Bichromatic Line Segment Intersection Counting in $O(n\sqrt{\log n})$ Time

Timothy M. Chan[*] Bryan T. Wilkinson[†]

Abstract

We give an algorithm for bichromatic line segment intersection counting that runs in $O(n\sqrt{\log n})$ time under the word RAM model via a reduction to dynamic predecessor search, offline point location, and offline dynamic ranking. This algorithm is the first to solve bichromatic line segment intersection counting in $o(n \log n)$ time.

1 Introduction

We consider *bichromatic line segment intersection counting*: given a set of disjoint blue line segments and a set of disjoint red line segments in the plane, output the number of intersections of blue segments with red segments. Bichromatic line segment intersection problems arise in applications such as map overlay in GIS. We give an algorithm that runs in $O(n\sqrt{\log n})$ time, where n is the total number of segments, under the standard word RAM model with w-bit words. The input segments are specified by their endpoints, which are given as $O(w)$-bit integer or rational coordinates. Our result answers an open question of Chan and Pătraşcu [6] by showing that bichromatic line segment intersection counting can be solved in $o(n \log n)$ time. Thus, the problem finally joins the ranks of many other problems in computational geometry that have $\Omega(n \log n)$ lower bounds under the comparison model but $o(n \log n)$ upper bounds under the word RAM model. Recent examples include 2-d Voronoi diagrams [2], 3-d convex hulls [6], and 3-d layers-of-maxima [12]. The word RAM model is important because its power corresponds very closely to that of actual programming languages (our algorithm uses only standard operations, such as arithmetic and bitwise logical operations) and it includes only the reasonable assumption that each input value is an integer (or rational) that fits in a word.

Chazelle et al. [7] give algorithms based on segment trees for bichromatic segment intersection reporting in $O(n \log n + k)$ time, where k is the number of intersections reported, and for counting in $O(n \log n)$ time. Chan's *trapezoid sweep* algorithm [3] for the reporting problem also achieves $O(n \log n + k)$ time, but has smaller constant factors and can be extended to curve segments. Mantler and Snoeyink [11] modify the trapezoid sweep to use operations of algebraic degree at most 2 and also to support counting in $O(n \log n)$ time.

Our algorithm follows the high-level idea of Mantler and Snoeyink [11] but reduces the low-level computations to dynamic predecessor search, offline point location, and offline dynamic ranking. Note that we cannot base similar reductions on just any high-level algorithm; the algorithm of Mantler and Snoeyink [11] gives particularly exploitable structure to the bichromatic line segment intersection counting problem. The algorithm of Chazelle et al. [7] inherently requires $O(n \log n)$ time due to the use of segment trees, and Chan's trapezoid sweep does not address the counting problem.

Recently, efficient algorithms have arisen for both offline point location and offline dynamic ranking under the word RAM model. Chan and Pătraşcu [6] give an algorithm for offline point location that locates $O(n)$ points in a subdivision of the plane defined by $O(n)$ segments in $n \cdot 2^{O(\sqrt{\log \log n})}$ time. Chan and Pătraşcu [5] also give an algorithm for offline dynamic ranking that processes $O(n)$ queries and updates in $O(n\sqrt{\log n})$ time. We use both of these algorithms as subroutines.

In Section 2 we give an overview of Mantler and Snoeyink's algorithm [11] and describe a 1-d data structure problem to which it reduces. In Section 3 we discuss a rank space reduction which is integral to achieving speed ups under the word RAM model. In Section 4 we describe our data structure, which we analyse in Sections 5 and 6.

2 High-Level Algorithm

We assume that the endpoints of all of the given segments are in general position. Otherwise, perturbation techniques can be used to handle any degeneracies [10]. We begin with an overview of the algorithm of Mantler and Snoeyink [11], simplified under our non-degeneracy assumption. The algorithm is a plane sweep that keeps track of all of the segments that intersect the vertical *sweep line*. The efficiency of the algorithm is achieved by storing the segments in maximal monochromatic *bundles* of segments. The algorithm keeps an alternating list of red and blue bundles. The order of the bundles in this list may not be consistent with the order of the segments along the sweep line. However, the order of the

[*]David R. Cheriton School of Computer Science, University of Waterloo, `tmchan@uwaterloo.ca`

[†]David R. Cheriton School of Computer Science, University of Waterloo, `b3wilkin@uwaterloo.ca`. Supported by NSERC.

bundles of a single colour in the list is always consistent with the order of segments of the same colour along the sweep line. Figure 1 shows the grouping of segments into bundles at various positions of an example plane sweep.

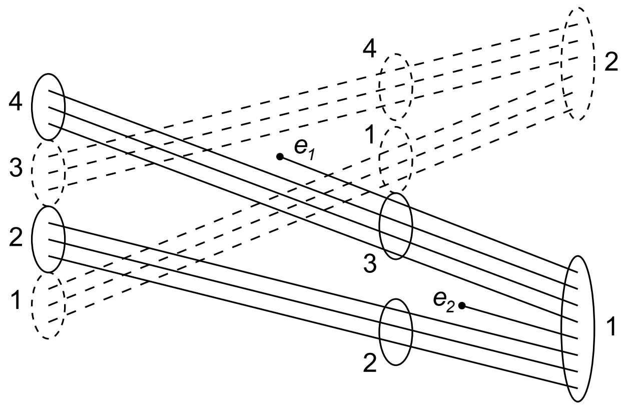

Figure 1: Ordering of red (dashed) and blue (solid) bundles before and after processing endpoints e_1 and e_2.

When the plane sweep reaches an endpoint e, our goal is to swap red and blue bundles until all blue bundles below e are below all red bundles above e, and all blue bundles above e are above all red bundles below e. When we swap a red bundle b_r and a blue bundle b_b it is because all segments of b_r intersect with all segments of b_b. So, whenever we perform a swap, we add $|b_r| \cdot |b_b|$ to our bichromatic segment intersection counter.

First, we find the red bundles immediately above and below e. If e is inside a red bundle, we split this bundle into two bundles such that one is above e and the other is below e. If b_r is the red bundle above e, we check if the blue bundle b_b that follows b_r in our list of bundles is below e. If so, we swap b_r and b_b. Doing so may result in two adjacent red bundles and/or two adjacent blue bundles. We merge these adjacent bundles of the same colour. We repeat this process until b_b is not below e. In the last repetition, e may be inside b_b, in which case we split b_b around e before swapping. We follow a similar process in the other direction.

After all of the bundle swapping has occurred, we still need to handle the inclusion or exclusion of the segment with endpoint e. If e is a left endpoint, we insert the segment into a new bundle which is placed between the lowest bundle above e and the highest bundle below e. If e is a right endpoint, we delete the segment. If the segment is the only segment in its bundle, deleting the segment removes its bundle. In either the insertion or deletion case, we merge adjacent bundles of the same colour as necessary.

For a proof of correctness, we refer the reader to Mantler and Snoeyink's paper [11]. The main idea of the proof is that the order of the bundles is always consistent with the order of a certain deformation of the segments along the sweep line. This deformation pushes intersections as far to the right as possible without moving endpoints, adding intersections, or removing intersections.

The low-level computations of the algorithm can be encapsulated into a data structure that supports the following operations:

Insert(s, b_ℓ, b_h)
: Inserts a new bundle containing only segment s between bundles b_ℓ and b_h.

Delete(s)
: Deletes segment s, removing its bundle b if s is the only segment in b.

IsAbove(e, b) / **IsBelow**(e, b)
: Determines whether or not endpoint e lies above or below all segments in bundle b.

Split(e, b)
: Splits bundle b into two bundles b_ℓ and b_h such that b_ℓ contains all segments below endpoint e and b_h contains all segments above endpoint e. Figure 2 shows an example of a bundle being split.

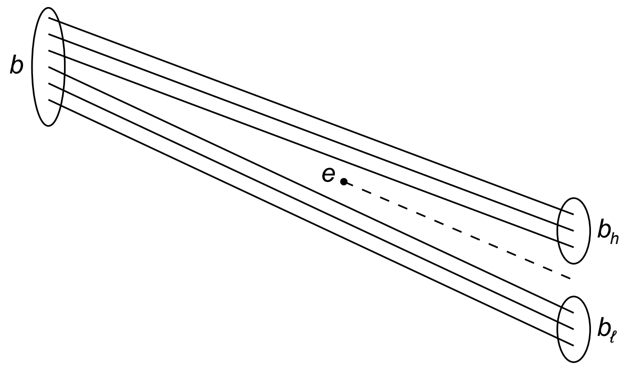

Figure 2: Splitting bundle b at endpoint e.

Merge(b_ℓ, b_h)
: Merges two adjacent bundles b_ℓ and b_h of the same colour into a single bundle.

Swap(b_ℓ, b_h)
: Swaps the order of two adjacent bundles b_ℓ and b_h of different colours.

HighestBelow(e) / **LowestAbove**(e)
: Finds the highest (lowest) red bundle in which all segments are below (above) endpoint e.

Next(b) / **Previous**(b)
: Finds the bundle that follows or precedes bundle b in sorted order.

Size(b)
Calculates the number of segments in bundle b.

The analysis presented by Mantler and Snoeyink [11] reveals that each of these operations is invoked at most $O(n)$ times. The main idea of their analysis is that during the processing of each endpoint, there is a constant upper bound on the number of bundle splits, and thus there are a linear number of bundles over the course of the algorithm. All of the other operations can be charged to bundles.

3 Rank Space Reduction

Before we describe our data structure, we discuss a preprocessing step in which we perform rank space reduction. Rank space reduction is important under the word RAM model to, for example, reduce the cost of predecessor search with van Emde Boas trees [14] from $O(\log\log U)$, where U is the size of the universe, to $O(\log\log n)$. In particular, we want to assign *ids* to segments such that segment s_1's id is greater than segment s_2's id if and only if s_1 is above s_2. Since red and blue segments can intersect, there may be no mapping that is consistent for all segments throughout the entire plane sweep. However, Palazzi and Snoeyink [13] give a topological ordering that is consistent for a set of disjoint segments. We can thus assign ids that are consistent with aboveness to all red segments and all blue segments separately, based on their own separate topological orderings. The topological sort involves a plane sweep that runs in $O(n\log n)$ time, since it uses a balanced search tree in order to find the segment above the current endpoint.

Instead of using this balanced search tree, we can find the segment above every endpoint in advance by computing the trapezoidal decomposition of the segments. Chan and Pătraşcu [4] reduce general offline 2-d point location to offline 2-d point location in a vertical slab that is divided into regions by disjoint segments that cut across the slab. We call this latter problem the *slab problem*. The same techniques (persistence and exponential search trees) can be used to reduce trapezoidal decomposition of disjoint segments to the slab problem. Chan and Pătraşcu [6] give an algorithm for the slab problem that yields an algorithm for trapezoidal decomposition of disjoint segments that runs in $n\cdot 2^{O(\sqrt{\log\log n})}$ deterministic time. The topological sort thus has the same runtime.

4 Data Structure

All segments have ids, as assigned by the rank space reduction described in Section 3. We keep a doubly linked list of bundles. A bundle stores pointers to its top and bottom segments. Only segments which are top or bottom segments of a bundle have pointers back to their bundle. In addition to this doubly linked list, we keep two predecessor search data structures that use segment ids as keys. These trees, T and B, contain only those red segments that are at the tops and bottoms of their bundles, respectively. If we were to use van Emde Boas trees [14], updates would run in $O(\log\log U)$ expected time. Since we want to avoid randomization, we use instead a data structure of Andersson and Thorup [1], which supports queries and updates in $O(\log\log n \frac{\log\log U}{\log\log\log U})$ deterministic time. Due to the rank space reduction, these operations actually run in $O(\frac{\log^2\log n}{\log\log\log n})$ time. Finally, we keep dynamic ranking data structures R_r and R_b, also using segment ids as keys, for all red segments and blue segments, respectively, that intersect the sweep line. We defer our selection of a particular dynamic ranking data structure to Section 5.

Lemma 1 *After a preprocessing step that runs in $n\cdot 2^{O(\sqrt{\log\log n})}$ time, we can find the segment of a given colour above or below endpoint e along the sweep line in $O(1)$ time.*

Proof. Assume without loss of generality that e is red. The red segments above and below e can be found in $O(1)$ time by navigating the red trapezoidal decomposition that was computed in $n\cdot 2^{O(\sqrt{\log\log n})}$ time in Section 3. Finding the blue segments above and below e is equivalent to locating e within the blue trapezoidal decomposition. So, in the preprocessing step, we perform offline planar point location of all red endpoints in the blue trapezoidal decomposition. The blue trapezoidal decomposition has linear complexity and has vertices with $O(w)$-bit rational coordinates. We can perform offline planar point location of the red endpoints in such a subdivision of the plane in $n\cdot 2^{O(\sqrt{\log\log n})}$ time using another algorithm of Chan and Pătraşcu [6]. $\square$

We now describe how we implement all of the operations of the data structure, using the ability to find the segments above and below an endpoint in constant time via Lemma 1 as a primitive.

Insert(s, b_ℓ, b_h)
We create a new bundle b in which s is both the top and bottom segment and rewire the next/previous bundle pointers between b, b_ℓ, and b_h. We insert s into R_c, where c is the colour of s. If s is red, we also insert it into both T and B.

Delete(s)
If s is both the top and bottom segment of its bundle b, s has a pointer to b. We rewire the next/previous bundle pointers around b to exclude b. If s is only the top (bottom) segment of b, we can find the new boundary of b by finding the

segment of the same colour below (above) the endpoint of s on the sweep line. If s is red, we delete s from T and/or B, as necessary. In any case, we delete s from R_c, where c is the colour of s.

IsAbove$(e,\ b)$ / **IsBelow**$(e,\ b)$
We check in constant time whether or not e is above (below) the top (bottom) segment of b. The endpoint e must then be above (below) all other segments of b.

Split$(e,\ b)$
We find the segments of b's colour above and below e. The segment below e becomes the top segment of b, which we relabel to b_ℓ. The original top segment of b becomes the top segment of a new bundle b_h. The bottom segment of b_h is the segment above e. We rewire the next/previous bundle pointers to include b_h after b_ℓ. If b was red, we add the segment below e to T and the segment above e to B.

Merge$(b_\ell,\ b_h)$
If b_ℓ and b_h are red, we remove the top segment of b_ℓ from T and the bottom segment of b_h from B. We replace the top segment of b_ℓ with the top segment of b_h and rewire the next/previous bundle pointers to exclude b_h.

Swap$(b_\ell,\ b_h)$
We rewire next/previous bundle pointers to swap the order of b_ℓ and b_h.

HighestBelow(e) / **LowestAbove**(e)
We find the red segment immediately below e and find its predecessor in T. The bundle of the resulting top segment is the highest bundle below e. A similar process finds the lowest bundle above e.

Next(b) / **Previous**(b)
We follow b's next or previous bundle pointer.

Size(b)
We query R_c, where c is the colour of b, for the ranks of the top and bottom segments of b. We obtain the size of b by subtracting the latter from the former and adding 1.

5 Handling Dynamic Ranking

Dynamic ranking can be solved using Dietz's data structure [8], which supports both queries and updates in $O(\frac{\log n}{\log\log n})$ time. The high-level algorithm described in Section 2 must determine the sizes of at most $O(n)$ bundles. Also, each endpoint causes a single insertion or deletion from a rank query data structure. Thus, if we were to use Dietz's data structure, dynamic ranking would contribute $O(n\frac{\log n}{\log\log n})$ time to the runtime of our algorithm. However, by considering the purpose of our rank queries, it turns out that we can do better.

We use dynamic ranking data structures to determine the sizes of bundles. The high-level algorithm requires these sizes for a single purpose: to calculate the total number of bichromatic intersections between a red bundle and a blue bundle that intersect. It is important to note that the results of these calculations have no effect on future decisions of the algorithm. In fact, the results only affect the algorithm's output value. Another way to calculate the same output value is to perform all of the rank queries and updates offline at the end of the algorithm. Offline dynamic ranking can be solved faster than its online counterpart. Specifically, Chan and Pătraşcu [5] give an algorithm for offline dynamic ranking that handles $O(n)$ queries and updates in $O(n\sqrt{\log n})$ time.

6 Analysis

The plane sweep requires that all endpoints are sorted by their x-coordinate, which can be performed in $O(n\log\log n)$ deterministic time [9]. The rank space reduction of Section 3 is performed in $n\cdot 2^{O(\sqrt{\log\log n})}$ time. The preprocessing step of Lemma 1 also runs in $n\cdot 2^{O(\sqrt{\log\log n})}$ time. The high-level algorithm invokes each operation of our data structure at most $O(n)$ times. All operations consist of a constant number of pointer assignments, queries and updates to dynamic predecessor search data structures, and queries and updates to dynamic ranking data structures. The queries and updates to the dynamic predecessor search data structures contribute at most $O(n\frac{\log^2\log n}{\log\log\log n})$ deterministic time to the runtime of the algorithm, using Andersson and Thorup's data structure [1]. As discussed in Section 5, the queries and updates to the dynamic ranking data structures can be handled offline in $O(n\sqrt{\log n})$ time. This final contribution to the algorithm's runtime dominates all others; thus, the algorithm as a whole runs in $O(n\sqrt{\log n})$ time.

References

[1] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54, June 2007.

[2] K. Buchin and W. Mulzer. Delaunay triangulations in $O(\text{sort}(n))$ time and more. *J. ACM*, 58:6:1–6:27, April 2011.

[3] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *In Proc. 6th Canad. Conf. Comput. Geom*, pages 263–268, 1994.

[4] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. *SIAM J. Comput.*, 39:703–729, July 2009.

[5] T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 161–173, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[6] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, II: Offline search. *CoRR*, abs/1010.1948, 2010. Also in *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 31–39, New York, NY, USA, 2007. ACM.

[7] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994. 10.1007/BF01182771.

[8] P. F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS '89, pages 39–46, London, UK, 1989. Springer-Verlag.

[9] Y. Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 602–608, New York, NY, USA, 2002. ACM.

[10] H. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *Proceedings of the NATO Advanced Science Institute, Series F*, pages 307–326. Springer-Verlag, 1988.

[11] A. Mantler and J. Snoeyink. Intersecting red and blue line segments in optimal time and precision. In J. Akiyama, M. Kano, and M. Urabe, editors, *Discrete and Computational Geometry*, volume 2098 of *Lecture Notes in Computer Science*, pages 244–251. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-47738-1_23.

[12] Y. Nekrich. A fast algorithm for three-dimensional layers of maxima problem. *CoRR*, abs/1007.1593, 2010.

[13] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP: Graph. Models Image Process.*, 56:304–310, July 1994.

[14] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, Washington, DC, USA, 1975. IEEE Computer Society.