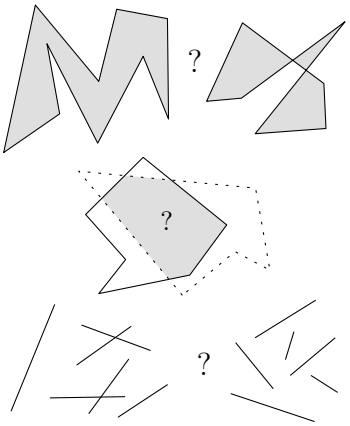
3. Line Segment Intersections

Lecture on Monday 29th September, 2008 by Michael Hoffmann <hoffmann@inf.ethz.ch>

3.1 Motivation

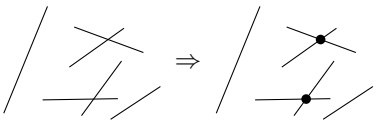


Problem 1 *Given a sequence $P = (p_1, \ldots, p_n)$ of points in $\mathbb{R}^2$, does P describe the boundary of a simple polygon?*

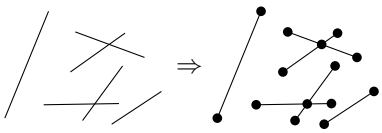
Problem 2 (Polygon Intersection) *Given two simple polygons P and Q in $\mathbb{R}^2$ as a (counterclockwise) sequence of their vertices, is $P \cap Q = \emptyset$?*

Problem 3 (Segment Intersection Test) *Given a set of n closed line segments in $\mathbb{R}^2$, do any two of them intersect?*

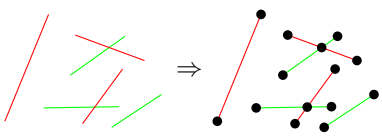
When do two line segments intersect? If one endpoint lies on the other? If two segments share an endpoint? What about overlapping segments and segments of length zero? For the time being, let us count all these as intersections. Except in case of simple polygons, where we do not want to consider the shared endpoint between two consecutive edges of the boundary.



Problem 4 (Segment Intersections) *Given a set of n closed line segments in $\mathbb{R}^2$, compute all pairs of segments that intersect.*



Problem 5 (Segment Arrangement) *Given a set of n closed line segments in $\mathbb{R}^2$, construct the arrangement induced by S, that is, the subdivision of $\mathbb{R}^2$ induced by S.*



Problem 6 (Map Overlay) *Given two sets S and T of n and m, respectively, pairwise interior disjoint line segments in $\mathbb{R}^2$, construct the arrangement induced by $S \cup T$.*

Two segments. Given two line segments $s = \lambda a + (1-\lambda)b, \lambda \in [0,1]$ and $s = \mu c + (1-\mu)d, \mu \in [0,1]$, it is a simple exercise in linear algebra to compute $s \cap t$. Note that $s \cap t$ is either empty or a single point or a line segment.

For $a = (a_x, a_y)$, $b = (b_x, b_y)$, $c = (c_x, c_y)$, and $d = (d_x, d_y)$ we obtain two linear equations in two variables λ and μ.

$$\begin{aligned} \lambda a_x + (1-\lambda)b_x &= \mu c_x + (1-\mu)d_x \\ \lambda a_y + (1-\lambda)b_y &= \mu c_y + (1-\mu)d_y \end{aligned}$$

Rearranging terms yields

$$\begin{aligned}\lambda(a_x - b_x) + \mu(d_x - c_x) &= d_x - b_x \\ \lambda(a_y - b_y) + \mu(d_y - c_y) &= d_y - b_y\end{aligned}$$

Assuming that the lines underlying s and t have distinct slopes (that is, they are neither identical nor parallel) we have

$$D = \begin{vmatrix} a_x - b_x & d_x - c_x \\ a_y - b_y & d_y - c_y \end{vmatrix} = \begin{vmatrix} a_x & a_y & 1 & 0 \\ b_x & b_y & 1 & 0 \\ c_x & c_y & 0 & 1 \\ d_x & d_y & 0 & 1 \end{vmatrix} \neq 0$$

and using Cramer's rule

$$\lambda = \frac{1}{D} \begin{vmatrix} d_x - b_x & d_x - c_x \\ d_y - b_y & d_y - c_y \end{vmatrix} \text{ and } \mu = \frac{1}{D} \begin{vmatrix} a_x - b_x & d_x - b_x \\ a_y - b_y & d_y - b_y \end{vmatrix}.$$

To test if s and t intersect, we can—after having sorted out the degenerate case in which both segments have the same slope—compute λ and μ and then check whether $\lambda, \mu \in [0, 1]$.

Algebraic degree. When implementing geometric algorithms, there are some fundamental geometric predicates and constructions on the bottom level which in turn are based on fundamental arithmetic operations. For instance, we formulated planar convex hull algorithms in terms of an orientation predicate which can be implemented using multiplication and addition/subtraction of coordinates/numbers. When using limited precision arithmetic, it is important to keep an eye on the size of the expressions that occur during an evaluation of such a predicate.

The rightturn predicate—given three points $p = (p_x, p_y), q = (q_x, q_y), r = (r_x, r_y) \in \mathbb{R}^2$, is r strictly to the right of the oriented line pr—can be computed as $(q_x - p_x)(r_y - p_y) - (r_x - p_x)(q_y - p_y) < 0$, that is, by evaluating a polynomial of degree two in the input coordinates. Therefore we say that the *algebraic degree* or, for short, degree of the orientation test is two. Observe that the definition of degree depends on the algebraic expression used to evaluate the predicate.

The degree of a predicate corresponds to the size of numbers that arise during its evaluation. If all input coordinates are k-bit integers then the numbers that occur during an evaluation of a degree d predicate on these coordinates are of size about[1] dk. If the number type used for the computation can represent all such numbers, the predicate can be evaluated exactly and thus always correctly. For instance, when using a standard IEEE double precision floating point implementation which has a mantissa length of 53

[1] It is only *about* dk because not only multiplications play a role but also additions. As a rule of thumb, a multiplication may double the bitsize, while an addition may increase it by one.

bit then the above orientation predicate can be evaluated exactly if the input coordinates are integers between 0 and 2^{25}, say.

Getting back to the line segment intersection test, observe that both λ and D result from multiplying two differences of input coordinates. Computing the x-coordinate of the point of intersection via $b_x + \lambda(a_x - b_x)$ uses another multiplication. Overall this computation yields a fraction whose numerator is a polynomial of degree three in the input coordinates and whose denominator is a polynomial of degree two in the input coordinates. That is, if the input coordinates are k-bit integers then the coordinates need about $5k$ bits to be represented, and thus lexicographic comparison of two intersection points of line segments is a degree five predicate.

Trivial Algorithm. Test all the $\binom{n}{2}$ pairs of segments from S in $O(n^2)$ time and $O(n)$ space. For Problem 4 this is worst-case optimal because there may by $\Omega(n^2)$ intersecting pairs.

But in case that the number of intersecting pairs is, say, linear in n there is still hope to obtain a subquadratic algorithm. Given that there is a lower bound of $\Omega(n \log n)$ for *Element Uniqueness* (Given $x_1, \ldots, x_n \in \mathbb{R}$, is there an $i \neq j$ such that $x_i = x_j$?) in the algebraic computation tree model, all we can hope for is an output-sensitive runtime of the form $O(n \log n + k)$, where k denotes the number of intersecting pairs (output size).

3.2 Interval Intersections

As a warmup let us consider the corresponding problem in $\mathbb{R}^1$.

Problem 7 *Given a set I of n intervals $[l_i, r_i] \subset \mathbb{R}$, $1 \leq i \leq n$. Compute all pairs of intervals from that intersect.*

Theorem 3.1 *Problem 7 can be solved in $O(n \log n + k)$ time and $O(n)$ space, where k is the number of intersecting pairs from $\binom{I}{2}$.*

Proof. First observe that two real intervals intersect if and only if one contains the right endpoint of the other.

Sort the set $\{(l_i, 0) \mid 1 \leq i \leq n\} \cup \{(r_i, 1) \mid 1 \leq i \leq n\}$ in increasing lexicographic order and denote the resulting sequence by P. Store along with each point from P its origin (i). Walk through P from start to end while maintaining a list L of intervals that contain the current point $p \in P$.

Whenever $p = (l_i, 0)$, $1 \leq i \leq n$, insert i into L. Whenever $p = (r_i, 1)$, $1 \leq i \leq n$, remove i from L and then report for all $j \in L$ the pair $\{i, j\}$ as intersecting. □

3.3 Segment Intersections

How can we transfer the (optimal) algorithm for the corresponding problem in $\mathbb{R}^1$ to the plane? In $\mathbb{R}^1$ we moved a point from left to right and at any point resolved the situation

locally around this point. More precisely, at any point during the algorithm, we knew all intersections that are to the left of the current (moving) point. A point can be regarded a hyperplane in $\mathbb{R}^1$, and the corresponding object in $\mathbb{R}^2$ is a line.

General idea. Move a line ℓ (so-called *sweep line*) from left to right over the plane, such that at any point during this process all intersections to the left of ℓ have been reported.

Sweep line status. The list of intervals containing the current point corresponds to a list L of segments (sorted by y-coordinate) that intersect the current sweep line ℓ. This list L is called *sweep line status* (SLS). Considering the situation locally around L it is obvious that only segments that are adjacent in L can intersect each other. This observation allows to reduce the overall number of intersection tests, as we will see. In order to allow for efficient insertion and removal of segments, the SLS is usually implemented as a balanced binary search tree.

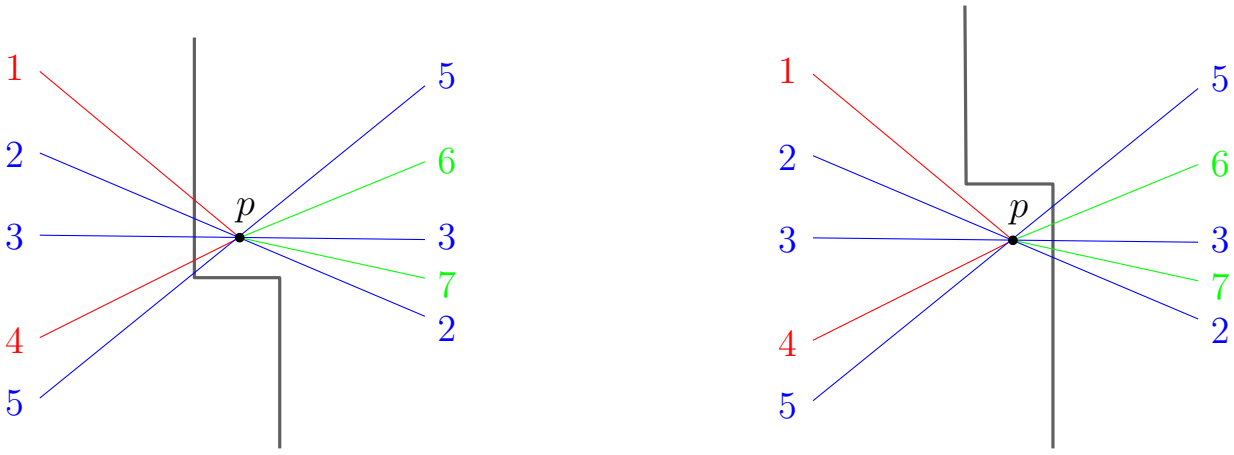
Event points. The order of segments in SLS can change at certain points only: whenever the sweep line moves over a segment endpoint or a point of intersection of two segments from S. Such a point is referred to as an *event points* (EP) of the sweep. Therefore we can reduce the conceptually continuous process of moving the sweep line over the plane by a discrete process that moves the line from EP to EP. This discretization allows for an efficient computation.

At any EP several events can happen simultaneously: several segments can start and/or end and at the same point a couple of other segments can intersect. In fact the sweep line does not even make a difference between any two event points that have the same x-coordinate. To properly resolve the order of processing, EPs are considered in lexicographic order and wherever several events happen at a single point, these are considered simultaneously as a single EP. In this light, the sweep line is actually not a line but an infinitesimal step function (see Figure 3.1).

Event point schedule. In contrast to the one-dimensional situation, in the plane not all EP are known in advance because the points of intersection are discovered during the algorithm only. In order to at any time be able to determine the next EP, we use a priority queue data structure, the so-called *event point schedule* (EPS).

Along with every EP p store a list $\text{end}(p)$ of all segments that end at p, a list $\text{begin}(p)$ of all segments that begin at p, and a list $\text{int}(p)$ of all segments in SLS that intersect at p a segment that is adjacent to it in SLS.

Along with every segment we store pointers to all its appearances in an $\text{int}(\cdot)$ list of some EP. As a segment appears in such a list only if it intersects one of its neighbors



(a) Before. (b) After.

Figure 3.1: *Handling an event point* p.

there, every segment needs to store at most two such pointers.

Invariants.

1. L is the sequence of segments from S that intersect ℓ, ordered by y-coordinate of their point of intersection.
2. E contains all endpoints of segments from S and all points where two segments that are adjacent in L intersect to the right of ℓ.
3. All pairs from $\binom{S}{2}$ that intersect to the left of ℓ have been reported.

Event point handling. An EP p is processed as follows.

1. If $\text{end}(p) \cup \text{int}(p) = \emptyset$, localize p in L.
2. Report all pairs of segments from $\text{end}(p) \cup \text{begin}(p) \cup \text{int}(p)$ as intersecting.
3. Remove all segments in $\text{end}(p)$ from L.
4. Reverse the subsequence in L that is formed by the segments from $\text{int}(p)$.
5. Insert segments from $\text{begin}(p)$ into L, sorted by slope.
6. Test the topmost and bottommost segment in SLS from $\text{begin}(p) \cup \text{int}(p)$ for intersection with its successor and predecessor, respectively, and update EP if necessary.

Updating EPS. Insert an EP p corresponding to an intersection of two segments s and t.

1. If p does not yet appear in E, insert it.
2. If s or t are contained in some $\text{int}(\cdot)$ list of some other EP q, remove them there and possibly remove q from E (if $\text{end}(q) \cup \text{begin}(q) \cup \text{int}(q) = \emptyset$).
3. Insert s and t into $\text{int}(p)$.

Sweep.

1. Insert all segment endpoints into $\text{begin}(\cdot)$ and $\text{end}(\cdot)$ lists of a corresponding EP in E.
2. As long as $E \neq \emptyset$, handle the first EP and then remove it from E.

Runtime analysis. Initialization: $O(n \log n)$. Processing of an EP p:

$$O(\#\text{intersecting pairs} + |\text{end}(p)| \log n + |\text{int}(p)| + |\text{begin}(p)| \log n + 2 \log n).$$

In total: $O(k + n \log n + k \log n) = O((n + k) \log n)$.

Space analysis. Clearly $|S| \leq n$. At begin we have $|E| \leq 2n$ and $|S| = 0$. Furthermore the number of additional EPs corresponding to points of intersection is always bounded by $2|S|$. Thus the space needed is $O(n)$.

Theorem 3.2 *Problem 4 and Problem 5 can be solved in* $O((n + k) \log n)$ *time and* $O(n)$ *space.* □

Theorem 3.3 *Problem 1, Problem 2 and Problem 3 can be solved in* $O(n \log n)$ *time and* $O(n)$ *time.* □

3.4 Improvements

The basic ingredients of the algorithm described above go back to Jon Bentley and Thomas Ottmann [?] from 1979.

Whereas Theorem 3.3 is obviously optimal, it is not clear why $O(n \log n + k)$ time should not be possible in Theorem 3.2. Indeed this was a prominent open problem in the 1980's that has been solved in several steps only by Bernard Chazelle and Herbert Edelsbrunner in 1988. The journal version of their paper [?] consists of 54 pages and the space usage is suboptimal $O(n + k)$.

Kenneth Clarkson and Peter Shor [?] (1989) and independently Ketan Mulmuley [?] (1988) described randomized algorithms with expected runtime $O(n \log n + k)$ using $O(n)$ and $O(n + k)$, respectively, space.

An optimal deterministic algorithm, with runtime $O(n \log n + k)$ and using $O(n)$ space, is known since 1995 only due to Ivan Balaban [**?**].

4. Red-Blue Intersections

Lecture on Thursday 2nd October, 2008 by Michael Hoffmann <hoffmann@inf.ethz.ch>

4.1 Red-Blue Intersections

Although the Bentley-Ottmann sweep appears to be rather simple, its implementation is not straightforward. For once, the original formulation did not take care of possible degeneracies—as we did in the preceding section. But also the algebraic degree of the predicates used is comparatively high. In particular, comparing two points of intersection lexicographically is a predicate of degree five, that is, in order to compute it, we need to evaluate a polynomial of degree five. When evaluating such a predicate with plain floating point arithmetic, one easily gets incorrect results due to limited precision roundoff errors. As we have seen for Jarvis' Wrap, such failures frequently render the whole computation useless. The line sweep algorithm is similarly problematic in this respect, as a failure to detect one single point of intersection often implies that no other intersection of the involved segments to the right is found.

In general, predicates of degree four are needed to construct the arrangement of line segments because one needs to determine the orientation of a triangle formed by three segments. Motivated by the Map overlay application we consider here a restricted case. In the *red-blue intersection* problem, the input consists of two sets R (red) and B (blue) of segments such that the segments in each set are interior-disjoint.

Definition 4.1 *As set S of line segments in $\mathbb{R}^d$ is* **interior-disjoint** $\iff$ *there are no two segments $s, t \in S$ for which $(s \setminus V(s)) \cap t \neq \emptyset$, where $V(s)$ denotes the set of (at most two) endpoints of s.*

In this case there are no triangles of intersecting segments, and it turns out that predicates of degree two suffice to construct the arrangement. This is optimal because already the intersection test for two segments is a predicate of degree two.

Predicates of degree 2. Restricting to degree two predicates has certain consequences. While it is possible to determine the position of a segment endpoint relative to a(nother) segment, one cannot, for example, compare a segment endpoint with a point of intersection lexicographically. Even computing the coordinates for a point of intersection is not possible. Therefore the output of intersection points is done implicitly, as "intersection of s and t".

Graphically speaking we can deform any segment as long as it remains monotone and it does not reach or cross any segment endpoint (it did not touch before). With help of degree two predicates there is no way to tell the difference.

Witnesses. Using such transformations the processing of intersecting points is deferred as long as possible (lazy computation). The last possible point $w(s, t)$ where an intersection

between two segments s and t has to be processed we call the *witness* of the intersection. The witness $w(s,t)$ is the lexicographically smallest segment endpoint that is located within the wedge formed by the two intersecting segments s and t to the right of the point of intersection (Figure 4.1).

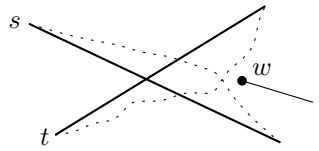


Figure 4.1: *The witness* $w(s,t)$ *of an intersection* $s \cap t$.

As a consequence only the segment endpoints are EPs and the EPS can be determined by lexicographic sorting during initialization. On the other hand, the SLS structure gets more complicated because its order of segments does not necessarily reflect the order in which the segments intersect the sweep line.

Invariants.

1. L is the sequence of segments from S intersecting ℓ; s appears before t in $L \Longrightarrow s$ intersects ℓ above t or s intersects t and the witness of this intersection is to the right of ℓ.
2. All intersections of segments from S whose witness is to the left of ℓ, have been reported.

SLS Data Structure. The SLS structure consist of three levels.

1. Collect adjacent segments of the same color in *bundles*, stored as balanced search trees. For each bundle store pointers to the topmost and bottommost segment. (As the segments within one bundle are interior-disjoint, their order remains static and thus correct under possible deformations due to lazy computation.)
2. All bundles are stored in a doubly linked list, sorted by y-coordinate.
3. All red bundles are stored in a balanced search tree (*bundle tree*).

The search tree structure should support insert, delete, split and merge in (amortized) logarithmic time each. For instance, splay trees [**?**] meet these requirements.

EP handling. An EP p is processed as follows.

1. Localize p in bundle tree $\rightarrow \leq 2$ bundles containing p (all red bundles in between must end at p). (For the localization we need the pointers to the topmost and bottommost segment within a bundle.)

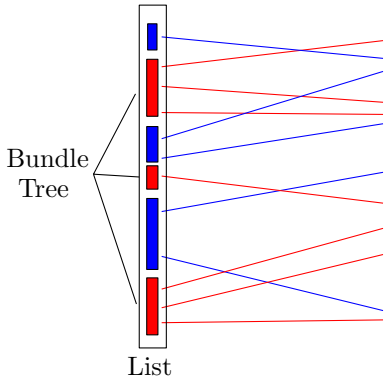


Figure 4.2: *Graphical representation of the SLS data structure.*

2. Localize p in ≤ 2 red bundles found and split them at p. All red bundles are now either above, ending, or below with respect to p.
3. Localize p within the blue bundles by linear search.
4. Localize p in the ≤ 2 blue bundles found and split them at p. All bundles are now either above, ending, or below with respect to p.
5. Run through the list of bundles around p. Handle all adjacent pairs of bundles (A, B) that are in wrong order and report all pairs of segments as intersecting. (Exchange A and B in the bundle list and merge them with their new neighbors.)
6. Report all pairs from $\text{begin}(p) \times \text{end}(p)$ as intersecting.
7. Remove ending bundles and insert starting segments, sorted by slope and bundled by color.

Remark: As both red and blue segments are interior-disjoint, at every EP there can be at most one segment that passes through the EP. Should this happen, for the purpose of EP processing split this segment into two, one ending and one starting at this EP. But make sure to not report an intersection between the two parts!

Analysis. Sorting the EPS: $O(n \log n)$ time. Every EP generates a constant number of tree searches and splits of $O(\log n)$ each. Every exchange in Step 5 generates at least one intersection. New bundles are created only by inserting a new segment or by one of the constant number of splits at an EP. Therefore $O(n)$ bundles are created in total. The total number of merge operations is $O(n)$, as every merge kills one bundle and $O(n)$ bundles are created overall. The linear search in Step 5 (beyond the ending bundles) can be charged to the subsequent merge operation. (Anyway, considering all bundles touched in that step at a single EP, notice that all but a constant number of them consists of ending segments.) In summary, we have a runtime of $O(n \log n + k)$ and space usage is linear obviously.

Theorem 4.2 *For two sets* R *and* B*, each consisting of interior-disjoint line segments in* $\mathbb{R}^2$*, one can find all intersecting pairs of segments in* $O(n \log n + k)$ *time and linear space, using predicates of maximum degree two. Here* $n = |R| + |B|$ *and* k *is the number of intersecting pairs.*

Remarks. The first optimal algorithm for the red-blue intersection problem was published in 1988 by Harry Mairson and Jorge Stolfi [**?**]. In 1994 Timothy Chan [**?**] described a trapezoidal-sweep algorithm that uses predicates of degree three only. The approach discussed above is due to Andrea Mantler and Jack Snoeyink [**?**] from 2000.

References