Chapter 1

Line Segment Intersection

(with material from [1], [3], and [5], pictures are missing)

1.1 Interval case

We first think of having n intervals (e.g., the x-range of horizontal line segments) and want to know whether any two of them intersect. Intersection means to have a common endpoint or to overlap. It is obvious that this can be done in time $O(n^2)$ by inspecting all pairs of intervals $I_a = [l_a, r_a]$, $I_b = [l_b, r_b]$ being disjoint, that is, whether $l_r < l_b$ and $r_a < l_b$.[1]

A much better solution is possible, if we first sort the all endpoints and mark the $2n$ points a left or right endpoint of an interval. In sorting we already detect intersections at endpoints only. If no endpoints match, we eventually have a sequence of characters 'L' and 'R'. The intervals are disjoint iff the characters appear in alternating order in the sequence. Sorting requires $O(n \log n)$ time, the alternating check is $O(n)$, so in total we need time $O(n \log n)$ to decide the problem.

1.2 Line segments

Let us introduce another motivation: Given two maps, e.g., for streets and rivers, compute all the points were bridges are required. We assume for simplicity that the streets and rivers are approximated by (connected) line segments (polygonal chains). Actually, in this setting we are interested whether the chains intersect in their interior, that is, to decide whether the line segments are considered as open or closed. Usually segments are interpreted as closed, that is, they can also intersect at their endpoints, or if an endpoint lies on another segment.

[1] Assuming the real-RAM, that is, ignoring bit-lengths of actual representations of interval boundaries.

In some sense, the distinction into two sets is not needed. One could consider a set of labelled (colored) segments. To simplify our discussion, we assume first that we only consider a single set of segments, but later, when discussing overlays of arrangements and boolean operations in more detail. So, we next discuss efficient solutions for the following problems:

Problem 1. *Given n line segments in the plane $S := \{s_1, \ldots, s_n\}$, determine whether they intersect.*

Problem 2. *Compute the intersections of n given line segments in the plane.*

A very naively approach would be: Check for any pair of segments (s_i, s_j), $1 \leq i, j \leq n$, $i \neq j$ whether $s_i \cap s_j \neq \emptyset$. And if so, report the single point (or the overlapping part) if also the constructive problem is to be solved. This algorithm has running time $O(n^2)$ — which is (even) optimal if each pair has an intersection, that is, the number of intersection is $\Omega(n^2)$. Though this situation is possible in real situations, it is not very likely for most cases.

Usually, practical applications have a number of intersections that is less than quadratic. A segment intersects only a few (neighboring) segments, if at all. It would be nice to have an algorithm that takes this into account, an algorithm that is output-sensitive. That is, the running time depends on the number of actual intersections. The less intersections exists, the less time is spent (mainly in geometric operations, as we will learn later). Note that it also means, that if the segments do not intersect at all, no time is spent to try to construct intersections.

A crucial observation for line segments is that they are more likely to intersect if they are close together. We next see an algorithm that benefits from the special geometry. But what is the intuition? We want to check which segments are far apart? However, obtaining a full order of segment distances requires $O(n^2 \log n^2)$ time, which is even worse than the previous naive approach. So pairwise-distances do not lead to improvements. A interesting idea is to reduce the dimension, and to only check for segments intersecting, if they already share a common x-range.[2] We have already seen that it takes $O(n \log n)$ time to decide whether x-ranges intersect. An extension to really compute the intersections is not complicated. The crucial observation is that after determining the x-range intersection, we only have to check segments for a real intersection for which already the x-ranges intersect — and even more: We only have to search in the intersected x-range of two segments.

1.3 Sweep line algorithm

Let us try formalize this idea more algorithmically. Suppose we sweep with a line ℓ from left to right over all segments. We assume that we start to the left

[2] Considering the y-range instead leads to a symmetric discussion of what follows.

of the leftmost endpoint of a segment, and end to the right of each segment's right endpoint. While we move the line, we always store which segments are currently intersecting the line. This allows us to find pairs of segments that possibly intersect, that is, that we can check for a real intersection. The line ℓ is called the sweep line, and so we call the algorithm. The careful reader already observes that the move is not needed to be continuous, as only at a some points structural changes happen. What do we mean with a structural change? In this example, the set of segments intersected by ℓ changes. Either we add a segment s_i, namely, when the sweep line moves over s_i's left endpoint, or we remove s_i from this list, namely when ℓ moves over s_i's right endpoint. We call these (x-)positions *events*.

The first crucial observation for this kind of algorithm is, that instead of really moving the line continuously, we maintain a sorted set of events and process it in increasing order. In this situation, we initialize the set with a all segment's endpoints. This requires to sort them (time $O(n \log n)$). Sweeping reduces to pick the next event, and to update the maintenance structure. Here the set of segments that intersect ℓ. Adding a segment, if the event is a left endpoint, deleting a segment if the event is a right segment. After changing the structure, we check the segments in the set for possible intersections. This way we ensure to only check segments that have a common x-range. There is already a subtlety here: Namely to check whether there are intersections at the events itself. That is, whether two segments only intersect at their endpoints, or whether the endpoint of a segment lies in the interior of another segment. But these degenerate cases are solvable.

Exercise 3. *Sketch modifications of the algorithm, how to handle the degenerate cases.*

The major problem with this algorithm is that it is not yet output-sensitive. There are still configurations, where a quadratic number of segments must be checked for intersection, namely: All share a common x-range. However, if they are all disjoint (e.g., parallel), not even a single intersection will be found.

So far we only considered proximity in x-direction. How about introducing proximity in y-direction as well? More precisely, let us order the segments intersecting the sweep line from bottom to top. It is easy to see, that at some position of the sweep line, only segments that have the same y-coordinate intersect at this x-coordinate. But there is more to see: Before a segment intersects with a non-neighbor, it either has to cross a neighbor (which means to intersect with the neighbor), or the "blocking" neighbor has to vanish. Vanishing means that the blocking segments ends, while the sweep line has moved on to the right. To summarize: The idea is to only check for intersection of segments that are currently neighbors in the sweep line. This leads to three important insights/changes:

First, the structure is now a sorted list (compared to an unsorted list before). That is, in order to correctly maintain structural changes, we have to reconsider our events. Beyond the case that a segment starts or ends, we now also have to consider the case when the order of segments already intersecting the sweep line changes. And this exactly happens when two segments intersect. If they are sorted $s_1 < s_2$ to the left of the intersection, their sorting is $s_2 < s_1$ to the right of the intersection.[3] Following, we introduce another *event* namely one for an intersection of segments.

The second insight is to detect cases, when segments become adjacent (neighbored) in the sorted sequence. There are three: A new segment is added, and we have to check for its intersection with its possible two neighbors above and below. That is, we check up to two pairs of segments for an intersection. A segment is deleted, and the possible neighbor below (if exists) can now intersect the neighbor above (if exists, too). Here we only check at most one pair of segments for an intersection. And the third case is that two segments intersected in a point. Following the neighboring segment above the two (if existing) now has a new neighbor below, and similar for the neighboring segment below the two (if existing) now also has a new neighbor. In this case, we again check up to two pairs for intersections. Note that each segment that is updated gets at most two new neighbors.

The third change is that now the event queue is not static after its initializing. Whenever we detect an intersection of two segments (see previous case distinction) we add it to the event queue as an intersection event. That is, the sweep now also stops at those events and processes them (as just described). Or more general: When processing an event the following happens. We first update the sorting, and then check for intersection. It is sufficient to check only for intersection to the right of the sweep line, as intersections to the left are already found and processed (and reported/stored).

The sweep line stops when the last event (usually the rightmost endpoint of a segment) is processed. Then no more segments can start to the right of this position, and we should see sweep line with an empty list of segments intersecting it (as at the very beginning of the algorithm). For the entire run we can guarantee the following invariant: All intersection points to the left of the sweep line have correctly been computed.

We next show that we really reduced the number of pairs to be checked for intersection, and that we still compute all intersections. The proof idea is to show that for any pair of segments s_i, s_j there is a position of the sweep line where the two segments are neighboring, such that we check them for their intersection (to be inserted in the queue). We assume some general position, in particular, segments should not overlap, not being vertical, and no three share a common point. Later we will "easily" remove these assumptions. The

[3] This follows from the segments being linear, and none of them to be considered vertical so far.

special case where an intersection takes place at a segment's endpoint is easy to detect when the sweep line hits the corresponding event. So we are left with proper intersections in the segments' interior.

Theorem 4. *Let s_i, s_j be non-vertical segments that intersect in a single point p and no third segments passes through p. Then, there is an event E to the left of p (i.e., $x(E) < p(x)$) where s_i and s_j become adjacent and are tested for intersections.*

Proof. Immediately to the left of p the two segments are adjacent in ℓ. That is, there is a position $x = x_0$ for ℓ. Consider the event smaller than x_0. This is the event we are searching for. This event must exist, as initially the two segments are not adjacent as the sweep starts to the left of all segments and the sweep line is empty. More: The event E's x-coordinate is no smaller than the larger of the leftmost x-coordinates of the two segments. □

1.3.1 Algorithm and data structure

We next discuss the algorithm more in-depth and also try to discuss the degenerate cases. First of all, it is questionable what to report as output. We will see that the sweep line approach is a rather generic algorithm that can produce different kind of output by just looking at how the algorithms progresses. For example, whenever an intersection is found, we can simply report it. But we can also postpone it and wait until we process the corresponding event, which then allows us to report which segments start, end, and intersect at that point. Reporting overlapping segments is also required to be complete.

For the event queue that is required by the algorithm we need a data structure that maintains events and supports the following operations: Insertions of a new event (which includes to check whether an event already exists), update of an event (in case, an event refers, e.g., to a starting point of a segment, and is also an intersection of two other segments – more cases are easy to derive) and to remove the smallest event – with respect to the chosen sorting. We somehow assumed that we sort events with increasing x-coordinate. But if two events share their x-coordinate but have different y-coordinate, we should still handle them separately, that is, to consider them as two events. For that, the sorting of events is achieved by a lexicographical order. Comparison is first done by x-coordinate, and only if equal, we compare y-coordinates. Note that this way, we do not talk about a sweep "line" anymore. Its more a sweep "point". But there are two common views: Either we consider the sweep line as slightly tilt to the right, or the sweep line has a "step" of infinitesimal width at the sweep point. A good candidate is a balanced search tree reflecting the lexicographic order of events. For each event maintained in it, we store which segments start $R(E)$ (are to the 'right' of E), which segments end $L(E)$ (are to the 'left' of E), and which segments

continue through $C(E)$. All operations require time $O(\log m)$ where m is the number of events currently in the tree.

For the status structure we need a sorted sequence of segments currently intersecting the sweep line. Note that the structure dynamically changes, whenever we process an event. But also notice that we only change a certain subset of the segments maintained. That is, it is not required to compute the full order of the k segments currently maintained using a sorting by y-coordinates. This would need $O(k \log k)$ time. Again we can use a balanced search tree which perfectly supports dynamic additions, deletions and to relocate subtrees. This is possible as for the current position we always have a well-defined order of the segments.

Exercise 5. *Describe how to maintain the segments in the leaves of the search tree, and to use to interior nodes as guides for queries, such that to easily locate the segment that is below a given point "on the sweep line" (not necessarily an intersection of the sweep line with a segment in the tree).*

It is clear that the operations requires time as most $O(\log n)$.

We next see the overview of the algorithm that gets a set of segments, and returns a list of intersection points together with segments that meet in that point.

Algorithm 1 Sweep line algorithm

Initialize an empty event queue $\mathcal{Q}$, and add all segments' leftmost and rightmost points as starting and ending events (note that events know which segments start or end). We see at most $2n$ events, but it can also be less.
Initialize an empty status structure $\mathcal{T}$
while $\mathcal{Q}$ is not empty **do**
  determine lexicographically smallest event E in $\mathcal{Q}$ and delete it from $\mathcal{Q}$

  Process(E)
end while

The subfunction Process(E) is basically doing what we described before. If $k_E = |R(E) \cup C(E) \cup L(E)| > 1$, we report the event as intersection together with that set of segments. In addition, it checks newly adjacent segments for "future" intersection and to updates $\mathcal{Q}$ accordingly. But this step first requires to correctly maintain $\mathcal{T}$. That is, to remove all segments $L(E) \cup C(E)$ from it and to insert at that position all segments $C(E) \cup R(E)$. We have to ensure that the order of inserted segments in $\mathcal{T}$ is identical to the order of those segments immediately to the right of E (if one of them is vertical, it will be inserted as last element). The order can be ensured by sorting the k_E segments in time $O(k_E \log k_E)$. But we can do better: If we first remove $L(E)$ from $\mathcal{T}$, we next simply reverse the order of segments

$C(E)$ in $\mathcal{T}$, which only requires time $O(k_C)$ with $k_C = |C(E)|$. It remains to insert the missing segments $R(E)$ to the now modified tree which can be done in time $O(k_R \log(k_C + k_R)$, with $k_R = |R(E)|$.

Below we discuss which geometric operations are required for the sweep line algorithm. The correctness of the algorithms follows from the fact that endpoints are already inserted at the beginning, and that we can sort the segments passing through a common vertex by angle and then apply Theorem 4 to neighboring segments. We next proof the running time and see that we have reached an output-sensitive algorithm:

Theorem 6. *The sweep line algorithm runs in time $O(n \log n + k \log n)$ where k is the number of intersection points of segments.*

Proof. The initializing of $\mathcal{Q}$ with start and end events takes time $O(n \log n)$. Initializing of the status structure is constant. An event is deleted only once, and for each deletion we can have up to 2 insertions. Each such operation requires $O(\log n)$ time. In addition each operation on $\mathcal{T}$ can be executed in time $O(\log n)$. For a single event, the number of such operations is bounded by k_E. Let $m = \sum_{\text{all events } E} k_E$. That is, we have in total $O(m \log n)$ time for operations on $\mathcal{T}$. We can bound $m = O(n + l)$, where l is the size of the output. But, whenever $k_E > 1$, all segments involved in E are reported, and events that have at most one segment are events where this segments starts or end. That is, it remains to show that $m = O(n + k)$. This is done by interpreting S as a planar graph embedded in the plane (we will do so in the next lecture anyhow). Vertices are given by end- and intersection points, segments are defined by subsegments connecting vertices. Any event E is a vertex of the graph, and k_E denotes its degree. Consequently m is bounded by the sum of all degrees of the graph. Every edge can contribute one at most twice. Namely to the vertices of the graph. That is $m \leq 2n_e$. We only have to bound n_e in terms of n and k now. We know that n_v, the number of vertices, is bounded by $2n + k$, but we also know for planar graphs that $n_e = O(n_v)$. And we are done. In fact we can even bound the number of faces n_f by $2n_e/3$. Then using Euler's formula which states $n_v - n_e + n_f \geq 2$, we have

$$2 \leq (2n + k) - n_e + 2n_e/3 = 2n + k - n_e/3$$

. Following, $n_e \leq 6n + 3k - 6$ and $m \leq 12n + 6k - 12$. □

What is the amount of storage required? Obviously, the status structure holds every segment at most once. So we have $O(n)$ space for it. However, the size of the event queue can be larger, as we also insert intersection points. In worst case, we will have up to $O(n + k)$ events, where k is again the number of reported intersection points. Is it possible to reduce working space? In fact, there is an easy way to do so. We only have to ensure that no intersection of two segments is stored if the two segments are not

adjacent in the status structure. Obviously, we will only have linearly many intersection events in the queue. The algorithm must be adapted to now also delete events when two segments stop being adjacent. It is clear that the two segments will become adjacent again, before the will finally intersect (if at all). Though the overall running time is not harmed by this modification, it is questionable of whether to really do so. If one is interested in the arrangement (see Lecture **??**), its output complexity is $O(n+k)$ anyhow. If the output has less complexity, the change might be useful.

Note that the algorithm can also be used for checking whether there is an intersection, that is, we could stop after the first intersection is found. Really reporting the intersection is not required.

Theorem 7. *Given n line segments, the decision of whether they intersect can be done in $O(n \log n)$ time, and this is optimal.*

Corollary 8. *As a consequence, the following problems can also be solved in time $O(n \log n)$:*

- *Is a given polygon simple?*
- *Do two given simple polygons (of size n) intersect?*
- *Does a chosen straight-line embedding of a planar graph contain any crossing edges?*

In the exercise we ask you to analyze the a sweep line algorithm to compute the intersections of n circles. It is clear that a slightly modified algorithm suffices to decide whether any two of n circles intersect in time $O(n \log n)$.

1.3.2 Geometric operations required for the sweep line algorithm

We next analyze more detailed which geometric operations are required by the sweep line algorithm. We discuss them for segments, but they generalize for other one-dimensional (and x-monotone) planar curves, too.

This operation is obvious:

Intersection of two segments Given two segments, report their intersecting set, which is either empty, or consists of a single point, or is an overlapping part of the two segments. In general, we must be able to report a set of isolated intersection points (possibly empty) and a set of overlapping parts (possibly empty).

The next operation is required to maintain the event queue:

Lexicographic comparison of points Given two points, decide whether one is lexicographically smaller or larger than the other, or whether they are equal.

Remember that the status structure is a sorted list of curves/segments currently intersecting the sweep line. For its maintenance we need the following geometric predicates:

Point position Given an x-monotone curve and a point in the curve's x-range. Decide whether the point is below the segment, above it, or lies on it. Below and above are with respect to the y-coordinates of the point and the curve at the point's x-coordinate. The predicate is used to find the position of a curve that starts at the position of the sweep line in the status structure (i.e., among the segments already contained in the structure).

Compare to right Given two segments/curves intersecting in a common point and not overlapping. Decide which of the two curves is smaller (in y-direction) than the other just to the right of the intersection point. This predicate is also used to insert a starting curve into the status structure. Namely, when the previous predicate reported 'on a curve' for the starting curve's minimal point. Then it did not succeed to find the right position of the curve. Remember that we aim to eventually obtain a status structure that reflects the vertical order of segments/curves immediately to the right of the event currently being process.

In case we consider non-linear curves (see Lecture **??**), we also need a way to split curves into (weakly) x-monotone subpieces:

Make x-monotone Splits a given non-x-monotone curve into a set of (weakly) x-monotone curves (and maybe isolated points).

Exercise 9. *Describe how to implement the predicates/operations for bounded arcs of rational functions.*

Chapter 2

Arrangements

When thinking about maps, they are often labeled. That is, some regions contain names such as streets, places of interest, or areas of cities. We next aim to represent such a decomposition in a data structure. Just collecting the lists of curves bounding them is often no sufficient for quickly reporting, for instance, the boundary of a city. That is why we aim for a more combinatorial and topological description: Which curves bound a region, which curves emanate from a point. More precisely: To transform the continuous problem into a set of finite objects.

In computational geometry, this is supported by arrangements, that we present in this lecture.

Definition 10. *Given a finite collection S of geometric objects such as hyperplanes or spheres in $\mathbb{R}^d$, the* arrangement $\mathcal{A}(S)$ *is the decomposition of $\mathbb{R}^d$ into maximal connected open cells of dimensions $0, 1, \ldots, d-1$ induced by S.*

Arrangements are interesting for their own, but can also serve as unifying structure. We here focus on two-dimensional arrangements of segments, lines, and later algebraic curves. Our actual assumption is that curves are (weakly) x-monotone Jordan arcs. It simplifies computations and their analyses.

The 0-, 1-, and 2-dimensional cells of a planar arrangement are called *vertices, edges, and faces.* An arrangement is *simple* if every two curves meet in a single point and any three curves do not have a point in common.[1]

Example 11 (Arrangement of lines)**.** *Given a set of lines L inducing a simple arrangement $\mathcal{A}(L)$. Then, a vertex of the arrangement is unique specified by a pair of lines; an edge is the maximal connected portion of a line that is not intersected by any other line. Finally, a face is a maximal connected 2-dimensional region that is not intersected by any line. An arrangement*

[1] We omit d-dimensional definitions.

of n lines has up to $O(n^2)$ cells, where equality holds if the arrangement is simple (as assumed).

In an arrangement of bounded curves, vertices are also modelled by endpoints of curves. In addition to curves, we can also assume the existence of isolated points (that will form vertices on their own, if not covered by a curve). Often we assume general position, that consists of conditions as: Intersections are interior to input curves, or no three curves intersect in a single point. However, we try to avoid such assumptions. An arrangement of n Jordan arcs has up to $O(n^2)$ cells.

Arrangements also induce various interesting substructures. Some of them already suffice to solve particular problems, which makes constructing and querying an entire arrangement obsolete. As the complexity is usually smaller, this has positive implications on an algorithm's performance. We only sketch main structures:

Lower envelope We interpret each curve c_i as a graph of a continuous function $c_i(x)$. The lower envelope is the pointwise minimum of these functions. Similarly, we can define the upper envelope (or any k-level).

Minimization diagram This structure is related to the lower envelope. It is the decomposition of the x-axis into intervals so that on each interval the same subset of functions attains the minimum.

Zone Consider another curve γ, its zone consists of all faces of a given arrangement that are intersected by γ.

Single cell A single cell of an arrangement. Often specified by a point contained in the cell.

For more details and an overview of complexities of substructures we refer the reader to [3, Chapter 24].

Beyond substructures there are decompositions. Resulting cells usually have finite complexity, for instance, the one of a disk. It offers simplification, and allows to analyze algorithms and the arrangement itself. A famous one is the vertical decomposition where each vertex is extended in upward and downward direction. The extension is a vertical segment if it next hits another curve, and a vertical ray, if it extends to infinity.

Exercise 12. *Show that $O(n)$ is the complexity of a vertical decomposition of a planar arrangement of complexity n.*

2.1 Representations

We next present possible representations for an arrangement.

Incidence graph This graph $G = (V, E)$ has a node for each cell of the arrangement, and two nodes are linked if the two cells are incident to each other.

Skeleton A *skeleton* is a connected subset of the edges and vertices of the arrangement. The *complete skeleton* is the union of all vertices and edges.

Cell-tuple structure The incidence graph misses to capture order information between cells. For example, edges bounding a face can be naturally ordered (e.g., counterclockwisely). The famous two-dimensional cell-tuple structure is the *doubly-connected-edge-list*, DCEL for short. It provides simple and unified access to incidence and ordering information in the arrangement. A DCEL contains a record for each face, each edge, and each vertex of the arrangement. A record can store an "attribute", that is, some information in addition to the geometric and topological data that are used to provide easy access to operations as:

- Which are the edges bounding a face?
- To which vertex is an edge connected?
- What are inner components of a face?
- What are neighboring faces of a vertex or an edge?
- What is the geometric point of a vertex?
- What is the curve supporting an edge?

These operations are enabled by adding pointers between the records, or to geometric objects. Note that faces usually do not have their own geometric object, as they are implicitly given by the surrounding curves (and maybe excluded isolated points). Edges are the central objects of a DCEL. Each edge stores a pointer to a geometric curve that it represent (as a vertex stores a pointer to a geometric point). Actually for one edge we store a pair of *twin halfedges* in opposite direction. Each halfedge then has a pointer to the face that is to the left of the halfedge when travelling the halfedge in its orientation. Due to the orientation we can also define source and target of a halfedge, but we only have to store one, as the other can be accessed by the halfedge's twin. In order to travel along the cycle of halfedges surrounding a face we have to store a pointer to the next halfedge in that order. To get from a face to such a cycle we store a pointer from the face to an arbitrary halfedge in the cycle. Note that there can be more than one cycle for a face: Usually one is oriented counter-clockwisely (the outer cycle), while there can be an arbitrary number of clockwisely oriented cycles

(separating inner components). Isolated vertices excluded in a face need additional pointers inside a face. Further note that a DCEL is not needed to be connected. One might think of introducing "dummy" edges to avoid inner components. This can even be done without sacrificing the DCEL's complexity. With the next and twin pointers, we obtain a way to visit all edges incident to a vertex, that is, jumping from an edge to the next in circular order is given by twin(next(h)).

To summarize,

a vertex stores a constant number of information: Pointer to geometric point and a pointer to an arbitrary incident edge (we just told how to visit all its incident edges starting from any)

an halfedge also stores a constant number of information: A pointer to a geometric curve that is represented by the halfedge. A pointer to its twin halfedge, a pointer to its target vertex, and a pointer to the face incident to the left of the halfedge. Finally a pointer that gives the next halfedge along the boundary of the incident face. Note that this will form a cyclic list that is either an outer edge cycle of the face, or an inner edge cycle of the incident face.

a face stores linearly many information — linear in the number of inner components. We have a pointer to one outer edge cycle, which is 'nil' in case of the single unbounded face. For each inner edge cycle there is another pointer. Similarly there is a pointer to every isolated vertex excluded from the face.

However, we can see that the total amount of storage is linear in the number of cells, that is, the complexity of the subdivision, as each inner component is only referenced once. For more details we strongly encourage to read [2, Chapter 1] and [1, Chapter 2.2]

2.2 Algorithms

In this section we answer the question: How to get the DCEL from a given set of curves? We will discuss three approaches.

Naive The first is simple, naive and direct. Decompose the set of curves C into (weakly) x-monotone subpieces. Call the set C'. We do so, as this makes a vertical decomposition and other algorithms easier to handle. We next compute all intersections of curves in C' and decompose the curves further such that they are interior disjoint from any intersection point (endpoints of overlapping parts can also serve for this purpose). Let B the resulting set. Now we can construct the DCEL for the set B. For each edge we may store pointers to the original curves in C, in case of overlaps there is more than one curve to be linked.

Sweep line We next discuss how to extend the sweep line algorithm to construct a DCEL, that is, we have to create vertices, edges connecting them, and create faces for which we have to find edges surrounding them. Vertices are trivial to create, namely whenever we see an event, we create a vertex. To create edges, we have to know the two vertices. For that we store with each curve maintained in the status structure a pointer to the vertex from which the curve emanates. Once we reach an event where a curve is involved (be it ending or passing), we create an edge from the stored source vertex to the newly created vertex of the event currently processed. It is questionable what geometric information to store with the edge: Two options are common. Either we store a link to the original curve, although the edge only represents some part of that curve (i.e., if a curve is intersected by another, we see at least two edges, but both would be linked to the original curve). Or, we split the curve at vertices, and only store with an edge the split part, that is, the curve stored in an edge directly represents what is meant with the edge (the part between two geometric points). The latter requires a split operation on an edge (and a merge if we later allow to remove vertices of an arrangement). However, the latter has advantages when processing the arrangement further, for example, when we compute overlays below, or when we extract a single cell. The order of edges around the vertices is given by the concatenated order of curves incident to an event. Remember that each event maintains a list of curves ending, passing, and starting. In worst case, we would require m operations to correctly find the cyclic order (where m is the number of curves in an event). Note that this does not harm the overall running time of the sweep. It remains to care about the DCEL's faces. Assume first, that we correctly created all vertices and edges with a sweep as above. Correctly link by the next pointers. That is, we have all edges (pairs of twins) available, that can form the outer and inner cycles of edges surrounding the faces. And we know that a face is always to the left of such a cycles (the cycles are counterclockwisely and clockwisely oriented). That is, we only have to collect the cycles, to determine their orientation, and to assign the cycles to faces and vice versa. To find the cycles is simple: Pick an edge, follow its next pointer until the first edge is seen again. This closes a cycle. Do so, until all edges are seen. Next we have to determine the cycle's orientation. This can be done as seen in Lecture **??** for the orientation of the polygon. Search for the lexicographical minimal vertex (point) and check the angle formed by the edge towards the vertex and the next edge (using the stored curves). If it is smaller than 180 degrees, then the cycle is outer. If it is larger, the cycle is inner. The test can also be done by comparing the vertical order (in y-direction) of the curves right of the lexicographically minimal point. We finally have to assign cycles to faces and vice versa. For that we create a graph, where each cycle forms a node (there is also a node for the imaginary outer cycle of the unbounded face). There is an arc between two nodes (cycles) if and only if one of the cycles

is a boundary of a hole (an inner one) and the other has an edge (vertex) immediately below the lexicographical minimal point of the first (i.e., inner) cycle. Note that each inner cycle must be connected to an outer. It can be transitive via other inner cycles.

Lemma 13 (Lemma 2.5 in [1])**.** *Each connected component of the graph corresponds to exactly the set of cycles incident to one face.*

Proof. Fix one cycle C of a face f. The face has to lie locally below the lexicographical minimal point of C. Thus, the other cycle to which it is linked must belong to the same face (there is no other face in between).

It remains to show that every inner cycle (bounding hole) is eventually linked to the outer cycle of the face. Suppose there is a cycle for which it is not the case and let C be the bottom most, which also has a lexicographically minimal point. However, by definition, there is another cycle C' to which C is connected. C' is below. Hence, C' is in the same connected component as C, which is not the connected component of f's outer cycle. This contradicts the definition of C. □

How can we construct the graph? We enhance the sweep by detecting the lexicographical minimal points of cycles. This happens when a curve starts in an event where no ending or passing curves exist (observe that cycles maybe one-dimensional). Note that in addition the status line gives an order of curves (and edges) below the current event (and thus below a lexicographically minimal point). The edge below is either another inner cycle, or an outer cycle of the face. Note that the orientation of a cycle (or even is stub only) is already determined when having processed the lexicographically minimal point — which is luckily ensured by the sweep order.

(Random) Incremental Suppose we are given an arrangement $\mathcal{A}$ and we are aiming to insert another curve γ into it. We can assume that γ is x-monotone, as all curves in the arrangement. If the arrangement is empty, the insertion is trivial. If it already consists of curves we need to do more:

1. Locate γ's minimal point in the arrangement (we discuss point location strategies below). This gives a cell.
   - If it is a vertex, then γ starts in it, and we have to update the DCEL accordingly, namely to insert a new edge in the right position of the cyclic lists of edges around the vertex.
   - If it is an edge, we have to split the edge. The new vertex has three incident edges then.
   - If it is a face, we create a new vertex of degree one contained in the face.
2. Next we traverse the zone of γ. That is, we compute which faces are crossed by the curve until we reach the curve's endpoint (see next step). In order to get from one face to the next, we have to check which of

the current face's edges γ intersects. This implies that whenever this happens we have to split an edge and we also have to split a face if it is totally crossed by γ.

3. γ's endpoint can be within a face, but also lie on an edge or an existing vertex. We proceed analogously to the starting point.

A special case is given when the entire γ lies within one face.

We remark that the order of insertion for n curves influences the (expected) running time (also in constants). A usual trick is to first compute a random permutation and then to insert the curves according to this permutation. However, the worst case running time for n line segments is still $O(n^2)$, which is optimal if all curves pairwisely intersect.

As mentioned, the algorithm relies on the possibility to perform exact point location for a curve's starting point in an existing arrangement. We next analyze this task in more detail.

Exercise 14. *Design a sweep-like algorithm to insert a set of curves into an existing arrangement.*

2.3 Point Location

In this section we are facing the problem of locating a point in a given arrangement (subdivision). That is, determine which cell (vertex, edge, face) contains a specified query point.

Naive A very naive approach is to just iterate over all cells. We start with the vertices and check whether one of the stored points is equal to the given point. This can be done by lexicographical comparison. We next check all edges whether they contain the point, which can be done with the point position predicate as described above. Note that we only check the interior of edges, as vertices are already checked. If not successful we similarly check the interior of faces. If no holes exists, this can be done by only checking the edges of the outer cycle of the face. For instance, whether they all have the point "to their left". Or to determine the two edges that contain the point's x-coordinate in their x-range and then to query the point position predicate for the two. In case that holes exists, we have to exclude that the point is actually inside a hole. For that we can either modify the previous tests to also consider edges of inner cycles. Or we descend to holes and ensure that the query point is not contained in any face contained in a hole (note that there can be more than one face contained in a hole) to finally decide whether the query point is inside the current query face. There are possibilities to tune this naive search.

Walk along a (vertical) line The previous exhaustive search can be improved by only searching cells that are met when walking in reverse order

along an "imaginary" ray starting in the query point. That is, we compute a special zone (see below) starting in the unbounded face, until we eventually meet the query point. On the way we may cross vertices, edges, and other faces. Once we reach the query point we exactly know which cell contains it.

Using landmarks We maintain a list of landmarks, that is, points for which we know their exact location in the subdivision. Given a new query point, we first perform some nearest-neighbor search (using a proper data structure, e.g., a Kd-tree) to find the nearest landmark and then perform "walk along a segment" from the landmark to the query point.

Vertical decomposition We decompose the arrangement further by extending vertices vertically; see construction above. We get a set of pseudo-trapezoids. We have to link those together which requires in total linear space. For the search we require a search structure which is formed by a directed acyclic graph (DAG) (of expected linear size), if the constructions are random incremental. More details can be found in [1, Chapter 6].

While the first two do not need additional data structures, the other two need. Those must be updated whenever the arrangement structurally changes. For that reason, one often introduces observers that get notified upon changes, that is, when a vertex, edge, or face is created (or deleted), or when a face is split, and more cases. This way the external entity can maintain and update auxiliary data based on local changes in the arrangement.

Exercise 15. *Design an efficient algorithm that computes the locations of a set of points in an arrangement using the sweep line paradigm.*

2.4 Overlay and Boolean Set Operations

We next face the problem that we are given two arrangements, and we want to compute their overlay, for example to match agricultural fields with a rain map. The goal is to subdivide each arrangement with respect to the other such that we finally know for each plant (on a field) how much water it received. That is, we want to label each face (to be more precise: each cell) with a pointer to the cells in the two original arrangements. A crucial observation is that a cell of one arrangement is only changed, if its intersection with a non-face of the other arrangement is non-empty. For that we assume that we are given a red-colored and a blue-colored arrangement. A red face is only split (or modified), if there is a blue edge running in the face, or it contains a blue vertex. And a red edge is only split, if there is a blue vertex or blue edge intersecting the red edge (there is the special case of overlapping edges, that we ignore here. Its handling is possible, though tedious). We

again use the sweep line paradigm to compute the overlay. Here, we rely on the fact that sweeping the curves in the red (or the blue) arrangements only, do not result in new intersection points. That is, no new event will ever be added to the initial list of events. However, if we sweep the red and blue arrangement in parallel, we add events, namely those where a red curve and a blue curve intersects. To obtain the resulting DCEL we simply do exactly the same as for a single set of curves (ignoring the colors at first hand). Note that this even allows to use the same tricks to determine the faces and their cycles.

It only remains to consider the colors, namely for the labelling of the newly created DCEL records. Remember that the recent sweep newly creates vertices, edges, and faces that will eventually be linked together and form the DCEL of the overlay. Each such record should be enhanced by two pointers: One into a red cell and one into a blue cell. However, this can be done when creating the records, using the fact that we sweep in both arrangements in parallel, and exactly know in which cell of each arrangement we are (or at least to which cells the current sweep event is incident). For example, think of an event that is formed by a red vertex only. It can match with a blue vertex, so we immediately have the originating cells. Or it can lie on a blue edge, which is determined by a geometric predicate. And we again know the cells. Or it can fall into a blue face, which is known from the last blue event that we encountered. Note that the 'active' face of one arrangement only changes at events of that arrangement. Similar considerations are possible for the red edges and faces, and for all blue records. In total we have ten cases: vertex-vertex = vertex, vertex-edge or edge-vertex = vertex, vertex-face or face-vertex = point, edge-edge = vertex, edge-edge = edge, edge-face or face-edge = edge, and finally face-face = face.

Boolean operations The map overlay allows to perform Boolean set operations. Think of each cell of an arrangement labelled with a Boolean flag as attribute. It is true if the cell (actually the points of the cell) is contained in the set, and false otherwise. Given two such sets, we want to perform operations as union, intersection, or difference. Observe that even though the input is a polygonally bounded set, the output can be more complicated. Just think of isolated points removed from a set. To compute such an operation, say intersection, we eventually traverse the overlay and report all cells whose originating cell both carry a true flag. To get the union, we report all cells where at least one of the originating cells carries a true flag. And similar for other Boolean operations.

This approach leads to unregularized Boolean set operations. If you want to have regularized operations, that is, the output is cleaned from lower-dimensional features such as isolated points, isolated edges, antennas, or open boundaries, you have to post-process the result. A more direct way

is presented in [4, Chaper 10.8].

Bibliography

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark H. Overmars. *Computational geometry*. Springer, Berlin [u.a.], 3., ed. edition, 2008.

[2] Jean-Daniel Boissonnat and Monique Teillaud, editors. *Effective computational geometry for curves and surfaces*, volume - of *Mathematics and visualization*. Springer, Berlin ; Heidelberg [u.a.], 2007.

[3] Jacob E. Goodman and Joseph O'Rourke. *Handbook of discrete and computational geometry*, volume - of *Discrete mathematics and its applications*. Chapman & Hall/CRC, Boca Raton, 2nd. ed. edition, 2004.

[4] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000. Available online at `http://www.mpi-inf.mpg.de/~mehlhorn/LEDAbook.html`.

[5] Mariette Yvinec. 2d triangulations. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.5 edition, 2009.

Chapter 3

Non-linear curves

Sweep needs modifications:

More than one intersection (e.g., several points, several overlapping parts). We have to insert only the leftmost to the event queue, unless all are available. Then without running time penalty (but with higher space consumption – not higher than the output complexity of the corresponding arrangement), we can add all of them.

We have to consider the *Multiplicity* of an intersection point of two curves. An exact definition is given in Lecture . It is odd, the two curves swap vertical ordering when processing the intersection point (as line segments), if it is even, the order is preserved. In that case, no new adjacency is created.

If $k > 2$ curves intersect in a common point (pass through), it is non-trivial to compute their ordering to the right. This can be done in $O(Mk)$ where M is the maximal pairwise multiplicity.