An Experimental Study of Point Location in Planar Arrangements in CGAL

IDIT HARAN and DAN HALPERIN
Tel-Aviv University

We study the performance in practice of various point-location algorithms implemented in CGAL (the Computational Geometry Algorithms Library), including a newly devised *landmarks* algorithm. Among the other algorithms studied are: a naïve approach, a "walk along a line" strategy, and a trapezoidal decomposition-based search structure. The current implementation addresses general arrangements of planar curves, including arrangements of nonlinear segments (e.g., conic arcs) and allows for degenerate input (for example, more than two curves intersecting in a single point or overlapping curves). The algorithms use exact geometric computation and thus result in the correct point location. In our landmarks algorithm (a.k.a. jump & walk), special points, "landmarks," are chosen in a preprocessing stage, their place in the arrangement is found, and they are inserted into a data structure that enables efficient nearest-neighbor search. Given a query point, the nearest landmark is located and a "walk" strategy is applied from the landmark to the query point. We report on various experiments with arrangements composed of line segments or conic arcs. The results indicate that compared to the other algorithms tested, the landmarks approach is the most efficient, when the overall (amortized) cost of a query is taken into account, combining both preprocessing and query time. The simplicity of the algorithm enables an almost straightforward implementation and rather easy maintenance. The generic programming implementation allows versatility both in the selected type of landmarks and in the choice of the nearest-neighbor search structure. The end result is an efficient point-location algorithm that bypasses the alternative CGAL implementations in most practical aspects.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Geometric algorithms*; F.2.2 [**Analysis of Algorithms**]: Nonnumerical Algorithms and Problems—*Geometrical problems*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Generic programming*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Point location, arrangements, CGAL, computational geometry, generic programming

This work has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes), by the Israel Science Foundation (grant no. 236/06), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.
Author's address: Idit Haran and Dan Halperin, School of Computer Science, Tel-Aviv University, 69978, Tel-Aviv, Israel; email: iditharan@gmail.com; danha@post.tau.ac.il.

ACM Reference Format:
Haran, I. and Halperin, D. 2008. An experimental study of point location in planar arrangements in CGAL. ACM J. Exp. Algor. 13, Article 2.3 (September 2008), 32 pages DOI 10.1145/1412228.1412237 http://doi.acm.org 10.1145/1412228.1412237

1. INTRODUCTION

Given a set $\mathcal{C}$ of n planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane induced by the curves in $\mathcal{C}$ into maximally connected cells. The cells can be 0- (*vertices*), 1- (*edges*) or 2-dimensional (*faces*). The *planar map* of $\mathcal{A}(\mathcal{C})$ is the embedding of the arrangement as a planar graph, such that each arrangement vertex corresponds to a planar point, and each edge corresponds to a planar subcurve of one of the curves in $\mathcal{C}$. Arrangements and planar maps are ubiquitous in computational geometry, and have numerous applications [Agarwal and Sharir 2000; Halperin 2004]. Figure 1 shows two different types of arrangements, one induced by line segments and the other by conic arcs.[1]

The planar point-location problem is a fundamental problem in the study of arrangements: Preprocess an arrangement into a data structure so that, given any query point q, the cell (i.e. face, edge or vertex) of the arrangement containing q can be efficiently retrieved.

The planar point-location problem may be solved naïvely by traversing all the edges and vertices in the arrangement and finding the geometric entity that is exactly on, or directly above, the query point. The time it takes to perform the query using this approach is proportional to the number of edges n of the arrangement, both in the average and worst-case scenarios.

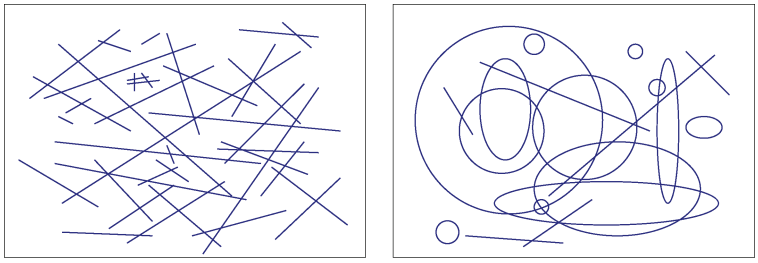
In case the arrangement remains unmodified once it is constructed, it may be useful to invest considerable amount of time in preprocessing in order to achieve good performance of point-location queries. However, if new curves are inserted into the arrangement or removed from it, an auxiliary point-location data structure that can be efficiently updated must be employed, perhaps at the expense of the query answering speed.

The point-location problem was extensively studied. Some algorithms that were developed for solving the problem achieve worst-case query time $O(\log n)$ and data structure of size $O(n)$, while other approaches aim at good average query time, in practice.

In this work, we study known algorithms for planar point location and compare different implementations of these algorithms. We have limited this work to cover algorithms implemented in CGAL (the Computational Geometry Algorithms Library). We show that no existing strategy within CGAL simultaneously addresses all the issues of preprocessing complexity, memory usage, and query time.

We propose a new point location method, called *landmarks*. In this algorithm, special points, which we call "landmarks," are chosen in a preprocessing stage,

[1]A *conic curve* is an algebraic planar curve of degree 2. A *conic arc* is a bounded segment of such a curve.



(a) (b)

Fig. 1. Arrangements of line segments (a) and of conic arcs (b).

their place in the arrangement is found, and they are inserted into a hierarchical data structure enabling fast nearest-neighbor search. Given a query point, the nearest landmark is located, and a "walk" strategy is applied, starting at the landmark and advancing toward the query point.

We have implemented our algorithm in CGAL. The simplicity of the algorithm and the use of generic programming enable an elegant implementation, which allows versatility both in the selected type of landmarks, and in the choice of the nearest-neighbor search structure.

Various experiments using the landmarks algorithm were conducted on arrangements of varying size and density, composed of either line segments or conic arcs. Our implementation was compared against various point-location algorithms in CGAL, including a naïve approach, a "walk along a line" strategy, and a trapezoidal decomposition-based search structure. The results indicate that among the studied algorithms, the landmarks approach is the most efficient when the overall (amortized) cost of a query is taken into account, combining both preprocessing and query time.

The rest of the paper is organized as follows: In Section 2 we introduce the terminology and notation used in the work, and we review related work on other point location strategies, including some implementations of point-location algorithms in CGAL. Section 3 describes the landmarks algorithm in details. Implementation details are given in Section 4. Section 5 presents a thorough point-location benchmark conducted on arrangements of varying size and density, composed of either line segments or conic arcs, with an emphasis on studying the behavior of the landmarks algorithm. Concluding remarks are given in Section 6.

2. PRELIMINARIES AND RELATED WORK

2.1 CGAL

CGAL[2] is the product of a collaborative effort of several sites in Europe and Israel, aiming to provide a generic and robust, yet efficient, implementation of widely used geometric data structures and algorithms. It is a software library written in C++ according to the generic programming paradigm

[2]The CGAL project homepage: `http://www.cgal.org/`.

[Austern 1999]. Robustness of the algorithms is achieved by both handling all degenerate cases and by using exact geometric computation. Exact computation means to ensure that all decisions made by the algorithm are correct decisions for the actual input. It does not mean that in all calculations exact representations for all numerical values have to be computed. Approximations that are sufficiently close to the exact value can often be used to guarantee the correctness of a decision. Whenever it is needed for ensuring exact computations, we use *exact number types*. A representation of a real number r should be called exact only if it allows one to compute an approximation of r to whatever precision, i.e., no information has been lost. Rational numbers are represented by a numerator and a denominator, where both are arbitrary precision integers. For example, we use the GMP[3] library to represent rational numbers. When we need to use algebraic numbers, as employed, for example, in construction of arrangement of conics, we use exact predicates. For example, a presentation of an algebraic number α could be by saving an integer polynomial P having a root α and an interval that isolates α from the other roots of P. Among the number-type libraries that are used for algebraic numbers are CORE[4] [Karamcheti et al. 1999] and LEDA[5] [Mehlhorn and Näher 2000]. The CGAL library also provides several number types that can be used for exact computation. (See [Schirra 2000] and [Yap 2004] for surveys on robustness issues in computational geometry.)

CGAL's arrangement package was the first generic software implementation designed for constructing arrangements of planar curves and supporting operations and queries on such arrangements [Flato et al. 2000; Fogel et al. 2004; Wein et al. 2007]. The arrangement package supports configurations of curves with finite x-monotone subcurve decomposition. The arrangement class template is parameterized by a traits class that encapsulates the geometry of the family of curves it handles. Robustness is guaranteed, as long as the traits classes use exact number types whenever it is needed for the computations they perform.

2.2 KD Trees

In the nearest-neighbor problem, a set P of data points in d-dimensional space is given. These points are preprocessed into a data structure, so that given any query point q, the nearest (or more generally s-nearest) points of P to q can be reported efficiently.

A KD tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in k-dimensional space. The idea is that at each level of the tree the space is partitioned by a hyperplane that is perpendicular to one of the coordinate system axes. The points of the KD tree are stored only in the leaf nodes of the tree. In fixed dimension, the KD tree can be built in $O(n \log n)$ time and uses linear space. If the input points are well distributed in space, it is known to answer nearest-neighbor queries in logarithmic time [Bentley 1975].

[3]Gnu's multiprecision library: `http://www.swox.com/gmp/`.
[4]`http://www.cs.nyu.edu/exact/core_pages/`.
[5]`http://www.algorithmic-solutions.com/enleda.htm`.

2.2.1 *ANN.* ANN (the Approximate Nearest-Neighbor library) is a library written in the C++ programming language to support both exact and Approximate Nearest-Neighbor searching in spaces of various dimensions.[6] It was implemented by David M. Mount of the University of Maryland and Sunil Arya of the Hong Kong University of Science and Technology. ANN is also a testbed containing programs and procedures for generating data sets, collecting and analyzing statistics on the performance of nearest-neighbor algorithms and data structures and visualizing the geometric structure of these data structures. It has been shown by Arya and Mount [1993] and Arya et al. [1998] that if the user is willing to tolerate a small amount of error in the search (returning a point that may not be the nearest neighbor, but is not significantly further away from the query point than the true nearest neighbor) then it is possible to achieve a significant improvement in running time.

2.3 Solving the Point-Location Problem

The point-location problem has been studied for many years. Given an arrangement of planar curves, that consists of n edges (that do not intersect in their interior), and a query point q, the problem is to find the cell of the arrangement containing q. Several approaches for solving the point-location problem are known with worst-case query time $O(\log n)$ and data structure of size $O(n)$ [Snoeyink 2004]. One approach is based on the vertical decomposition of the arrangement. By drawing a vertical line through every vertex of the arrangement, we obtain vertical *slabs* in which point location is almost one-dimensional. Two binary searches suffice to answer a query: one on the x coordinates for the slabs containing q, and one on the edges that cross the slab. The query time is $O(\log n)$, but the space may be quadratic if all edges are stored with the slabs that they cross [Dobkin and Lipton 1976]. Since the location structures for adjacent slabs are similar, one can sweep from left to right to construct balanced binary search trees on edges for all slabs [Preparata 1981; Shamos 1975]. To obtain linear storage space, Sarnak and Tarjan [1986] used *persistent search trees*.

Edahiro et al. [1984] used the slabs idea and developed a point-location algorithm that is based on a grid. The plane is divided into cells of equal size called buckets using horizontal and vertical partition lines. These partition lines are determined by several parameters of the arrangement. In each bucket the local point location is performed using the naïve slabs algorithm described above. The slabs partition of each bucket is not computed in advance. Instead, it is created on the fly during query time.

A different approach which has an expected query time of $O(\log n)$, which is referred to in the CGAL documentation and in the paper as the *RIC (random incremental construction)* algorithm, was introduced by Mulmuley [1990] and Seidel [1991]. The algorithm uses two structures: (1) a *trapezoidal map* and (2) a *search structure*—the history DAG (directed acyclic graph). The trapezoidal map is created by subdividing each arrangement face into

[6]The ANN library homepage: `http://www.cs.umd.edu/~mount/ANN/`.

pseudotrapezoidal cells each of constant complexity. Each such cell is bounded above and below by curves and from the sides by vertical attachments (which on one side may degenerate to a point). The search structure and the trapezoidal map are interlinked: A cell in the trapezoidal map has a pointer to the leaf of the DAG corresponding to it and a leaf node of the DAG has a pointer to the corresponding cell in the trapezoidal map. The algorithm is incremental: it adds the segments one at a time, in a random order, and after each addition it updates the search structure and the trapezoidal map. The RIC algorithm gives an expected optimal point location scheme: $O(\log n)$ expected query time, $O(n \log n)$ expected preprocess time, and $O(n)$ expected space, when the expectation is taken over random choices made by the construction algorithm.

Another approach for point location, which achieves good performance, is called the chain method. A (vertical) monotone chain is a path such that the y coordinate never increases along the path. A simple polygon is (vertical) monotone if it is formed by two monotone chains, with the first and last vertices in common. It is possible to add some edges to a planar subdivision, in order to make all faces monotone, obtaining what is called a monotone subdivision. Edelsbrunner et al. [1986] devised an optimal data structure that only uses the edges in a monotone subdivision. The idea is to use vertical monotone chains, instead of using vertical lines to partition the subdivision. In order to achieve $O(\log n)$ query time, they used fractional cascading, in addition to a binary search, keeping pointers between the edges of different monotone chains.

Point location in triangulations was extensively studied. Kirkpatrick [1983] developed a method for point location in triangulation, which takes $O(\log n)$ query time, using a data structure of size $O(n)$. The method creates a hierarchy of subdivisions in which all faces are triangles. In every triangulation, one can find (in linear time) an independent set of low-degree vertices whose size is a constant fraction of all vertices. To build a higher level in the hierarchy, these vertices are removed and the polygons created by their removal are triangulated (if necessary). Repeating this process a logarithmic number of times gives a constant-size triangulation. To locate the triangle containing a query point q, we start by finding the triangle in the coarsest triangulation. Then, knowing the hole that this triangle came from, we replace the missing vertex, and check the incident triangles to locate q in the previous, finer triangulation.

Devillers et al. [2002] proposed a *walk along a line* algorithm for point location in triangulations, which does not require the generation of additional data structures, and offers $O(\sqrt{n})$ query time, on average, if the vertices are distributed uniformly at random and $O(n)$ query time in the worst case. The walk may begin at an arbitrary vertex v of the triangulation and advance toward the query point q, using four walk strategies: (1) A straight walk, which traverses all triangles that are intersected by a line segment connecting the vertex v and q. (2) The orthogonal walk, that visits all triangles along an axis-parallel path moving from the vertex v to q by changing one coordinate at a time. (3) The visibility walk, which advances from one triangle to another through the first edge of the triangle e if the line supporting e separates the vertex v from q (if not, it checks the second edge of the triangle, and so on). (4) A stochastic walk

is obtained by replacing the access to the first edge of a triangle in the visibility walk by a random edge. Because of the simplicity of the structure (triangles), all walk strategies consist of low-cost primitive operations.

Devillers later proposed a walk strategy based on a Delaunay hierarchy [Devillers 2002], which uses a hierarchy of triangulations. The triangulation at the lowest level is the original triangulation where operations and point location are to be performed. Each succeeding higher level consists of a data structure that stores a triangulation of a small random sample of the vertices of the triangulation at the preceding lower level. Point location is done through a top-down nearest-neighbor query. The nearest-neighbor query is first performed naïvely in the top level triangulation. Then, at each following level, the nearest-neighbor at that level is found through a linear walk performed from the nearest-neighbor found at the lower level. Because the number of vertices in each triangulation is only a small fraction of the number of vertices of the preceding triangulation, the data structure remains small and rapidly answers point-location queries. This structure behaves best when it is built for Delaunay triangulations.

Other algorithms that were developed only for Delaunay triangulations, often referred to as *jump & walk* algorithms, were proposed by Mücke et al. [1996] and by Devroye et al. [1998]. The *jump & walk* proceeds as follows: Given a triangulation of n sites, pick $k \in [1, n]$ random sites in the data. Given a query point q, select the site ζ closest to q, and traverse the triangulation from ζ to q, exploiting the adjacency relationship between the successive triangles crossed by segment ζq. The expected query time of this method is $O(k + \sqrt{n/k})$, which reaches its optimum $O(n^{1/3})$ when k is $\Theta(n^{1/3})$. Devroye et al. [2004] later improved this method to an algorithm called *binsearch & walk*. They keep $n^{1/4}$ points with known locations and use a weighted-balanced binary search tree, based on the lexicographic order of the points, to find the nearest point from which to start the walk to the query. They also developed a method called *2-d search & walk*, that uses a balanced two-dimensional KD tree in which the partition alternates directions between x and y and member sets in the partition are rectangles (which is basically a KD tree in two dimensions). The latter method achieves $O(\log n)$ expected time to locate a random query point in the Delaunay triangulation of n sites uniformly and independently distributed in the plane.

Arya et al. [2001b] showed that a simple modification of the RIC algorithm to include weights gives expected query times satisfying entropy bounds. Suppose that we have a planar subdivision with regions of constant complexity, such as trapezoids or triangles, and that we know the probability p_i of a query falling in the ith region. The entropy H is defined to be $\sum_i -p_i \cdot \log_2 p_i$. For a constant K, assign to a subdivision edge that is incident on regions with total probability P the weight $\lceil KPn \rceil$, and perform a randomized incremental construction. The use of integral weights ensures that ratios of weights are bounded by $O(n)$, which is important to achieve query time bounded by $O(H)$. Entropy-preserving cuttings can be used to give a method whose query time of $H + o(H)$ approaches the optimal entropy bound [Arya et al. 2001a], at the cost of increased space and programming complexity.

As the point-location problem is of practical importance, many works (some mentioned above) include extensive experiments in their study, beside the theoretical analysis. Devillers et al. [2002] have tested the different "walk" algorithms in triangulations, and showed that the best "walking" strategy among straight, orthogonal, visibility or stochastic walk may depend on the triangulation at hand. Devroye et al. [2004] have tested their *jump & walk* algorithm using different "jump" alternatives. Their experiments show that the most efficient option would be to use a balanced two-dimensional KD tree to get to the nearest starting point.

2.4 Point Location in CGAL

Point location constitutes a significant part of the arrangement package in CGAL, as it is a basic query applied to arrangements during their construction. Various point-location algorithms (also referred to as point-location strategies) have been implemented as part of CGAL's arrangement package. The best point-location strategy is dependent on the arrangement's size, topology, and the frequency of modifications. In the new design of the arrangement package (see Section 4) the different point-location strategies can be used simultaneously on the same arrangement, which enables flexibility in their use. For example, one can use a specific algorithm for constructing the arrangement and another algorithm while the arrangement is unmodified and many queries are issued on it. The point-location strategies implemented in CGAL are:

1. The *naïve* strategy traverses all vertices and edges of the arrangement and locates the nearest edge or vertex that is situated exactly on, or immediately above, the query point. It maintains no data structures, beyond the basic representation of the arrangement, and does not require any preprocessing stage. The naïve algorithm takes $O(n)$ time, where n is the number of edges in the arrangement, both in the worst and in the average case.
2. The *walk* algorithm traces (in reverse order) a vertical ray r emanating from the query point to infinity; it traverses the *zone*[7] of r in the arrangement. This vertical walk is simpler than a walk along an arbitrary direction (that will be explained in details below, as part of the landmarks algorithm), as it requires simpler predicates ("above/below" comparisons). Simple predicates are desirable in exact computing especially when nonlinear curves are used. Like the naïve strategy, the walk strategy maintains no extra data structures, and does not require any preprocessing stage. The walk strategy has a faster query time than the naïve algorithm in the average case, although it may also take $O(n)$ time in the worst case. Figure 2 illustrates an example of the walk algorithm.
3. The *triangulation* strategy was implemented only for line segment arrangements. It consists of a preprocessing stage where each arrangement face is subdivided using constrained Delaunay triangulation (CDT).[8] A CDT is

[7]The *zone* of a curve is the collection of all the cells in the arrangement that the curve intersects.
[8]This phase is done only once in preprocessing time and saved in parallel to the arrangement. The triangulation algorithm does not support dynamic insertion of segments into the arrangement—it builds a new triangulation upon any insertion to (or deletion from) the arrangement.

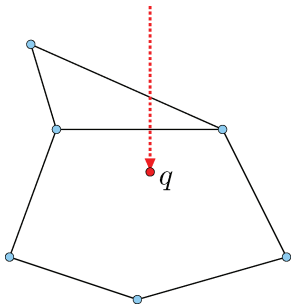


Fig. 2. An example of walking along a vertical ray starting from infinity and advancing toward the query point q. During the walk, all the faces that are crossed by the ray are tested, until the face containing q is found.

a triangulation with constrained edges, which tries to be "as much Delaunay as possible." As constrained edges are not necessarily Delaunay edges, the triangles of a CDT do not necessarily fulfill the empty-circle property, but they fulfill a weaker constrained empty-circle property. To state this property, it is convenient to think of constrained edges as blocking the view. Then, a triangulation is constrained Delaunay if and only if the circumscribing circle of any facet encloses no vertex visible from the interior of the facet. In this triangulation, point location is implemented using a triangulation hierarchy [Devillers 2002], which uses (as explained above) a hierarchy of triangulations and performs a hierarchical search from the highest level in the hierarchy to the lowest. At each level of the hierarchical search, a walk is performed to find the triangle in the next lower level that contains q, until the triangle in the lowest level is found. The algorithm uses the triangulation package of CGAL [Boissonnat et al. 2002].

4. The *RIC (random incremental construction)* algorithm is an implementation of the dynamic algorithm introduced by Mulmuley [1990] and Seidel [1991], described in Section 2.3. The implementation consists of two structures: (i) a *trapezoidal map* and (ii) a *search structure*—the history DAG. The expected time for constructing the search structure is $O(n \log n)$, and the expected query time is $O(\log n)$, where n is the number of nonintersecting edges in the arrangement. (For a detailed explanation of the algorithm, see de Berg et al. [2000], Chapter 6).

3. POINT LOCATION WITH LANDMARKS

The motivation behind the development of the new *landmarks* algorithm was to address both issues of preprocessing complexity and query time at once, something that none of the existing strategies do well in practice. The naïve and the walk algorithms have, in general, bad query time, which precludes their use in large arrangements. The RIC algorithm answers queries very fast, but it uses relatively large amount of memory and requires a complex preprocessing stage. In the case of dynamic arrangements, where curves are constantly being inserted into or removed from the arrangement, this is a major

drawback. Moreover, in real-life applications the curves are typically inserted to the arrangement in nonrandom order. This reduces the performance of the RIC algorithm, as it relies on random order of insertion, unless special procedures are followed [Devillers and Guigue 2001].

The basic idea behind the landmarks algorithm is to choose and locate points (landmarks) within the arrangement and store them in a data structure that supports nearest-neighbor search. During query time, the landmark closest to the query point is found using the nearest-neighbor search and a short "walk along a line" is performed from the landmark toward the query point. The key incentive behind the landmarks algorithm is to reduce the number of costly algebraic predicates involved in the walk or the RIC algorithms at the expense of increased number of the relatively inexpensive coordinate comparisons (in nearest-neighbor search).

The algorithm is composed of three independent components, each of which can be optimized or replaced with a different component (of the same functionality):

1. Choosing the landmarks that faithfully[9] represent the arrangement and locating them in the arrangement.
2. Constructing a data structure that supports nearest-neighbor search (such as KD trees), and using this structure to find the nearest landmark to a given query point.
3. Applying a "walk along a line" procedure, moving from the landmark toward the query point.

The following sections elaborate on these components.

3.1 Choosing the Landmarks

When choosing the landmarks, we aim to minimize the expected length of the "walk" inside the arrangement toward a query point. The search for a good set of landmarks has two aspects:

1. Choosing the number of landmarks.
2. Choosing the distribution of the landmarks throughout the arrangement.

It is clear that as the number of landmarks grows, the walk stage becomes faster. However, this results in longer preprocessing time and larger memory space. We found out that, in certain cases, the nearest-neighbor search consumes a significant portion of the overall query time (when "overshooting" with the number of landmarks; see Section 5.2.2 below).

What constitutes a good set of landmarks depends on the specific structure of the arrangement at hand. In order to assess the quality of the landmarks, we defined a measure representing the complexity of the walk stage: The *arrangement distance* (AD) between two points is the number of faces in which the straight line segment that connects these points passes. If two points

[9]Meaning that the local density of the landmarks distribution will roughly follow the one of the arrangement.

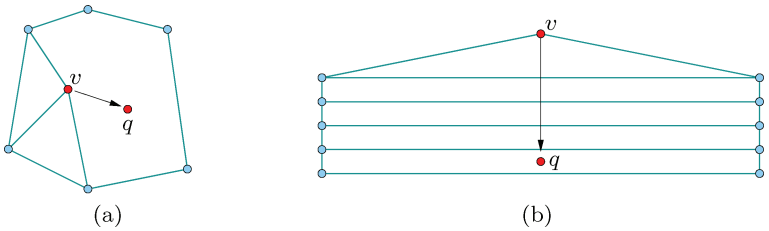


Fig. 3. The arrangement distance from the closest vertex v to the query point q: (a) AD = 0 and (b) AD = 4.

reside in the same face of the arrangement, the arrangement distance is defined to be zero. The arrangement distance may differ substantially from the Euclidean distance, as two points, which are spatially close, can be separated in an arrangement by many small faces. Figure 3 shows the arrangement distance from the closest vertex v of the arrangement to a query point q: in 3(a) AD = 0, and in 3(b) AD = 4.

The landmarks may be chosen with respect to the (0, 1 or 2-dimensional) cells of the arrangement. One can use the vertices of the arrangement, points along the edges (e.g., the edges midpoints), or interior points in the faces, as landmarks. In order to choose representative points inside the faces, it may be useful to preprocess the arrangement faces, which are possibly nonconvex, for example, using vertical decomposition or triangulation.[10] Such preprocessing will result in simple faces (pseudotrapezoids and triangles, respectively) for which interior points can be easily determined. Landmarks may also be chosen independently of the arrangement geometry. One option is to spread the landmarks randomly throughout a rectangle bounding the arrangement. Another is to use a uniform grid, or to use other structured point sets, such as Halton sequences or Hammersley points [Matoušek 1999; Niederreiter 1992]. Each choice has its advantages and disadvantages and improved performance may be achieved using combinations of different types of landmarks choices.

In the current implementation the landmarks type is given as a template parameter, called *generator*, to the landmarks algorithm, and can be easily replaced. This generator is responsible for creating the sets of landmark points and updating them if necessary. The following types of landmark generators were implemented:

—*LM(vert).* All the arrangement vertices are used as landmarks. The benefit of using the vertices of the arrangement as landmarks, is that their location in the arrangement is known, and they represent the arrangement well (dense areas contain more vertices). The drawback is that walking from a vertex requires a preparatory step in which we examine all incident faces around the vertex to decide on the startup face (see more details in Section 3.3 below).

[10]Triangulation is relevant only in case of arrangements of line segments.

—*LM(mide).* The midpoints of all the arrangement edges are chosen. This generator was implemented only for line-segment arrangements. The benefit of using the middle of the edges as landmarks, similarly to the LM(vert), is that their location in the arrangement is known (on the edges), and they also represent the arrangement well. However, walking from the midpoints of the edges also requires a small preparatory step to choose between the two faces incident to the edge.

—*LM(rand).* Random points are selected. These points are randomly sampled from a uniform distribution inside the arrangement bounding rectangle. The bounding rectangle is defined by the minimal and maximal x and y coordinates of elements (edges, vertices) in the arrangement. The number of random points is given as a parameter to the generator and is set to be the number of vertices by default. After choosing the points, we have to locate them in the arrangement. To this end, we use the newly implemented batched point location in CGAL [Wein et al. 2007], which uses the sweep algorithm for constructing the arrangement, while adding the landmark points as special events in the sweep. When reaching such a special event during the sweep, we search the y (status) structure to find the edge that is just above the point.

—*LM(grid).* The landmarks are chosen on a uniform grid. As in the previous generator, the number of landmarks n is given as a parameter to the generator, and is set to be the number of vertices by default. The landmarks are chosen on a $\lceil\sqrt{n}\,\rceil \times \lceil\sqrt{n}\,\rceil$ grid that covers the bounding rectangle of the arrangement. The location of the grid points in the arrangement is done in the same manner as was described for random points. The benefit of using grid points as landmarks is that the closest grid point to a given query point can be found in constant time (no need for a search structure).

—*LM(halton).* Halton sequence points are used. The motivation for constructing this sequence is to get a low discrepancy point set in the plane [Matoušek 1999, Chapter 2]. The Halton points landmarks are first calculated on the unit square $[0, 1] \times [0, 1]$ and then scaled to the arrangement's bounding rectangle. The Halton sequence in the unit square is calculated as follows [LaValle 2006, Chapter 5]: We choose two prime integers (for 2D points: $p_1 = 2$, $p_2 = 3$). To construct the ith sample, consider the base-p representation for i, which takes the form $i = a_0 + pa_1 + p^2a_2 + p^3a_3 + \cdots$. The following point in the interval [0,1] is obtained by reversing the order of the bits and moving the decimal point:

$$r(i, p) = \frac{a_0}{p} + \frac{a_1}{p^2} + \frac{a_2}{p^3} + \frac{a_3}{p^4} + \cdots \tag{1}$$

Starting from $i = 0$, the ith sample (point) in the Halton sequence is $(r(i, p_1), r(i, p_2))$. For example, assume the base p to be 2. For $I = 1, 2, 3, \ldots$, we take each number I, write it in base 2, and reverse the digits, including the decimal point, and convert back to base 10, and so on.

$$
\begin{array}{rcrcrcl}
1 & = & 1.0 & \Rightarrow & 0.1 & = & 1/2 \\
2 & = & 10.0 & \Rightarrow & 0.01 & = & 1/4 \\
3 & = & 11.0 & \Rightarrow & 0.11 & = & 3/4 \\
4 & = & 100.0 & \Rightarrow & 0.001 & = & 1/8 \\
5 & = & 101.0 & \Rightarrow & 0.101 & = & 5/8 \\
6 & = & 110.0 & \Rightarrow & 0.011 & = & 3/8 \\
7 & = & 111.0 & \Rightarrow & 0.111 & = & 7/8
\end{array}
$$

Now to get a "good" sequence of Halton points in the plane, we compute the x coordinates using base 2, and the y coordinates using base 3. Similar to the random and the grid points, the number of landmarks is given as a parameter to the generator, and is set to be the number of vertices by default. The location of the points in the arrangement is also found in the same manner as for random points.

When random points, grid points, or Halton points are used, it is in most cases clear in which face a landmark is located (as opposed to the case of vertices or edge midpoints). Thus, no preparatory step is required at the beginning of the walk stage.

The number of landmarks in LM(vert), LM(rand), LM(grid), and LM(halton) is equal to the number of vertices in the arrangement. The number of landmarks in LM(mide) is equal to the number of edges in the arrangement.

We have implemented five types of landmarks sets. The design of the landmarks class (see Section 4.3) enables to extend the types of landmarks and create other types of landmarks, optionally by combining several of the above types (for example, using random points and vertices together).

3.2 Nearest-Neighbor Search Structure

Following the choice and location of the landmarks, we have to store them in a data structure that supports nearest-neighbor queries. We note that a search structure should allow for fast preprocessing and query. A search structure that supports approximate nearest-neighbor search can also fit our needs, since the landmarks are used as starting points for the walk, and the final accurate result of the point location is computed in the walk stage.

Exact nearest-neighbor search results can be obtained by constructing a Voronoi diagram of the landmarks. However, locating the query point in the Voronoi diagram is again a point-location problem. Thus, using Voronoi diagrams as our search structure takes us back to the problem we are trying to solve. Instead, we look for a simple data structure that will answer nearest-neighbor queries quickly, even if only approximately.

The nearest-neighbor search structure is a template parameter to the landmarks algorithm. This modularity enables us to test several nearest-neighbor structures. One implementation uses the CGAL's spatial searching package [Tangelder and Fabri 2006], which is based on KD trees. The input points provided to this structure (landmarks, query points) are approximations of the original points (rounded to double), which leads to extremely fast search. Again, we emphasize that the end result is always exact.

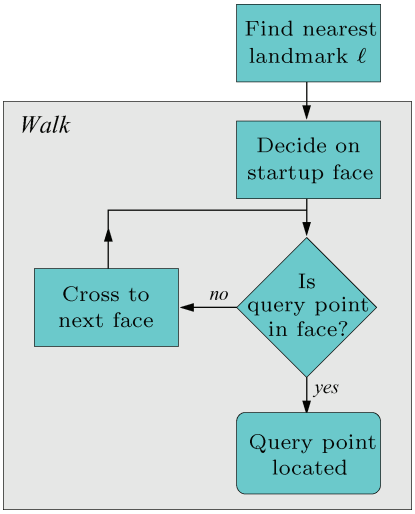


Fig. 4. The "walk" part of the query algorithm.

Another implementation uses the ANN package (see Arya et al. [1998] and Section 2.2.1), which supports data structures and algorithms for both exact and approximate nearest-neighbor searching. This library implements a number of different data structures, based on KD trees and box-decomposition trees, and employs a couple of different search strategies.

In the special case of LM(grid), no search structure is needed, and the closest landmark can be found in $O(1)$ time.

3.3 Walking from the Landmark to the Query Point

This walk algorithm developed as part of this work differs from other walk algorithms that were tailored for triangulations (especially Delaunay triangulations), as it is geared toward general arrangements that may contain faces of arbitrary topology, with unbounded complexity, and a variety of degeneracies. It also differs from the walk algorithm implemented in CGAL as the walk direction here is arbitrary, rather than vertical. The "walk" stage is illustrated in the diagram in Figure 4.

The walk starts by determining the startup face. The startup face is the face f that is most likely to contain the query point q. As explained in the previous section, certain types of landmarks (vertices, mid-edges) are not associated with a single startup face. If the landmark is located inside a face, then this is the startup face. If the landmark is located on an edge, then we need to choose between the two incident faces to this edge. Last, if the landmark is located on a vertex, we need to check all the faces incident to the vertex in order to find the face in q's direction.

After the startup face f was found, a test whether the query point q lies inside f is applied. This operation requires passing over all the edges on the face boundary, but this passage is quick, since we only count the number of f's edges above q. If this number is odd, then q is inside f, and the query is

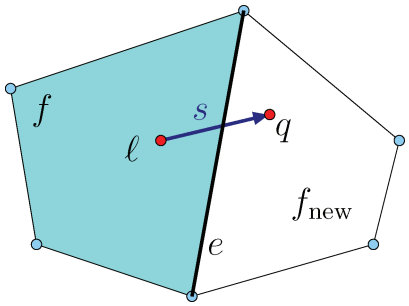


Fig. 5. An example of walking from the landmark ℓ to the query point q. In this example, ℓ is located in the face f, which is then set as the startup face. Since q is not inside f, we cross to f_{new} through the edge e, which is the edge of f's boundary that intersects the segment s connecting ℓ to q.

terminated. During this pass over the edges on the boundary of f, we also test whether q is on an edge or a vertex on the boundary and, if that is the case, we return this element as the result of the query.

However, if this number is even, then the actual "walk" part begins. A virtual line segment s is then drawn from the landmark (whose location in the arrangement is known) to the query point q. Then, we find the first edge e on the boundary of f that intersects s. Exploiting the arrangement data structure that enables $O(1)$ time access to a face from a neighboring face through a separating edge, we cross to the face on the other side of e. Figure 5 shows an example of crossing from a face f where the landmark ℓ is located, through the edge e, to the face f_{new} where the query point q is located.

As explained above, crossing to the next face requires finding the edge e on the boundary of f that intersects s. It is important to notice that there is no need to find the exact intersection point between e and s, as this may be an expensive operation. Instead, it is sufficient to perform a simpler operation. The idea is to consider the x range that contains both the curves s and e and compare the vertical order of these curves on the left- and right-hand sides of this range. If the vertical order changes, it implies that the curves intersect (see, e.g., Figure 6(a)).

In case several edges on f's boundary intersects s, probably the best edge to cross would have been the one that is the closest to q. However, we cannot decide what is the closest edge to q, since we do not compute the exact intersection point between e and s. Hence, we cross using the first edge that was found and mark this edge as used. This edge will not be crossed again during this walk, which assures that the walk process ends.

Care should be exercised when dealing with special cases, such as when s and e share a common endpoint, as shown in Figure 6(b). In this case, we need to compare the curves slightly to the right of this endpoint (the endpoint of e is the landmark ℓ).

Another case that is relevant to nonlinear curves, shown in Figure 6(c), is when e and s intersect an even number of times (two in the figure), and thus no crossing is needed. In this case, comparing the vertical order of these curves on the left- and right-hand sides of the common x range will not find an

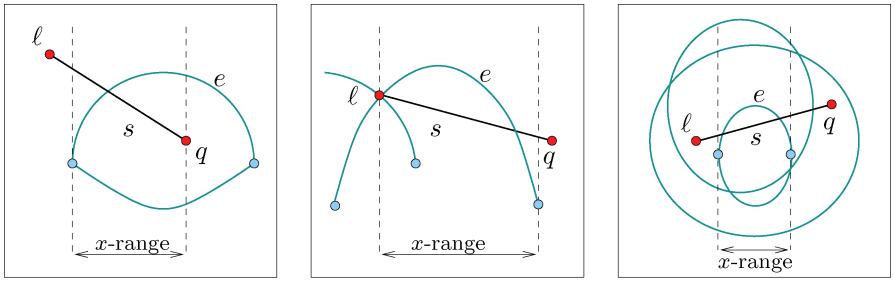


Fig. 6. Walk algorithms, crossing to the next face. In all cases, the vertical order of the curves is compared on the left- and right-hand sides of the marked x range. (a) s and e swap their y order therefore, we should use e to cross to the next face. (b) s and e share a common left endpoint, but e is above s immediately to the right of this point, and below s at the right boundary of the marked x range, so e is used for crossing (c) The y order does not change, as s and e have an even number (two) of intersections. Therefore, e is not used for crossing.

intersection. This is the desired behavior of the predicate in this case, as we do not want to use this edge for crossing.

4. IMPLEMENTATION DETAILS

In this section we present the implementation details of the landmarks point-location algorithm. Being a part of the arrangement package in CGAL, we start by explaining about the implementation of the arrangement class (Section 4.1). We also give details about the design of the point-location strategies, in general (Section 4.2), and the implementation of the class `Arr_landmarks_point_location` (Section 4.3), in particular. The code implementing the landmarks point-location algorithm can be downloaded along with the new arrangement package in CGAL.[11] This section assumes some familiarity of the reader with generic programming and it presents detailed information regarding the implementation of the algorithm in CGAL. The main principles of generic programming can be found in Austern [1999]. The reader not familiar with these techniques, may wish to proceed directly to Section 5.

4.1 The Main Arrangement Class

The `Arrangement_2<Traits,Dcel>`[12] class-template represents the planar embedding of a set of weakly x monotone[13] planar curves that are pairwise disjoint in their interiors. It provides the necessary capabilities for maintaining the planar graph, while associating geometric data with the vertices, edges, and faces of the graph. The arrangement is represented using a *doubly connected edge list* (DCEL), a data structure that enables efficient maintenance of 2D subdivisions. For more details on the DCEL data structure see de Berg et al. [2000, Chapter 2].

[11]The CGAL project homepage: `http://www.cgal.org/`.

[12]CGAL prescribes the suffix _2 for all data structures of planar objects as a convention.

[13]A continuous planar curve C is x *monotone* if every vertical line intersects it at most once. Vertical segments are defined to be *weakly* x monotone and can also be handled by the arrangement class.

The `Arrangement_2<Traits,Dcel>` class template should be instantiated with two classes as follows:

—A traits class [Myers 1997], which provides the geometric functionality, and is tailored to handle a specific family of curves. It encapsulates implementation details, such as the number type used, the coordinate representation, and the geometric or algebraic computation methods. The arrangement package in CGAL contains several traits classes that can handle line segments, polylines (continuous piecewise-linear curves), conic arc, and arcs of rational–function graphs.

—A DCEL class, which represents the underlying topological data structure, and defaults to `Arr_default_dcel<Traits>`. It associates a point with each DCEL vertex and an x monotone curve with each halfedge pair, where the geometric types of the point and the x monotone curve are defined by the traits class. However, users may extend the default DCEL implementation, and attach additional data to the DCEL records, or even supply their own DCEL class written from scratch.

The two template parameters enable the separation between the topological and geometric aspects of the planar subdivision. This separation is advantageous, as it allows users with limited expertise in computational geometry to employ the package with their own representation of any special family of curves. They must, however, supply the relevant traits-class methods, which mainly involve algebraic computation. (For further details regarding the new design of the arrangement class see Wein et al. [2007].)

The arrangement package also offers a notification mechanism that uses *observers* [Gamma et al. 1995], which can be attached to an arrangement instance and receive notifications about the changes this arrangement goes through. This may be essential for some applications that need to know exactly what happens inside a specific arrangement-instance; for example, point-location strategies that require auxiliary data structures, which must be notified on various local changes in the arrangement, in order to keep their data structures up-to-date.

4.2 Point-Location Queries

As we separate the arrangement representation from algorithms that operate on it, the `Arrangement_2` class does not support point-location queries directly. Instead, the package provides a set of classes that are capable of answering such queries, all being models of the concept *ArrangementPointLocation*. Each class employs a different algorithm or *strategy* for answering queries. A model of this concept must define the `locate()` function that accepts an input query point and returns an object representing the arrangement cell that contains this point.

The following models for the concept *ArrangementPointLocation* are included in the arrangement package: `Arr_naive_point_location`, `Arr_walk_along_a_line_point_location`, `Arr_Triangulation_point_location`, `Arr_trapezoid_ric_point_location`, and `Arr_landmarks_point_location`. Each employs a different point-location strategy, as explained in details in Section 2.4.

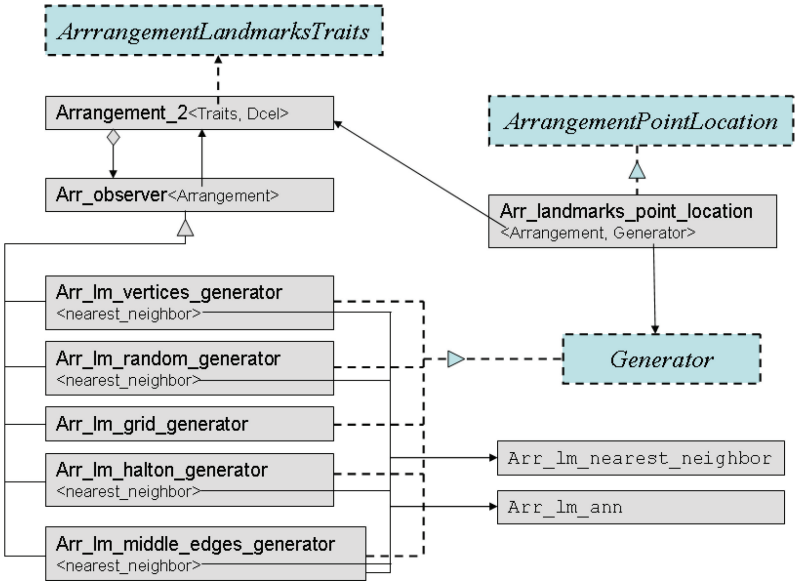


Fig. 7. The classes and concepts involved in the implementation of the landmarks point location. A rectangle with a solid frame designates a class and a rectangle with a dashed frame designates a concept. A plain arrow designates a reference to an instance of a class or of a concept, solid lines directed through a triangle mark an inheritance and directed dashed lines designate "is a model of" relation. A rhombus-shaped tail indicates that the source class stores a container of objects of the target type.

Each of the following point location classes: the landmarks, the trapezoidal-ric, and the triangulation defines a nested observer class that inherits from `Arr_observer`, and is used to receive notifications whenever the arrangement is modified. The usage of the notification mechanism makes it possible to associate several point-location objects with the same arrangement simultaneously.

4.3 The class *Arr_landmarks_point_location*

The class `Arr_landmarks_point_location<Arrangement,Generator>` is the main class of the landmarks point-location strategy. It has a member function `Object locate(Point_2 q)` that implements the landmarks point-location algorithm. It is templated by two parameters: the *Arrangement*, which is a common parameter for all point-location strategies, which represents the arrangement, and a *Generator*. Figure 7 shows the connections between all the classes and concepts that are involved in the landmarks algorithm.

4.3.1 *The Generator.* The *Generator* class represents the type of landmarks that are used. It is responsible for creating the set of landmarks along with their locations in the arrangement, storing them in a nearest-neighbor search structure and finding the closest landmark to a query point. As mentioned in Section 3.1, five types of landmarks generators were implemented, with respect to the kind of landmarks they generate: `Arr_landmarks_vertices_generator`, `Arr_middle_edges_landmarks_generator`, `Arr_random_landmarks_generator`, `Arr_grid_landmark_generator`, and `Arr_halton_landmarks_generator`.

All generators inherit from the class `Arr_observer` in order to use the notification mechanism in the arrangement class, so that whenever the arrangement changes, the generator updates its set of landmarks and their location in the arrangement. For most of the changes the arrangement goes through, such as inserting a new curve into the arrangement, the landmark set is rebuilt along with the search structure. The need to rebuild at each change is due to two reasons: First, all the current implementations of KD trees (that we are aware of) do not support dynamic insertions and deletions and, therefore, need to be rebuilt after any change. Second, in order to know what landmarks were effected by a certain change in the arrangement, so that we can relocate only those landmarks, there should be "backward" pointers from each cell of the arrangement to the associated landmarks. This is a relatively complicated data structure that was not implemented in the current work.

The only generator that partially supports dynamic insertions is the `Arr_landmarks_vertices_generator`. The reason that it is possible only for this generator (and not for the other generators) is that when a landmark is located on a vertex, its location does not change when new curves are inserted into the arrangement. This is in contrast to landmarks that are located inside a face, where the face may be split by the insertion of a new curve. Therefore, when curves are added to the arrangement, and new vertices are created, the search structure is not rebuilt. Instead, we maintain a counter of the number of vertices that were changed and only when this number exceeds the square-root of the number of landmarks, the search structure is rebuilt. The `Arr_landmarks_vertices_generator` does not support deletion of vertices, since we have no "backward" pointers from a vertex to the corresponding landmark. Therefore, when a vertex is deleted from the arrangement, the search structure is always rebuilt.

4.3.2 *The Nearest-Neighbor Class.* The *Generator* class template is parameterized with a *nearest-neighbor* class that wraps the nearest-neighbor search structure. This allows for each generator to hold a different nearest-neighbor search structure, to change the search structure easily, or to create several generators of the same type with different nearest-neighbor strategies. The nearest-neighbor search structure should define an `NN_Point_2` class, which represents the coordinates of a point in a fixed precision number type, and the arrangement `object` where this point is located. The `NN_Point_2` should have a constructor from a regular `Point_2` object, which might be represented with unlimited precision. We have implemented two types of nearest-neighbor classes:

—A nearest-neighbor class that wraps CGAL's KD trees, which is a part of the *spatial searching* package. This KD tree is an implementation of a standard KD tree as explained in Section 2.2.

—A nearest-neighbor class that wraps the ANN package (see Section 2.2.1 for further details).

The `Arr_grid_landmark_generator` is not templated by a nearest-neighbor search structure. Instead, it stores the landmarks in a vector representing a $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ grid, and finds the closest landmark simply by rounding the coordinates of a given query point to the grid lines.

4.3.3 *The Walk.* As explained in Section 3.3, the main part of the location algorithm is walking from the closest landmark ℓ to the query point q. A vital test in this part is to check whether the segment s, connecting ℓ and q, intersects with an edge e on the boundary of the startup face f. In this section, we elaborate on this test, and show that the test can be computed using only the predicates that are supported by the arrangement traits.
An edge e on the boundary of f that intersect s must fulfill two requirements:

1. e shares a common x range with s.
2. The y order of e and s changes between the left- and right-hand sides of the common x range.

The verification of the first requirement is straightforward. To test the second requirement, we need to determine the y order between two curves on both sides of the common x range. Let us consider the comparison on the left-hand side of the common x range (the comparison on the right-hand side is done in a similar manner). In the general case, when the curves do not share a common (left) endpoint, we take the rightmost point p between the two left endpoints of the curves, and compare p to the curve that p is not its endpoint. (This comparison is supported by a basic operation in the traits concept, which determines whether a given point is above, below or on a given curve.) In the special case where s and e share a common endpoint (this may happen frequently when the landmark is a vertex), we determine the relative position of the two curves immediately to the right of this endpoint. (This operation is also supported by the traits class.)

5. EXPERIMENTAL RESULTS

5.1 The Benchmark

In this section, we describe the benchmark we used to study the behavior of various point-location algorithms and specifically the newly proposed landmarks algorithm.

The benchmark includes six types of arrangements: *random segments (uniform)*, *random segments (normal)*, *random conics*, *robotics*, *Norway* and *low-E*. Each arrangement of the first type consists of line segments that were generated by connecting pairs of points whose coordinates x, y are each chosen uniformly at random in the range [0, 1000]. We generated arrangements of various sizes, up to arrangements consisting of more than 1,350,000 edges. The second type was generated in the same way as the first type, except that the coordinates were chosen from a normal distribution instead of a uniform one.

The third type of arrangements, random conics, are composed of 20% random line segments, 40% circles, and 40% canonical ellipses. The circles centers were chosen uniformly at random in the range $[0, 1000] \times [0, 1000]$ and their radii were chosen uniformly at random in the range [0, 250]. The ellipses were chosen in a similar manner, with their axes lengths chosen independently at random in the range [0, 250].

The other arrangements in the benchmark are more structured. The forth arrangement, *robotics*, is a line-segment arrangement that was constructed by

computing the Minkowski sum[14] of a star-shaped robot and a set of obstacles. This arrangement consists of 25,533 edges. The *Norway* arrangement is also a line segment arrangement, that was constructed by computing the Minkowski sum of the border of Norway and a flower-shaped polygon with 23 edges. The resulting arrangement consists of 42,786 edges. The last arrangement, *low-E*, is the arrangement representing the lower envelope[15] of 3D triangles, which contains 6632 edges.

For each arrangement, we selected 1000 random query points in the arrangement's bounding rectangle, to be located in the arrangement. For the comparison between the various algorithms, we measured the preprocessing time, the average query time, and the memory usage of the algorithms. All algorithms were executed on the same set of arrangements and same sets of query points.

Several point-location algorithms were studied. We tested the different variants of the landmarks algorithm: LM(vert), LM(rand), LM(grid), LM(halton), and LM(mide) (see Section 3.1). The number of landmarks used in the LM(vert), LM(rand), LM(grid), and LM(halton) is equal to the number of vertices of the arrangement. The number of landmarks used in the LM(mide) is equal to the number of edges of the arrangement. All landmarks algorithms, besides LM(grid), used CGAL's KD trees as their nearest-neighbor search structure. We have also tested the ANN library, but the few tests that were made using the ANN library as the nearest-neighbor search structure show similar results to those using CGAL's KD-tree. Therefore, we do not include these tests in the results below.

We also used the benchmark to study the naïve algorithm, the walk (from infinity) algorithm, the RIC algorithm, and the triangulation algorithm (only for line segments). The LM(mide) was not implemented on conic-arc arrangements, since finding the midpoint of a conic arc connecting two vertices of the arrangement, which may have been constructed by intersection of two conic curves, is not a trivial operation, and the middle point may possibly be of high algebraic degree.

As stated above, we use exact computations and thus report in the exact point location. We use exact number types whenever they are needed to ensure the exactness of the algorithm (GMP for rational numbers and CORE [Karamcheti et al. 1999] for algebraic numbers).

The benchmark was conducted on a single 2.4-GHz PC with 1 GB of RAM, running under LINUX.

5.2 Results

5.2.1 *Comparing Point-Location Strategies.* Table I shows the average time it takes to answer a single point-location query in arrangements of varying types and sizes using the different point-location algorithms. The number of edges mentioned in these tables is the number of undirected edges of the arrangement. (In the CGAL implementation each edge is represented by two halfedges with opposite orientations.)

[14]The *Minkowski sum* of sets A and B is the set $\{a + b \mid a \in A, b \in B\}$.

[15]The representation of an upper/lower envelope as an arrangement is also called the *minimization diagram*. (See Sharir and Agarwal [1995] for more information.)

Table I. Average Time (in ms) for One Point-Location Query

Arrang.Type	#Edges	Naïve	Walk	RIC	Triang.	LM (vert)	LM (rand)	LM (grid)
	2112	2.2	0.8	0.06	0.86	0.16	0.13	0.13
Random	37,046	36.7	3.6	0.09	1.17	0.20	0.16	0.15
segments	235,446	241.4	9.7	0.12	1.96	0.38	0.35	0.18
(uniform)	955,866	1636.1	15.0	0.23	1.83	1.27	1.45	0.18
	1,366,364	2443.6	18.0	0.27	2.10	1.80	2.06	0.19
Random	2230	4.5	0.7	0.08	0.39	0.13	0.15	0.09
segments	59,816	92.7	1.4	0.07	0.56	0.60	0.48	0.11
(normal)	247,350	397.9	2.9	0.09	0.86	2.07	1.88	0.13
Random	1001	1.4	0.2	0.05	N/A	0.31	0.08	0.07
conics	3418	5.6	0.5	0.07	N/A	0.32	0.07	0.06
	13,743	21.7	1.1	0.09	N/A	0.38	0.07	0.07
Robotics	25,533	37.6	1.3	0.08	0.39	0.12	0.11	0.07
Norway	42,786	65.7	0.9	0.10	0.52	0.15	0.15	0.08
Low-E	6632	17.2	2.2	0.17	0.79	0.32	0.26	0.18

Table II. Preprocessing Time (in s)

Arrang.Type	#Edges	Arrang. Construct.	RIC	Triang.	LM (vert)	LM (rand)	LM (grid)
	2112	0.1	0.5	11.2	0.1	0.1	0.1
Random	37,046	1.3	29.7	360.2	0.1	2.9	2.9
segments	235,446	8.9	115.0	3360.1	0.3	24.2	22.2
(uniform)	955,866	60.5	616.5	21172.2	2.2	141.8	100.8
	1,366,364	97.6	1302.3	33949.1	3.4	212.8	148.6
Random	2230	0.2	1.2	3.8	0.1	0.2	0.2
segments	59,816	5.6	33.9	188.6	0.2	5.8	5.6
(normal)	247,350	22.9	183.4	1094.4	0.9	26.2	25.8
Random	1001	8.2	2.2	N/A	0.1	0.2	0.2
conics	3418	29.2	6.1	N/A	0.1	0.6	0.8
	13,743	127.1	28.2	N/A	0.1	2.7	3.5
Robotics	25,533	2.6	8.3	34.7	0.1	1.7	0.3
Norway	42,786	5.3	20.1	70.3	0.1	3.2	2.3
Low-E	6632	2.3	6.3	23.4	0.1	1.3	1.2

Table II shows the preprocessing time for the same arrangements and same algorithms as in Table I. The preprocessing consist of two parts: Construction of the arrangement (common to all algorithms) and construction of auxiliary data structures needed for the point location, which are algorithm specific. As mentioned above, the naïve and the walk strategies do not require any specific preprocessing stage besides constructing the arrangement and, therefore, do not appear in the table.

Table III shows the memory usage of the point-location strategies for the same arrangements and same algorithms as in Tables I and II. The memory used by the arrangement data structure before using any point-location strategy is also presented in this table (Column 3). Columns 4–8 show the memory used by the different point-location algorithms. As in the previous table, the naïve and the walk strategies do not appear in the table, since they do not require any data structure besides the basic arrangement representation.

Table III. Memory Usage (in MB) by the Point-Location Data Structures

Arrang. Type	#Edges	Arrang. Size	RIC	Triang.	LM (vert)	LM (rand)	LM (grid)
	2112	0.8	1.3	0.3	0.2	0.5	0.5
Random	37,046	9.5	21.5	7.7	2.6	8.1	6.8
segments	235,446	57.3	136.5	46.4	17.0	51.9	44.4
(uniform)	955,866	231.3	555.0	206.1	55.8	208.5	178.1
	1,366,364	333.8	793.2	268.9	86.8	307.0	258.9
Random	2230	0.6	1.3	0.3	0.1	0.5	0.4
segments	59,816	13.3	38.8	11.6	2.5	12.5	10.7
(normal)	247,350	54.8	161.9	47.9	10.8	52.0	44.4
Random	1001	3.1	2.6	N/A	0.1	0.6	1.0
conics	3418	10.2	9.0	N/A	0.3	2.1	2.6
	13,743	40.6	3.7	N/A	1.4	8.4	11.3
Robotics	25,533	7.2	14.9	3.7	1.3	3.9	3.1
Norway	42,786	12.2	24.4	5.9	1.9	6.2	4.7
Low-E	6632	1.8	4.3	0.7	0.2	1.3	1.0

The information presented in these tables shows that, unsurprisingly, the naïve and the walk strategies, although they do not require any preprocessing stage and any memory besides the basic arrangement representation, result with the longest query time in most cases, especially in case of large arrangements.

The triangulation algorithm has the worst preprocessing time, which is mainly because of the time required for subdividing the faces of the arrangement using CDT; this implies that resorting to CDT is probably not the way to go for point location in arrangements of segments. The query time of this algorithm is quite fast, since it uses the Delaunay hierarchy, although it is not as fast as the RIC or the LM(grid) algorithm.

The RIC algorithm results with fast query time, but it consumes the largest amount of memory, and its preprocessing stage is very slow. The memory usage of the RIC algorithm is its major drawback. For an arrangement of about million edges, the RIC data structure (along with the arrangement data structure) already consumes about 1 GB, which, at the time of writing, is, in many cases, the size of the RAM memory of a home PC.

All the landmarks algorithms have rather fast preprocessing time and fast query time. The LM(vert) has by far the fastest preprocessing time, since the location of the landmarks is known, and there is no need to locate them in the preprocessing stage. The LM(grid) has the fastest query time for large-size arrangements induced by both random line-segments (both uniform and normal distributed) and random conic arcs, for the reason that the location of the closest grid point is carried out in constant time. The size of the memory used by the LM(vert) algorithm is the smallest of all algorithms. This is because the vertices are already part of the arrangement, as opposed to random or grid points, for which we need to allocate additional memory.

The other two variants of landmarks that were examined but are not reported in the tables are: (1) the LM(halton), which yields similar results to those of the LM(rand), and (2) the LM(mide) which yields similar results to those of

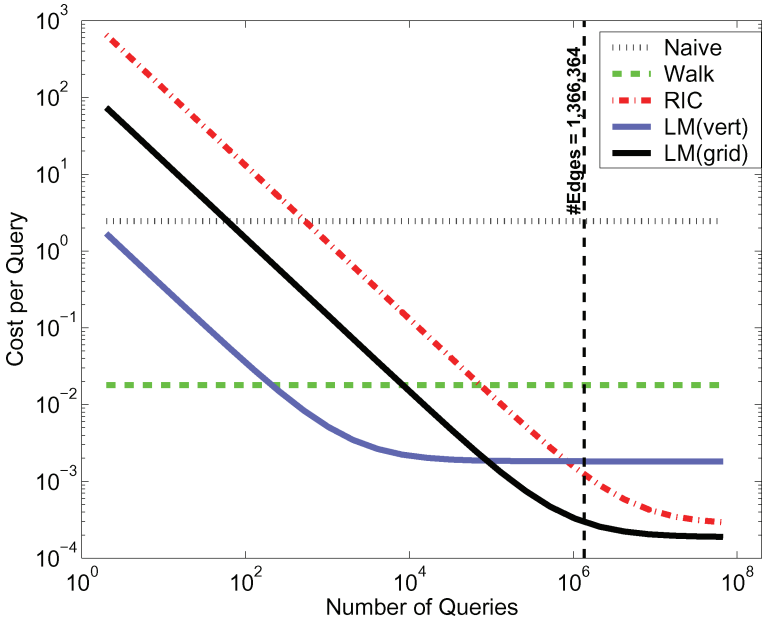


Fig. 8. The average combined (amortized) cost per query in a large arrangement, with 1,366,384 edges.

the LM(vert), although since it uses more landmarks, it has a little longer query and preprocessing stages, which makes it less efficient for these types of arrangements.

Figure 8 presents the combined cost of a query (amortizing also the preprocessing time over all queries) on the last random-segments arrangement shown in the tables, which consists of more than 1,350,000 edges. The x axis indicates the number of queries m. The y axis indicates the average amortized cost-per-query, $cost(m)$, which is calculated in the following manner:

$$cost(m) = \frac{\text{preprocessing time}}{m} + \text{average query time}. \tag{2}$$

We can see that when m is small, the cost is a function of the preprocessing time of the algorithm. Clearly, when $m \to \infty$, $cost(m)$ becomes the query time. For the naïve and the walk algorithms, which do not require preprocessing, $cost(m) = query\ time = constant$. Looking at the lower envelope of these graphs, we can see that for $m < 100$ the walk algorithm is the most efficient. For $100 < m < 100{,}000$ the LM(vert) algorithm is the most efficient and, for $m > 100{,}000$, the LM(grid) algorithm gives the best performance. As we can see, for each number of queries, there exists a landmarks algorithm, which is better than the RIC algorithm.

5.2.2 *Analysis of the Landmarks Strategy.* As mentioned above, there are various parameters that affect the performance of the landmarks algorithm, such as the number of landmarks, their distribution over the arrangement,

Table IV. LM(rand) Algorithm Performance for a Fixed Arrangement of Segments with 1,366,384 Edges and a Varying Number of Random Landmarks

Number of Landmarks	Preprocessing Time [s]	Query Time [ms]	% Queries with AD = 0
100	61.7	4.93	3.4
1000	59.0	1.60	7.6
10,000	60.8	0.58	19.2
100,000	74.3	0.48	42.3
1,000,000	207.2	3.02	71.9

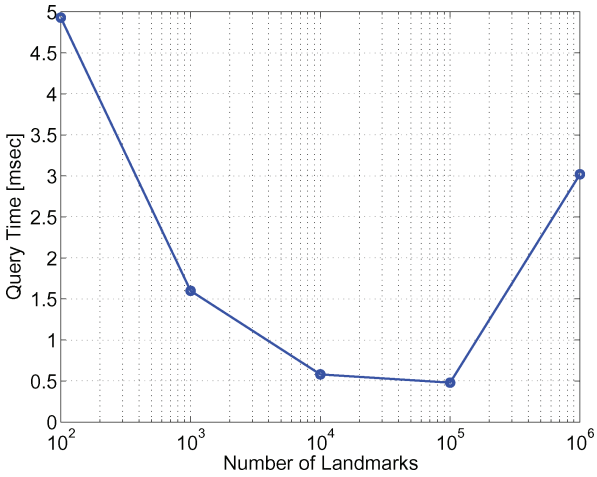


Fig. 9. The query time (ms) for one point-location query using the LM (rand) algorithm for a fixed arrangement of segments with 1,366,384 edges and a varying number of random landmarks.

and the structure used for the nearest-neighbor search. We checked the effect of varying the number of landmarks on the performance of the algorithm, using several random arrangements.

Table IV shows typical results, obtained for the largest random-segments arrangement of our benchmark. The landmarks used for these tests were random points sampled uniformly in the bounding rectangle of the arrangement. As expected, increasing the number of random landmarks increases the preprocessing time of the algorithm. However, as also shown in Figure 9, the query time decreases only until a certain minimum around 100,000 landmarks and it is much larger for 1,000,000 landmarks. The reason that the query time increases for a very large number of landmarks is that the time for finding the closest landmark to the query points becomes significant as the number of landmarks grows. The last column in the table shows the percentage of queries, where the chosen startup landmark was in the same face as the query point. As expected, this number increases with the number of landmarks.

An in-depth analysis of the duration of the landmarks algorithm reveals that the major time-consuming operations vary with the size of the arrangement

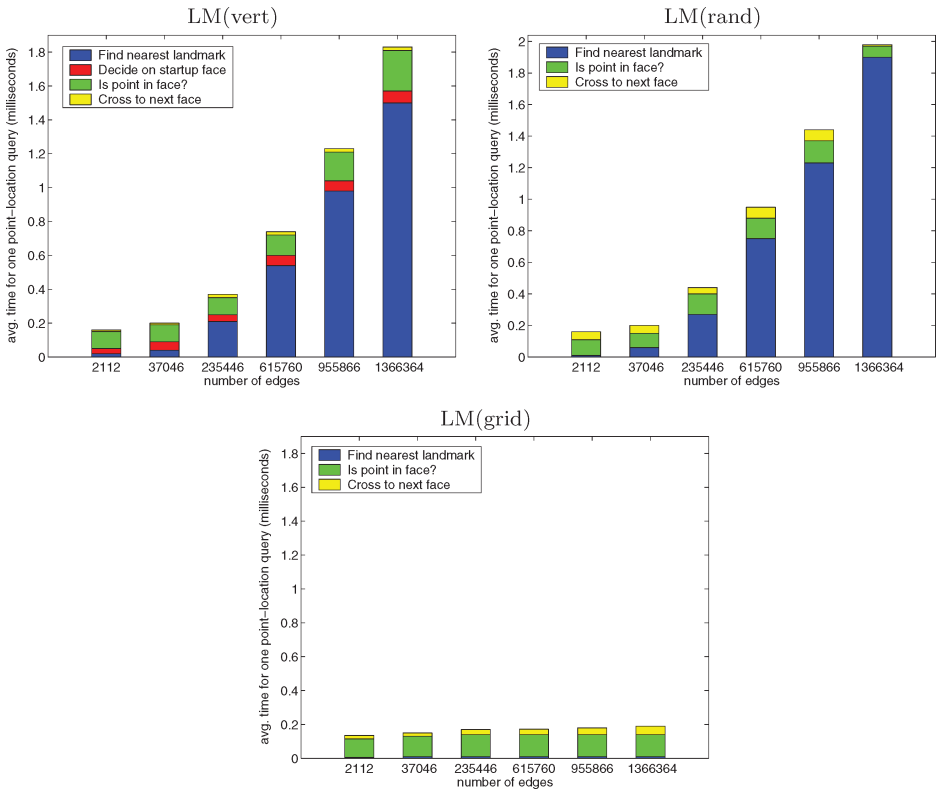


Fig. 10. The average breakdown of the time required by the main steps of the landmarks algorithms in a single point-location query, for arrangements of varying size.

(and consequently, the number of landmarks used), and with the landmarks type used. Figure 10 shows the duration percentages of the various steps of the query operation, in the LM(vert), LM(rand), and LM(grid) algorithms. As can be seen in the LM(vert) and LM(rand) diagrams, the nearest-neighbor search part increases when more landmarks are present and becomes the most time-consuming part in large arrangements. The results indicate that the dependence of the specific KD tree implementation used in this work on the number of points is stronger than the expected theoretical $O(\log n)$ dependence. As stated above, the search algorithm can be replaced by a more efficient one. In the LM(grid) algorithm, this step is negligible.

A significant step that is common to all landmarks algorithms, checking whether the query point is in the current face, also consumes a significant part of the query time. This part consumes a major portion of the LM(grid) algorithm runtime.

An additional operation shown in the LM(vert) diagram is finding the startup face in a specified direction. This step is relevant only in the LM(vert) and the LM(mide) algorithms. The last operation, crossing to the next face, is relatively short in LM(vert), as in most cases (more than 90%) the query point is found to

Table V. Average Time (in ms) for One Point-Location Query in the Delaunay Triangulation or the Corresponding Arrangement

#Random Points	Triangulation Package	LM (vert)	LM (rand)	LM (grid)	LM (halton)	LM (mide)
1000	0.278	0.073	0.117	0.106	0.110	0.067
100,000	0.458	0.198	0.269	0.083	0.257	0.449
350,000	0.597	0.577	0.752	0.078	0.710	1.580

be inside the startup face. This step is a little longer in LM(grid) and LM(rand) than in LM(vert), since only about 70% of the query points are found in the same face as the landmark point.

5.3 Comparison with Point-Location Algorithms in CGAL's Triangulations

Many applications that use triangulations in their implementation and, in particular, Delaunay triangulations and their dual data structures Voronoi diagrams, also require intensive use of point-location queries. A lot of effort (see Section 2) has been invested in speeding up the point location in these structures. We have tested the performance of our landmarks strategy on triangulations and compared it with the location time achieved by the triangulation package in CGAL. The strategy used with the triangulation package is the one that employs the Delaunay hierarchy (as explained in Section 2).

In the first test, we chose a set of random points and created the triangulation of these points. We converted the triangulation into an arrangement, so that each triangle is a face in the arrangement. We then created a new set of 1000 random query points, located them in the triangulation using the Delaunay hierarchy, and in the arrangement using our landmarks strategy. Table V shows the results of this test. We can see that the location using LM(grid) is much faster than the location in the triangulation package. As mentioned, this strategy is the most efficient here since the nearest landmark (grid) point can be found in constant time. The other landmarks strategies sometimes perform better and sometimes worse than using the point location in the triangulation package. The LM(vert) is relatively efficient in this case, since it probably represents the triangulation well, while LM(rand) and LM(halton) may require a longer "walk" from the landmark to the query point. The LM(mide) contains more landmarks (equal to the number of edges) and thus requires a longer nearest-neighbor search.

In the second test, we chose random segments and created the CDT of these segments. We then chose, as before, 1000 random query points and located them in the arrangement using the landmarks algorithm and in the triangulation using the Delaunay hierarchy. Table VI shows the results of this test. We can see that all the landmarks strategies perform better than the CDT package.

In the third test, we selected random points and built the Voronoi diagram of these points using the new CGAL package, which is an adaptor over CGAL's Delaunay triangulations. We converted the Voronoi diagram into an arrangement. For each diagram (and corresponding arrangement), we used two sets of query points (with the same number of points). The first set contains the sites that were used for building the Voronoi diagram. The points in the second

Table VI. Average Time (in ms) for One Point-Location Query in the Constrained Delaunay Triangulation or the Corresponding Arrangement

#Random Segments	CDT Package	LM (vert)	LM (rand)	LM (grid)	LM (halton)	LM (mide)
400	1.065	0.167	0.191	0.142	0.169	0.184
600	1.135	0.199	0.225	0.138	0.211	0.282
800	1.364	0.251	0.282	0.143	0.255	0.418
1000	1.358	0.319	0.348	0.146	0.332	0.625

Table VII. Average Time (in ms) for One Point-Location Query in the Voronoi Diagram or the Corresponding Arrangement

#Random Sites	Queries	Voronoi diag.	LM (vert)	LM (rand)	LM (grid)	LM (halton)	LM (mide)
2000	the sites	0.812	0.159	4.17	10.30	6.51	0.132
2000	random	1.722	0.174	4.20	10.06	6.50	0.147
4000	the sites	1.766	0.160	3.58	16.54	7.89	0.132
4000	random	1.716	0.185	9.03	17.09	7.86	0.153

set are random points that were selected uniformly at random inside the same bounding rectangle from which the original sites were selected.

Table VII shows the location time in the Voronoi diagram and in the arrangement using the landmarks strategies. The results indicate that the LM(vert) and LM(mide) perform much better than the point location in the Voronoi diagram. The other landmarks strategies perform poorly on these arrangements. Looking closely at the Voronoi diagram, we observe that the Voronoi diagram's bounding rectangle is very large and most cells of the arrangement are located in a small area compared to the bounding rectangle (since there are few vertices very far from the others). Therefore, the landmarks that are not based on the arrangement's entities (i.e. LM(rand), LM(grid), and LM(halton)) and are chosen uniformly do not represent the arrangement well, and the walk from these landmarks to the query points (that are located in the dense area of the arrangement) takes a lot of time, since it requires crossing many faces.

Figure 11 aims to clarify this point. The figure presents a Voronoi diagram and several landmarks on a grid. One can see that a Voronoi diagram is very sparse in most places and very dense only in a small area. If we now take the bounding rectangle of this arrangement (its finite entities bounding rectangle), and choose landmarks uniformly (uniform grid, random or halton points), we would probably get only few landmarks in the dense and, hence interesting, part of the arrangement. Now, if we take the query points to be in the dense area (such as the sites themselves), the algorithm will look for the closest landmark to each query point. Since there are only few landmarks in this area, the one that would be found close to the query point would be quite far in the "arrangement distance" measure defined above. Therefore, the walk part would have to cross many edges from the landmark to the query point (up to $O(n)$ edges). Obviously, the query time of the algorithms using such landmarks (LM(grid), LM(rand), and LM(halton)) would be long. It is clear that in such arrangements, that are

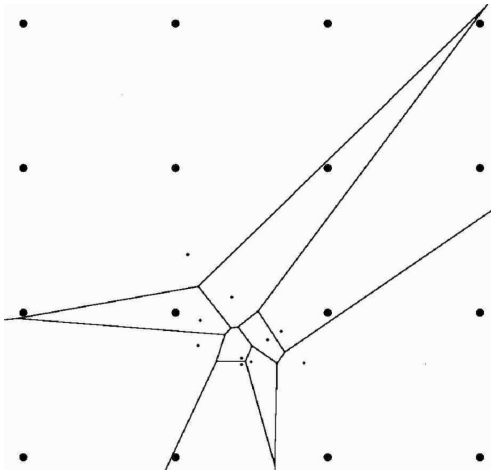


Fig. 11. A Voronoi diagram of ten random points and grid landmarks spread on the diagram's bounding rectangle (its finite entities bounding rectangle). The small points represent the Voronoi sites and the large points are the landmarks.

known to be nonuniform, it is better to use the vertices of the arrangement or the middle points of the edges as landmarks.

6. CONCLUSIONS AND FUTURE WORK

We have proposed a new landmarks algorithm for exact point location in general planar arrangements and have integrated the implementation of our algorithm into CGAL. We use generic programming, which allows for the adjustment and extension for any type of planar arrangements. We tested the performance of the algorithm on arrangements constructed of different types of curves, i.e., line segments and conic arcs, and compared it with other point-location algorithms.

The main conclusion from our experiments is that, when choosing an appropriate type of landmark, the landmarks algorithm is the best strategy, among CGAL's strategies, considering the cost per query, which takes into account both (amortized) preprocessing time and query time. Moreover, the memory space required by the algorithm (no matter what kind of landmarks are used) is small compared to other algorithms that use auxiliary data structures for point location. The algorithm is easy to implement, maintain, and adjust for different needs using different types of landmarks and search structures.

Given an arrangement, in order to decide what kind of landmarks should be used, we should consider the distribution of the arrangement in the plane. Generally we would say that if the arrangement is known to be uniformly (or even normally) distributed inside some bounding rectangle, the landmarks that give the best results are those on a grid. On the other hand, if the arrangement is very sparse in some areas and dense in others, with major differences between those areas, we would recommend using the arrangement's vertices as landmarks, since they represent the arrangement best.

Although the tests were made on curves of first and second degree only, we have reason to believe that when using curves of higher order, the landmarks algorithm will still be relatively efficient. This is because the algorithm uses approximate calculations whenever possible and requires exact predicates only for the last, and relatively short, part of the computation (the "walk" part).

There are still many ways to improve the algorithm. For example, one can make the landmarks generator dynamic. That is, implement a data structure that will not have to be rebuilt every time a new curve is added to or deleted from the arrangement. Another improvement may come from other types of landmarks, for example, by combining different types of landmarks, or by subdivision of the faces of the arrangement into simple cells and putting a landmark in each of these cells.

Another question is to determine the optimal number of landmarks. We have seen that the larger the number of landmarks, the probability that the query point will be in the same cell as the landmark is higher. However, when the number of landmarks is large the time it takes to find the closest landmark to a query point becomes significant. Thus, it may be instructive to look for the optimal number of landmarks versus the arrangement type and size.

Another aspect that was not included in the current work is a theoretical analysis of the algorithm. It may be interesting to estimate, for example, the average time it should take to locate a query point in an arrangement of random segments. For this purpose one may first analyze the probability that the closest landmark lies in the same cell of the arrangement as the query point.

Finally, the landmarks algorithm can be improved to handle arrangements with large faces. The main drawback of the landmarks strategy is that when the arrangement contains very large faces and we want to decide if the query point is inside a face, we have to go over all the edges of the face's boundary. In case a face f is very large, it may be useful to maintain an extra structure for ray shooting inside f. Such a structure should answer queries of the form: Given a query point q and a direction d, find the first edge on f's boundary that the ray emerging from q in direction d hits. Using such a structure inside large faces can decrease the time for testing whether a query point is inside the face and, if not, the edge that it hits on the boundary of f can be used for crossing to the next face during the walk algorithm. Of course, we will have to pay in increased preprocessing time and storage space.

REFERENCES

AGARWAL, P. K. AND SHARIR, M. 2000. Arrangements and their applications. In *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds. Elsevier Science Publi. North-Holland, Amsterdam. 49–119.

ARYA, S. AND MOUNT, D. M. 1993. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*. 271–280.

ARYA, S., MOUNT, D. M., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. 1998. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM 45*, 891–923.

ARYA, S., MALAMATOS, T., AND MOUNT, D. M. 2001a. Entropy-preserving cutting and space-efficient planar point location. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*. 256–261.

ARYA, S., MALAMATOS, T., AND MOUNT, D. M. 2001b. A simple entropy-based algorithm for planar point location. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*. 262–268.

AUSTERN, M. H. 1999. *Generic Programming and the STL*. Addison-Wesley, Reading, MA.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM 18,* 9 (Sept.), 509–517.

BOISSONNAT, J.-D., DEVILLERS, O., PION, S., TEILLAUD, M., AND YVINEC, M. 2002. Triangulations in CGAL. *Comput. Geom. Theory Appl. 22,* 1–3, 5–19.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 2000. *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer-Verlag, New York.

DEVILLERS, O. 2002. The Delaunay hierarchy. *Intern. J. Found. Comput. Sci. 13*, 163–180.

DEVILLERS, O. AND GUIGUE, P. 2001. The shuffling buffer. *Intern. J. Comput. Geom. Appl. 11*, 555–572.

DEVILLERS, O., PION, S., AND TEILLAUD, M. 2002. Walking in a triangulation. *Intern. J. Found. Comput. Sci. 13*, 181–199.

DEVROYE, L., MÜCKE, E. P., AND ZHU, B. 1998. A note on point location in Delaunay triangulations of random points. *Algorithmica 22*, 477–482.

DEVROYE, L., LEMAIRE, C., AND MOREAU, J.-M. 2004. Expected time analysis for Delaunay point location. Comput. Geom. Theory Appl. *29*, 2, 61–89.

DOBKIN, D. P. AND LIPTON, R. J. 1976. Multidimensional searching problems. *SIAM J. Comput. 5,* 2, 181–186.

EDAHIRO, M., KOKUBO, I., AND ASANO, T. 1984. A new point-location algorithm and its practical efficiency—comparison with existing algorithms. *ACM Trans. Graph. 3*, 86–109.

EDELSBRUNNER, H., GUIBAS, L. J., AND STOLFI, J. 1986. Optimal point location in a monotone subdivision. *SIAM J. Comput. 15,* 2, 317–340.

FLATO, E., HALPERIN, D., HANNIEL, I., NECHUSHTAN, O., AND EZRA, E. 2000. The design and implementation of planar maps in CGAL. *ACM J. Exp. Algorithmics 5*. Special Issue, selected papers of the Workshop on Algorithm Engineering (WAE).

FOGEL, E., WEIN, R., AND HALPERIN, D. 2004. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA)*. LNCS, vol. 3221. Springer-Verlag, New York. 664–676.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns—Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

HALPERIN, D. 2004. Arrangements. In *Handbook of Discrete and Computational Geometry*, 2nd ed., J. E. Goodman and J. O'Rourke, Eds. Chapman & Hall/CRC, Boca Raton, FL. 529–562.

KARAMCHETI, V., LI, C., PECHTCHANSKI, I., AND YAP, C. 1999. A Core library for robust numeric and geometric computation. In *Proceedings of the 15th Annual Symposium on Computational Geometry.* 351–359.

KIRKPATRICK, D. G. 1983. Optimal search in planar subdivisions. *SIAM J. Comput. 12,* 1, 28–35.

LAVALLE, S. M. 2006. *Planning Algorithms*. Cambridge University Press Cambridge. Also available at http://msl.cs.uiuc.edu/planning/).

MATOUŠEK, J. 1999. *Geometric Discrepancy—An Illustrated Guide*. Springer, New York

MEHLHORN, K. AND NÄHER, S. 2000. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK.

MÜCKE, E. P., SAIAS, I., AND ZHU, B. 1996. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry.* 274–283.

MULMULEY, K. 1990. A fast planar partition algorithm, I. *J. Symbolic Comput. 10,* 3-4, 253–280.

MYERS, N. 1997. A new and useful template technique: "Traits". In *C++ Gems*, S. B. Lippman, Ed. SIGS Reference Library, vol. 5. 451–458.

NIEDERREITER, H. 1992. *Random Number Generation and Quasi-Monte Carlo Methods*. Regional Conference Series in Applied Mathematics, vol. 63. CBMS-NSF.

PREPARATA, F. P. 1981. A new approach to planar point location. *SIAM J. Comput. 10,* 3, 473–482.

SARNAK, N. AND TARJAN, R. E. 1986. Planar point location using persistent search trees. *Commun. ACM 29,* 7 (July), 669–679.

SCHIRRA, S. 2000. Robustness and precision issues in geometric computation. In *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds. Elsevier Science Publ. North-Holland, Amsterdam. 597–632.

SEIDEL, R. 1991. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl. 1,* 1, 51–64.

SHAMOS, M. I. 1975. Geometric complexity. In *Proceedings of the 7th ACM Symposium on Theory of Computing*. 224–233.

SHARIR, M. AND AGARWAL, P. K. 1995. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, Cambridge.

SNOEYINK, J. 2004. Point location. In *Handbook of Discrete and Computational Geometry*, 2nd ed., J. E. Goodman and J. O'Rourke, Eds. Chapman & Hall/CRC, Boca Raton. Chapter 34, 529–562.

TANGELDER, H. AND FABRI, A. 2006. dD spatial searching. In CGAL*-3.2 User and Reference Manual*, CGAL Editorial Board, Ed. `http://www.cgal.org/Manual/3.2/doc_html/cgal_manual/Spatial_searching/Chapter_main.html`.

WEIN, R., FOGEL, E., ZUKERMAN, B., AND HALPERIN, D. 2007. Advanced programming techniques applied to CGAL's arrangement package. *Comput. Geom. Theory Appl. 38,* 1-2, 37–63.

YAP, C. 2004. Robust geometric computation. In *Handbook of Discrete and Computational Geometry*, 2nd ed., J. E. Goodman and J. O'Rourke, Eds. Chapman & Hall/CRC, Boca Raton, FL. 927–952.

Received June 2006; revised March 2007 and January 2008; accepted May 2008