Space-Efficient Plane-Sweep Algorithms

Amr Elmasry∗

Department of Computer Engineering and Systems, Alexandria University, Alexandria 21544, Egypt

Frank Kammer†

Institut für Informatik, Universität Augsburg, Germany

Abstract

We introduce space-efficient plane-sweep algorithms for basic planar geometric problems. It is assumed that the input is in a read-only array of \(n\) items and that the available workspace is \(\Theta(s)\) bits, where \(\log n \leq s \leq n \cdot \log n\). In particular, we give an almost-optimal algorithm for finding the closest pair among a set of \(n\) points that runs in \(O(n^2/s + n \cdot \log s)\) time. We give a simple algorithm to enumerate the intersections of \(n\) line segments that runs in \(O((n^2/s) \cdot \log^2 s + k)\) time, where \(k\) is the number of reported intersections. When the segments are axis-parallel, we give an \(O(n^2/s + n \cdot \log s)\)-time algorithm for counting the intersections, and an algorithm for enumerating the intersections whose running time is \(O((n^2/s) \cdot \log s \cdot \log \log s + n \cdot \log s + k)\). We also present space-efficient algorithms to calculate the measure of axis-parallel rectangles.

Keywords: space-efficient, plane-sweep algorithm, closest pair, line-segment intersection, measure

ACM classification: F.2.2

1 Introduction

Because of the rapid growth of the input data sizes in nowadays applications, algorithms that are designed to efficiently utilize space are becoming even more important than before. One other reason for the need for space-efficient algorithms is the limitations in the memory sizes that can be deployed to modern embedded systems. In accordance, many algorithms have been developed with the objective of optimizing the time-space product.

Several models of computation have been considered for the case when writing in the input area is restricted. In the multi-pass streaming model [20] the input is assumed to be held in a read-only sequentially-accessible media, and the main optimization target is the number of passes an algorithm makes over the input. In the read-only random-access model [17]—the model that we consider in this paper—the input is assumed to be stored on a read-only randomly-accessible media, and arithmetic operations on operands that fit in a computer...
word are assumed to take constant time each. Another model, introduced in [9], allows the input to be permuted but not destroyed. For a variant of the latter model, called the restore model [11], the input array is allowed to be modified while answering a query but it has to be restored to its original state afterwards.

Furthermore, we assume that the available workspace is \( \Theta(s) \) bits, where \( \lfloor \log n \rfloor \leq s \leq n \cdot \lfloor \log n \rfloor \). In addition, and as usual, it is assumed that the input coordinates for all the problems have a space-efficient representation, i.e., can be represented with \( O(\log n) \) bits, and each operation can be performed in constant time. Next, we survey some results for the read-only random-access model.

Pagter and Rauhe [24] gave an asymptotically-optimal algorithm for sorting \( n \) elements, which runs in \( O(n^2/s + n \cdot \log s) \) time. A simplified variation of this sorting algorithm is given in [2]. Beame [6] established a matching \( \Omega(n^2) \) lower bound for the time-space product for sorting in the stronger branching-program model. Several papers [15, 17, 22, 25] considered the selection problem in the read-only random-access model. Elmasry et al. [16] introduced space-efficient algorithms for basic graph problems. Concerning geometric problems, Darwish and Elmasry [14] gave an optimal convex-hull construction algorithm that runs in \( O(n^2/s + n \cdot \log s) \) time. Konagaya and Asano [18] gave an algorithm for reporting line-segments intersections that runs in \( O((n^2/\sqrt{s}) \cdot \sqrt{\log n} + k) \) time, where \( k \) is the number of reported intersections. Other papers that deal with space-efficient geometric algorithms include [1, 3, 5]. Recently, Korman et al. [19] gave space-efficient algorithms for triangulations and for constructing Voronoi diagrams.

In this paper we give space-efficient plane-sweep algorithms for solving planar geometric problems. As a building block for our algorithms we use the memory-adjustable navigation pile [2]: a priority-queue-like data structure that allows to stream the elements of the input array in sorted order in \( O(n/s + \log s) \) time per element in the read-only random-access model of computation. Another ingredient that we use in some of our algorithms is a rank-select data structure [12, 21]. A rank-select data structure can be built on a vector of \( n \) bits using \( O(n) \) time and \( o(n) \) extra bits, and supports in \( O(1) \) time the queries \( \text{rank}(i) \), which returns the number of 1-bits in the first \( i \) positions of the bit vector, and \( \text{select}(j) \), which returns the index of the \( j \)-th 1-bit in the bit vector. In accordance, one can sequentially scan the entries that have 1-bits each in \( O(1) \) time.

In Section 2 we introduce a general lemma that can be used to design space-efficient algorithms, and later employ it in our algorithms. In Section 3 we give a simple algorithm for enumerating the intersections among \( n \) line-segments that runs in \( O((n^2/s) \cdot \log^2 s + k) \) time, where \( k \) is the number of intersections returned. Our algorithm is asymptotically faster than that of Konagaya and Asano for all values of \( s \). In Section 4 we give a space-efficient algorithm for the closest-pair problem whose running time is \( O(n^2/s + n \cdot \log s) \), where \( n \) is the number of points. A lower bound of \( \Omega(n^2-\epsilon) \) was given by Yao [26] for the time-space product of the element-distinctness problem, where \( \epsilon \) is an arbitrarily small positive constant. This lower bound applies for the closest-pair problem, indicating that our algorithm is close to optimal. Subsequently, we revisit the line-segments-intersections problem for the special case when the line segments are axis-parallel (horizontal or vertical). In Section 5 we give an algorithm for counting the intersections whose running time is \( O(n^2/s + n \cdot \log s) \). In Section 6 we sketch a technique to represent the sweep line for special plane-
sweep algorithms using fewer bits than usual, and then utilize this technique in Section 7 for enumerating the intersections in \( O(n^2/s) \cdot \lg s \cdot \lg s + n \cdot \lg s + k \) time. In Section 8 we show that the measure (the size of the area of the union) for axis-parallel rectangles can be calculated in \( O(n^2/s) \cdot \lg n + n \cdot \lg s \) time if the corners of the rectangles are given in sorted x-coordinate order, and in \( O(n^2/s) \cdot \lg s + (n^2/s) \cdot \lg^2 s \) time otherwise. This means that, if the input is sorted and \( s = \Omega(n) \), our algorithm matches the \( O(n \lg n) \) running time of the standard algorithm of Bentley 7. We conclude the paper in Section 9 with some comments.

2 A Stretching Lemma

A problem is called decomposable when a global solution can be easily computed by combining the partial solutions for overlapping subsets of the input. Examples of such problems are the closest-pair problem, the farthest-pair problem, and the (red-blue) line-segments intersections problem. Assume that the available workspace is enough to only handle a subset of the input that comprises \( O(r) \) elements at a time, for some parameter \( r \leq n \). The following idea can be applied.

Let \( n \) be the number of elements in the input array. Divide the array into \([n/r]\) batches \( B_1, \ldots, B_{[n/r]} \) of at most (the last batch may have less) \( r \) consecutive elements each and proceed as follows: For \( i = 1, \ldots, [n/r] \) and \( j = i + 1, \ldots, [n/r] \), apply the algorithm within \( B_i \cup B_j \). Compute the overall answer by combining the partial results. As we try all pairs of subproblems, the algorithm correctly explores all the possible subproblems \( B_i \cup B_j \) for some \( i \) and \( j \), and accordingly produces the output correctly.

The number of the subproblems handled in sequence is \( \Theta(n^2/r^2) \). Let the time needed to solve a subproblem of size \( O(r) \) be \( t(r) \), it follows that the overall time spent by the algorithm is \( O((n^2/r^2) \cdot t(r)) \), in addition to the time needed to combine the partial results (which is shorter for the problems we deal with).

Suppose we know how to solve a problem \( P \) of size \( n \) using \( s' = \Theta(f(n)) \) bits in \( O((n^2/s') \cdot g(s') + n \cdot \lg s') \) time, where \( \lg n \leq f(n) \leq n \cdot \lg n \) and \( g(s') = \Omega(1) \). Assume that the problem \( P \) is decomposable. For any \( s \) where \( \lg n \leq s \leq s' \), we can solve instances of \( P \) of size \( r = \lceil f^{-1}(s) \rceil \) using \( s \) bits in \( t(r) = O((n^2/s) \cdot g(s) + r \cdot \lg s) \) time. By applying the above construction, we can solve any instance of \( P \) of size \( n \) in \( O((n^2/r^2) \cdot t(r)) = O((n^2/s) \cdot g(s) + (n^2/r) \cdot \lg s) = O((n^2/s) \cdot g(s) + (n^2/f^{-1}(s)) \cdot \lg s) \) time. In particular, when \( f(n) = O(n/\lg n) \) it follows that \( f^{-1}(s) = \Omega(s \cdot \lg s) \), and we can solve instances of \( P \) of size \( n \) in \( O((n^2/s) \cdot g(s)) \) time for any \( \lg n \leq s \leq s' \).

3 Line-Segments Intersections

Given a set of \( n \) line segments in the plane, the line-segments-intersections problem is to enumerate all the intersection points among these line segments. The counting version of the problem is to only produce the number of intersections. We give next a straightforward application of the stretching lemma.

An optimal algorithm to enumerate all the intersections runs in \( O(n \cdot \lg n + k) \) time 4, where \( n \) is the number of the input line segments and \( k \) is the number
of the intersections returned. This algorithm mandates to store the indices of the \( n \) segments using \( \lg n \) bits per index. If the available workspace is \( \Theta(s) \) bits with \( \lg n \leq s \leq n \cdot \lg n \), we can apply the algorithm on batches of size \( O(r) \) line segments, where \( r = [s/\lg s] \). It follows that \( t(r) = O(r \cdot \lg r + k') \), where \( k' \) is the number of intersections found when solving that subproblem. The reported intersections are the union of the intersections found by solving the subproblems. Hence, the overall time for this algorithm is \( O((n^2/r^2) \cdot t(r)) = O((n^2/r) \cdot \lg r + k) = O((n^2/s) \cdot \lg^2 s + k) \).

In the same vein, one can adapt the standard counting algorithm that runs in \( O(n \cdot \lg n) \) time \cite{Nievergelt1992} for a space-efficient variant. The running time for the counting version will be \( O((n^2/s) \cdot \lg^2 s) \).

Lemma 3.1 Given a read-only array of \( n \) elements and \( \Theta(s) \) bits of workspace, where \( \lg n \leq s \leq n \cdot \lg n \), the planar line-segments-intersections enumeration problem can be solved in \( O((n^2/s) \cdot \lg^2 s + k) \) time, where \( k \) is the number of intersections returned. The counting version can be solved in \( O((n^2/s) \cdot \lg^2 s) \) time.

4 Closest Pair

Given a set of \( n \) points in the plane, with coordinates that can be represented in \( O(\log n) \) bits each, the planar closest-pair problem is to identify a pair of points that are closest to each other or to compute the distance between such a pair.

Assume that the available workspace is \( \Theta(s) \) bits, where \( \sqrt{n} \cdot \lg n \leq s \leq n \cdot \lg n \). First, produce the points in sorted order according to their \( x \)-coordinate value using an adjustable navigation pile, and consider them in order in groups having \( \lfloor s/\lg n \rfloor \) points each (except possibly the last group). Call the vertical regions containing these groups the **vertical stripes**, and call the boundary vertical lines separating the vertical stripes the **vertical separators**. Since there are at most \( m = \lceil (n/s) \cdot \lg n \rceil \) vertical stripes, the \( x \)-coordinate values of all the vertical separators can be stored in \( O((n/s) \cdot \lg^2 n) \) bits, which is \( O(s) \) as long as \( s = \Omega(\sqrt{n} \cdot \lg n) \). It follows that the coordinates of all the separators can be simultaneously stored within the available workspace. Note that the points of a vertical stripe can all fit in the available workspace. Thus, a standard closest-pair algorithm \cite{Nievergelt1992} can be applied to identify the closest pair among the points of each vertical stripe one after the other. Find the pair with the minimum closest distance among all the vertical subproblems, and call this minimum distance \( \delta \).

Next, produce the points in sorted order according to their \( y \)-coordinate values using another adjustable navigation pile. Only retain the points that lie within a horizontal distance \( \delta \) from any of the vertical separators and ignore the other points. Call the selected points the **candidate points**. Consider the candidate points in the \( y \)-coordinate order in groups having \( 7 \cdot m \) points each (except the last group that may have less points). Call the horizontal regions containing these groups the **horizontal stripes**. Note that the points of a horizontal stripe can be stored in \( O((n/s) \cdot \lg^2 n) = O(s) \) bits, which can all fit in the available workspace. Since the candidate points within two horizontal stripes can all fit in the available workspace, apply a standard closest-pair algorithm to identify the closest pair among the candidate points of every two consecutive horizontal stripes in order. Let \( \delta' \) be the minimum closest distance among all the horizontal subproblems. Finally, return \( \min(\delta, \delta') \) as the closest-pair distance.
We prove next the correctness of the algorithm. Since the distance between any pair of points within a vertical stripe is at least \( \delta \), any point that is at horizontal distance more than \( \delta \) from all the vertical separators can not be closer than \( \delta \) to any other point. We then need to only proceed with the candidate points that lie within a horizontal distance \( \delta \) from any of the vertical separators.

Fix a candidate point \( p \). Given a specific vertical separator, for the candidate points above \( p \) to be closer than \( \delta \) to \( p \) they must lie together with \( p \) within a \( 2\delta \times \delta \) rectangle centered at the vertical separator. It follows that there could be at most 7 points above \( p \) within a horizontal distance \( \delta \) from this separator whose distances to \( p \) are less than \( \delta \). For an illustration of this fact, see, e.g., Cormen et al. [13, Fig. 33.11]. Since there are \( m \) separators, the number of the \( p \)-related candidate points above \( p \) that have to be checked for possibly having a distance less than \( \delta \) from \( p \) is at most \( 7 \cdot m \); no other point above \( p \) can be at distance less than \( \delta \) from \( p \). Obviously, the \( p \)-related candidate points must be consecutive in the \( y \)-coordinate values. Since we store \( 7 \cdot m \) candidate points per stripe, the \( p \)-related candidate points above \( p \) lie in only two horizontal stripes, the horizontal stripe that spans \( p \) and the horizontal stripe above it.

We conclude that we need to only consider the mutual distances among the points of each two consecutive horizontal stripes. Finally, we point out that our settings allow us to store all the vertical separators as well as the points in a vertical stripe and those in a couple of horizontal stripes.

The time needed to produce the points in sorted order in both coordinates is bounded by the time for sorting using the adjustable navigation pile, which is \( O(n^2/s + n \cdot \lg s) \) [2]. The time needed to execute the standard closest-pair algorithm for all the stripes is \( O(n \cdot \lg n) = O(n \cdot \lg s) \). The time needed to check whether each point is close to one of the separators or not is \( O(n \cdot \lg n) = O(n \cdot \lg s) \) using a binary search among the \( x \)-coordinate values of the separators for each point. It follows that the running time of the algorithm is \( O(n^2/s + n \cdot \lg s) \).

Assume now that we have \( \Theta(s) \) bits available, where \( \lg n \leq s < \sqrt{m} \cdot \lg n \). Let \( r = s^2/\lg^2 s \). As \( s = \Theta(\sqrt{r} \cdot \lg r) \), we can apply the above algorithm on instances of size \( \Theta(r) \). In such case, the running time for each such instance would be \( t(r) = O(r^2/s + r \cdot \lg s) = O(r^2/s) \). We then divide the input into \( \lceil n/r \rceil \) batches of points and apply the stretching lemma. We compute the closest pair within every pair of batches, and return the overall closest pair. The space needed is indeed \( O(s) \), and the time consumed is \( O((n/r)^2 \cdot t(r)) = O(n^2/s) \).

**Lemma 4.1** Given a read-only array of \( n \) elements and \( \Theta(s) \) bits of workspace, where \( \lg n \leq s \leq n \cdot \lg n \), the planar closest-pair problem can be solved in \( O(n^2/s + n \cdot \lg s) \) time.

It is well-known that the closest-pair algorithm can be generalized from two to higher dimensions [14] to run in \( O(n \lg^{d-1} n) \) time in \( d \) dimensions. Applying the stretching lemma in a similar way as described above, we can solve the closest-pair problem in \( d \) dimensions with \( \Theta(s) \) bits of workspace in \( O(n^2/s + n \cdot \lg^{d-1} s) \) time, where \( \lg n \leq s \leq n \cdot \lg n \).
5 Counting Axis-Parallel Line-Segments Intersections

Given a set of \( n \) axis-parallel (horizontal or vertical) line segments in the plane, we want to count the intersection points among these line segments.

Assume that the available workspace is \( \Theta(s) \) bits, where \( n^{2/3} \cdot \lg n \leq s \leq n \cdot \lg n \). First, produce the endpoints of the line segments in sorted order according to their \( x \)-coordinate values using an adjustable navigation pile, and consider them in order in groups having \( \lceil \frac{s}{\lg n} \rceil \) points each (except possibly the last group). Again, call these groups the **vertical stripes**, and call the boundary vertical lines separating the vertical stripes the **vertical separators**. Since there are \( \lceil (n/s) \cdot \lg n \rceil = O(n^{1/3}) \) vertical stripes, the \( x \)-coordinate values of all the vertical separators can be stored within the available workspace. We include a line segment in a stripe if at least one of its two endpoints lie inside the stripe. The line segments of a vertical stripe can then all fit in the available workspace. Then, we apply a standard line-segments-intersections counting algorithm to each vertical stripe one after the other, and add these counts together.

Next, produce the points in sorted order according to their \( y \)-coordinate values using another adjustable navigation pile, and consider them in **horizontal stripes** having \( \lceil \frac{s}{\lg n} \rceil \) points each (except possibly the last group). It also follows that all of the \( O(n^{1/3}) \) so-called **horizontal separators** can be simultaneously stored in the available workspace. In a similar fashion as above, we apply a standard line-segments-intersections counting algorithm to each horizontal stripe one after the other, and add these counts to the accumulated count.

To avoid counting intersections twice, we truncate the horizontal segments before dealing with them such that each new endpoint lies on the closest vertical separator to the old endpoint intersecting the segment. Note that the intersections of the truncated parts of the horizontal segments with vertical segments have been accounted for while dealing with the vertical stripes.

Let \( R_{i,j} \) be the **region** formed by the intersection of the \( i \)th horizontal stripe with the \( j \)th vertical stripe. A line segment spans a region if it crosses the region’s boundaries. What is left is to account for the intersections among these spanning (horizontal and vertical) line segments and add it to the accumulated counts. We show next how to count the spanning horizontal line segments for each region. The treatment for the vertical line segments is similar. For every horizontal line segment, we locate the starting and ending regions using binary search among the separators. We store, \( b_{i,j} \), the number of horizontal segments beginning in each region and, \( f_{i,j} \), the number of horizontal segments finishing in each region. Since there are \( O(n^{2/3}) \) regions, all these values can be stored in \( O(n^{2/3} \cdot \lg n) \) bits, which is \( O(s) \). We scan the regions of every horizontal stripe sequentially while calculating, \( e_{i,j} \), the number of horizontal line segments entering \( R_{i,j} \), i.e., the number of segments that have a non-empty intersection with \( R_{i,j} \) and \( R_{i,j-1} \). We set \( e_{i,0} = 0 \) and \( e_{i,j} = e_{i,j-1} + b_{i,j-1} - f_{i,j-1} \), and assume w.l.o.g. that the boundaries of the stripes are disjoint from the endpoints of the segments. We then compute the number of horizontal segments spanning \( R_{i,j} \) as \( e_{i,j} - f_{i,j} \). The number of intersections of the spanning line segments of \( R_{i,j} \) is the product of its spanning horizontal and vertical segments. We finally sum the counts of these intersections for all the regions.

The time needed to produce the endpoints in sorted order in both coordinates
using the adjustable navigation pile is $O(n^2/s + n \cdot \lg s)$ \[2\]. The time needed to execute the standard segments-intersection counting algorithm for all the stripes is $O(n \cdot \lg s)$. The time needed to perform binary search among the separators is $O(n \cdot \lg s)$. The time needed to count the spanning segments of all the regions is constant per region and sums up to $O(n^{2/3})$. It follows that the overall running time of the algorithm is $O(n^2/s + n \cdot \lg s)$.

Assume now that we have $\Theta(s)$ bits available, where $\lg n \leq s < n^{2/3} \cdot \lg n$. Let $r = s^{3/2}/\lg^{3/2} s$. Since $s = \Theta(r^{2/3} \lg r)$, we can apply the above algorithm on instances of size $\Theta(r)$. In such case, the running time for each such instance would be $t(r) = O(r^2/s + r \cdot \lg s) = O(r^2/s)$. We then divide the input array into $\lfloor n/r \rfloor$ batches of consecutive segments and apply the stretching lemma. The space needed is $O(s)$, and the time consumed is $O((n/r)^2 \cdot t(r)) = O(n^2/s)$.

**Lemma 5.1** Given a read-only array containing the endpoints of $n$ line segments and $\Theta(s)$ bits of workspace, where $\lg n \leq s \leq n \cdot \lg n$, counting the planar axis-parallel line-segments intersections can be done in $O(n^2/s + n \cdot \lg s)$ time.

### 6 A Batching Lemma

A folk approach to solve geometric problems is to apply a plane-sweep algorithm \[5\]. We assume that the sweep line moves over the plane from left to right. Only at particular event points is an update of the status required. Typically, a plane-sweep algorithm uses a priority queue (event queue) to produce the upcoming events in order and a balanced binary search tree (status structure) to store the objects that cross the sweep line in order. Since $\Theta(n)$ objects might be part of the search tree, a standard plane-sweep algorithm needs $\Theta(n \cdot \log n)$ bits. To reduce the number of bits necessary to store the objects that are part of the status structure to $\Theta(n)$ bits, the following idea may be applied.

Suppose that our plane is divided into $\ell$ vertical and horizontal stripes such that each contains $\Theta(n/\ell)$ local objects, where an object is local for a stripe if it starts or ends within the stripe. The boundaries of the stripes are called separators. The intersection of a horizontal stripe with a vertical stripe is called a region. To apply the batching lemma, we need the following two conditions: (1) Each event of the event queue is on a vertical separator. (2) Each object that is part of the status structure starts and ends on a horizontal separator, i.e., it is a so-called vertically spanning object. Assume the available workspace is $\Theta(s)$ bits, where $s = n$. By setting $\ell = \lceil (n/s) \cdot \lg n \rceil = \Theta(\lg n)$, we have $s = \Omega(\ell \cdot \lg n)$. This allows us to store all local objects of a stripe as well as the coordinates of the separators of the stripes in the working storage.

Because of (1) and (2), we can treat the sweep line as a vertical stripe, and so it is enough to update the status structure only once per vertical stripe with a batch of objects. To ‘represent’ the status structure, we split the vertical stripe to $\ell$ regions formed by the intersections with the horizontal stripes. We store the indices of the $\Theta(n/\ell)$ vertically spanning objects of the regions of the vertical stripe in an array using a total of $\Theta((n/\ell) \cdot \lg n) = \Theta(s)$ bits. In addition, we store for each of the $\ell$ regions a bit vector of $\Theta(n/\ell)$ bits indicating whether each of these vertically spanning objects spans the region or not. Over and above, for each bit vector of a region, we build a rank-select data structure that allows us to scan the vertical spanning objects of the region in constant time per object.
The bit vectors and the rank-select structures are enough to represent the status structure. This way, the sweep line can be stored in a total of $\Theta(n)$ bits.

We use an adjustable navigation pile as our event queue to produce the events in order. Since $s = n$, the time to produce all the events in order throughout the procedure is $O(n \cdot \lg n)$. When the sweep line moves to a new vertical stripe, we update the representation of the status structure as follows: the vertical spanning objects in the new stripe are produced by the navigation pile. For each such object, the regions it spans are allocated in $O(\ell)$ time per object by simply comparing the object coordinates with the horizontal separators. The bit-vectors entries and the rank-select structures are updated accordingly. The time to update the status structure (build a new one) is $O(n)$. Throughout the algorithm, the total time to update the status structure is $O(n \cdot \ell) = O(n \cdot \lg n)$.

What is left is to show how to allocate an event point within the status structure representing the sweep line. We would be satisfied with only identifying the region that contains this event point within the vertical stripe. We do that in a straightforward manner using binary search against the $\ell$ horizontal separators, consuming $O(\lg \ell) = O(\lg \lg n)$ time per event point.

7 Enumerating Axis-Parallel Line-Segments Intersections

Assume first that the available workspace is $\Theta(s)$ bits, where $n \leq s \leq n \cdot \lg n$.

We use the same ideas as our counting algorithm. As before, we split the plane into $\ell$ horizontal and vertical stripes, where $\ell = \lceil (n/s) \cdot \lg n \rceil$. Accordingly, we can similarly enumerate the intersection points among the local parts of the line segments by applying a standard line-segments-intersection enumeration algorithm instead of the counting algorithm.

By chopping the segments, we assume from now on that all the segment endpoints lie on the boundaries of the regions and span the regions they cross. Note that each horizontal line segment that spans a region must intersect all the vertical segments spanning the same region. By applying the ideas of the batching lemma, we treat the vertical stripes from left to right as the sweep line. We store the vertical line segments that lie in the vertical stripe and build a status structure that consumes $\Theta(s)$ bits in $O(n)$ time, as illustrated in the batching lemma. Using this data structure it is possible to enumerate the vertical segments that span a given region in time proportional to the number of the reported segments. For each horizontal segment, we check if it crosses any of the regions of the sweep line. We do that using binary search for each of the horizontal segments against the $\ell$ horizontal separators. After every binary search for a horizontal segment, we query the status structure to find the vertical segments spanning the same region, and so their intersection points are computed and reported. After locating the crossing regions of all the horizontal segments with the sweep line, the sweep line is advanced to the next vertical stripe.

The total time needed to execute the standard algorithm locally within every stripe is $O(n \cdot \lg s)$, which matches the time bound to build the status structure of all vertical stripes using the batching lemma. The time to perform binary search for each of the $O(n)$ horizontal segments against the $\ell$ regions of the status
structure is $O(n \cdot \lg \ell)$. Hence, we can compute all intersection points of a vertical stripe in $O(n \cdot \lg \ell + k')$ time, where $k'$ is the number of these intersections. Since we repeat these actions for every vertical stripe as the sweep line advances, the total time is $O(n \cdot \ell \cdot \lg \ell + k) = O((n^2/s) \cdot \lg s \cdot \lg g s + k)$, where $k$ is the number of intersections returned. Since we can partition the plane into stripes using a navigation pile in $O(n^2/s + n \cdot \lg s)$ time, the total time consumed by the whole algorithm is $O((n^2/s) \cdot \lg s \cdot \lg g s + n \cdot \lg s + k)$.

Assume next that we have $\Theta(s)$ bits, where $\lg n \leq s < n$. Let $r = s$. We can therefore apply the above algorithm on instances of size $\Theta(r)$. In such case, the running time for each instance would be $t(r) = O((r^2/s) \cdot \lg s \cdot \lg g s + r \cdot \lg s + k') = O((r^2/s) \cdot \lg s \cdot \lg g s + k')$, where $k'$ is the number of intersections for the subproblem. We then divide the input into $\lceil n/r \rceil$ batches of segments and apply the stretching lemma. The time consumed is $O((n/r)^2 \cdot t(r)) = O((n^2/s) \cdot \lg s \cdot \lg g s + k)$, where $k$ is the number of intersections returned.

**Lemma 7.1** Given a read-only array containing the endpoints of $n$ line segments and $\Theta(s)$ bits of workspace, where $\lg n \leq s \leq n \cdot \lg n$, enumerating the planar axis-parallel line-segments intersections is done in $O((n^2/s) \cdot \lg s \cdot \lg g s + n \cdot \lg s + k)$ time, where $k$ is the number of intersections returned.

8 Measure of axis-parallel rectangles

In this section we consider the problem of computing the measure of a set of axis-parallel rectangles, i.e., the size of the area of the union. Bentley [7] described an $O(n \cdot \lg n)$-time algorithm that, given a set of $n$ axis-parallel rectangles, computes their measure. His algorithm can be implemented with $O(n \cdot \lg n)$ bits of working space. Bentley’s algorithm sweeps a vertical line from left to right across the rectangles and maintains the intersection of the rectangles and the sweep line. A generalization of the algorithm to $d$ dimensions was given by Chan [10]. Another algorithm to compute the measure was presented by Overmars and Yap [23].

We assume that the available workspace is $\Theta(s)$ bits, where $\lg n \leq s \leq n \cdot \lg n$, and that the corners of the rectangles are sorted by their $x$-coordinates. In addition, we assume that the coordinates can be represented with $O(\lg n)$ bits each and the operations on them can be performed in constant time each. To compute the measure of a set of $n$ axis-parallel rectangles, we use a slightly modified version of Bentley’s algorithm as a subroutine.

We partition the plane into $\ell = \Theta((n/s) \cdot \lg n)$ horizontal stripes, where each stripe consists of $\Theta(s/\lg n)$ corners of the rectangles and such that the separators of the stripes are disjoint from the corners. The rectangles with corners in a stripe are declared as *cornered within* the stripe. It follows that the rectangles cornered in one stripe can be simultaneously stored in the available workspace. A rectangle is *spanning* a stripe if its vertical line segments span the whole stripe. We process the stripes in sorted $y$-coordinate order, one after the other. By using an adjustable navigation pile, we can produce and store the set of rectangles cornered within each stripe in sequence. Before processing a stripe and storing the rectangles in the workspace, we truncate the rectangles cornered within the stripe such that they are shrunk to their intersection with the stripe. We would then run Bentley’s algorithm on the rectangles cornered
in the stripe. However, we need to also take into consideration the rectangles spanning the stripe. We show next how to do that efficiently.

Each time we run Bentley’s plane-sweep algorithm locally on a horizontal stripe, we handle the spanning rectangles as follows. For each maximal rectangular area $A$ that is part of the stripe and covered by a set of spanning rectangles to the stripe, we do not account for any part of $A$ in our area calculations, while still update the sweep line. In other words, the local plane-sweep algorithm computes the measure of (the parts of) the rectangles that are in the stripe, but outside $A$. We call $A$ a global area and account for its size separately.

Technically, we run simultaneously a local plane-sweep algorithm on a stripe and iterate over all vertical segments of the rectangles spanning the stripe in sorted $x$-coordinate order. This can be done by iterating over all rectangles and skipping over those which are not spanning the stripe. We call the modified algorithm an extended local plane-sweep algorithm. We store $z$ as the difference between the number of scanned spanning line segments that are left boundaries of a rectangle and the number of scanned spanning line segments that are right boundaries. Whenever $z$ changes from $z = 0$ to $z > 0$ at some coordinate, we record this coordinate as $x_1$. We determine the smallest $x_2$ coordinate larger than $x_1$ where the value of $z$ returns back to $z = 0$. We have found a global area, and accordingly update the total size of the global areas. (The area of the found global area $A$ is the product of $x_2 - x_1$ and the width of the horizontal stripe.) We extend the local plane-sweep algorithm not to change the measured area in between $x_1$ and $x_2$, while still update the sweep line.

Concerning the running time, the time to run a local plane-sweep algorithm within each stripe is $O((s/\lg n) \cdot \lg s)$ and the time to sequentially scan the segments for finding the spanning ones is $O(n)$ (as the segments are already sorted). Hence, the extended local plane-sweep algorithm of a stripe runs in $O(n + (s/\lg n) \cdot \lg s)$ time. In accordance, the total time to process all $\ell$ stripes is $O((\ell^2/n) \cdot \lg n + n \cdot \lg s)$.

**Lemma 8.1** Given a read-only array containing the corners of $n$ rectangles in sorted $x$-coordinate order such that each coordinate is stored in $O(\lg n)$ bits, and the available workspace is $\Theta(s)$ bits, where $\lg n \leq s \leq n \cdot \lg n$, the measure can be computed in $O((n^2/s) \cdot \lg n + n \cdot \lg s)$ time.

Note that we used the fact that the input is sorted only for efficiently getting the scanning rectangles of each stripe in sorted $x$-coordinate order. If the input is not sorted, we can instead use $\ell$ times an adjustable navigation pile to produce the segments in sorted order for each stripe. Each time we use a navigation pile to produce all segments requires $O(n^2/s + n \cdot \lg s)$ time, for a total running time of $O((n^3/s^2) \cdot \lg n + (n^2/s) \cdot \lg s \cdot \lg n)$ for the whole algorithm.

**Lemma 8.2** Given a read-only array containing the corners of $n$ rectangles such that each coordinate is stored in $O(\lg n)$ bits, and the available workspace is $\Theta(s)$ bits, where $\lg n \leq s \leq n \cdot \lg n$, the measure can be computed in $O((n^3/s^2) \cdot \lg n + (n^2/s) \cdot \lg s \cdot \lg n)$ time.

## 9 Concluding Comments

We have given space-efficient plane-sweep algorithms for some basic geometric problems. We believe that the techniques we introduce cover a range of ideas
to handle many other plane-sweep algorithms in a space-efficient manner. We also believe that our techniques can easily be extended to higher dimensions.

Except for $\epsilon$ in the $\Omega(n^2 - \epsilon)$ Yao’s lower bound for the element-distinctness problem, the $n^2/s + n \cdot \lg s$ bound for the running time of the problems we solve would be optimal. It is an intriguing open problem to get rid of this $\epsilon$.

Another question is if it is possible to get around with the extra logarithmic factors in the running times of the problem of enumerating the general and the axis-parallel line-segments intersections. It also remains open if it is possible to solve the measure problem more efficiently when the input is not sorted.

References

[1] Tetsuo Asano, Kevin Buchin, Maike Buchin, Matias Korman, Wolfgang Mulzer, Günter Rote, and André Schulz. Reprint of: Memory-constrained algorithms for simple polygons. *Comput. Geom. Theory Appl.*, 47(3, Part B):469–479, 2014.

[2] Tetsuo Asano, Amr Elmasry, and Jyrki Katajainen. Priority queues and sorting for read-only data. In *Proc. 10th International Conference on Theory and Applications of Models of Computation (TAMC 2013)*, volume 7876 of *LNCS*, pages 32–41. Springer, 2013.

[3] Tetsuo Asano, Wolfgang Mulzer, Günter Rote, and Yajun Wang. Constant-work-space algorithms for geometric problems. *J. Comput. Geom.*, 2(1):46–68, 2011.

[4] Ivan J. Balaban. An optimal algorithm for finding segments intersections. In *Proc. 11th Symposium on Computational Geometry*, pages 211–219, 1995.

[5] Luis Barba, Matias Korman, Stefan Langerman, Rodrigo I. Silveira, and Kunihiko Sadakane. Space-time trade-offs for stack-based algorithms. In *Proc. 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *LIPIcs*, pages 281–292. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.

[6] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991.

[7] Jon Bentley. Algorithms for klee’s rectangle problems, 1977. Unpublished manuscript.

[8] Jon Louis Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, 28(9):643–647, 1979.

[9] Hervé Brönnimann, John Iacono, Jyrki Katajainen, Pat Morin, Jason Morrison, and Godfried Toussaint. Space-efficient planar convex hull algorithms. *Theor. Comput. Sci.*, 321(1):25–40, 2004.

[10] Timothy M. Chan. Klee’s measure problem made easy. In *Proc. 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013*, pages 410–419. IEEE Computer Society, 2013.
[11] Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Selection and sorting in the “restore” model. In Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2014), pages 995–1004. SIAM, 2014.

[12] David Clark. Compact Pat Trees. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1996.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 3rd edition, 2009.

[14] Omar Darwish and Amr Elmasry. Optimal time-space tradeoff for the 2D convex-hull problem. In Proc. 22nd Annual European Symposium on Algorithms (ESA 2014), volume 8737 of LNCS, pages 284–295. Springer, 2014.

[15] Amr Elmasry, Daniel Dahl Juhl, Jyrki Katajainen, and Srinivasa Rao Satti. Selection from read-only memory with limited workspace. Theor. Comput. Sci., 554:64–73, 2014.

[16] Amr Elmasry, Frank Kammer, and Torben Hagerup. Space-efficient basic graph algorithms. In Proc. 32nd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2015), LIPIcs, pages 288–301, 2015.

[17] Greg N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. J. Comput. Syst. Sci., 34(1):19–26, 1987.

[18] Matsuo Konagaya and Tetsuo Asano. Reporting all segment intersections using an arbitrary sized work space. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 96-A(6):1066–1071, 2013.

[19] Matias Korman, Wolfgang Mulzer, Andr van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Time-space trade-offs for triangulations and Voronoi diagrams. In Proc. 14th Algorithms and Data Structures Symposium (WADS 2015), 2015.

[20] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. Theor. Comput. Sci., 12(3):315–323, 1980.

[21] J. Ian Munro. Tables. In PROC 16th FSTTCS, volume 1180 of LNCS, pages 37–42. Springer, 1996.

[22] J. Ian Munro and Venkatesh Raman. Selection from read-only memory and sorting with minimum data movement. Theor. Comput. Sci., 165(2):311–323, 1996.

[23] Mark H. Overmars and Chee-Keng Yap. New upper bounds in klee’s measure problem (extended abstract). In Proc. 29th Annual Symposium on Foundations of Computer Science (FOCS 1988), pages 550–556. IEEE Computer Society, 1988.

[24] Jakob Pagter and Theis Rauhe. Optimal time-space trade-offs for sorting. In Proc. 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1998), pages 264–268. IEEE Computer Society, 1998.
[25] Venkatesh Raman and Sarnath Ramnath. Improved upper bounds for time-space trade-offs for selection. *Nord. J. Comput.*, 6(2):162–180, 1999.

[26] Andrew Chi-Chih Yao. Near-optimal time-space tradeoff for element distinctness. *SIAM J. Comput.*, 23(5):966–975, 1994.