pp. 211–219.
[2] J.L. Bentley, T.A. Ottmann, Algorithms for reporting and counting geometric intersections, IEEE Transactions on Computers C-28 (9) (1979) 643–647.
[3] J.-D. Boissonnat, J. Snoeyink, Efficient algorithms for line and curve segment intersection using restricted predicates, Computational Geometry: Theory and Applications 16 (1) (May 2000) 35–52.
[4] J.-D. Boissonnat, A. Vigneron, An elementary algorithm for reporting intersections of red/blue curve segments, Computational Geometry: Theory Applications 21 (3) (March 2002) 167–175.
[5] P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, J. Vahrenhold, Space-efficient geometric divide-and-conquer algorithms, Computational Geometry: Theory & Applications (2006), doi:10.1016/j.comgeo.2006.03.006, in press, accepted November 2004. An extended abstract appeared in: Proceedings of the 20th European Workshop on Computational Geometry, 2004, pp. 65–68.
[6] H. Brönnimann, T.M.-Y. Chan, E.Y. Chen, Towards in-place geometric algorithms, in: Proceedings of the Twentieth Annual Symposium on Computational Geometry, ACM Press, New York, 2004, pp. 239–246.
[7] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, G.T. Toussaint, Space-efficient planar convex hull algorithms, Theoretical Computer Science 321 (1) (June 2004) 25–40. An extended abstract appeared in the Proceedings of the Fifth Latin American Symposium on Theoretical Informatics, 2002, pp. 494–507.
[8] B. Chazelle, Intersecting is easier than sorting, in: Proceedings of the 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 125–134.
[9] B. Chazelle, H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, Journal of the ACM 39 (1) (January 1992) 1–54.
[10] B. Chazelle, L.J. Guibas, Fractional cascading: I. A data structuring technique, Algorithmica 1 (2) (1986) 133–162.
[11] E.Y. Chen, T.M.-Y. Chan, A space-efficient algorithm for line segment intersection, in: Proceedings of the 15th Canadian Conference on Computational Geometry, 2003, pp. 68–71.
[12] J.-C. Chen, Optimizing stable in-place merging, Theoretical Computer Science 302 (1–3) (June 2003) 191–210.
[13] J.-C. Chen, A simple algorithm for in-place merging, Information Processing Letters 98 (1) (April 2006) 34–40.
[14] R.W. Floyd, Algorithm 245: Treesort, Communications of the ACM 7 (12) (December 1964) 701.
[15] V. Geffert, J. Katajainen, T. Pasanen, Asymptotically efficient in-place merging, Theoretical Computer Science 237 (1–2) (April 2000) 159–181.
[16] J.E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, second ed., Discrete Mathematics and its Applications, CRC Press, Boca Raton, FL, 2004.
[17] J. Katajainen, T. Pasanen, Stable minimum space partitioning in linear time, BIT: Computer Science 32 (1992) 580–585.
[18] J. Katajainen, T. Pasanen, Sorting multisets stably in minimum space, Acta Informatica 31 (4) (1994) 301–313.
[19] J. Kleinberg, É. Tardos, Algorithm Design, Addison-Wesley, Boston, MA, 2006.
[20] D.M. Mount, Geometric intersection, in: [16], Chapter 38, pp. 857–876.
[21] J.I. Munro, An implicit data structure supporting insertion, deletion, and search in $o(\log^2 n)$ time, Journal of Computer and System Sciences 33 (1) (August 1986) 66–74.
[22] L.T. Pardo, Stable sorting and merging with optimal space and time bounds, SIAM Journal on Computing 6 (2) (June 1977) 351–372.
[23] J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier, Amsterdam, 2000.
[24] J.S. Salowe, W.L. Steiger, Stable unmerging in linear time and constant space, Information Processing Letters 25 (3) (July 1987) 285–294.
[25] S. Schirra, Robustness and precision issues in geometric computation, in: [23], Chapter 14, pp. 597–632.
[26] J. Vahrenhold, Line-segment intersection made in-place, in: A. López-Ortiz, F. Dehne, J.-R. Sack (Eds.), Algorithms and Data Structures, 9th International Workshop, WADS 2005, in: Lecture Notes in Computer Science, vol. 3608, Springer, Berlin, 2005, pp. 146–157.
[27] J.W.J. Williams, Algorithm 232: Heapsort, Communications of the ACM 7 (6) (June 1964) 347–348.