Line-Segment Intersection Made In-Place

Jan Vahrenhold

Westfälische Wilhelms-Universität Münster,
Institut für Informatik, 48149 Münster, Germany
jan@math.uni-muenster.de

Abstract. We present a space-efficient algorithm for reporting all k intersections induced by a set of n line segments in the place. Our algorithm is an in-place variant of Balaban's algorithm and runs in $\mathcal{O}(n \log_2^2 n + k)$ time using $\mathcal{O}(1)$ extra words of memory over and above the space used for the input to the algorithm.

1 Introduction

Researchers have studied space-efficient algorithms since the early 70's. Examples include merging, (multiset) sorting, and partitioning problems; see [8, 9, 11]. Brönnimann *et al.* [5] 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 $\mathcal{O}(n \log_2 h)$ time using $\mathcal{O}(1)$ extra space, where h denotes the size of the output. Recently, Brönnimann *et al.* [4] 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.* [3] developed a general framework for geometric divide-and-conquer algorithmus and derived space-efficient algorithms for the nearest neighbor, bichromatic nearest neighbor, and orthogonal line segment intersection problems, and Chen and Chan [7] 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.

The Model. The goal is to design algorithms that use very little extra space over and above 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 $\mathcal{O}(\log_2 n)$ bit integer, and a constant number of words can hold one element of the input array. The extra memory used by an algorithm is measured in terms of the number of extra words. 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 $\Omega(n^2)$. 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 [7] for variable size output, space-efficient algorithms and accurately models algorithms that have output streams with write-only buffer

F. Dehne, A. López-Ortiz, and J.-R. Sack (Eds.): WADS 2005, LNCS 3608, pp. 146–157, 2005.
© Springer-Verlag Berlin Heidelberg 2005

space. In the space-efficient model, an algorithm is said to work *in-place* iff it uses $\mathcal{O}(1)$ extra words of memory.

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 [12]. In the space-efficient model of computation, Bose *et al.* [3] 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 $\mathcal{O}(n \log_2 n + k)$ time and uses $\mathcal{O}(1)$ words of extra memory. Chen and Chan [7] modified the well-known algorithm of Bentley and Ottmann [2] and obtained a space-efficient algorithm that runs in $\mathcal{O}((n+k)\log_2^2 n)$ time and uses $\mathcal{O}(\log_2^2 n)$ extra words of memory.[1] We will improve these bounds to $\mathcal{O}(n \log_2^2 n + k)$ 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.

2 The Algorithm

Our algorithm is an in-place version of the optimal $\mathcal{O}(n \log_2 n + k)$ algorithm proposed by Balaban [1]. Balaban obtained this complexity by first developing an intermediate algorithm with running time $\mathcal{O}(n \log_2^2 n + k)$ and then applying the well-known concept of *fractional cascading* [6]. As fractional cascading relies on explicitly maintained copies of certain elements, this concept can only applied with $\mathcal{O}(n)$ extra space which is prohibitive for an in-place algorithm. Thus, we build upon the (suboptimal) intermediate algorithm.

2.1 Divide-and-Conquer and the Recursion Tree

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 Figure 1. While doing so, the algorithm maintains the following invariants:

Invariant 1: Prior to processing a strip, all segments crossing the left strip boundary are vertically ordered at the x-coordinate of the left strip boundary.

Invariant 2: During the sweep over a strip, all intersections inside the strip are reported.

Invariant 3: 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.

The base of recursion 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.

[1] If the model is changed such that the input can be destroyed, the bounds can be improved to $\mathcal{O}((n+k)\log_2 n)$ time and $\mathcal{O}(1)$ extra space.

Invariant 1 implies that this set of segments is sorted according to $<_b$, the vertical order at x-coordinate b.

Balaban explains his algorithm based upon the intuition that the recursive calls of a divide-and-conquer algorithm can be modelled as a recursion tree where each node is assigned the subproblem to be solved in the corresponding recursive call. The recursion starts at the root node of the recursion tree $\mathcal{T}$, and hence the algorithm can been said to process the nodes (and hence the strips) along an Euler tour of $\mathcal{T}$. A closer look at the algorithm will reveal that, during the execution of the algorithm, some of the intersections detected while processing a strip corresponding to a node $v \in \mathcal{T}$ are found while v is being visited for the first time whereas some of these intersections are found while v is being visited for the last time. This in turn implies that the algorithm follows a divide-and-conquer strategy similar to the one described in Algorithm 1:

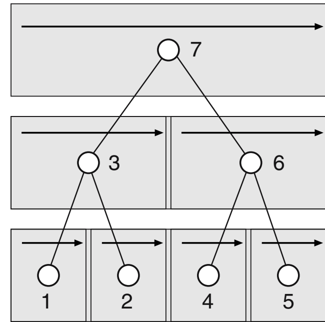


Fig. 1. Processing the recursion tree. Numbers indicate the order in which the strips are *finished*

Algorithm 1. RECURSIVE($\mathtt{A}, b, e$): Recursive divide-and-conquer [3]

1: **if** $e - b \leq s$ where s is the size of the recursion base. **then**
2: BASE-CODE($\mathtt{A}, b, e$) {Code for solving small instances}
3: **else**
4: PRE-CODE($\mathtt{A}, b, e$) {Setup Subproblem 1 in $\mathtt{A}[b, \ldots, \lfloor e/2 \rfloor - 1]$}
5: RECURSIVE($\mathtt{A}, b, \lfloor e/2 \rfloor$) {First recursive call}
6: MID-CODE($\mathtt{A}, b, e$) {Setup Subproblem 2 in $\mathtt{A}[\lfloor e/2 \rfloor, \ldots, e - 1]$}
7: RECURSIVE($\mathtt{A}, \lfloor e/2 \rfloor, e$) {Second recursive call}
8: POST-CODE($\mathtt{A}, b, e$) {Merge Subproblems 1 and 2 in $\mathtt{A}[b, \ldots, e - 1]$}

This algorithm operates on an array $\mathtt{A}[0, \ldots, n-1]$ and makes calls to 4 subroutines: BASE-CODE is used to solve small instances, PRE-CODE is executed before any recursive calls, MID-CODE is executed after the first recursive call but before the second, and POST-CODE is executed after the second recursive call. In our previous work, we have shown that this general template can be realized in-place [3]. In the following subsections, we will demonstrate how both the subroutines and the partitioning of the segments to be processed can be realized using only $\mathcal{O}(1)$ extra space.

2.2 The Base of Recursion

As mentioned above, the base of recursion 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. By Invariant 1, the set $\mathcal{L}$ of segments is sorted according to $<_b$, the vertical order at x-coordinate b.

Algorithm 2. The algorithm $\textsc{Split}_{b,e}(\mathcal{L}, \mathcal{Q}, \mathcal{L}')$ [1]

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: $\mathcal{L}' \leftarrow s_j$.

Algorithm 2 partitions $\mathcal{L}$ into two sets $\mathcal{Q}$ and $\mathcal{L}' = \mathcal{L} \setminus \mathcal{Q}$ such that both sets are sorted according to $<_b$, that there are no intersections induced by the segments in $\mathcal{Q}$ ($\mathcal{Q}$ is called a *staircase*), and that $\mathcal{Q}$ is maximal: that is, *complete relative to* $\langle b, e \rangle$.
The correctness of the algorithms depends on the invariant that both the staircase $\mathcal{Q}$ and the remaining subset $\mathcal{L}'$ remain ordered by $<_b$. This condition cannot be enforced with a linear-time in-place algorithm as the only known such algorithm for stable partitioning [11] is a variant of $\{0, 1\}$-sorting. This implies that the algorithm has to be able to decide for any given element whether it should belong to $\mathcal{Q}$ or $\mathcal{L}'$—independent of whether the algorithm has seen any other element before and independent of the processing order of the elements. As constructing a staircase has to be done incrementally, using this non-incremental stable in-place partitioning is not feasible.

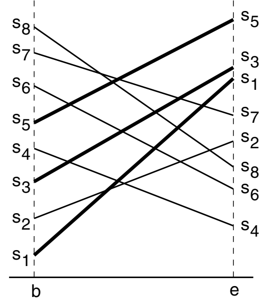


Fig. 2. Base of recursion. Fat lines indicate a maximal staircase

Let us for the moment, however, assume that such an in-place partitioning algorithm exists, and let us see how it can be used as a subroutine. Algorithm 3 recursively uses Split to partition a set $\mathcal{L}$ of segments spanning $\langle b, e \rangle$ and sorted by $<_b$ such that the set $\mathrm{Int}_{b,e}(\mathcal{L})$ 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, Algorithm 3 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 $b' = e$.

The running time of Algorithm 3 is linear in the number of segments in $\mathcal{L}$ and the number of intersections reported. To see this, note that Steps 1, 3, and 7 run in time linear in $|\mathcal{L}|$ and that Step 5 runs in time linear in $|\mathcal{L}|$ plus the number of intersections reported. For the recursive calls, observe that a segment is *not* assigned to a staircase (during the executing of Algorithm 2) hence being processed in a recursive call iff there exists at least one intersection with a staircase. The effects of the recursive calls to Split are reverted by the repeated calls to Merge (Line 7 of Algorithm 3). This operation is a linear-time operation as both $\mathcal{Q}$ and $\mathcal{R}'$ are ordered by $<_e$, and using the algorithm by Geffert *et al.* [9], it can also be performed in-place.

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

Require: $\mathcal{L} = (s_1, \ldots, s_k)$ is ordered by $<_b$; each s_i spans $\langle b, e \rangle$.
Ensure: $\mathcal{R} = (s_{\pi(1)}, \ldots, s_{\pi(k)})$ is ordered by $<_e$.
1: $\textsc{Split}_{b,e}(\mathcal{L}, \mathcal{Q}, \mathcal{L}')$. {Partition $\mathcal{L}$ into (a staircase) $\mathcal{Q}$ and $\mathcal{L}' = \mathcal{L} \setminus \mathcal{Q}$.}
2: **if** $\mathcal{L}' = \emptyset$ **then**
3: $\quad \mathcal{R} := \mathcal{Q}$. {Base of recursion: No intersections within $\mathcal{L}$; $\mathcal{R}$ is ordered by $<_b$.}
4: **else**
5: $\quad$ Find all intersections of $\mathcal{Q}$ and $\mathcal{L}'$ inside $\langle b, e \rangle$ using a synchronized scan.
6: $\quad \textsc{SearchInStrip}_{b,e}(\mathcal{L}', \mathcal{R}')$. {Recursively find intersections within $\mathcal{L}'$.}
7: $\quad \mathcal{R} := \textsc{Merge}_e(\mathcal{Q}, \mathcal{R}')$. {Both $\mathcal{Q}$ and $\mathcal{R}'$ are ordered by $<_e$.}

We have noted [3] that, due to the use of a recursion stack, algorithms that use recursion generally are not in-place. However, if we assume the existence of an in-place partitioning algorithm $\textsc{InPlaceSplit}(\mathtt{A}, b, e, \ell_b, \ell_e)$ that partitions $\mathcal{L} := \mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ into $\mathcal{L}' := \mathtt{A}[\ell_b, \ldots, \ell_c - 1]$ and $\mathcal{Q} := \mathtt{A}[\ell_c, \ldots, \ell_e - 1]$ and returns the split index ℓ_c, Algorithm 3, can be made in-place using this subroutine inside a simple `repeat-until`-loop (see Algorithm 4):

Algorithm 4. Algorithm $\textsc{InPlaceSearchInStrip}(\mathtt{A}, b, e, \ell_b, \ell_e)$

Require: $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ is ordered by $<_b$; each s_i spans $\langle b, e \rangle$.
Ensure: $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ is ordered by $<_e$.
1: Let $\ell_c := \ell_e$.
2: **repeat**
3: $\quad$ Let $\ell := \ell_c$.
4: $\quad \ell_c := \textsc{InPlaceSplit}(\mathtt{A}, b, e, \ell_b, \ell)$ $\{\mathcal{L}' = \mathtt{A}[\ell_b, \ldots, \ell_c - 1]; \mathcal{Q} = \mathtt{A}[\ell_c, \ldots, \ell - 1]\}$
5: $\quad$ **if** $\ell_c > \ell_b$ **then**
6: $\quad\quad$ Find all intersections of $\mathcal{Q}$ and $\mathcal{L}'$ inside $\langle b, e \rangle$ using a synchronized scan.
7: **until** $\ell_c = \ell_b$
8: Repeatedly identify and merge staircases into $\mathcal{R} = \mathtt{A}[\ell_b, \ldots, \ell_e - 1]$.

Because Algorithm $\textsc{InPlaceSplit}$ partitions the set $\mathcal{L}$ (that is the subarray $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$) such that the elements of the non-staircase set $\mathcal{L}'$ always appear in the front of the subarray, Algorithm 4 only needs to maintain the following pointers: one pointer to the beginning and the end of the original set $\mathcal{L}$, say ℓ_b and ℓ_e, and one pointer ℓ to the end of the *current* set $\mathcal{L}$. The final merging step is implemented as follows: starting from ℓ and advancing towards at most ℓ_e, we find the index ℓ' denoting the end of the next staircase to be merged by exploiting the fact that each staircase is sorted according to $<_e$. We then merge the staircases $\mathtt{A}[\ell_b, \ldots, \ell - 1]$ and $\mathtt{A}[\ell, \ldots, \ell' - 1]$ according to $<_e$ using an in-place merging algorithm [9], update ℓ with the value of ℓ', and repeat the merging process as long as $\ell < \ell_e$. By the argument used above, the overall runtime for merging all staircases is linear in $|\mathcal{L}| + |\mathrm{Int}_{b,e}(\mathcal{L})|$.

Let us now come back to the problem of finding a stable in-place partitioning of $\mathcal{L} = \mathcal{L}' + \mathcal{Q}$. In order to still obtain both sets in sorted order, we proceed as

follows: we implement InPlaceSplit such as to use the approach of Algorithm SortedSubsetSelection [3] to stably move the non-stairs, i.e. the set $\mathcal{L}'$ to the front of $\mathcal{L}$. We can do so incrementally, as we 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 sort the segments in $\mathcal{Q}$ using an in-place sorting algorithm, e.g. *heapsort* [8].

Lemma 1. *Algorithm* InPlaceSplit*, when invoked at a node v of the recursion tree $\mathcal{T}$, runs in time $\mathcal{O}(|\mathcal{L}|+|\mathrm{Int}_{b,e}(\mathcal{L})|+H_v)$ where $\sum_{v\in\mathcal{T}} H_v \in \mathcal{O}(n\log_2 n)$.*

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 staircase exactly once, so the overall running time of sorting all staircases is in $\mathcal{O}(n\log_2 n)$.

2.3 The "Divide" and "Conquer" Phases

The main concept of Balaban's algorithm is to report a pair (s,t) of intersecting segment at the highest node v in the recursion tree where one of the segments, say s, is part of the staircase $\mathcal{Q}_v$ spanning the strip $\langle b,e\rangle$ assigned to v and where the intersection point lies within $\langle b,e\rangle$. The other segment t cannot be part of the staircase at v because segments in the same staircase do not intersect. There are three possible situations: (1) t crosses the left boundary of $\langle b,e\rangle$, (2) t lies completely within $\langle b,e\rangle$, or (3) t crosses the right boundary of $\langle b,e\rangle$.[2]

Invariant 1 implies that, upon entering a node v, all segments intersecting the left boundary of the strip $\langle b,e\rangle$ are available in the form of an ordered set $\mathcal{L}_v$ that is sorted according to $<_b$. Similarly, Invariant 3 requires the existence of an ordered set $\mathcal{R}_v$ (which, in the parameter list of Balaban's algorithm (Algorithm 5) is a reference parameter to be modified by the algorithm) that contains the segments crossing the right strip boundary—again in sorted order. The unordered set $\mathcal{I}_v$ contains all segments that lie completely within the strip $\langle b,e\rangle$. Handling Situations (1)–(3) then consists of computing $\mathrm{Int}_{b,e}(\mathcal{Q}_v,\mathcal{L}_v)$, $\mathrm{Int}_{b,e}(\mathcal{Q}_v,\mathcal{I}_v)$, and $\mathrm{Int}_{b,e}(\mathcal{Q}_v,\mathcal{R}_v)$, the sets of intersections inside $\langle b,e\rangle$ and induced by segments in the staircase $\mathcal{Q}_v$ and in the sets $\mathcal{L}_v$, $\mathcal{I}_v$, and $\mathcal{R}_v$, respectively. The intersections inside $\langle b,e\rangle$ that do not involve any $s\in\mathcal{Q}_v$ are found recursively.

To obtain a logarithmic depth of recursion, Balaban subdivides the set of segments that are not part of the staircase at the current node in such a way that the same number of *endpoints* is processed in each of the recursive call. Under the simplifying assumption that the x-coordinates of the segments are the

[2] 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.

integers $[1 \ldots 2n]$,[3] this corresponds to subdividing with respect to the median $c := \lfloor (b+e)/2 \rfloor$, and the Balaban's algorithm can be stated as follows ($\mathtt{LSON}(v)$ and $\mathtt{RSON}(v)$ denote the left and right child of v, respectively):

Algorithm 5. The algorithm $\textsc{TreeSearch}(\mathcal{L}_v, \mathcal{I}_v, b, e, \mathcal{R}_v)$ [1]

1: **if** $e - b = 1$ **then**
2: $\quad \textsc{SearchInStrip}_{b,e}(\mathcal{L}_v, \mathcal{R}_v)$;
3: **else**
4: $\quad \textsc{Split}_{b,e}(\mathcal{L}_v, \mathcal{Q}_v, \mathcal{L}_{\mathtt{LSON}(v)})$; {Compute staircase.}
5: $\quad$ Compute $\mathrm{Int}_{b,e}(\mathcal{Q}_v, \mathcal{L}_{\mathtt{LSON}(v)})$. {Handle Situation (1).}
6: $\quad c := \lfloor (b+e)/2 \rfloor$;
7: $\quad$ Construct $\mathcal{I}_{\mathtt{LSON}(v)}$ and $\mathcal{I}_{\mathtt{RSON}(v)}$ from $\mathcal{I}_v$;
8: $\quad \textsc{TreeSearch}(\mathcal{L}_{\mathtt{LSON}(v)}, \mathcal{I}_{\mathtt{LSON}(v)}, b, c, \mathcal{R}_{\mathtt{LSON}(v)})$;
9: $\quad$ Construct $\mathcal{L}_{\mathtt{RSON}(v)}$ from $\mathcal{R}_{\mathtt{LSON}(v)}$ by insertion/deletion;
10: $\quad \textsc{TreeSearch}(\mathcal{L}_{\mathtt{RSON}(v)}, \mathcal{I}_{\mathtt{RSON}(v)}, c, e, \mathcal{R}_{\mathtt{RSON}(v)})$;
11: $\quad$ Compute $\mathrm{Int}_{b,e}(\mathcal{Q}_v, \mathcal{R}_{\mathtt{RSON}(v)})$. {Handle Situation (3).}
12: $\quad$ Compute $\mathrm{Loc}(\mathcal{Q}_v, \{s\})$ for each $s \in \mathcal{I}_v$.
13: $\quad$ Compute $\mathrm{Int}(\mathcal{Q}_v, \mathcal{I}_v)$ based upon $\mathrm{Loc}(\mathcal{Q}_v, \mathcal{I}_v)$. {Handle Situation (2).}
14: $\quad \mathcal{R}_v := \textsc{Merge}_e(\mathcal{Q}_v, \mathcal{R}_{\mathtt{RSON}(v)})$; {Establish Invariant (3).}

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 $\Omega(\log_2 n)$ 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, is infeasible in an in-place setting.

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

Invariant A: All segments that cross the left boundary of $\langle b, e \rangle$ are stored in sorted $<_b$ order at the front of $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ (see Invariant (1)).

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 $\textsc{InPlaceTreeSearch}(\mathtt{A}, b, e, \ell_b, \ell_e)$:

[3] 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.

Invariant C: The strip boundaries $\langle b', e' \rangle$ of the "parent strip" are known.
Invariant D: There exists an integer $i \in \{0, \dots \ell_b - \ell_e\}$ such that all segments of $\mathtt{A}[\ell_b, \ell_e - 1]$ that do not cross the right strip boundary are stored in $\mathtt{A}[\ell_b, \dots, \ell_b + i - 1]$ and that all other segments are stored in $\mathtt{A}[\ell_b + i, \dots, \ell_e - 1]$ sorted according to $<_e$ (see Invariant (3)).

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 move it (using a linear number of swaps) to the front of the staircase $\mathcal{Q}_v$. 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 $\mathcal{L}_v$ and/or $\mathcal{R}_v$ (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.

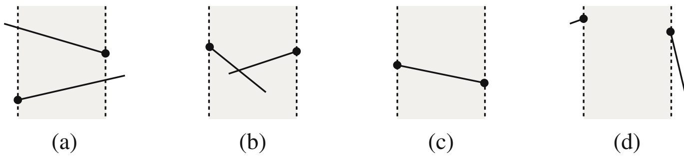
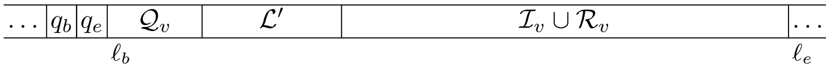


Fig. 3. Some of the configurations of segments whose endpoints define $\langle b, e \rangle$

Any combination of these configurations is possible, 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 $\mathcal{O}(\log_2 n)$ 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 InPlaceSplit to compute the staircase $\mathcal{Q}_v$ and interchange the subarrays containing $\mathcal{L}'$ and $\mathcal{Q}_v$[4] such that the subarray looks as follows:



We also maintain a "staircase" stack $\mathcal{S}$ of depth $\mathcal{O}(\log_2 n)$ to indicate whether $\mathcal{Q}_v$ contains zero, one, or more segments in addition to $(q_b \cup q_e) \cap \mathcal{Q}_v$. 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 $\mathcal{Q}_v$ in front of q_b, and prior to going into the "left recursion" we also prepare for establishing Invariant (C):

[4] Interchanging two blocks $\mathtt{A}[x_0, \dots, x_1 - 1]$ and $\mathtt{A}[x_1, \dots, x_2 - 1]$ can be done in-place and in linear time by first using swaps to revert the order of the elements in each of the blocks separately and by then reverting the order of $\mathtt{A}[x_0, \dots, x_2 - 1]$ again using swaps. Katajainen and Pasanen [11] attribute this to "computer folklore".

We determine the segment q_c whose endpoint induces the right boundary of the left subslab (see Section 2.4 for the details of how to do this in-place) and shift the segments q_e, q_b, and q_c in front of $\mathcal{L}_{\mathtt{LSON}(v)} := \mathcal{L}'$. We then establish Invariant (B) by moving all elements in $\mathcal{I}_{\mathtt{LSON}(v)} \cup \mathcal{R}_{\mathtt{LSON}(v)}$ to immediately behind $\mathcal{L}_{\mathtt{LSON}}(v)$ using simple swaps (we use $\mathcal{N}_v$ to denote the set of segments not moved). We also update ℓ_b to point to the first element in $\mathcal{L}_{\mathtt{LSON}(v)}$ and update ℓ_e to point to the first segment not in $\mathcal{I}_{\mathtt{LSON}(v)} \cup \mathcal{R}_{\mathtt{LSON}(v)}$:

...	$\mathcal{Q}_v$	q_e	q_b	q_c	$\mathcal{L}_{\mathtt{LSON}(v)}$	$\mathcal{I}_{\mathtt{LSON}(v)} \cup \mathcal{R}_{\mathtt{LSON}(v)}$	$\mathcal{N}_v$	...
					ℓ_b		ℓ_e	

By Invariant (D), we know that upon returning from the "left recursive" call, the array has the following form ($\mathcal{O}_{\mathtt{LSON}(v)}$ denotes the segments whose right endpoint lies inside the left subslab):

...	$\mathcal{Q}_v$	q_e	q_b	q_c	$\mathcal{O}_{\mathtt{LSON}(v)}$	$\mathcal{R}_{\mathtt{LSON}(v)}$	$\mathcal{N}_v$	...
					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 $\mathcal{C}$). As doing this we have also recovered 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 interchange $\mathcal{R}_{\mathtt{LSON}(v)} \cup \mathcal{N}_v$ and $\mathcal{O}_{\mathtt{LSON}(v)}$. Note that $\mathcal{R}_{\mathtt{LSON}(v)} = \mathcal{L}_{\mathtt{RSON}(v)}$ in our setting as these sets only differ by the segment q_c which is stored separately. Also, relative to the subslab at $\mathtt{RSON}(v)$, $\mathcal{N}_v = \mathcal{I}_{\mathtt{RSON}(v)} \cup \mathcal{R}_{\mathtt{RSON}(v)}$. We update ℓ_e to point to the first element in $\mathcal{O}_v$, and shift q_e, q_b, and q_c.

...	$\mathcal{Q}_v$	q_b	q_c	q_e	$\mathcal{L}_{\mathtt{RSON}(v)}$	$\mathcal{I}_{\mathtt{RSON}(v)} \cup \mathcal{R}_{\mathtt{RSON}(v)}$	$\mathcal{O}_{\mathtt{LSON}(v)}$	...
					ℓ_b		ℓ_e	ℓ

By Invariant (D), we know that upon returning from the "right recursive" call, the array has the following form:

...	$\mathcal{Q}_v$	q_b	q_c	q_e	$\mathcal{O}_{\mathtt{RSON}(v)}$	$\mathcal{R}_{\mathtt{RSON}(v)}$	$\mathcal{O}_{\mathtt{LSON}(v)}$	...
					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 of whether q_c crosses the right boundary of $\langle b, e\rangle$ or not, we insert q_c into $\mathcal{R}_{\mathtt{RSON}(v)}$ or into $\mathcal{O}_{\mathtt{RSON}(v)}$. 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 $\mathcal{Q}_v$ span $\langle b, e\rangle$, are non-intersecting, and are ordered by $<_b$, we determine the start of the subarray in which $\mathcal{Q}_v$ is stored. We then interchange the blocks such that $\mathcal{O}_{\mathtt{LSON}(v)}$ and $\mathcal{O}_{\mathtt{RSON}(v)}$ as well as $\mathcal{Q}_v$ and $\mathcal{R}_{\mathtt{RSON}(v)}$ appear next to each other. Finally, we use an in-place merging algorithm [9] to construct $\mathcal{R}_v (= \mathcal{R}_{\mathtt{RSON}(v)}) \cup \mathcal{Q}_v$, thus establishing Invariant (D). Note that all interchanging, shifting, and scanning done so far takes time linear in $|\mathcal{L}_v \cup \mathcal{I}_v \cup \mathcal{R}_v|$.

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:

Algorithm 6. Algorithm InPlaceTreeSearch($\mathtt{A}, b, e, \ell_b, \ell_e$)

1: **if** $\langle b, e\rangle$ does not contain any endpoint of a segment $s \in \mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ **then**
2: InPlaceSearchInStrip($\mathtt{A}, b, e, \ell_b, \ell_e$);
3: **else**
4: Let ℓ_l be the index of the first segment in $\mathtt{A}[\ell_b, \ldots, \ell_e - 1]$ that does not cross the left strip boundary. $\{\mathcal{L}_v = \mathtt{A}[\ell_b, \ldots, \ell_l - 1]\}$
5: $\ell_c :=$ InPlaceSplit($\mathtt{A}, b, e, \ell_b, \ell_l$). $\{\mathcal{L}' = \mathtt{A}[\ell_b, \ldots, \ell_c - 1]; \mathcal{Q}_v = \mathtt{A}[\ell_c, \ldots, \ell_l - 1]\}$
6: Stably exchange the subarrays $\mathtt{A}[\ell_b, \ldots, \ell_c - 1]$ and $\mathtt{A}[\ell_c, \ldots, \ell_l - 1]$.
7: Compute $\mathrm{Int}_{b,e}(\mathcal{Q}_v, \mathcal{L}')$. {Handle Situation (1).}
8: **for** each segment $s \in \mathtt{A}[\ell_l, \ldots, \ell_e - 1]$ than lies inside $\langle b, e\rangle$ **do**
9: Using binary search, locate the lower endpoint of s w.r.t. the stairs of $\mathcal{Q}_v$ and compute $\mathrm{Int}_{b,e}(\mathcal{Q}_v, \{s\})$. {Handle Situation (2).}
10: Find the index of the median of the endpoints inside the current strip. Let c be the x-coordinate of this endpoint.
11: Establish Invariants (A) and (B), update ℓ_b and ℓ_e.
12: InPlaceTreeSearch($\mathtt{A}, b, c, \ell_b, \ell_e$);
13: Recover the old values of e and ℓ_e. Establish Invariants (A) and (B), update ℓ_b and ℓ_e.
14: InPlaceTreeSearch($\mathtt{A}, c, e, \ell_b, \ell_e$);
15: Recover the old values of b and ℓ_e.
16: Compute $\mathrm{Int}_{b,e}(\mathcal{Q}_v, \mathcal{R}')$. {Handle Situation (3); check for duplicates.}
17: $\mathcal{R}_v :=$ Merge$_e(\mathcal{Q}_v, \mathcal{R}_{\mathtt{RSON}(v)})$; Establish Invariant (D).

Due to space constraints, the above description does not explicitly contains code for simulating the two "recursive" calls to InPlaceTreeSearch, since we have shown previously [3] that it is possible to handle these calls using a stack of $\mathcal{O}(\log_2 n)$ bits, that is using $\mathcal{O}(1)$ extra space. 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. We need, however, to fill in the details of how to select the endpoint with median x-coordinate.

2.4 Selecting the Median In-Place

When selecting the median of the endpoints in line 10 of Algorithm 6, we have to do so while maintaining the set $\mathcal{L}'$ in sorted 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 make sure to only

process segments not spanning $\langle b, e \rangle$. Doing so, we can guarantee that each segment participates in $\mathcal{O}(\log_2 n)$ 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 $\langle b, e \rangle$. The segments that have at least one endpoint in $\langle b, e \rangle$ are then stored consecutively in $\mathtt{A}[\ell_b + i, \ldots, \ell_e - 1]$ (for some $i \in \{0, \ldots, \ell_e - \ell_b\}$).

Lemma 2. *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 in $\mathcal{O}(m \log_2 m)$ time.*

Proof. We simulate a plane-sweep over the set of segments and maintain the current x-coordinate ξ as well as the number o of endpoints inside $\langle b, e \rangle$ that have already be swept over. The segments are maintained in-place in a heap-based priority queue $\mathcal{H}$, the priority of s being the smallest x-coordinate of s's endpoints that still is at least ξ. When deleting the minimal element s from $\mathcal{H}$ we increment o iff $\xi \in [b, e]$ and re-insert s iff the x-coordinate of its right endpoint is larger than ξ. If $o = k$, we report s and ξ, else we continue. As there are at most $2m$ priority queue operations, the algorithm runs in time $\mathcal{O}(m \log_2 m)$.

After we have found the median using the algorithm implied by Lemma 2, we need to restore $\mathcal{L}'$ in sorted $<_b$ order. To this end, we then select the elements from $\mathtt{A}[\ell_b + i, \ldots, \ell_e - 1]$ that cross the left strip boundary, sort them in-place by $<_b$, and then merge them in-place with the segments in $\mathtt{A}[\ell_b, \ldots, \ell_b + i - 1]$.

Lemma 3. *The global cost incurred by median-finding is $\mathcal{O}(n \log_2^2 n)$.*

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 $\mathcal{O}(\log_2 n)$ cost per level for median-finding (see Lemma 2). We charge s an additional $\mathcal{O}(\log_2 n)$ cost per level for the at most one sorting step it participates in (when restoring $\mathcal{L}'$). As all other operations require only linear time per level, the global cost incurred by median-finding is $\mathcal{O}(n \log_2^2 n)$ as claimed.

2.5 Analysis of the Running Time

For the main part of the analysis, Balaban's results carry over. Using the notation $\mathcal{S}_v = \mathcal{L}_v \cup \mathcal{I}_v \cup \mathcal{R}_v$, the following theorem holds for the recursion tree $\mathcal{T}$:

Theorem 1 (Theorem 2 in [1]). $\sum_{v \in \mathcal{T}} |\mathcal{S}_v| \leq n \lceil 4 \log_2 n + 5 \rceil + 2k$.

To make the algorithm in-place, we had to resort to some algorithmic techniques not captured in Balaban's analysis. The global extra cost for making the algorithm Split in-place is $\mathcal{O}(n \log_2 n)$ (see Lemma 1). From Theorem 1 follows that the overall extra cost for establishing the invariants is $\mathcal{O}(n \log_2 n + k)$ 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 and restoring the original order of the elements. By Lemma 3, the overall cost for this is in $\mathcal{O}(n \log_2^2 n)$. The last component of the analysis is the `for`-loop in Line 8 of Algorithm 4: Each iteration

of this loop takes $\mathcal{O}(\log_2 |\mathcal{Q}_v|) \subseteq \mathcal{O}(\log_2 n)$ time, and each of the n segments can be part of $\mathcal{I}_w$ for $\mathcal{O}(\log_2 n)$ nodes $w \in \mathcal{T}$. Combining this with Balaban's original analysis, we obtain the main result of this paper:

Theorem 2. *All k intersections induced by a set of n segments in the plane can be computed in $\mathcal{O}(n \log_2^2 n + k)$ 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 $\mathcal{O}(n \log_2 n + k)$?

References

1. I. J. Balaban. An optimal algorithm for finding segments [*sic!*] intersections. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pages 211–219, New York, 1995. ACM Press.
2. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, Sept. 1979.
3. P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry: Theory & Applications*, 2005. To appear, accepted November 2004. An extend abstract appeared in *Proceedings of the 20th European Workshop on Computational Geometry*, pages 65–68, 2004.
4. H. Brönnimann, T. M.-Y. Chan, and E. Y. Chen. Towards in-place geometric algorithms. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pages 239–246. ACM Press, 2004.
5. H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Optimal in-place planar convex hull algorithms. *Theoretical Computer Science*, 321(1):25–40, June 2004. An extended abstract appeared in the *Proceedings of the Fifth Latin American Symposium on Theoretical Informatics* (2002), pages 494–507.
6. B. M. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.
7. E. Y. Chen and T. M.-Y. Chan. A space-efficient algorithm for line segment intersection. In *Proceedings of the 15th Canadian Conference on Computational Geometry*, pages 68–71, 2003.
8. R. W. Floyd. Algorithm 245: Treesort. *Communications of the ACM*, 7(12):701, Dec. 1964.
9. V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1–2):159–181, Apr. 2000.
10. J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, second edition, 2004.
11. J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32:580–585, 1992.
12. D. M. Mount. Geometric intersection. In Goodman and O'Rourke [10], chapter 38, pages 857–876.