On-line Zone Construction in Arrangements of Lines in the Plane*

Y. Aharoni D. Halperin I. Hanniel S. Har-Peled C. Linhart
Department of Computer Science
Tel Aviv University

Abstract

Given a finite set $\mathcal{L}$ of lines in the plane we wish to compute the zone of an additional curve γ in the arrangement $\mathcal{A}(\mathcal{L})$, namely the set of faces of the planar subdivision induced by the lines in $\mathcal{L}$ that are crossed by γ, where γ is not given in advance but rather provided *on-line* portion by portion. This problem is motivated by the computation of the area bisectors of a polygonal set in the plane. We present four algorithms which solve this problem efficiently and exactly (giving precise results even on degenerate input). We implemented the four algorithms. We present implementation details, comparison of performance, and a discussion of the advantages and shortcomings of each of the proposed algorithms.

1 Introduction

Given a finite collection $\mathcal{L}$ of lines in the plane, the *arrangement* $\mathcal{A}(\mathcal{L})$ is the subdivision of the plane into vertices, edges and faces induced by $\mathcal{L}$. Arrangements of lines in the plane, as well as arrangements of other objects and in higher dimensional spaces, have been extensively studied in computational geometry [9, 15, 23], and they occur as the underlying structure of the algorithmic solution to geometric problems in a large variety of application domains. The *zone* of a curve γ in an arrangement $\mathcal{A}(\mathcal{L})$ is the collection of (open) faces of the arrangement crossed by γ (see Figure 1 for an illustration).

In this paper we study the following algorithmic problem: Given a set $\mathcal{L}$ of n lines in the plane, efficiently construct the zone of a curve γ in the arrangement $\mathcal{A}(\mathcal{L})$, where γ consists of a single connected component and is given on-line, namely γ is not given in its entirety as part of the initial input but rather given (contiguously) piece after piece and at any moment the algorithm has to report all the faces of the arrangement crossed by the part of γ given so far.

There is a straightforward solution to our problem. We can start by constructing $\mathcal{A}(\mathcal{L})$ in $\Theta(n^2)$ time, represent it in a graph-like structure $\mathcal{G}$ (say, the half-edge data structure [8, Chapter 2]), and then explore the zone of γ by walking through $\mathcal{G}$. However, in general we are not interested in the entire arrangement. We are only interested in the zone of γ, and this zone can be anything from a single triangular face to the entire arrangement.

The *combinatorial complexity* (complexity, for short) of a face in an arrangement is the overall number of vertices and edges on its boundary. The complexity of a collection of faces is the sum of the face complexity over all faces in the collection. The complexity of an arrangement of n lines is $\Theta(n^2)$ (and if we allow degeneracies, possibly less). The complexity of the zone of an arbitrary curve in the arrangement can range from $\Theta(1)$ to $\Theta(n^2)$. The simple algorithm sketched above will always require

*This work has been supported in part by ESPRIT IV LTR Projects No. 21957 (CGAL) and No. 28155 (GALIA), by The Israel Science Foundation founded by the Israel Academy of Sciences and Humanities, by a Franco-Israeli research grant (monitored by AFIRST/France and The Israeli Ministry of Science), and by the Hermann Minkowski – Minerva Center for Geometry at Tel Aviv University. Dan Halperin has been also supported in part by an Alon fellowship and by the USA-Israel Binational Science Foundation.

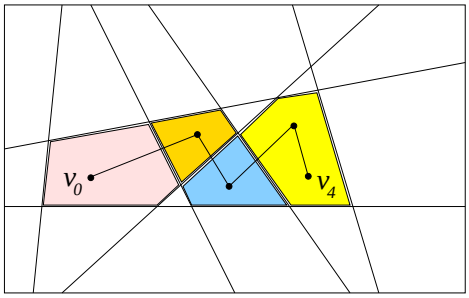

Figure 1: An example of a simple arrangement, induced by 7 lines and a bounding box, and the zone of a polygonal curve in it. The polyline $v_0, \ldots, v_4$ crosses 4 faces (the shaded faces).

$\Omega(n^2)$ time, which may be too much when the complexity of the zone is significantly below quadratic. (Note that even an optimal solution to our problem may require more than $\Theta(n^2)$ running time for certain input curves, since we do not impose any restriction on the curve and it may cross a single face an arbitrary large number of times.)

If the whole curve γ were given as part of the initial input (together with the lines in $\mathcal{L}$) then we could have used one of several worst-case near-optimal algorithms to solve the problem. The idea is to transform the problem into that of computing a single face in an arrangement of segments, where the segments are pieces of our original lines that are cut out by γ [10]. Denote by k the overall number of intersections between γ and the lines in $\mathcal{L}$, the complexity of the zone of γ in $\mathcal{A}(\mathcal{L})$ (or the complexity of the single face containing γ in the modified arrangement) is bounded by[1] $O((n+k)\alpha(n))$ and several algorithms exist that compute a single face in time that is only a polylogarithmic factor above the worst-case combinatorial bound, for example [7].

However, since in our problem we do not know the curve γ in advance, we cannot use these algorithms, and we need alternative, on-line solutions. Our problem, the on-line version of the zone construction, is motivated by the study of area bisectors of polygonal sets in the plane [3] which is in turn motivated by algorithms for part orienting using Micro Electro Mechanical Systems [4]. The curve γ that arises in connection with the area bisectors of polygons is determined by the faces of the arrangement through which it passes, it changes from face to face, and its shape within a face f is dependent on f.

In the work reported here we assume that γ is a *polyline* given as a set of $m+1$ points in the plane $v_0, v_1, \ldots, v_m$, namely the collection of m segments $\overline{v_0v_1}, \overline{v_1v_2}, \ldots, \overline{v_{m-1}v_m}$, given to the algorithm in this order. In the motivating problem γ is a piecewise algebraic curve where each piece can have an arbitrarily high degree. We simplify the problem by assuming that the degree of each piece is one since our focus here is on the on-line exploration of the faces of the arrangement. The additional problems that arise when each piece of the curve can be of high degree are (almost completely) independent of the problems that we consider in this paper and we discuss them elsewhere [1]. We further assume that we are given a *bounding box* B, i.e., the set $\mathcal{L}$ contains four lines that define a rectangle which contains $v_0, \ldots, v_m$ (see Figure 1).

An efficient solution to the on-line zone problem is given in [3], and it is based on the well-known algorithm of Overmars and van Leeuwen for the dynamic maintenance of halfplane intersection [21]. As with many other papers in computational geometry, the data structure described in [21] is rather intricate and we anticipated that it would be difficult to implement. We resorted instead to simpler solutions which are nevertheless non-trivial.

We devised and implemented four algorithms for on-line construction of the zone. The first algorithm is based on halfplane intersection and maintains a balanced binary tree on a set of halfplanes induced by $\mathcal{L}$ (it is reminiscent of the Overmars-van Leeuwen structure, but it is simpler and does not have the

[1] $\alpha(\cdot)$ denotes the extremely slowly growing inverse of the Ackermann function.

good theoretical guarantee of running time as the latter). The second algorithm works in the dual plane and maintains the convex hull of the set of points dual to the lines in $\mathcal{L}$. Its efficiency stems from a heuristic to recompute the convex hull when the hull changes (the set of points does not change but their contribution to the lower or upper hull changes according to face of the arrangements that γ visits). Algorithms 1 and 2 could be viewed as simple variants of the algorithm described in [3]. The third algorithm presents a completely novel approach to the problem. It combines randomized construction of the binary plane partition induced by the lines in $\mathcal{L}$ together with maintaining a doubly connected edge list for the faces of the zone that have already been built. We give two variants of this algorithm, where the second variant (called Algorithm 3b) handles conflict lists [20] more carefully than the first (Algorithm 3). Finally, the fourth algorithm is also a variant of Algorithm 3; however it differs from Algorithm 3 in a slight but crucial manner: it refines the faces of the zone as they are constructed such that in the refinement no face has more than some small constant number of edges on its boundary.

The four algorithms are presented in the next section, together with implementation details and description of optimizations applied to speed up the running time of the algorithms. In Section 3 we describe a test suite of ten input sets on which the algorithms have been examined, followed by a chart of experiment results. Then in Section 4 we point out the advantages and shortcomings of each of the algorithms, and summarize the lessons we learnt from the implementation, optimization and experiments. Concluding remarks and suggestions for future work are given in Section 5.

2 Algorithms and Implementation

2.1 Algorithm 1: Halfplane Intersection

Given a point p and a set $\mathcal{L}$ of n lines, the face that contains p in the arrangement $\mathcal{A}(\mathcal{L})$ can be computed in $O(n \log n)$ time by intersecting the set of halfplanes induced by the lines and containing p. The idea is to divide the set of halfplanes into two subsets, recursively intersect each subset, and then use a linear-time algorithm for the intersection of the two convex polygons (see, for example, [8, Chapter 4]). We shall extend this scheme to the on-line zone construction by maintaining the *recursion tree*.

2.1.1 The General Scheme

Let us assume we have a procedure `ConvexIntersect` that intersects two convex polygons in linear time. Given the first point v_0 of γ, we construct the recursion tree in a bottom-up manner in $O(n \log n)$ time, as follows. The lowest level of the tree contains n leaves, which hold the intersection of the bounding box with each halfplane induced by a line of $\mathcal{L}$ and containing p. The next levels are constructed recursively by applying `ConvexIntersect` in postorder. The resulting data structure is a balanced binary tree, in which each internal node holds the intersection of the convex polygons of its two children.

When the polyline moves from one face of the arrangement to its neighbor it crosses a line in $\mathcal{L}$. All we need to do is update the leaf in the recursion tree that corresponds to this line, and then reconstruct the polygons of its ancestors, again in a bottom-up manner. Clearly, for each update the number of calls to `ConvexIntersect` is $O(\log n)$. The time it takes to move from one face to its neighbor therefore depends on the total complexity of the polygons in the path up the tree. In the worst case this can add up to $O(n)$ even if the returned face (i.e., the root of the tree) is of low complexity. However, if the complexity of each polygon along the path is constant (as in some of our test cases), then the update time is only $O(\log n)$.

2.1.2 Convex-Polygon Intersection

The main "building block" of the algorithm is the `ConvexIntersect` procedure. We started by implementing a variation of the algorithm described in [8, Chapter 4]. The idea is to decompose each convex polygon into two chains — *left* and *right*. We use a sweep algorithm to sweep down these four

chains, finding the intersections as we proceed, and handling the various possible cases in each event (this algorithm corresponds, with minor modifications, to the Shamos-Hoey method described in [22]). This algorithm turned out to be quite expensive — in each event, several cases of edge intersections have to be checked, although some of them cannot appear more than twice.

Therefore, we implemented a different algorithm that sweeps the two left chains and the two right chains separately, and then sweeps through the resulting left and right chains to find the top and bottom vertices of the intersection polygon. These sweeps are very simple, and can be easily implemented efficiently. For two polygons with 40 vertices, whose intersection contains 80 vertices, the new algorithm runs 4 times faster than the original one.

2.1.3 Optimization

In addition to the improved intersection algorithm described above, we applied several other optimization techniques. The most important of these was the use of *floating-point filters*. We implemented the segment intersection predicate in terms of the orientation predicate in LEDA's rational-geometry kernel [19]. This predicate performs a fast floating-point initial check, and only if it fails does it revert to the exact, but slow, rational computation (see Section 4.2 for more details). In addition, in the function for the intersection of two segments we made use of the fact that the function is called in our implementation only after we know that the segments intersect, and that these segments always lie on the set of lines predefined in the problem. This enabled us to ignore cases that a more general function (such as CGAL's[2] intersection function) has to take into consideration (e.g., two parallel lines).

Another method we used to accelerate the intersection function was the use of a hashing scheme to avoid re-computation of intersections already computed (see Section 4.3). However, this did not yield a significant speedup — for an input of 1000 random lines and a hash table with 60000 entries, we got an improvement of only 10-15 percent.

2.2 Algorithm 2: Dual Approach

The algorithm is based on the duality between the problems of computing the intersection of halfplanes and calculating the convex hull of a set of points. The duality transform [9] maps lines (points) in the primal plane to points (lines) in the dual plane, preserving above/below relations — for example, a primal point above a primal line is mapped to a dual line which is above the line's dual point.

Given a set of lines in the plane and a point p, we can find the intersection of halfplanes induced by the lines and containing p by first transforming the set of lines to a set of points P in the dual plane, and transforming p to a dual line ℓ. Next, we need to partition the set P into two subsets according to whether a point lies above or below ℓ. We then compute the convex hull of each of the two subsets. Now, by spending an additional linear amount of work (specifically, finding the tangents connecting the two convex hulls) we obtain a list of points whose original primal lines make up the boundary of the respective intersection of halfplanes and in the desired order. For more details on the duality between convex hulls and halfplane intersection see, e.g., [8, Section 11.4].

Our algorithm is based on a modification by Andrew [2] of Graham's scan algorithm [14]. The first step of this algorithm calls for sorting the set of points, and then the actual convex hull is computed in an $O(n)$ scan. Since our set of points is fixed, we can pre-sort them and, as the traversal progresses, shift points from one subset to the other without destroying the implied order – all we have to do is keep a tag with each point indicating the subset it currently belongs to. When moving from a face to one of its neighbors through an edge, a single dual point must be moved from one subset to the other (the point whose primal line contains the crossed edge) and the convex hulls should be recomputed. Thus, the algorithm reports each face of the zone in $O(n)$ time. However, a few simple observations enable us to reduce the needed amount of work, effectively improving the algorithm's performance on some inputs. A detailed description of the algorithm will be given in the full version of this paper.

[2]CGAL [6] — the Computational Geometry Algorithms Library, *http://www.cs.uu.nl/CGAL*.

2.3 Algorithm 3: Binary Plane Partition

As explained in Section 1, a straightforward solution to the zone problem would be to construct the arrangement $\mathcal{A}(\mathcal{L})$, and then explore the zone of γ, face by face. However, constructing $\mathcal{A}(\mathcal{L})$ takes $\Theta(n^2)$ time (and space), whereas the actual zone may be of considerably lower complexity. In this section we shall describe an algorithm that maintains a partial arrangement, namely a subset of the faces induced by $\mathcal{L}$. The idea would be to construct only the parts of $\mathcal{A}(\mathcal{L})$ that are intersected by γ, adding faces to the partial arrangement on-line, as we advance along the curve. The algorithm we have developed is a variant of the randomized *binary plane partition* (BPP) technique. The partial arrangement $\mathcal{A}^*(\mathcal{L})$ is represented in a data structure that is a combination of a *doubly connected edge list* (DCEL) and the BPP's binary search tree, both of which are described in [8]. We call this data structure a *face tree*.

2.3.1 The Face Tree Data Structure

The face tree is a binary search tree whose nodes correspond to faces in the plane. Each face is split into two sub-faces, using one of the input lines that intersects it. We assign an arbitrary orientation to each line in $\mathcal{L}$. The sub-face that is to the left of the splitting line is set as the node's left child, and the right sub-face as its right child.

Formally, denote by $f(u)$ the face that corresponds to node u, and let $l(u)$ and $r(u)$ be the left and right children (sub-nodes) of u respectively. The set of *inner segments* of u is the intersection of the lines $\mathcal{L}$ with the face $f(u)$: $I(u) = \{\ \ell \cap f(u) \mid \ell \in \mathcal{L},\ \ell \cap f(u) \neq \emptyset\ \}$ (the set $I(u)$ is often referred to as the *conflict list* of $f(u)$). The face $f(u)$ is partitioned into two sub-faces using one of these inner segments, which we shall call the *splitter* of $f(u)$, and is denoted $s(u)$. We say that a face is *final* if $I(u) = \emptyset$, in which case it always corresponds to a leaf in the face tree. Locating a point p in a tree whose depth is d takes $O(d)$ time, using a series of calls to the `SideOfLine` predicate, i.e., simple queries of the form – "is p to the left of $s(u)$?".

As we have already mentioned, the faces are not only part of a binary search tree – they also form a DCEL. Each face is described by a doubly connected cyclic linked list of its edges (or half-edges, as they are called in [8]). Thus, splitting the edges into two sub-faces is very efficient, and takes constant time once we have chosen the splitter – we simply "cut" the two edges, $e1$ and $e2$, which the splitter intersects, and "glue" the new parts, together with the splitter and its reverse (twin-edge), to form two new faces. For each half-edge in the DCEL, we maintain its *twin half-edge* and a pointer to its *incident face*, as in a standard DCEL. These pointers enable us to follow γ efficiently along adjacent faces, instead of performing a new point location query each time it exits a face, which is extremely useful when γ revisits faces it has already traversed before.

Let u be a leaf corresponding to a non-final face in $\mathcal{A}^*(\mathcal{L})$, with n_u inner segments. In order to split the face $f(u)$, we first choose a splitter $s(u)$ from $I(u)$ at random; we then prepare the lists that describe the edges of the two sub-faces, as explained above; and, finally, we prepare the inner segments list of each sub-face. Splitting a face takes a total of $O(n_u)$ time (the exact details will be described in the full version of this paper).

2.3.2 The General Scheme

The algorithm starts by initializing the root r of the face tree as the bounding box B (see the Introduction). The set of inner segments $I(r)$ is initialized as $\{\ \ell \cap B \mid \ell \in \mathcal{L}\ \}$. The initialization phase requires $O(n)$ time and space. Given the first point v_0 of the polyline, we locate it in the face tree. Obviously, we will find it in $f(r)$. Since r is not a final leaf (assuming $n > 0$), we split r, and continue recursively in the sub-face that contains v_0, as illustrated in Figure 2. The search ends when we reach a final leaf, that is, a face in $\mathcal{A}(\mathcal{L})$ that is not intersected by any line in $\mathcal{L}$. We report this face, and wait for the next point along the polyline.

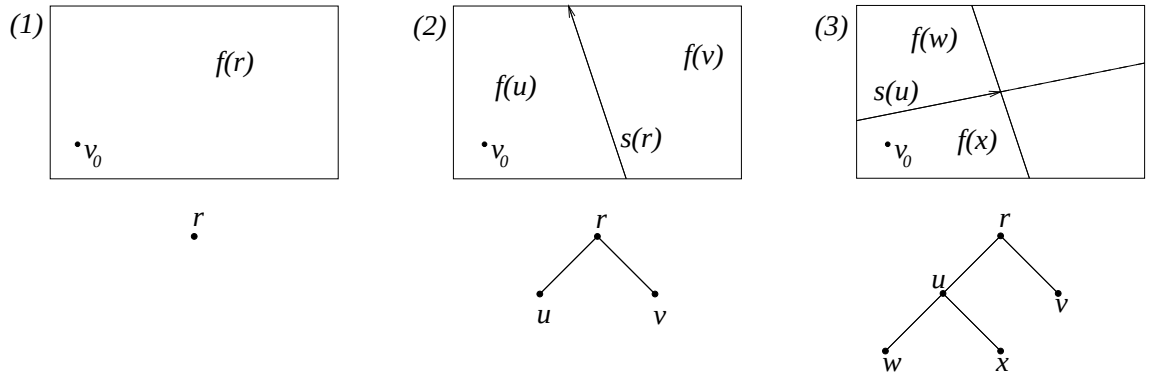

Figure 2: An illustration of the construction of the face tree given the first point v_0 of the polyline: first, the root face $f(r)$ is set as the bounding-box (1); then, $f(r)$ is split into two sub-faces – $f(u)$, which contains v_0, and $f(v)$ (2); similarly, $f(u)$ is split into two sub-faces – $f(w)$ and $f(x)$ (3); and so on until no input line crosses the face f_0 which contains v_0.

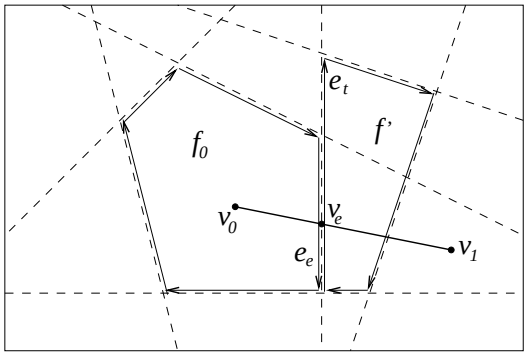

Figure 3: Following γ along $\mathcal{A}^*(\mathcal{L})$ after the first face (f_0) has been built: first, we find the exit point v_e (on edge e_e) and skip to the adjacent face f' using the twin edge e_t; then, from f' we build the face f_e through which γ passes using the same recursive procedure we used for building f_0 from $f(r)$ (as shown in Figure 2).

Suppose we are now given the next vertex v_1 along γ. First, we find the intersection between the segment $\overline{v_0 v_1}$ and the edges of the current face f_0, that we have reported earlier. This can be performed in $O(\log m_0)$, where m_0 is the complexity of f_0, but we implemented a trivial $O(m_0)$ solution. If $\overline{v_0 v_1}$ exits f_0 at point v_e on edge e_e, as illustrated in Figure 3, then we skip to the opposite face using e_e's twin-edge pointer – e_t, and locate v_e in it – as long as we are in an internal node of the face tree, we simply check on which side of the corresponding splitter lies v_e, and continue the search there; once we reach a leaf f', we repeat the same splitting algorithm as described above, until we finally find the face f_e that contains v_e (we now update the twin-edge pointers of both e_e and $e_t \cap f_e$, so that next time γ moves from f_0 to f_e, or vice versa, we could follow it in $O(1)$ time). Next, we need to locate the point at which $\overline{v_e v_1}$ exits f_e, and report the next face that γ crosses. This procedure is repeated until we reach the face that contains v_1. We then wait for the next vertex along γ, and report the faces in the zone in the same manner.

2.3.3 Improved Maintenance of the Conflict Lists

The main shortcoming of the algorithm we have just described is its worst-case behavior of $\Theta(n^2)$ time for preparing a face with $\Theta(n)$ edges. For example, in input 5 (Section 3) there are $n = 2000$ lines

that all contribute edges to a single large face. Initially, the root face contains 2000 inner segments, so splitting it requires $2000c$ operations, where c is some constant. After the root is split, the face containing the first point in the polyline has 1999 inner segments. Splitting it takes $1999c$ operations, and results in a face with 1998 inner segments, and so on. Thus, locating the first point is performed in $\Theta(n^2)$ time, which is the time it would require to construct the full arrangement.

To overcome this shortcoming, we developed a data structure that enables us to split a face without having to check all its inner segments. Instead of maintaining the inner segments in a simple array, we distribute them among the vertices of the face in *conflict lists*. Denote by p the point that is being located (p is either the first vertex of the polyline, or its exit point from the previous face). An inner segment s in face f is said to be in *conflict* with the vertex v if v and p lie on different sides of s. According to our new approach, we maintain a list of inner segments in each vertex, so that all the segments that are held in the list of a vertex v are in conflict with v. Each inner segment is kept in only one list, at one of the vertices that it is in conflict with.

In the first stage of the algorithm, we wish to locate the first vertex v_0 of the polyline in the bounding box. To this end, we first prepare the conflict lists by simply checking for each input line with which vertices it is in conflict, and adding the line to one of the corresponding conflict lists at random. We then split the bounding box using a randomly chosen splitter, and continue recursively until we reach a final face, that is, a face that contains v_0 and that is not intersected by lines from $\mathcal{L}$. It remains to show how the conflict lists can be updated when the face is split. Let f be a face that contains v_0, and denote by s its splitter. s divides f into two sub-faces – f_g, or the *green face*, that contains v_0, and f_r, the *red face*. Let u_1 and u_2 be the endpoints of s. In order to construct the final face that contains v_0, we need to work only on the green face. Notice that all the inner segments that are in the conflict lists of the green face (i.e., correspond to vertices of f that are also vertices of f_g) are still in conflict with v_0, so they need not be changed. The only inner segments we have to check are those that are in the red face – for each such segment, we add it to the conflict list of u_1 (if it is in conflict only with u_1), u_2 (similarly), one of u_1 and u_2 at random (if it is in conflict with both vertices), or none (in case it is in conflict with neither u_1 nor u_2, which means that it does not intersect the green face). This algorithm guarantees that the face that contains v_0 is constructed in expected time $O(n \log n)$ (see [20]).

The rest of the algorithm is very similar to the original scheme – given the next vertex along the polyline, we first locate the exit point from the current face, and construct the next face γ intersects. However, the conflict lists in the red faces need to be updated, since the definition of a conflict depends on the point being located, and it has changed. Furthermore, some of the inner segments might be missing, and we need to collect them from the relevant green face, i.e., the sibling of the red face. The solution we implemented is to gather all the inner segments from the current face and from the sub-tree of the sibling node, and prepare new conflict lists, as we have done for the bounding box. Once the conflict lists are built, we continue as before.

This new algorithm, which we will denote 3b, gave a substantial performance improvement for the cases of complex faces, with only a small degradation in the running times on other inputs, as the results in Section 3 show.

2.4 Algorithm 4: Randomized Incremental Construction of the Zone

Recently, Har-Peled [17] gave an $O((n+m)\alpha(n)\log n)$ expected time algorithm for the on-line construction of the zone, where m is the number of intersections between γ and the input lines. This improves by almost a logarithmic factor over the application of Overmars and van Leeuwen [21] for this restricted case [3]. The algorithm of [17] relies on a careful simulation of an off-line randomized incremental algorithm (which uses an oracle) that constructs the zone. Motivated by the algorithm of [17], we had implemented a somewhat simpler variant. The resulting algorithm is similar to the Binary Plane Partition (BPP, for short) algorithm presented in Section 2.3. The original algorithm of [17] is a bit complicated, and it is not clear that it will perform in practice better than the algorithms presented in this paper.

The algorithm implemented is similar to the algorithm of Section 2.3, with the difference that we cut complex faces so that every internal node in the face tree will hold a polygon of constant complexity. For a node v, let R_v be the polygonal region that corresponds to it (this may be a whole face in the arrangement, as in algorithm 3, or part of a face), and let $\mathrm{cl}(v)$ be its *conflict list*, i.e., the list of input lines that intersect R_v.

Computing a leaf of T which contains a given point p is performed by carrying out a point-location query, as in algorithm 3. We divert from this algorithm, in the following way: if a face R_v created by the algorithm has more than c vertices (where c is an arbitrary constant which is a parameter of the algorithm), we split it into two regions, by a segment s that connects two of its vertices (chosen arbitrarily), thus generating two new children in the tree T, such that the complexity of each of their polygons is at most $\lceil c/2 \rceil + 1$ (note that s does not lie on one of the lines in $\mathcal{L}$). Now, a node in T corresponds to a polygon having at most c vertices (a similar idea was investigated in [16]). The result is that each split operation requires only constant time (ignoring the extra time required for computing the conflict lists). This also seems to reduce the expected depth of the tree, and we conjecture that it is logarithmic.

To compute a face, we first compute a leaf v of T that contains our current point p, as described above. We next compute all the leaves that correspond to the face that contains p, by performing a point-location query in the middle of a splitter lying on the boundary of R_v. By performing a sequence of such point-location queries in T, we can compute all the leaves of T that correspond to the face that contains p. Reconstructing the whole face from those leaves is straightforward.

The walk itself is carried out by computing for each face the point where the walk leaves the face, and performing a point-location query in T for this exit point (as in algorithm 3).

2.4.1 Computing the Conflict Lists Using Less Geometry

When splitting a region R_v into two regions R_{v^+}, R_{v^-}, we have to compute the conflict lists of R_{v^+}, R_{v^-}. For a line $\ell \in \mathrm{cl}(R_v)$, this can be done by computing the intersection between ℓ and R_{v^+}, R_{v^-}, but this is rather expensive. Instead, we do the following: for each line in the conflict list of R_v, we keep the indices of the two edges of ∂R_v that ℓ intersects. As we split R_v into R_{v^+} and R_{v^-}, we compute for each edge of R_v the edges of R_v^+, R_v^- it is being mapped to. Thus, if the line ℓ does not intersect the two edges of R_v that are crossed by the splitting line, then we can decide, by merely inspecting this mapping between indices, whether ℓ intersects either R_{v^+}, R_{v^-}, or both, and what edges of R_v^+, R_v^- are being intersected by ℓ. However, there are situations where this mapping mechanism is insufficient. In such cases, we use the geometric predicate `SideOfLine`, which determines on which side of a line a given point lies, as illustrated in Figure 4. While there is a non-negligible overhead in computing the mapping, the above technique reduces the number of calls to the `SideOfLine` predicate by a factor of two (the predicate `SideOfLine` is an expensive operation when using exact arithmetic, even if filtering is applied).

2.4.2 Optimization

In addition to caching results of computations (see Section 4.2), massive filtering was performed throughout the program. The lines are represented both in rational and floating-point representation. The points are represented as two pointers to the lines whose intersection forms the point. Whenever the result of a geometric predicate is unreliable (i.e., the floating-point result lies below a certain threshold) the exact representation of the geometric entities involved are computed, and the predicate is recomputed using exact arithmetic. This filtering results in that almost all computations are carried out using floating point arithmetic, which gives a speedup by a factor of two.

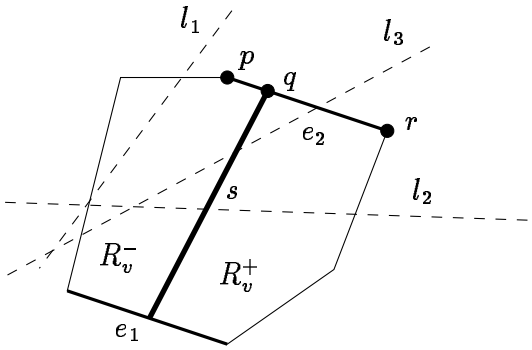

Figure 4: Computing the conflict lists of R_v^+, R_v^-: The lines l_1, l_2 can be classified by inspecting their edge indices, and the indices mapping. The classification of l_3 is done by inspecting its edge indices, and computing `SideOfLine` of the points p, q (or q, r) relative to l_3.

3 Experiments

In order to test the programs that implement the four algorithms we described in the previous section, we have created 10 input files (see Figure 5). Inputs 1–4 are relatively small, with input 3 checking that the algorithm copes with the degenerate case that the polyline crosses *vertices* of the arrangement. In input 5, the polyline is contained in a very complex face, with 2000 edges. Input 6 is similar – the polyline intersects the same three faces 17 times; two of these faces are complex, with about 500 edges each. Input 7 consists of 500 random lines and a polyline which is a single line segment. Input 8 contains many small faces – there are 498 vertical lines and 2 horizontal lines, forming a grid of almost 1500 rectangular faces; the polyline traverses 499 of them, almost 50 times each. The polyline in input 9 repeats almost the same path 3 times. Input 10 is a larger version of input 7 – the complex arrangement contains 2000 random lines, and the polyline crosses thousands of faces of the arrangement.

input no.	1	2	3	4	5	6	7	8	9	10
# of lines	14	45	38	104	2000	1002	500	500	228	2000
# of vertices of the polyline	7	24	5	141	48	17	2	50	23	45
# of faces in the zone (without multiplicities)	19 (18)	95 (94)	27 (26)	950 (906)	1 (1)	33 (3)	232 (232)	24,403 (499)	917 (337)	7,144 (6,983)
Algorithm 1	0.064	0.387	0.225	4.003	4.582	2.866	1.917	120.09	4.900	58.803
Algorithm 2	0.040	0.294	0.096	3.659	13.076	3.330	1.418	96.160	5.400	120.91
Algorithm 3	0.004	0.029	0.006	0.274	4.926	1.134	0.379	0.130	0.203	15.627
Algorithm 3b	0.003	0.032	0.016	0.391	1.573	0.451	0.258	0.161	0.184	9.568
Algorithm 4	0.009	0.030	0.012	0.275	0.982	0.358	0.090	0.184	0.134	2.701

Table 1: Summary of the test data and running times of the four algorithms described in Section 2. Times are average running times in seconds, based on 25 executions on a Pentium-II 450MHz PC with 528MB RAM memory.

We ran the programs on the test suite and measured average running times, based on 25 executions (see Table 1). In general, the best results were achieved by algorithms 3, 3b, and 4, whose running times were usually 4 to 20 times faster than those of algorithms 1 and 2. It is important to note that each algorithm was implemented by a different programmer, applying different optimization techniques, as will be discussed in Section 4. Comparing the performance of the programs might therefore be

misleading with regard to the algorithms' potential performance. Nevertheless, some conclusions can be drawn from our experiments.

Algorithms 1 and 2 do not maintain a partial arrangement, which explains why they perform very poorly on input 8 — they need to re-compute the same faces again and again. The performance of these algorithms is rather similar in most cases. An exception are inputs 5 and 10, where the better asymptotic performance of algorithm 1 is apparent. As mentioned earlier, the improved maintenance of the conflict lists reduced the running time of algorithm 3b, compared to algorithm 3, on inputs 5 and 6. Interestingly, it also performs better on the large random arrangement of input 10. Algorithm 4 gave similar results to algorithm 3b, but after massive optimization (Section 2.4.2) it surpassed it by a factor of up to 3.

4 Discussion

In the following section, we summarize the major conclusions drawn from our experience, especially with regard to optimization techniques.

4.1 Software Design

Motivated by the structure of the CGAL basic library [6], in which geometric predicates are separated from the algorithms (using the traits mechanism [12]), we restricted all geometric computations in the programs to a small set of geometric predicates. This enabled us to debug and profile the programs easily, and deploy caching, filtering, and exact arithmetic effortlessly. Writing such predicates is not always trivial, but fortunately LEDA [18] and CGAL [11] provide such implementations.

4.2 Number Type and Filtering

To get good performance, one would like to use the standard floating-point arithmetic. However, this is infeasible in geometric computing, where exact results are required, due to precision limitations. Since our input is composed of lines and vertices in rational coordinates, we can perform all computations exactly using LEDA rational type, and our first choice was to use this number type. However, LEDA rational suffers from several drawbacks: (i) computations are slow (up to 20-40 times slower than floating point), (ii) they consume a lot of memory, and (iii) the bit length of the representation of the numbers doubles with each operation, which in turn causes a noticeable slowdown in program execution time.

One possible approach to improve the efficiency of the representation of a LEDA rational number is to normalize the number explicitly. However, this operation is expensive, and the decision where to do such normalization is not straightforward.

A different approach is to use filtering [13]. Generally speaking, filtering is a method of carrying out the computations using floating-point (i.e., fast and inexact) arithmetic, and performing the computations using exact arithmetic only when necessary. LEDA [5] provides a real number type, which facilitates such filtered computations (it is not restricted to rational operators or geometric computations). Additionally, LEDA provides a fine-tuned computational geometry kernel that performs filtering. Algorithms 1, 3, and 3b used LEDA's filtered predicates (e.g., `SideOfLine`), which resulted in a speedup by a factor of two or more, compared to the non-filtered rational computations. Furthermore, one can also implement the filtering directly on the geometric representation of the lines and points (for example, computing the exact coordinates of the points only if necessary), as was done in algorithm 4 (see Section 2.4.2). As mentioned earlier, this technique saves a considerable amount of exact computations, and gives an additional speedup of two.

Overall, usage of filtering resulted in the most drastic improvement in the running times of the programs, and was easy to implement.

4.3 Caching

One possible approach to avoid repeated exact computations is to cache results of geometric computations (i.e., intersection of lines, `SideOfLine` predicate, etc.). Such a caching scheme is easy to implement, but does not result in a substantial improvement. Especially, as the filtered computations might require less time than the lookup time. Caching was implemented in algorithms 1 and 4.

4.4 Geometry

The average number of vertices of a face in an arrangement of lines is about 4 (this follows directly from Euler's formula). Classical algorithms rely on vertical decomposition, which have the drawback of splitting all the faces of the arrangement into vertical trapezoids. An alternative approach is to use constant complexity convex polygons instead of vertical trapezoids (as was done in algorithm 4 described in Section 2.4). The results indicate that this approach is simple and efficient. This idea was suggested by Matoušek, and was also tested out in [16].

4.5 Miscellaneous

Additional improvement in running time can be achieved by tailoring predicates and functions to special cases, instead of using ready-made library (e.g., CGAL) ones. For example, knowing that two segments intersect at a point, liberates us from the necessity to check if they overlap. Such techniques were used in algorithms 1, 3, and 3b. Significant improvement can also be accomplished by improving "classical" algorithms for performing standard operations (see, for example, Section 2.1.2).

5 Conclusions

We have presented four algorithms for on-line zone construction in arrangements of lines in the plane. All algorithms were implemented, and we have also presented experimental results and comparisons between the algorithms.

A major question raised by our work is what could be said theoretically about the on-line zone construction problem. As mentioned in the Introduction, [3] proposes a near-optimal output-sensitive algorithm based on the Overmars-van Leeuwen data structure for dynamic maintenance of halfplane intersection. It would be interesting to implement this data structure and compare it with the algorithms that we have implemented.

If the number of faces in the zone of the curve γ is small (constant), then algorithm 3b described in Section 2.3.3 is guaranteed to run in expected time $O(n \log n)$, since in that case it is an almost verbatim adaptation of the randomized incremental algorithm for constructing the intersection of halfplanes, whose running time analysis is given in [20, Section 3.2]. Indeed it performs very well on inputs 5 and 6 (Section 3). Surprisingly it also performs well on an input of totally opposite nature, input 10, where there are thousands of small faces in the zone. An interesting open problem is to extend the analysis of [20] for the case of a single face to the case of the on-line zone construction.

Algorithm 4 is an attempt to implement a practical variant of a theoretical result giving a near-optimal output-sensitive algorithm for the on-line zone construction [17] (the result in [17] was motivated by the good results of algorithm 3 and differs from it only slightly but, as mentioned above, in a crucial factor — by subdividing large faces into constant size faces). Still, the practical variant which was implemented has no theoretical guarantee. Can the analysis of [17] be extended to explain the performance of algorithm 4 as well?

Finally, as mentioned earlier, the algorithms have been implemented independently, and hence there are many factors that influence their running time on the test suite beyond the fundamental algorithmic differences. We are currently considering alternative measures (such as the number of basic operations) that will allow for better comparison of algorithm performance.

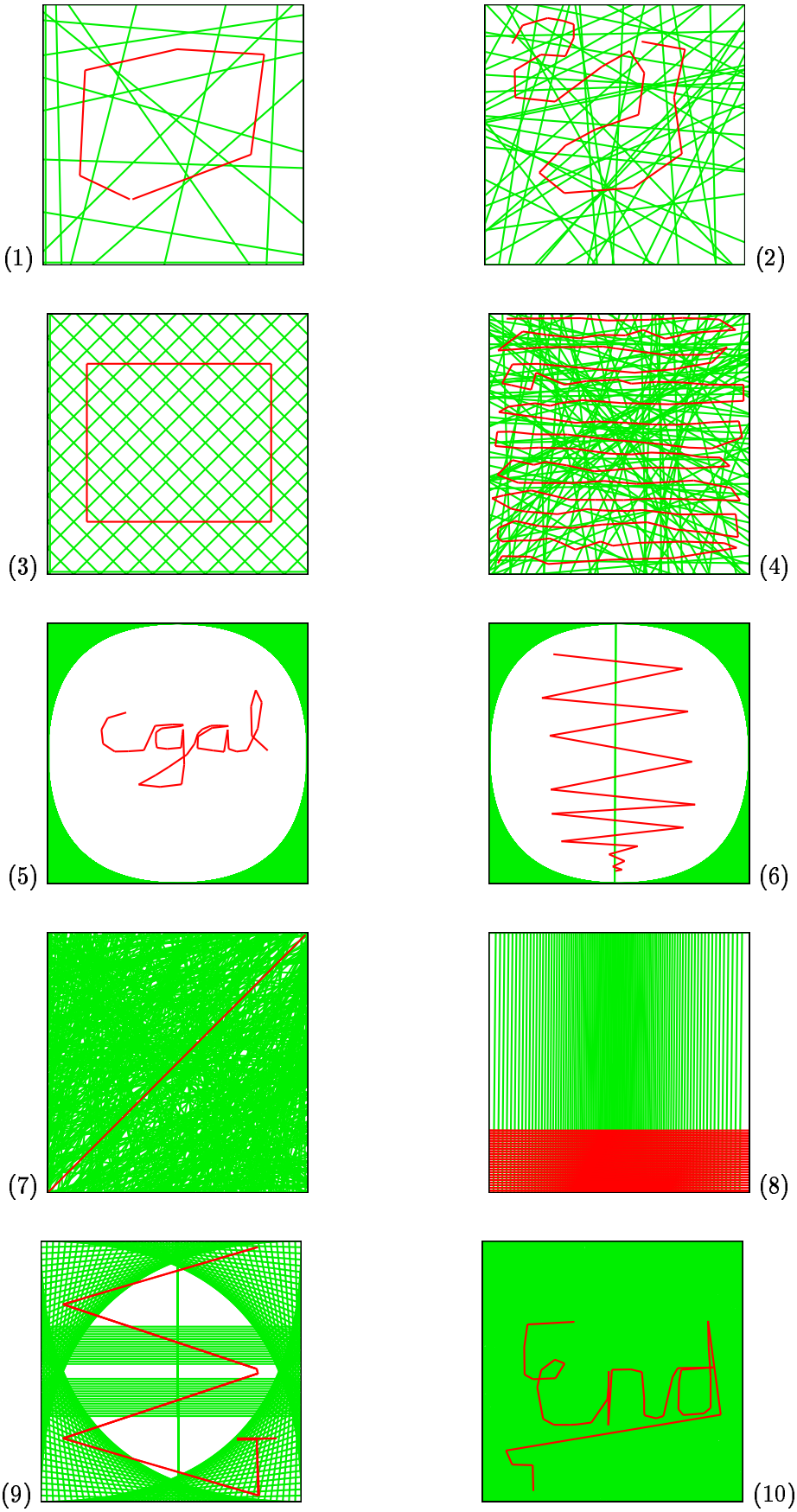

Figure 5: The ten input suite whose details, together with the running time of the algorithms on them, are given in Table 1.

References

[1] Y. Aharoni. Computing the area bisectors of polygonal sets: An implementation. In preparation, 1999.

[2] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979.

[3] K.-F. Böhringer, B. Donald, and D. Halperin. The area bisectors of a polygon and force equilibria in programmable vector fields. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 457–459, 1997. To appear in Disc. and Comput. Geom.

[4] K.-F. Böhringer, B. R. Donald, and N. C. MacDonald. Upper and lower bounds for programmable vector fields with applications to MEMS and vibratory plate parts feeders. In J.-P. Laumond and M. Overmars, editors, *Robotics Motion and Manipulation*, pages 255–276. A.K. Peters, 1996.

[5] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, Jan. 1996.

[6] *The CGAL User Manual, Version 1.2*, 1998.

[7] B. Chazelle, H. Edelsbrunner, L. J. Guibas, M. Sharir, and J. Snoeyink. Computing a face in an arrangement of line segments and related problems. *SIAM J. Comput.*, 22:1286–1302, 1993.

[8] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[9] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Heidelberg, Germany, 1987.

[10] H. Edelsbrunner, L. J. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir. Arrangements of curves in the plane: Topology, combinatorics, and algorithms. *Theoret. Comput. Sci.*, 92:319–336, 1992.

[11] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Proc. 1st ACM Workshop on Appl. Comput. Geom.*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 191–202. Springer-Verlag, 1996.

[12] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. Technical Report MPI-I-98-1-007, Max-Planck-Institut Inform., 1998.

[13] S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[14] R. L. Graham. An efficient algorithm for determining the convex hull of a set of points in the plane. *Information Processing Letters*, 1:132–133, 1972.

[15] D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 21, pages 389–412. CRC Press LLC, 1997.

[16] S. Har-Peled. Constructing cuttings in theory and practice. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 327–336, 1998.

[17] S. Har-Peled. Taking a walk in a planar arrangement. Manuscript, `http://www.math.tau.ac.il/~sariel/papers/98/walk.html`, 1999.

[18] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, New York. To appear.

[19] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual, Version 3.7*. Max-Planck-Institut für Informatik, 1998.

[20] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1994.

[21] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23:166–204, 1981.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[23] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, New York, 1995.