Space-Efficient Plane-Sweep Algorithms

Amr Elmasry[1] and Frank Kammer[2]

1 **Department of Computer Engineering and Systems, Alexandria University, Alexandria, Egypt**
elmasry@alexu.edu.eg
2 **Institut für Informatik, Universität Augsburg, Augsburg, Germany**
kammer@informatik.uni-augsburg.de

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 $\lg n \leq s \leq n \cdot \lg n$. Three techniques that can be used as general tools in different space-efficient algorithms are introduced and employed within our algorithms. In particular, we give an almost-optimal algorithm for finding the closest pair among a set of n points that runs in $O(n^2/s + n \cdot \lg s)$ time. We also give a simple algorithm to enumerate the intersections of n line segments that runs in $O((n^2/s^{2/3}) \cdot \lg s + k)$ time, where k is the number of intersections. The counting version can be solved in $O((n^2/s^{2/3}) \cdot \lg s)$ time. When the segments are axis-parallel, we give an $O((n^2/s) \cdot \lg^{4/3} s + n^{4/3} \cdot \lg^{1/3} n)$-time algorithm that counts the intersections and an $O((n^2/s) \cdot \lg s \cdot \lg \lg s + n \cdot \lg s + k)$-time algorithm that enumerates the intersections, where k is the number of intersections. We finally present an algorithm that runs in $O((n^2/s + n \cdot \lg s) \cdot \sqrt{(n/s) \cdot \lg n})$ time to calculate Klee's measure of axis-parallel rectangles.

1998 ACM Subject Classification E.1 Data Structures F.2.2 Nonnumerical Algorithms and Problems, I.1.2 Analysis of Algorithms, I.3.5 Geometric Algorithms

Keywords and phrases closest pair, line-segments intersection, Klee's measure

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.30

1 Introduction

Because of the rapid growth of the input data sizes in current applications, algorithms that are designed to efficiently utilize space are becoming even more important than before. One other reason to run a space-efficient algorithm is the limitation in the memory sizes that can be deployed to modern embedded systems. Therefore, many algorithms have been developed with the objective to optimize the time-space product.

Several models of computation have been considered for the case when writing in the input area is restricted. The objective of a space-efficient algorithm is to optimize the amount of extra space needed to perform its task. In the *multi-pass streaming model* [19] the input is assumed to be held in a read-only sequentially-accessible working space, and the goal would be to optimize the number of passes an algorithm makes over the input. In the *read-only word RAM* [16]—the model that we consider in this paper—the input is assumed to be stored on a read-only randomly-accessible working space and arithmetic operations on operands that fit in one word are assumed to take constant time each.

Throughout the paper, it is assumed that n is the number of items of the input, each stored in a constant number of words, and that the available workspace is $\Theta(s)$ bits, where $\lg n \leq s \leq n \lg n$. Since a single cursor, which is necessary to iterate over the input, already needs $\Theta(\lg n)$ bits, there is no hope to solve any of the problems with less workspace. In

addition, and as usual on a word RAM, it is assumed that operations on the input coordinates can be performed in constant time each. We emphasize that this assumption is not essential for our algorithms to work, but only scales with their running times.

Next, we survey some of the major results known for the read-only random-access model. Pagter and Rauhe [21] gave an asymptotically-optimal algorithm for sorting n elements that runs in $O(n^2/s + n \lg s)$ time. Beame [5] established a matching $\Omega(n^2)$ lower bound for the time-space product for sorting in the stronger branching-program model. Elmasry et al. [15] introduced space-efficient algorithms for basic graph problems. Concerning geometric problems, Chan [9] presented an algorithm for the closest-pair problem with integer coordinates in the word RAM model, and his algorithm can be made to work in the read-only model. Darwish and Elmasry [14] gave an optimal planar convex-hull construction algorithm that runs in $O(n^2/s + n \lg s)$ time. Konagaya and Asano [17] gave an algorithm for reporting line-segments intersections that runs in $O((n^2/\sqrt{s}) \cdot \sqrt{\lg n} + k)$ time, where k is the number of intersections. Recently, Korman et al. [18] gave space-efficient algorithms to construct triangulations and Voronoi diagrams whenever $s = \Omega(\lg n \cdot \lg \lg n)$ bits of working space are available. Asano et al. [3] considered space-efficient plane-sweep algorithms for Delaunay triangulation and Voronoi diagram. However, they only considered the case where $s = \Theta(\log n)$ bits, and both algorithms run in $O(n^2)$ time for this case.

As a building block for our algorithms we use the *adjustable navigation pile* [2]; an efficient priority-queue-like data structure that uses $O(s)$ bits, where $\lg n \leq s \leq n \lg n$, in the read-only random-access model of computation. Given a read-only input array of n elements and a specified value, an adjustable navigation pile can be initialized in $O(n)$ time. Subsequently, the elements that are larger than the given value can be streamed in sorted order in $O(n/s + \lg s)$ time per element. Thus, it is possible to stream the next k elements starting with a specified value in sorted order in $O((n/s + \lg s) \cdot k + n)$ time, and all the elements of the array can be streamed in sorted order in $O(n^2/s + n \lg s)$ time.

Another ingredient that we use in some of our algorithms is a *rank-select* data structure [12]. A rank-select data structure can be built on a bit vector of length n using $O(n)$ time and $o(n)$ extra bits, and supports in $O(1)$ time the queries *rank*(i), which returns the number of 1-bits in the first i positions of the bit vector, and *select*(j), which returns the index of the j-th 1-bit in the bit vector. In accordance, one can sequentially scan the entries of the bit vector that have 1-bits in $O(1)$ time per entry.

In this paper we give space-efficient plane-sweep algorithms that solve planar geometric problems where one moves a line across the plane and maintains the intersection of that line with the objects of interest. Many geometric problems have been solved using this kind of algorithm [7, 8, 13]. We assume that the sweep line moves over the plane from left to right. 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 and query the objects that cross the sweep line in order. The status structure is updated only at particular *event points*. Since $\Theta(n)$ objects might be part of the search tree, a typical plane-sweep algorithm needs $\Theta(n \lg n)$ bits, which is true for the standard algorithms of all problems considered here. In contrast to Asano et al. [3], all our algorithms allow a trade-off between time and space and work for all values of s where $\lg n \leq s \leq n \lg n$.

In Section 2 we introduce a general technique that we call the *divide-and-compress technique* relying on splitting the input array, and later employ it in our algorithms. In Section 3 we give a simple algorithm that enumerates intersections among n line segments and runs in $O((n^2/s^{2/3}) \cdot \lg s + k)$ time, where k is the number of intersections. Our algorithm is asymptotically faster than that of Konagaya and Asano for all values of s. We point out that

the same approach can be used to count the number of intersections in $O((n^2/s^{2/3}) \cdot \lg s)$ time. In Section 4 we give an algorithm for finding the closest pair among n points whose running time is $O(n^2/s + n \lg s)$. To obtain this result, we combine new ideas with the classical plane-sweep and divide-and-conquer approaches for solving the closest-pair problem. A lower bound of $\Omega(n^{2-\epsilon})$ was shown by Yao [22] for the time-space product of the element-distinctness problem, where ϵ is an arbitrarily small positive constant. This lower bound applies for the closest-pair problem, indicating that our algorithm is close to optimal. In Section 5 we give an algorithm for counting the intersections among n axis-parallel line segments that runs in $O((n^2/s) \cdot \lg^{4/3} s + n^{4/3} \cdot \lg^{1/3} n)$ time. The idea is to partition the plane as a grid and to run local plane sweeps on parts of the plane with truncated segments. In Section 6 we sketch a so-called *batching 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 among n axis-parallel line segments in $O((n^2/s) \cdot \lg s \cdot \lg \lg s + n \lg s + k)$ time, where k is the number of intersections. In Section 8 we show how to calculate Klee's measure (the area of the union) for n axis-parallel rectangles in $O((n^2/s) \cdot \lg n + n \lg s)$ time if the corners of the rectangles are stored in sorted order. In Section 9 we introduce another general technique that we call the *multi-scanning technique* where we partition the plane and run several plane sweeps interleaved. We use this technique to calculate Klee's measure in $O((n^2/s + n \lg s) \cdot \sqrt{(n/s) \cdot \lg n})$ time if the corners of the rectangles are unsorted. We conclude the paper in Section 10 with some comments.

2 A Divide-and-Compress Technique: Splitting the Input Array

We call a problem *decomposable* if any partitioning of the input into subsets allows us to compute a solution for the input by computing the partial solutions for these subsets as well as for the unions of all pairs of subsets and by combining these partial solutions. We also assume that the time needed to combine the results is bounded by the time to compute the partial solutions. Examples of such problems that we deal with in this paper are the axis-parallel line-segments intersections problem (Section 3) and closest-pair problem (Section 4). For the closest-pair distance, the overall solution is the minimum among the partial solutions for the subproblems. For the enumeration of the axis-parallel line-segments intersections, the overall solution is the union of the non-overlapping partial solutions. The general line-segments intersections problem is also decomposable, and can be handled using the same approach with slight modifications.

The following technique divides the instance in smaller parts and so compresses the necessary workspace used to solve a decomposable 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 smaller than the number n of elements in the input array. Split the array into $\lceil n/r \rceil$ *batches* $B_1, \ldots, B_{\lceil n/r \rceil}$ of at most r consecutive elements each (the last batch may have less) and proceed as follows: For $i = 1, \ldots, \lceil n/r \rceil$ and $j = i + 1, \ldots, \lceil n/r \rceil$, apply the underlying 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 for decomposable problems.

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 $t(r) + k'$ where k' is the size of the output. Thus, the overall time spend by the algorithm is $O((n^2/r^2) \cdot t(r) + k)$ where k is the size of whole output.

▶ **Lemma 1.** *Suppose we know how to solve a decomposable problem $\mathcal{P}$ of size n using $s' = \Theta(f(n))$ bits in $O(n^2/g(s') + n \lg s')$ time, where $f, g : \mathbb{N} \to \mathbb{R}$ are functions with $\lg n \leq f(n) \leq n \lg n$. For any s where $\lg n \leq s \leq s'$, we can solve any instance I of $\mathcal{P}$ of size n in $O(n^2/g(s) + (n^2/f^{-1}(s)) \cdot \lg s)$ time with $O(s)$ bits. In particular, when $f(n) = O(n/\lg n)$ and $g(s) = O(s)$, we can solve I in $O(n^2/g(s))$ time and $O(s)$ bits.*

Proof. By definition of $\mathcal{P}$, we can solve instances of $\mathcal{P}$ that are of size $r = \lceil f^{-1}(s) \rceil$ using s bits in $t(r) = O(r^2/g(s) + r \lg s)$ time. By applying the above construction, we can solve I in $O((n^2/r^2) \cdot t(r)) = O(n^2/g(s) + (n^2/r) \cdot \lg s) = O(n^2/g(s) + (n^2/f^{-1}(s)) \cdot \lg s)$ time. If $f(n) = O(n/\lg n)$, then $f^{-1}(s) = \Omega(s \lg s)$, and we can solve I in $O(n^2/g(s) + n^2/s)$. If in addition $g(s) = O(s)$, the claimed time and space bounds follow. ◀

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 produce the number of these intersections. An optimal algorithm to enumerate all the intersections that runs in $O(n \lg n + k)$ time was given by Balaban [4], where n is the number of segments and k is the number of intersections returned. Chazelle [11], improving a result of Agarwal [1], showed how to count the intersections among n line segments in $O(n^{4/3} \lg^{1/3} n)$ time, and how to report k *bichromatic intersections* in $O(n^{4/3} \lg^{1/3} n + k)$ time, i.e., given sets of red and blue segments, to report all intersections between a red and a blue segment. All these algorithms require a linear number of words, i.e., $O(n \lg n)$ bits.

If the available workspace is $\Theta(s)$ bits with $\lg n \leq s \leq n \lg n$, we give next a straightforward application of the divide-and-compress technique. We can apply the reporting algorithms on batches of size $O(r)$ line segments, where we choose $r = \Theta(f^{-1}(s))$, i.e., $r = \Theta(s/\lg s)$. First we apply Balaban's algorithm for each batch separately, then we apply a bichromatic-intersections algorithm on every pair of batches (coloring one of them red and the other blue). Note that we cannot apply Balaban's algorithm on pairs of batches since the partial solutions will be overlapping (intersections among the segments of a batch will show up in several partial solutions), and hence combining the partial solutions would be problematic. Thus, the running time $t(r)$ on r segments is $O(r^{4/3} \cdot \lg^{1/3} r)$. The reported intersections are the union of the non-overlapping intersections found by solving the subproblems. Hence, the overall time for this algorithm is $O((n^2/r^2) \cdot t(r) + k) = O((n^2/r^{2/3}) \cdot \lg^{1/3} r + k) = O((n^2/s^{2/3}) \cdot \lg s + k)$ time, where k is the number of reported intersections.

▶ **Theorem 2.** *Given a read-only array of n elements and $\Theta(s)$ bits of workspace, where $\lg n \leq s \leq n \lg n$, the planar line-segments-intersections enumeration problem can be solved in $O((n^2/s^{2/3}) \cdot \lg s + k)$ time, where k is the number of intersections returned. The counting version can be solved in $O((n^2/s^{2/3}) \cdot \lg s)$ time.*

4 Closest Pair

Given a set of n points in the plane, the planar closest-pair problem is to identify a pair of points that are closest to each other.

Assume for the moment that the available workspace is $\Theta(s)$ bits, where $\sqrt{n} \cdot \lg n \leq s \leq n \lg n$. In a first step, we produce the points in sorted order according to their x-coordinate values using an adjustable navigation pile and partition them into groups having $\lceil s/\lg n \rceil$ successive points each (except possibly the last group). Partition the plane in vertical regions,

called *vertical stripes*, where each region contains one group—if necessary, rotate the plane slightly. More exactly, choose the vertical regions such that the vertical lines separating the vertical stripes, called the *vertical separators*, pass through a point. We deal with the vertical separators in our workspace by storing, for each of them, $O(\lg n)$ bits of the index of the corresponding point in the input array. Since there are at most $m = \lceil (n/s) \cdot \lg n \rceil$ vertical stripes, references to 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 $s = \Omega(\sqrt{n} \cdot \lg n)$. The entities of all the separators can then be simultaneously stored within the available workspace. Additionally, all the points of a vertical stripe can fit in the available workspace. Thus, a standard closest-pair algorithm [13] can be applied to identify the closest pair among the points of each vertical stripe one after the other. We then find the pair with the minimum closest distance among all the vertical subproblems, and call this minimum distance δ.

To find a closest pair that is spread over two different stripes, we use a standard idea from the divide-and-conquer algorithm for the closest-pair problem. In a second step, we produce the points in sorted y-coordinate order using another adjustable navigation pile, but retain only the points that lie within a horizontal distance δ from any of the vertical separators. Call these points the *candidate points*. We consider the candidate points in the y-coordinate order in groups having $8m$ points each (except the last group that may have less points). Call the horizontal regions containing these groups the *horizontal stripes*. Note that references to the points of a horizontal stripe can be stored in $O((n/s) \lg^2 n) = O(s)$ bits, which can all fit in the available workspace. We can then apply a standard closest-pair algorithm within the working storage to identify the closest pair among the candidate points of every two consecutive horizontal stripes in order. Let δ' be the minimum closest distance among all the horizontal subproblems. Finally, we return $\min(\delta, \delta')$ as the closest-pair distance.

We prove next the correctness of the algorithm. We only have to show that the restriction to the points close to the vertical separators in the second step is correct. We slightly generalize the proof for the standard divide-and-conquer algorithm for the closest-pair problem. Since the distance between any pair of points within a vertical stripe is at least δ, any point that is at horizontal distance more than δ from all the vertical separators can not be closer than δ to any other point. We then only need to proceed with the candidate points that lie within a horizontal distance δ 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 δ to p they must lie together with p within a $2\delta \times \delta$ rectangle centered at the vertical separator. Note that there could be at most 8 points above p within this rectangle whose distances to p are less than δ, since 2 rows of 3 circles of diameter δ can cover the whole rectangle and there can be at most one point in the two left and the two right circles as well as at most two points in the middle circles. Since there are m vertical separators, the number of candidate points P above p to be checked for possibly having a distance less than δ from p is at most $8m$; no other point above p can be at distance less than δ from p. (Actually, it suffices to check only 5 candidate points above p for each separator [13, Exercise 33.4-2].) Obviously, the points in P must be consecutive in the y-coordinate values. Since we store $8m$ candidate points per stripe, the points in 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.

We can produce the points in sorted order in both coordinates in $O(n^2/s + n \lg s)$ time using the adjustable navigation pile [2]. The time needed to execute the standard closest-pair algorithm for all the stripes is $O(n \lg s)$ [13]. The check whether each point is close to one of the separators or not runs in $O(n \lg n) = O(n \lg s)$ time using a binary search among

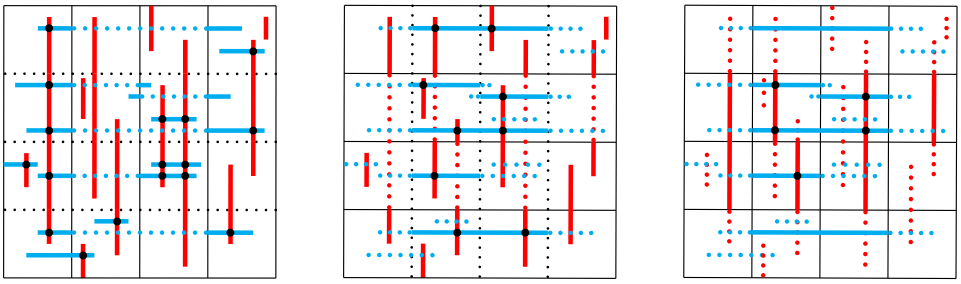

Figure 1 Counting axis-parallel line segments in three *phases*. Each stripe contains 12 points and, for clarity reasons, all stripes have the same size. The black dots on the crossings of two segments show the intersection points that are counted in each phase.

the x-coordinates of the separators for each point. Hence, the overall running time of the algorithm is $O(n^2/s + n \lg s)$.

Assume now that we have $\Theta(s)$ bits available, where $\lg n \leq s < \sqrt{n} \cdot \lg n$. 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 a case, the running time for each of these instances would be $t(r) = O(r^2/s + r \lg s) = O(r^2/s)$. We then divide the input into $\lceil n/r \rceil$ batches of points and apply the divide-and-compress technique, 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 $O((n/r)^2 t(r)) = O(n^2/s)$.

▶ **Theorem 3.** *Given a read-only array of n elements and $\Theta(s)$ bits of workspace, where $\lg n \leq s \leq n \lg n$, the planar closest-pair problem can be solved in $O(n^2/s + n \lg s)$ time.*

It is known that the closest-pair algorithm can be generalized from two to higher dimensions [13] to run in $O(n \lg^{d-1} n)$ time in d dimensions. Applying the divide-and-compress technique in a similar way as described above, we can solve the closest-pair problem in d dimensions with $\Theta(s)$ bits of workspace in $O(n^2/s + n \lg^{d-1} s)$ time, where $\lg n \leq s \leq n \lg n$.

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 for the moment that the available workspace is $\Theta(s)$ bits, where $n^{2/3} \cdot \lg n \leq s \leq n \lg n$. First, we 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 $\lceil s/\lg n \rceil$ points each (except possibly the last group that may have fewer points). As in the previous section, these groups define the *vertical stripes* and *vertical separators*. Since there are $\lceil (n/s) \cdot \lg n \rceil = O(n^{1/3})$ vertical stripes, references to the x-coordinate values of all the separators can be stored within the workspace. We associate a line segment to a stripe if at least one of its two endpoints lies inside the stripe. If we consider the line segments of a vertical stripe, or more exactly, their positions in the input array, they can all fit in the workspace. Thus, we can apply a standard line-segments-intersections counting algorithm to each vertical stripe one after the other, and add these counts together. See the left side of Fig. 1.

Subsequently, we produce the points in sorted order according to their y-coordinate values using another adjustable navigation pile, and partition the plane in *horizontal stripes* (analogous to the definition of the vertical stripes) such that each has $\lceil s/\lg n \rceil$ points (except

possibly the last group that may have less points). It follows that the $O(n^{1/3})$ references to the so-called *horizontal separators* can be simultaneously stored in the workspace. In a similar fashion as above, we apply a 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 modify the access to the horizontal segments such that further computations consider the segments to be truncated. 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 already been accounted for while dealing with the vertical stripes. See the middle of Fig. 1.

Let $\mathcal{R}_{i,j}$ be the *cell* formed by the intersection of the ith horizontal stripe with the jth vertical stripe. A line segment *spans* a cell if it crosses two of the cell's boundaries. It remains to account for the intersections among these spanning segments. The number of these intersections for each cell is the product of the numbers of its spanning horizontal and vertical segments. We show next how to count the spanning horizontal segments for each cell. The treatment for the vertical segments is similar. See the right side of Fig. 1.

A line segment is *interior* to a cell if both its endpoints lie inside the cell. For each cell $\mathcal{R}_{i,j}$, we store the count $b_{i,j}$ of horizontal segments beginning in the cell, the count $f_{i,j}$ of horizontal segments finishing in the cell, and the count $t_{i,j}$ of the horizontal segments interior to the cell. Since there are $O(n^{2/3})$ cells, all these values can be stored in $O(n^{2/3} \cdot \lg n)$ bits, which is $O(s)$ when $s \geq n^{2/3} \cdot \lg n$. For every horizontal segment, we locate the starting and ending cells using binary search among the separators, and increment the corresponding counters in accordance. We then scan the cells of every horizontal stripe sequentially while calculating $e_{i,j}$ the number of horizontal segments *entering* $\mathcal{R}_{i,j}$, i.e., the number of segments that have a non-empty intersection with $\mathcal{R}_{i,j}$ and $\mathcal{R}_{i,j-1}$; this is done using $e_{i,0} = 0$ and $e_{i,j} = e_{i,j-1} + b_{i,j-1} - f_{i,j-1}$. We finally obtain the number of horizontal segments spanning $\mathcal{R}_{i,j}$ as $e_{i,j} - f_{i,j} + t_{i,j}$. The time needed to produce the endpoints in sorted order in both coordinates using the adjustable navigation pile is $O(n^2/s + n \lg s)$ [2]. The time needed to execute the standard segments-intersection counting algorithm for all the stripes is $O(n^{4/3} \cdot \lg^{1/3} n)$. The time needed to perform binary search among the separators is $O(n \lg s)$. The time needed to count the intersections of the spanning segments of all the cells is constant per cell and sums up to $O(n^{2/3})$. It follows that the overall running time of the algorithm is $O(n^{4/3} \cdot \lg^{1/3} n)$.

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 $\Theta(r)$ elements. We divide the input array into $\lceil n/r \rceil$ batches of consecutive segments and apply the divide-and-compress technique. First apply the algorithm on instances for every batch individually, then on instances for every pair of batches. Using these computed counts, the overall count can be easily calculated. The running time for each instance would be $t(r) = O(r^{4/3} \cdot \lg^{1/3} s)$. The overall time consumed in this case is $O((n/r)^2 \cdot t(r)) = O((n^2/s) \cdot \lg^{4/3} s)$.

▶ **Theorem 4.** *Given a read-only array containing the endpoints of n line segments and $\Theta(s)$ bits of workspace, where $\lg n \leq s \leq n \lg n$, counting the planar axis-parallel line-segments intersections can be done in $O((n^2/s) \cdot \lg^{4/3} s + n^{4/3} \cdot \lg^{1/3} n)$ time.*

6 A Batching Technique: Processing Sweep-Line Events in Batches

We now show that, if the given objects are axis-parallel, one may reduce the working storage of the status structure to $\Theta(n)$ bits by processing the events in batches.

For a parameter m to be set later, suppose our plane is divided into m vertical and horizontal stripes such that each stripe contains $O(n/m)$ local objects, where an object is *local* for a stripe if it starts or ends within the stripe. As before, the boundaries of the stripes are called *separators*. The intersection of a horizontal stripe with a vertical stripe is called a *cell*. To apply the batching technique, we need a plane-sweep instance with the following two properties: (1) All the events of the event queue are on vertical separators, i.e., they result from so-called *horizontally spanning objects*. (2) All the objects of the status structure start and end on horizontal separators, i.e., they are so-called *vertically spanning objects*. Assume the available workspace is $\Theta(s)$ bits, where $n \leq s \leq n \lg n$. By setting $m = \lceil (n/s) \cdot \lg n \rceil$, we can store references to all local objects of a stripe and references to the coordinates of the separators in the working storage. Because of properties (1) and (2), 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 m cells formed by the intersections with the horizontal stripes. Recall that there are $O(n/m)$ vertically spanning objects that are in a cells of the vertical stripe. We store their positions in an array using a total of $O((n/m) \cdot \lg n) = O(s)$ bits. In addition, we store for each of the m cells a bit vector of $O(n/m)$ bits indicating whether each of these objects spans the cell or not. Over and above, for each bit vector of a cell, we build a rank-select data structure that allows us to scan the vertical spanning objects of the cell in constant time per object. The bit vectors and the rank-select structures are enough to represent the status structure. Thus, the sweep line can be stored in a total of $O(s)$ bits. We use an adjustable navigation pile as our event queue to produce the events and the spanning objects in order. Since $s \geq n$, the time to produce all the events in order throughout the procedure is $O(n \lg s)$. 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 cells that it spans are allocated in $O(m)$ 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 $O(n \cdot m) = O((n^2/s) \cdot \lg n) = O((n^2/s) \cdot \lg s)$.

It remains to show how to allocate an event point within the status structure representing the sweep line. We would be satisfied with only identifying the cell that contains this event point within the vertical stripe. We do that using binary search against the m horizontal separators, consuming $O(\lg m) = O(\lg \lg s)$ time per event point.

▶ **Lemma 5.** *Let $\mathcal{I}$ be a plane-sweep instance for which (1) and (2) holds. Using the batching technique, a sweep can be performed on a plane with n objects, using a data structure that can be stored in $\Theta(s)$ bits, where $n \leq s \leq n \lg n$. The sweep makes a total of $O((n/s) \cdot \lg s)$ stopovers, and the data structure can be rebuilt in $O(n)$ time per stopover plus a total of $O(n \lg s)$ time, and queried in $O(\lg \lg s)$ time per event. Handling all events at each stopover, we can run a plane-sweep algorithm on $\mathcal{I}$ in $O((n^2/s) \cdot \lg s \cdot \lg \lg s + n \cdot \lg s)$ total time.*

7 Enumerating Axis-Parallel Line-Segments Intersections

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

We use the same ideas as in Section 5. As before, we split the plane into $m = \lceil (n/s) \cdot \lg n \rceil$ horizontal and vertical stripes where each except the last contains $\lceil n/m \rceil$ line segments. We enumerate the intersections among the local parts of the segments within the stripes by applying a standard line-segments-intersection enumeration algorithm.

By truncating the segments, we assume from now on that all the endpoints lie on the boundaries of the cells and the segments span the cells they cross. Note that each horizontal line segment that spans a cell must intersect all the vertical segments spanning the same cell. By applying the ideas of the batching technique, we store the vertical spanning line segments that lie in the current vertical stripe and build a status structure that consumes $\Theta(s)$ bits in $O(n)$ time. Using this data structure it is possible to enumerate the vertical segments that span a given cell in time proportional to the number of the reported segments. For each horizontal segment, we check if it spans any of the cells of the sweep line. We do that using binary search for each horizontal segment against the m horizontal separators. After every binary search for a horizontal segment, we query the status structure to find the vertical segments spanning the same cell, and so their intersections with the horizontal segment are computed and reported. After locating the crossing cells 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 all the stripes is $O(n \lg s)$, which matches the time bound to build the status structure of all vertical stripes using the batching technique. The time to perform binary search for each of the horizontal segments against the m cells of the status structure is $O(n \lg m)$. Hence, we can compute all intersection points of a vertical stripe in $O(n \lg m + k')$ 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 $O(n \cdot m \cdot \lg m + k) = O((n^2/s) \cdot \lg s \cdot \lg \lg s + k)$, where k is the number of intersections returned. Since we can partition the plane into stripes using a navigation pile in $O(n^2/s + n \lg s)$ time, the total time consumed by the whole algorithm is $O((n^2/s) \cdot \lg s \cdot \lg \lg s + n \lg s + k)$.

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

▶ **Theorem 6.** *Given a read-only array containing the endpoints of n line segments and $\Theta(s)$ bits of workspace, where $\lg n \leq s \leq n \lg n$, enumerating the planar axis-parallel line-segments intersections is done in $O((n^2/s) \cdot \lg s \cdot \lg \lg s + n \lg s + k)$ time, where k is the number of intersections returned.*

8 Measure of Axis-Parallel Rectangles

We consider the problem of computing the *measure* of a set of n axis-parallel rectangles, i.e., the size of the area of the union. The problem was posed by V. Klee, and thus called *Klee's measure problem*. Bentley [6] described an $O(n \lg n)$-time algorithm that can be implemented with $\Theta(n \lg n)$ 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. Another algorithm to compute the measure was presented by Overmars and Yap [20]. A generalization of the algorithm to d dimensions was given by Chan [10].

Assume that the available workspace is $\Theta(s)$ bits, where $\lg n \leq s \leq n \lg n$. To compute the measure of a set of n axis-parallel rectangles, we use Bentley's algorithm as a subroutine. In this section, we restrict ourselves to the case where the corners of the rectangles are stored sorted by their x-coordinates. This restriction is dropped in the next section.

We split the plane into $m = \Theta((n/s) \cdot \lg n)$ horizontal stripes, where each stripe consists of $\Theta(s/\lg n)$ rectangle corners and accordingly fit in the available workspace. A rectangle is *spanning* a stripe if its vertical segments cross the two separators of the stripe. We process the stripes in sorted y-coordinate order, one after the other. By using an adjustable navigation pile, we produce and store the rectangles cornered within each stripe in sequence. Before processing a stripe and storing the rectangles, we truncate those rectangles such that they are shrunk to their intersection with the stripe. We would then run Bentley's algorithm on these rectangles. However, we need to also take into consideration the rectangles spanning the stripe. We show next how to do that efficiently.

We horizontally scan the stripe and keep track of the spanning segments and the corners. We accumulate as a global variable the width $\mathcal{W}$ of the union of the spanning rectangles so far. To do that, we maintain z as the difference between the number of scanned spanning segments that are left boundaries of a rectangle and the number of scanned spanning segments that are right boundaries. Whenever z becomes positive, we record this coordinate as x_1. Whenever z returns back to zero, we record this coordinate as x_2; we have just passed over a spanning area, and accordingly update $\mathcal{W}$ by adding to it the value $x_2 - x_1$. Whenever we meet a corner, we update its x-coordinate value as follows. If z is positive (the corner is in a spanning area), first set the x-coordinate of this corner to x_1. Either way, whether z is positive or zero, we subtract the current value of $\mathcal{W}$ from the x-coordinate of the corner. This process of relocating the corners is called *simplifying* the rectangles in [10]. After finishing the scan, we apply Bentley's algorithm to the relocated corners and calculate the measure within the current stripe. We also multiply $\mathcal{W}$ by the width of the stripe to get the area covered by the spanning rectangles, and add this area to the calculated measure. The total measure is the sum of the measures within all the stripes.

The time to sequentially scan the segments and simplify the rectangles within each stripe is $O(n)$ (as the segments are already sorted), and the time for applying Bentley's algorithm is $O((s/\lg n) \cdot \lg s)$. The total time to process all the m stripes is $O((n^2/s) \cdot \lg n + n \lg s)$.

▶ **Theorem 7.** *Given a read-only array storing the corners of n axis-parallel rectangles in sorted x-coordinate order, and the available workspace is $\Theta(s)$ bits, where $\lg n \leq s \leq n \lg n$, the measure (area of the union) can be computed in $O((n^2/s) \cdot \lg n + n \lg s)$ time.*

9 A Multi-Scanning Technique: Partitioning the Plane

In this section we introduce a technique to replace one global sweep with many local sweeps, and apply it to the measure problem if the input is not sorted. The idea is to perform alternating vertical and horizontal sweeps on parts of the plane to identify cells, each containing a set of objects that fit in the working storage. Once identified, we apply a local algorithm within each cell. By partitioning the plane into a grid of cells, we combine the local solutions for the cells together to obtain the final outcome. The details come next.

We partition the plane into $m = \lceil \sqrt{(n/s) \cdot \lg n} \rceil$ horizontal stripes, where each stripe consists of $O(n/m)$ corners. We process the horizontal stripes one after the other in sorted y-coordinate order using an adjustable navigation pile. Once the two separators of a horizontal stripe $\mathcal{H}$ are determined, we initialize an adjustable navigation pile $Y_{\mathcal{H}}$ for the stripe that allows us to stream the corners within $\mathcal{H}$ ordered by their y-coordinates. We start sweeping over the plane in sorted x-coordinate order using another adjustable navigation pile $X_{\mathcal{H}}$ that is initialized over the whole input. For this horizontal sweep, we are interested only in the corners in $\mathcal{H}$ as well as the vertical segments spanning $\mathcal{H}$—to find the spanning segments, we have to take all corners of the plane into consideration. Whenever the number of corners in

$\mathcal{H}$ produced by $X_{\mathcal{H}}$ is $\ell = \lceil s/\lg n \rceil$ (except for the last cell that may have less corners), we have reached a vertical separator that identifies, as a right boundary, a cell $\mathcal{V}$ within $\mathcal{H}$. The corners of a cell can be stored in $O(s)$ bits and hence fit in the working storage. During this horizontal sweep over $\mathcal{V}$, we calculate the horizontal width $\mathcal{W}_h$ of the area covered by the vertically spanning rectangles, and in the meantime simplify these corners of $\mathcal{V}$ (relocate the x-coordinates), as explained in the previous section, while storing them. We temporarily pause the horizontal sweep, and start a vertical sweep within $\mathcal{H}$ after initializing $Y_{\mathcal{H}}$ using the value of the horizontal separator between $\mathcal{H}$ and the stripe above it. During this vertical sweep, we calculate the vertical width $\mathcal{W}_v$ of the area covered by the horizontally spanning rectangles, and simplify the stored corners of $\mathcal{V}$ (this time, relocate the y-coordinates). Since the corners within $\mathcal{V}$ fit in the working storage, we compute Klee's measure of the parts of the simplified rectangles within the cell $\mathcal{V}$ using Bentley's algorithm. We add the areas covered by the spanning vertical and the spanning horizontal rectangles to adjust the measure, and subtract the intersection area $\mathcal{W}_h \times \mathcal{W}_v$ that has been added twice. We repeatedly proceed with the horizontal sweep using $X_{\mathcal{H}}$ to identify and partially process a cell, then alternately initialize $Y_{\mathcal{H}}$ and perform a vertical sweep within $\mathcal{H}$ to finish the processing of the cell. After all the cells of a horizontal stripe are processed, we repeat the same actions for the next horizontal stripes in sequence. Since we correctly calculate the measure within every cell, the overall sum of all the local measures is what we are looking for.

Concerning the running time, we consider the time to produce the segments by the navigation piles. Recall that $X_{\mathcal{H}}$ sweeps over all the n corners, whereas $Y_{\mathcal{H}}$ sweeps only over the $O(n/m)$ corners of $\mathcal{H}$. The navigation piles X for the horizontal sweeps repeatedly process all the input for every horizontal stripe. Since we have a total of m such sweeps, the total time consumed by the X navigation piles is $O((n^2/s + n \lg s) \cdot m)$. The navigation piles Y for the vertical sweeps process the $O(n/m)$ corners of a horizontal stripe in one sweep. Therefore, the total time for each of these vertical sweeps is $O((n/s + \lg s) \cdot n/m + n)$. It is straightforward to verify that $n/s + \lg s = \Omega(m)$ for all considered values of n and s (it is either true that $n/s > m$ or otherwise $\lg s = \Omega(m)$). The total number of vertical sweeps done within each horizontal stripe is $O((n/m)/\ell)$, which is $O(m)$ since $m = \lceil \sqrt{(n/s) \cdot \lg n} \rceil$. It follows that the total time of the vertical sweeps within one horizontal stripe is $O(n^2/s + n \lg s)$. Multiplying by the number of horizontal stripes m, the total time consumed by the Y navigation piles is $O((n^2/s + n \lg s) \cdot m)$, matching the bound for the X piles. The time needed by the extended local version of Bentley's algorithm within each cell is $O(\ell \cdot \lg \ell)$, resulting in a total of $O(n \cdot \lg s)$ time for all the calls to Bentley's algorithm. The time for the navigation piles is dominating.

▶ **Theorem 8.** *Given a read-only array containing the corners of n axis-parallel rectangles, and the available workspace is $\Theta(s)$ bits, where $\lg n \leq s \leq n \lg n$, the measure can be computed in $O((n^2/s + n \lg s) \cdot \sqrt{(n/s) \cdot \lg n})$ time.*

10 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. 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 Pankaj K. Agarwal. Partitioning arrangements of lines II: Applications. *Discrete Comput. Geom.*, 5(6):533–573, 1990.

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, 2013. doi:10.1007/978-3-642-38236-9_4.

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. doi:10.1145/220279.220302.

5 Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991. doi:10.1137/0220017.

6 Jon Louis Bentley. Algorithms for Klee's rectangle problems, 1977. Unpublished manuscript.

7 Jon Louis Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, 28(9):643–647, 1979. doi:10.1109/TC.1979.1675432.

8 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.

9 Timothy M. Chan. Closest-point problems simplified on the RAM. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 472–473, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545444.

10 Timothy M. Chan. Klee's measure problem made easy. In *Proc. 54th Anual IEEE Symposium on Foundations of Computer Science (FOCS 2013)*, pages 410–419, 2013. doi:10.1109/FOCS.2013.51.

11 Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(2):145–158, 1993.

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, 2014. doi:10.1007/978-3-662-44777-2_24.

15 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. doi:10.4230/LIPIcs.STACS.2015.288.

16 Greg N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *J. Comput. Syst. Sci.*, 34(1):19–26, 1987. doi:10.1016/0022-0000(87)90002-X.

17 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. URL: http://search.ieice.org/bin/summary.php?id=e96-a_6_1066.

18 Matias Korman, Wolfgang Mulzer, André 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.

19 J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12(3):315–323, 1980. doi:10.1016/0304-3975(80)90061-4.

20 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, 1988. doi:10.1109/SFCS.1988.21971.

21 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, 1998. doi:10.1109/SFCS.1998.743455.

22 Andrew Chi-Chih Yao. Near-optimal time-space tradeoff for element distinctness. *SIAM J. Comput.*, 23(5):966–975, 1994. doi:10.1137/S0097539788148959.