

Computing intersections in a set of line segments: the Bentley-Ottmann algorithm

Michiel Smid*

October 14, 2003

1 Introduction

In these notes, we introduce a powerful technique for solving geometric problems. This technique, called the *plane sweep technique*, appears in Shamos' Ph.D. thesis from 1978—which is considered the birthplace of computational geometry—although the concept was known already to geometers. The plane sweep technique gives efficient and reasonably simple algorithms for a large variety of geometric problems. In many cases, the only data structures needed are balanced binary search trees.

2 A general description of the plane sweep technique

The basic idea of the plane sweep technique is as follows. Let S be a set of planar objects for which we want to solve a given problem. We move (sweep) a vertical line SL —the *sweep line*—from left to right over the elements of S . During the sweep, we dynamically maintain (information based on) the intersection $SL \cap S$, which is a one-dimensional scene, and, simultaneously, compute the desired information for parts of the set S . We maintain the following invariant.

*School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6.
E-mail: michiel@scs.carleton.ca.

1. The output for those elements of S that are completely to the left of the sweep line has been computed already.
2. There are data structures containing information needed to solve the problem for the other elements of S , i.e., those elements that intersect or are completely to the right of the sweep line.

If the sweep line is to the right of all elements of S , then the first part of the invariant implies that we have solved our problem for the entire set S .

Let us consider this in some more detail. For any position of the sweep line SL , let S_{SL} be the set of elements of S that intersect SL . We maintain these elements in a so-called *Y-structure*, which depends on the problem to be solved. For many applications, the elements of S_{SL} can be sorted with respect to the y -coordinates of their intersections with the sweep line, and the *Y-structure* is just a balanced binary search tree storing S_{SL} according to this order.

The *Y-structure* changes if

1. the sweep line encounters an element of S , in which case we have to *insert* certain information into the *Y-structure*,
2. the sweep line encounters the rightmost part of an element of S , in which case we have to *delete* certain information from the *Y-structure*, or
3. the relative order of the elements of S_{SL} on the sweep line changes, in which case we have to *interchange* certain information in the *Y-structure*.

The positions of the sweep line where the *Y-structure* changes are called the *transition points* of the sweep. At the start of the sweep algorithm, certain (sometimes even all) transition points are known already. The remaining transition points are discovered during the sweep itself. In order to maintain these transition points, we maintain a dynamic data structure, called the *X-structure*. This structure must support the following operations.

1. *Insertion* of a new transition point.
2. *Delete-min*, i.e., find and delete the transition point with minimum x -coordinate, which is the next transition point of the sweep algorithm.

3. In some applications, we also have to be able to *delete* an arbitrary transition point.

In many applications, the X -structure is a balanced binary search tree or just a linked list or array, containing the transition points sorted by their x -coordinates.

The general description given above implies that the plane sweep technique reduces a *static* two-dimensional problem to a *dynamic* one-dimensional problem. As we mentioned already, the latter dynamic problem can often be solved using binary search trees. In the next section, we will apply this technique to solve a basic geometric problem.

3 The line segment intersection problem

As a concrete (and classical) application of the plane sweep technique, we consider the *line segment intersection problem*, which is defined as follows.

We are given a set $S = \{L_1, L_2, \dots, L_n\}$ of n line segments in the plane. Our task is to compute all pairs (L_i, L_j) , $i \neq j$, of segments that intersect.

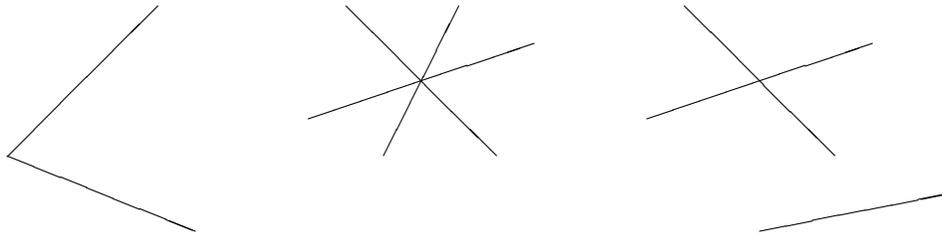
A trivial solution to this problem considers all $\binom{n}{2}$ pairs (L_i, L_j) , $i \neq j$, and checks for each such pair if the two segments intersect. Clearly, this algorithm has a running time of $O(n^2)$. This is worst-case optimal, because there can be a quadratic number of intersecting pairs. We want, however, an algorithm whose running time does not only depend on n , but also on the number of intersections. That is, if there are few intersections, the algorithm should be much faster than $\Theta(n^2)$. As we will see, the plane sweep technique gives us such an algorithm. This algorithm—which is probably the most famous algorithm in computational geometry—is due to Bentley and Ottmann (1979). Its running time is $O((n + k) \log n)$, where k is the number of intersecting pairs of segments. We say that this algorithm is *output-sensitive*, because its running time does not only depend on n , but also on the size k of the output.

Exercise 1 Let L and L' be two line segments. Assume that each segment is given by its two endpoints. How can we decide if L and L' intersect and, if they do, how can we compute their intersection?

3.1 The Bentley-Ottmann algorithm

We make the following simplifying assumptions about the input: (i) There are no vertical segments, (ii) no two segments intersect at their endpoints, (iii) no three (or more) segments have a common intersection, (iv) all endpoints of the segments and all intersection points have different x -coordinates, and (v) no two segments overlap. The algorithm that will be discussed in these notes can be extended such that it also works if these assumptions are not satisfied. (See the reference at the end of these notes.) The details, in particular those that arise when implementing the algorithm, become more complicated. Since these details do not give more insight into the plane sweep technique, they are omitted here.

The figure below illustrates our assumptions. Assumption (ii) excludes the left case, (iii) excludes a situation as in the middle, and (iv) excludes the right case.



In order to compute all intersecting pairs of segments, we move the vertical sweep line SL from left to right, starting to the left of all segments, say at position $x = -\infty$. For any position of the sweep line, we partition the segments into three groups.

1. *Dead segments*: these are the segments that are completely to the left of SL .
2. *Active segments*: these are the segments that intersect SL .
3. *Sleeping segments*: these are the segments that are completely to the right of SL .

First observe that the active segments can be sorted with respect to the y -coordinates of their intersections with the sweep line. As we saw in Section 2,

the active segments will be maintained in a data structure called the Y -structure. What are the *transition points*? In other words, when does the order of the active segments on the sweep line change? This order changes if the sweep line reaches the left or right endpoint of a segment, or if it reaches the intersection of two segments. We want to maintain these transition points in the X -structure. At the start of the algorithm, not all transition points are known: at that moment, we only know the endpoints of the segments. As we will see, the other transition points (the intersections of segments), are discovered during the sweep, and will be inserted into the X -structure (at some later stage, they will be deleted again).

Before we give the details, we need some definitions. Let L and L' be two active segments, and let p and p' be their intersections with the sweep line SL , respectively. Assume that p is below p' . Then we say that L and L' are *neighbors* on SL if there is no active segment L'' , different from L and L' , whose intersection with SL is between p and p' . In this case, we say that L' is the *successor* of L on SL , and L is the *predecessor* of L' on SL .

During the sweep, we maintain the following three invariants.

Invariant 1: For any position of the sweep line SL , the Y -structure contains all active segments, sorted with respect to the y -coordinates of their intersections with SL .

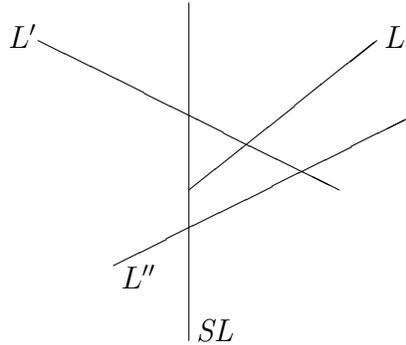
Invariant 2: For any position of the sweep line SL , the X -structure contains (i) all endpoints of sleeping segments, (ii) all endpoints to the right of SL of active segments, and (iii) all intersections to the right of SL of active segments that are neighbors on SL . The elements in the X -structure are sorted by x -coordinate.

Invariant 3: For any position of the sweep line SL , all pairs of intersecting dead segments have been reported.

In Section 3.3, we will see that we can take balanced binary search trees for both the X -structure and the Y -structure.

As mentioned already, the sweep line starts at position $x = -\infty$, and then moves to the right, until it reaches position $x = +\infty$. What happens if the sweep line moves from one transition point to the next one? There are three possible cases.

Case A: The sweep line encounters the left endpoint of a segment, say L .

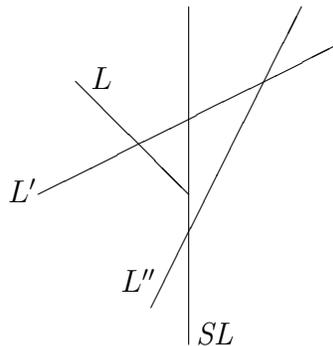


A.1: Since L becomes active, we insert it into the Y -structure.

A.2: We search for the segment L' in the Y -structure that is immediately above L on SL (i.e., the successor of L). Similarly, we search for the segment L'' in the Y -structure that is immediately below L on SL (i.e., the predecessor of L). We assume for simplicity that both L' and L'' exist. Since L and L' are neighbors on SL now, we test if they intersect. If they do, we insert their intersection into the X -structure. Similarly, we test if L and L'' intersect and, if they do, insert their intersection into the X -structure.

A.3: Since L' and L'' are not neighbors on SL any more, we test if they intersect to the right of SL . If they do, we delete their intersection from the X -structure.

Case B: The sweep line encounters the right endpoint of a segment, say L .

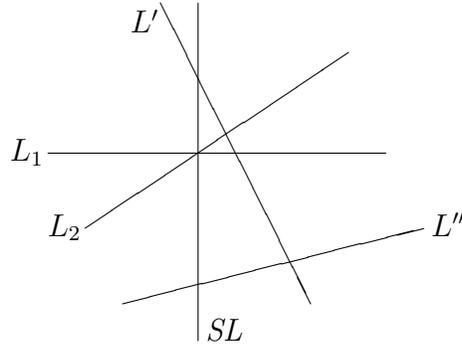


B.1: We search in the Y -structure for the successor L' and the predecessor L'' of L . Again, we assume for simplicity that both L' and L'' exist.

B.2: We delete the segment L from the Y -structure.

B.3: We test if L' and L'' intersect to the right of SL . If they do, we insert their intersection into the X -structure.

Case C: The sweep line encounters the intersection of two segments, say L_1 and L_2 .



Assume that immediately before the intersection, L_2 is below L_1 on the sweep line.

C.1: We report the intersecting pair (L_1, L_2) .

C.2: We search in the Y -structure for the successor L' of L_1 and the predecessor L'' of L_2 . We assume for simplicity that both L' and L'' exist.

C.3: We change the order of L_1 and L_2 in the Y -structure. In case L_2 and L' intersect to the right of SL , we insert their intersection into the X -structure. Similarly, in case L_1 and L'' intersect to the right of SL , we insert their intersection into the X -structure.

C.4: If L_1 and L' (resp. L_2 and L'') intersect to the right of SL , then we delete their intersection from the X -structure.

Remark 1 Steps A.3 and C.4, i.e., deleting intersections between segments that are not neighbors on the sweep line any more, are not necessary for the *correctness* of our algorithm. These steps guarantee that the X -structure only contains intersections between segments that are neighbors on SL . As a result, at any moment, the X -structure contains at most $3n - 1$ points: There are $2n$ endpoints of segments, and at most $n - 1$ neighboring active segment pairs.

Algorithm *Bentley-Ottmann*(S)
 (* S is a set of n line segments in the plane *)
 initialize an empty Y -structure;
 sort the $2n$ endpoints of all segments by x -coordinate, and
 store them in the X -structure;
 (* the sweep line SL is at position $x = -\infty$ *)
while X -structure $\neq \emptyset$
do let min be the minimum element in the X -structure;
 delete min from the X -structure;
 (* the sweep line SL moves to position $x = min$ *)
if min is the left endpoint of a segment
then carry out the steps as specified in Case A
endif;
if min is the right endpoint of a segment
then carry out the steps as specified in Case B
endif;
if min is an intersection of two segments
then carry out the steps as specified in Case C
endif
endwhile
 (* the sweep line SL is at position $x = +\infty$ *)

Figure 1: *The Bentley-Ottmann algorithm solving the line segment intersection problem.*

The algorithm for computing all pairs of intersecting segments is given in Figure 1. In the next sections, we will analyze this algorithm.

3.2 The correctness proof

Recall that the transition points of the sweep are (i) the endpoints of the segments and (ii) the intersections between segments. It is clear that between two transition points, the order of the active segments on the sweep line does not change. Using this, it can be shown that during the while-loop, Invariants 1 and 2 are correctly maintained. Intuitively, it should also be

clear that Invariant 3 is correctly maintained. We will prove this formally.

Lemma 1 *During the while-loop, Invariant 3 is maintained.*

Proof: Consider any position of the sweep line. We will show that each intersection among the dead segments has appeared as minimum element in the X -structure.

If this claim is true, then the third **if-then** statement in the algorithm in Figure 1 implies that all intersections among the dead segments have been reported. (Each such intersection is reported in an execution of this statement.) Hence, it will prove that Invariant 3 holds.

The claim is proved by contradiction. Assume there is an intersection among the dead segments that did not appear as minimum element in the X -structure. Let p be the leftmost intersection with this property. Let L and L' be the two dead segments that intersect in p . Let q be the rightmost transition point to the left of p . Observe that q exists and that its x -coordinate is at least equal to the x -coordinates of both endpoints of L and L' . Also, q has appeared as minimum element in the X -structure. We consider what happened at the moment when the sweep line just left q . At that moment, L and L' were both active and, hence, were both stored in the Y -structure. We distinguish two cases.

Case 1: Immediately after the sweep line visited point q , the segments L and L' were neighbors on SL .

By Invariant 1, point p was contained in the X -structure at that moment. Therefore, by our choice of q , point p was the next transition point. Hence, immediately after we processed q , point p was the minimum element in the X -structure. This is a contradiction.

Case 2: Immediately after the sweep line visited point q , the segments L and L' were not neighbors on SL .

In this case, there is an endpoint or intersection point r strictly between p and q . Since r is a transition point, this is a contradiction to our choice of q .

This proves the claim and, hence, the lemma. ■

Lemma 2 *Algorithm Bentley-Ottmann(S) computes all pairs of line segments of S that intersect.*

Proof: At the end of the algorithm, the X -structure is empty, and the sweep line is to the right of all segments. That is, at the end of the algorithm, all segments are dead. The claim then follows from Invariant 3, which is correctly maintained by the previous lemma. ■

Exercise 2 We have seen that our algorithm computes all pairs of intersecting segments. Prove that each intersecting pair is reported exactly once. How many times can an intersection be inserted and deleted in the X -structure?

3.3 Implementation of the X - and Y -trees and complexity of the algorithm

Before we can analyze the running time of the algorithm, we have to give some implementation details.

We take for the Y -structure a balanced binary search tree, in which we store all active segments, sorted with respect to the y -coordinates of their intersections with the sweep line. A segment L is given by its two endpoints ℓ_L and r_L . Each node of the tree stores the two endpoints of an active segment.

Suppose the sweep line is at the position $x = \alpha$. To search in the tree with a y -coordinate β , we start in the root and walk down a path. At each node (storing, say, the segment L), we use left/right turn tests, using the three points ℓ_L , r_L , and (α, β) , to decide whether to proceed to the left or right child.

Hence, we can search in the Y -structure in time that is proportional to the logarithm of the number of segments stored in it. Since the Y -structure contains at most $n - 1$ segments, the search time is $O(\log n)$. Similarly, we can in $O(\log n)$ time find the successor and predecessor of a segment, and insert and delete a segment. To change the order of two segments L_1 and L_2 in the Y -structure, we first search for the nodes u and v , respectively, that contain these segments. Then we write the two endpoints of L_1 in v , and the two endpoints of L_2 in u . Hence, changing the order of two segments can also be done in $O(\log n)$ time.

The X -structure contains a subset of the endpoints and intersections between segments. We have to be able to insert and delete points, and to find and delete the element with minimum x -coordinate. We again take a balanced binary search tree to implement this data structure. Observe that

at any time, the X -structure contains at most $3n - 1$ elements. Therefore, each operation takes $O(\log n)$ time.

Now we can bound the running time of algorithm *Bentley-Ottmann*(S). In the initialization step, we sort the $2n$ endpoints of the segments by x -coordinate, and store them in a binary search tree. This takes $O(n \log n)$ time. Let k be the number of intersections among the segments. Then the while-loop of the algorithm makes $2n + k$ iterations. In each iteration, a constant number of operations are performed in the X - and Y -structures. That is, each iteration takes $O(\log n)$ time. It follows that the total running time of the algorithm is

$$O(n \log n) + (2n + k) \cdot O(\log n) = O((n + k) \log n).$$

How much space does the algorithm use? The X - and Y -structures both have size $O(n)$. Clearly, it takes $O(k)$ space to store all k intersections. Hence, the algorithm uses $O(n + k)$ space.

There are applications in which an intersection is sent to the output at the moment when it is reported. In such applications, we do not count the number of intersections in the space bound, because they are not actually kept in memory. We only count the size of the data structures and speak about the amount of *working space* used. In our case, the algorithm uses $O(n)$ working space. We have proved the following result.

Theorem 1 *We can compute the k intersections in a set of n line segments in the plane in $O((n + k) \log n)$ time, using $O(n)$ working space.*

4 Remarks

We have described the Bentley-Ottmann algorithm for line segments that satisfy the assumptions mentioned at the beginning of Section 3.1. Unfortunately, these assumptions are not realistic in real life and, as a result, implementing the algorithm leads to non-trivial problems. Another source of problems is the finite precision arithmetic of real computers.

Bartuschka, Mehlhorn and Näher have given a *robust* and *exact* implementation of the Bentley-Ottmann algorithm. This implementation is guaranteed to give the correct and exact solution for any set of line segments. The paper can be found at

<http://www.dsi.unive.it/~wae97/proceedings/>

The Bentley-Ottmann algorithm has a running time of $O((n + k) \log n)$, which is not optimal. In 1988, Chazelle and Edelsbrunner showed how to solve the line segment intersection problem optimally, i.e., in $O(n \log n + k)$ time. This algorithm, however, is much more complicated. Simpler, but randomized, algorithms are known that solve the problem in $O(n \log n + k)$ expected time. The reader is referred to the books by Mulmuley and by Boissonnat and Yvinec.