A Simple Trapezoid Sweep Algorithm for Reporting Red/Blue Segment Intersections

Timothy M. Chan*
Department of Computer Science
University of British Columbia

Abstract

We present a new simple algorithm for computing all intersections between two collections of disjoint line segments. The algorithm runs in $O(n \log n + k)$ time and $O(n)$ space, where n and k are the number of segments and intersections respectively. We also show that the algorithm can be extended to handle single-valued curve segments with the same time and space bound.

1 Introduction

In this paper, we consider the *red/blue segment intersection problem*: Given a disjoint set of red line segments and a disjoint set of blue line segments in the plane, with a total of n segments, report all k intersections of red segments with blue segments.

This is a special case of the general segment intersection problem of reporting all k pairwise intersections of a given set of n line segments in the plane.

Although Chazelle and Edelsbrunner[4] gave an asymptotically time-optimal algorithm for the general segment intersection problem which runs in $O(n \log n + k)$ time and uses $O(n + k)$ space, simpler methods that use $O(n)$ space exist for the red/blue problem. Mairson and Stolfi[6], who also cited Mairson[5] and Mowchenko[7], presented a plane-sweep algorithm for the red/blue intersection problem that runs in asympotically optimal $O(n \log n + k)$ time and $O(n)$ space. Chazelle et al.[3] proposed another algorithm that can report red/blue intersections with the same time and space bound, and can also *count* red/blue intersections in $O(n \log n)$ time; their algorithm is later simplified by Palazzi and Snoeyink[8].

Here, we give a new simple algorithm that reports all red/blue intersections in $O(n \log n + k)$ time and $O(n)$ space, based on a variation of the plane sweep, which we call the *trapezoid sweep*, and we extend our algorithm to deal with curve segments. Our algorithm has smaller constant factors than the ones in [3, 4, 8], and is similar to Mairson and Stolfi's, but Mairson and Stolfi's requires a recursive cone-breaking procedure and needs an additional $O(k)$ space for curve segments.

The organization of the paper is as follows: Section 2 describes the basic idea of the algorithm, and Section 3, 4, and 5 gives the algorithm; in Section 6, we consider curve segments; and Section 7 is the conclusion.

2 The Trapezoid Sweep

The classic plane-sweep algorithm of Bentley and Ottmann[2] computes intersections of general line segments in the order of increasing x-coordinate and requires $O(n \log n + k \log n)$ time. In order to remove a logarithmic factor from k, we shall compute intersections in a somewhat different order that does not result in sorting all intersections.

Define the *blue trapezoidation* to be the decomposition of the plane into (closed) trapezoids obtained from the blue segments and the vertical extensions of each endpoint (*both* red and blue) to the blue segments just above and below it, as shown in Figure 1 (recall that the blue segments are disjoint). For each of these *blue trapezoids*, the left and right walls are vertical line segments, the top and bottom sides are parts of blue segments, and the interior contains no red or blue endpoints.

*Supported by a Killam Predoctoral Fellowship.

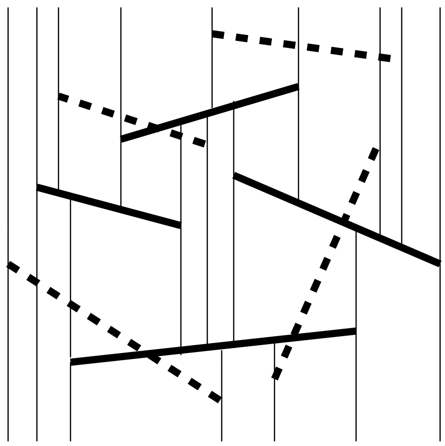

Figure 1: A blue trapezoidation. (Red segments are dashed.)

Our algorithm shall sweep through these blue trapezoids from left to right by a vertical *sweep line*, and whenever we hit the right wall of a blue trapezoid T, we add T to our *sweep front* F and maintain:

Invariant: For each red segment s_{red}, the intersections of s_{red} reported so far are exactly the intersections of s_{red} that are to the left of, or equal to, the rightmost point of s_{red} in F; furthermore, each such intersection is reported only once.

F is initially empty. After we have swept through all blue trapezoids, F shall become the entire plane, and we shall have computed all intersections.

3 Data Structures

We now specify the data structure requirements of our algorithm. To simplify our presentation, we shall ignore degeneracies, e.g. we assume that the endpoints and intersections have distinct x-coordinates. Standard perturbation techniques can be used to deal with these degeneracies (for example, see [6]).

We first assume the following $O(1)$ time operations:

- *report*(s_{red}, s_{blue}) reports the intersection of red segment s_{red} and blue segment s_{blue}.
- *meet*(s_{red}, s_{blue}) returns $-\infty$ if s_{red} and s_{blue} don't intersect (or s_{red} or s_{blue} is *nil*); otherwise, it returns the x-coordinate of the intersection of s_{red} and s_{blue}.

The input to our algorithm is the sequence of endpoints sorted by x-coordinate, along with the following information:

- $x[i]$ is the x-coordinate of the i-th endpoint (in the sorted order).
- $s[i]$ is the segment of the i-th endpoint.
- *type*$[i]$ is the type (*left* or *right*) of the i-th endpoint.
- $c[i]$ is the colour (*red* or *blue*) of $s[i]$.

The global variables that our algorithm uses are:

- x_{sweep} is the x-coordinate of the sweep line.
- $x_0[s_{red}]$ is the largest x-coordinate of the reported intersections of red segment s_{red}. Initially, $x_0[s_{red}]$ is set to the x-coordinate of the left endpoint of s_{red}.
- L_{red} and L_{blue} are the lists of red, and respectively, blue segments that intersect the sweep line. The two lists are ordered by the y-values of their segments at x-coordinate x_{sweep}. Initially, L_{red} and L_{blue} are empty.

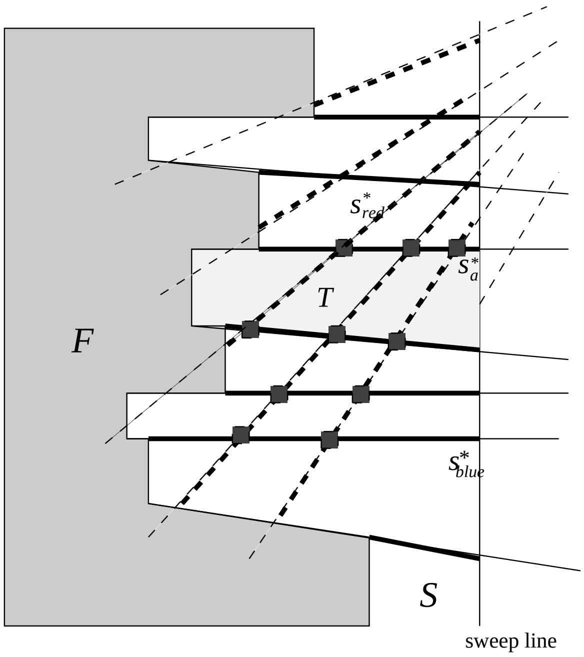

Figure 2: Adding a blue trapezoid T to the sweep front F. (Marked points are the intersections reported by the call $advance(s_a)$.)

We need the following operations on ordered lists of line segments:

- $insert(L, s)$ adds s to list L.
- $delete(L, s)$ deletes s from list L.
- $search(L, s, dir)$ returns the element in list L that is just greater (or less) than s if $dir = +1$ (or -1).
- $next(L, s, dir)$ returns the successor (or predecessor) of s in list L if $dir = +1$ (or -1).

search()/*next()* returns *nil* if its result is undefined. We can use balanced-tree structures, such as red-black trees or splay trees, to implement these operations in $O(\log n)$ time (amortized, for splay trees), and with extra pointers, *next()* in $O(1)$ time.

By the disjointness of red segments and of blue segments, the ordered lists L_{red} and L_{blue} can be maintained as long as we do an *insert()* whenever the sweep line hits a left endpoint, and a *delete()* whenever the sweep line hits a right endpoint.

4 Adding a Blue Trapezoid to the Sweep Front

Suppose the invariant currently holds, and the sweep line next hits the right wall of the blue trapezoid T, i.e. the right wall of T has the next smallest x-coordinate x_{sweep}.

Let $S = \{(x, y) | \ x < x_{sweep}\} \setminus F$. Then S is the part to the left of the sweep line of the union of all blue trapezoids that intersects the sweep line. In particular, S contains no red or blue endpoints.

For each $s_{blue} \in L_{blue}$, define s^*_{blue} to be the part of s_{blue} that is in S. For each $s_{red} \in L_{red}$, define s^*_{red} to be the part of s_{red} that is to the right of the rightmost point of s_{red} in F and is to the left of the sweep line. Then, for every $s \in L_{red} \cup L_{blue}$, s^* is a line segment, the left endpoint of s^* is on the left boundary of S (i.e. the boundary of F), the right endpoint of s^* is on the right boundary of S (i.e. the sweep line), and s^* is contained in S. We thus have Figure 2.

For every $s_{red} \in L_{red}$ and $s_{blue} \in L_{blue}$, note the following observation from the invariant: s^*_{red} and s^*_{blue} intersect iff s^*_{red} and s_{blue} intersect, iff s_{red} and s_{blue} have an unreported intersection to the left of the sweep line, iff s_{red} and s_{blue} intersect at a point with x-coordinate strictly between $x_0[s_{red}]$ and x_{sweep}.

We can define a procedure, $advance(s)$, which does the following given $s \in L_{blue}$: for each s^*_{red} that intersects s^*, report the intersection of s^*_{red} with s^* and the intersections of s^*_{red} with all other s^*_{blue} to left of that intersection. (See Figure 2.) Then it is not hard to show that we can add T to the sweep front F while maintaining the invariant by calling $advance(s_a)$ and $advance(s_b)$, where s_a and s_b are the blue segments on the top and bottom sides of T.

To find all $s_{red} \in L_{red}$ such that s^*_{red} and s^* intersect, we simply enumerate the s_{red}'s in L_{red} in increasing order of their y-values at the sweep line, starting with the one that is immediately above s at the sweep line, until we encounter the first s_{red} such that s^*_{red} and s^* don't intersect (then all other s'^*_{red} above that s^*_{red} don't intersect with s^* since the red segments are disjoint); we proceed in the other direction similarly.

Given one such s_{red} (w.l.o.g. assume that s_{red} is above s at the sweep line), to find all $s_{blue} \in L_{blue}$ such that s^*_{red} and s^*_{blue} intersect at a point to the left of the intersection of s^*_{red} and s^*, we simply enumerate the s_{blue}'s in L_{blue} in decreasing order of their y-values at the sweep line, starting with s, until we encounter the first s_{blue} such that s^*_{red} and s^*_{blue} don't intersect (then all other s'^*_{blue} below that s^*_{blue} don't intersect with s^*_{red} since the blue segments are disjoint).

Hence, the procedure *advance()* can be written as follows:

PROCEDURE $advance(s)$, where s is a blue segment:
    for $dir \in \{+1, -1\}$ do
        $s_{red} \leftarrow search(L_{red}, s, dir)$
        while $x_0[s_{red}] < meet(s_{red}, s) < x_{sweep}$ do
            $s_{blue} \leftarrow s$
            while $x_0[s_{red}] < meet(s_{red}, s_{blue}) < x_{sweep}$ do
                $report(s_{red}, s_{blue})$
                $s_{blue} \leftarrow next(L_{blue}, s_{blue}, -dir)$
            $x_0[s_{red}] \leftarrow meet(s_{red}, s)$
            $s_{red} \leftarrow next(L_{red}, s_{red}, dir)$

Remark: If we want to, we can easily modify *advance()* so that the intersections along each segment are reported in sorted order by executing each inner and outer while loop in "reverse" order.

Excluding the two calls to *search()*, which take $O(\log n)$ time, the cost of *advance()* is proportional to the number of intersections that it reports.

5 The Algorithm

Now, the algorithm works by sweeping the endpoints from left to right. When we encounter a red endpoint or a blue left endpoint, we have hit the right wall of one blue trapezoid, so we have one trapezoid to add to the sweep front F. When we encounter a blue right endpoint, we have hit the right wall of two blue trapezoids, so we have two trapezoids to add to F. To add a blue trapezoid to F, we call the *advance()* procedure from Section 4 on the blue segments on its top and bottom sides.

ALGORITHM:
    for $i = 1, \ldots, 2n$ do
        $x_{sweep} \leftarrow x[i]$
        if $c[i] = red$ or $type[i] = left$ then
            $advance(search(L_{blue}, s[i], +1))$
            $advance(search(L_{blue}, s[i], -1))$
        else
            $advance(next(L_{blue}, s[i], +1))$
            $advance(s[i])$
            $advance(next(L_{blue}, s[i], -1))$
        if $type[i] = left$ then
            $insert(L_{c[i]}, s[i])$
        else
            $delete(L_{c[i]}, s[i])$

The i-th step of the for loop takes $O(\log n + k_i)$ time, where k_i is the number of intersections reported at that step. Thus, the total time of the algorithm, including the pre-sorting of endpoints, is $O(n \log n + \sum_i k_i) = O(n \log n + k)$ (since each intersection is reported once). The space requirement is clearly $O(n)$.

6 Modifications for Curve Segments

Our algorithm can be extended to handle curve segments which satisfy the following conditions:

- Segments are continuous.
- Segments are *single-valued* (or *monotone*), i.e. a segment can intersect a vertical line in at most one point.
- The intersection of a segment and a vertical line can be computed in $O(1)$ time and space.
- The rightmost intersection of two segments to the left of a given vertical line can be computed in $O(1)$ time and space.

Since two segments may now intersect more than once, we first have to add an extra argument to *report()* and *meet()*:

- $report(s_{red}, s_{blue}, x)$ reports the intersection of red segment s_{red} and blue segment s_{blue} at x-coordinate x.
- $meet(s_{red}, s_{blue}, x_0)$ returns the largest x such that $x < x_0$ and x is the x-coordinate of an intersection of s_{red} and s_{blue}; it returns $-\infty$ if no such x exists.

The only major modification to the algorithm is then the inner while loop of *advance()*, where we report the intersections of s^*_{red} with all s^*_{blue}'s to the left of the intersection of s^*_{red} with s^*. As we enumerate the s^*_{blue}'s, we may now have to move in both the "upward" and "downward" directions in L_{blue}.

PROCEDURE $advance(s)$, where s is a blue segment:
    for $dir \in \{+1, -1\}$ do
        $s_{red} \leftarrow search(L_{red}, s, dir)$
        while $x_0[s_{red}] < meet(s_{red}, s, x_{sweep})$ do
            $s_{blue} \leftarrow s,\ \ x \leftarrow x_{sweep},\ \ dir' \leftarrow -dir$
            while $x_0[s_{red}] < meet(s_{red}, s_{blue}, x)$ do
                $x \leftarrow meet(s_{red}, s_{blue}, x)$
                $report(s_{red}, s_{blue}, x)$
                $s'_{blue} \leftarrow next(L_{blue}, s_{blue}, dir')$
                if $meet(s_{red}, s_{blue}, x) < meet(s_{red}, s'_{blue}, x)$ then
                    $s_{blue} \leftarrow s'_{blue}$
                else
                    $dir' \leftarrow -dir'$
            $x_0[s_{red}] \leftarrow meet(s_{red}, s, x_{sweep})$
            $s_{red} \leftarrow next(L_{red}, s_{red}, dir)$

Remark: If we want to, we can easily modify *advance()* so that the intersections along each red (but not blue) segment are reported in sorted order by executing each inner while loop in "reverse" order.

The time and space bound of the algorithm are as before.

7 Conclusions

Our algorithm has been implemented and tested on data from GIS map overlay applications. On these data, it outperforms the algorithms in [2, 8]. Furthermore, it is competitive with heuristic methods such as quadtree and binary space partition, especially if the endpoints are pre-sorted. [1] contains the experimental results.

Acknowledgements

I would like to express my thanks to Jack Snoeyink for suggesting this investigation and for reading the draft of this paper.

References

[1] D. S. Andrews, J. Snoeyink, J. Boritz, T. M. Chan, G. Denham, J. Harrison, and C. Zhu. Further Comparison of Algorithms for Geometric Intersection Problems. Accepted to *6th International Symposium on Spatial Data Handling*, 1994.

[2] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric interesections. *IEEE Transactions on Computers*, C-28(9):643-647, 1979.

[3] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Technical Report UIUC DCS-R-90-1578, Dept. Comp. Sci., Univ. Ill. Urbana, 1990.

[4] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *JACM*, 39:1-54, 1992.

[5] H. G. Mairson. Reporting line segment intersections. Manuscript, 1981.

[6] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. Earnshaw (ed.), *Theoretical Foundations of Computer Graphics and CAD*, NATO ASI Series, Vol. F40, 307-326, Springer-Verlag, 1988.

[7] J. T. Mowchenko. Ph. D. Thesis. Department of Electrical Engineering, University of Waterloo, Ontario, Canada, 1983.

[8] L. Palazzi and J. Snoeyink. Counting and Reporting Red/Blue Segment Intersections. In *Algorithms and Data Structures (WADS '93)*, number 709 in Lecture Notes in Computer Science, 530-540, Springer-Verlag, 1993.