An Experimental Study of Point Location in General Planar Arrangements*

Idit Haran[†]   Dan Halperin[†]

Abstract

We study the performance in practice of various point-location algorithms implemented in CGAL, 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 arbitrary planar curves, including arrangements of non-linear segments (e.g., conic arcs) and allows for degenerate input (for example, more than two curves intersecting in a single point, or overlapping curves). All calculations use exact number types 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 then the algorithm "walks" from the landmark to the query point. We report on extensive experiments with arrangements composed of line segments or conic arcs. The results indicate that the Landmarks approach is the most efficient when the overall 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 a highly effective point-location algorithm for most practical purposes.

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 maximal connected cells. The cells can be 0-dimensional (*vertices*), 1-dimensional (*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 (see, e.g., [5, 18].) Figure 1 shows two arrangements of different types of curves, one induced by line segments and the other by conic arcs.[1] The planar point-location problem is one of the most fundamental problems applied to arrangements: Preprocess an arrangement into a data structure, so that given any query point q, the cell of the arrangement containing q can be efficiently retrieved.

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 real-time performance of point-location queries. On the other hand, if the arrangement is dynamic, and new curves are inserted to it (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.

A naïve approach to point location might be traversing over 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, both in the average and worst-case scenarios.

A more economical approach [25] is to draw a vertical line through every vertex of the arrangement to obtain vertical *slabs* in which point location is almost one-dimensional. Then, two binary searches suffice to answer a query: one on x-coordinates for the slab containing q, and one on

[†]School of Computer Science, Tel-Aviv University, 69978, Israel. {`haranidi,danha`}`@post.tau.ac.il`

---

[1]A *conic curve* is an algebraic planar curve of degree 2. A *conic arc* is a bounded segment of a conic curve.
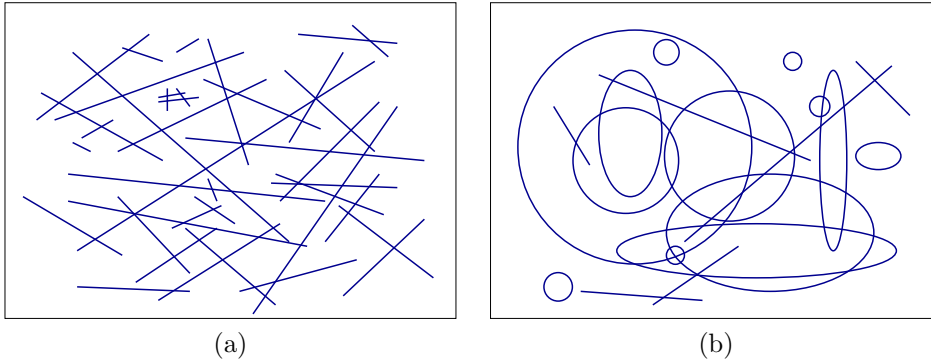
Figure 1: Random arrangements of line segments (a) and of conic arcs (b).

edges that cross the slab. Query time is $O(\log n)$, but the space may be quadratic. In order to reduce the space to linear storage space, Sarnack and Tarjan [26] used Persistent Search Trees. Edahiro et al. [15] used these ideas 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. In each bucket the local point location is performed using the slabs algorithm described above.

Another approach aiming at worst-case query time $O(\log n)$ was proposed by Kirkpatrick [19], using a data structure of size $O(n)$. Mulmuley [23] and Seidel [27] proposed an alternative method that uses the vertical decomposition of the arrangement into pseudo-trapezoidal cells, and constructs a search Directed Acyclic Graph (DAG) over these simple cells. We refer to the latter algorithm, which is based on Randomized Incremental Construction, as the *RIC* algorithm.

Point location in Delaunay triangulations was extensively studied: Early works on point location in triangulations can be found in [21] and [22]. Devillers et al. [12] proposed a *Walk along a line* algorithm, which does not require the generation of additional data structures, and offers $O(\sqrt{n})$ query time on the average ($O(n)$ in the worst case). The walk may begin at an arbitrary vertex of the triangulation, and advance towards the query point. Due to the simplicity of the structures (triangles), the walk consists of low-cost operations. Devillers later proposed a walk strategy based on a Delaunay hierarchy [10], which uses a hierarchy of triangles, 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, until the triangle in the lowest level is found. Other algorithms that were developed only for Delaunay triangulations, often referred to as *Jump & Walk* algorithms, were proposed by Devroye et al. [13, 14].

Arya et al. [6] devised point location algorithms aiming at good average (rather than worst-case) query time. The efficiency of these algorithms is measured with respect to the entropy of the arrangement.

The algorithms presented in this paper are part of the arrangement package in CGAL, the Computational Geometry Algorithms Library [1]. CGAL 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. Robustness of the algorithms is achieved by both handling all degenerate cases, and by using exact number types. CGAL's arrangement package was the first generic software implementation, designed for constructing arrangements of arbitrary planar curves and supporting operations and queries on such arrangements [16, 17]. 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 for the computations they perform. Among the number-type libraries that are used are GMP- Gnu's multi-precision library [4], for rational numbers, and CORE [2] and LEDA [3] for algebraic numbers.

Point location constitutes a significant part of the arrangement package, 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 the CGAL's arrangement package: The *Naïve* strategy traverses all vertices and edges, and locates the nearest edge or vertex that is situated exactly on, or immediately above, the query point. The *Walk* algorithm traces (in reverse order) a vertical ray r emanating from

the query point to infinity; it traverses the *zone*[2] 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 with non-linear curves. Both the Naïve and the Walk strategies maintain no data structures, beyond the basic representation of the arrangement, and do not require any preprocessing stage. Another point-location strategy implemented in CGAL for line-segments arrangement is a triangulation algorithm, which consists of a preprocessing stage where the arrangement is refined using a Constrained Delaunay Triangulation. In the triangulation, point location is implemented using a triangulation hierarchy [10]. The algorithm uses the triangulation package of CGAL [9]. The RIC point-location algorithm described above was also implemented in CGAL [16].

The motivation behind the development of the new, Landmarks, algorithm, was to address both issues of preprocessing complexity and query time, something that none of the existing strategies do well. 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 to or removed from, this is a major drawback. Moreover, in real-life applications the curves are typically inserted to the arrangement in non-random order. This reduces the performance of the RIC algorithm, as it relies on random order of insertion, unless special procedures are followed [11].

In the *Landmarks* algorithm, special points, which we call "landmarks", are chosen in a preprocessing stage, 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 towards the query point. This walk part differs from other walk algorithms that were tailored for triangulations (especially Delaunay triangulations), as it is geared towards 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 is arbitrary, rather than vertical. Tests that were carried out using the Landmarks

algorithm, reported in Section 3 indicate that the Landmarks algorithm has relatively short preprocessing stage, and it answers queries fast.

The rest of this paper is organized as follows: Section 2 describes the Landmarks algorithm in details. Section 3 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 4.

2 Point Location with Landmarks

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 towards the query point. The key motivation 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 relies on three independent components, each of which can be optimized or replaced by a different component (of the same functionality):

1. Choosing the landmarks that faithfully represent the arrangement, and locating them in the arrangement.

2. Constructing a data structure that supports nearest-neighbor search (such as a kd-trees [8]), and using this structure to find the nearest landmark given a query point.

3. Applying a "walk along a line" procedure, moving from the landmark towards the query point.

The following sections elaborate on these components.

2.1 Choosing the Landmarks. When choosing the landmarks we aim to minimize the expected length of the "walk" inside the arrangement towards 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.

---

[2]The *zone* of a curve is the collection of all the cells in the arrangement that the curve intersects.

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 usage. Indeed, 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 3.3 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 metric representing the complexity of the walk stage: The *arrangement distance* (AD) between two points is the number of faces crossed by the straight line segment that connects these points. If two points 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.

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 as landmarks, points along the edges (e.g., the edges midpoints), or interior points in the faces. In order to choose representative points inside the faces, it may be useful to preprocess the arrangement faces, which are possibly non-convex, for example using vertical decomposition or triangulation.[3] Such pre-processing will result in simple faces (pseudo trapezoids 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 inside 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 [20, 24]. Each choice has its advantages and disadvantages and improved performance may be achieved using combinations of different types of landmark choices.

In the current implementation the landmark 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, *LM(mide)* – midpoints of all the arrangement edges are chosen, *LM(rand)* – random points are selected, *LM(grid)* – the landmarks are chosen on a uniform grid, and *LM(halton)* – Halton sequence points are used. In the LM(rand), LM(grid) and LM(halton) the number of landmarks is given as a parameter to the generator, and is set to be the number of vertices by default. The benefit of using vertices or edge's midpoints 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. Walking from the midpoints of the edges also requires a small preparatory step to choose between the two faces incident to the edge.

For random landmarks, we use uniform samples inside the arrangement bounding-rectangle. 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, 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-structure to find the edge that is just above the point. Similar preprocessing is conducted on the uniform grid, when the grid points are used as landmarks, and also on the Halton 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, a preparatory step is scarcely required at the beginning of the walk stage.

2.2 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. The search structure should allow for fast preprocessing and query. A search structure that supports approximate nearest-neighbor search can also be suitable, 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 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

---

[3]Triangulation is relevant only in case of arrangements of line segments.
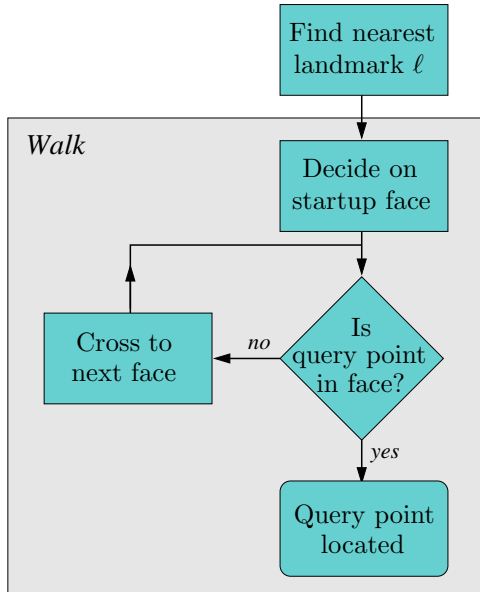
Figure 2: The query algorithm diagram.

the CGAL's spatial searching package, 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.

Another implementation uses the ANN package [7], which supports data structures and algorithms for both exact and approximate nearest neighbor searching. The library implements a number of different data structures, based on kd-trees and box-decomposition trees, and employs a couple of different search strategies. Few tests that were made using this package show similar results to those using CGAL's kd-tree.

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

2.3 Walking from the Landmark to the Query Point. The "walk" algorithm developed as part of this work is geared towards general arrangements, which may contain faces of arbitrary topology and of unbounded (not necessarily constant) complexity. This is different from previous Walk algorithms that were tailored for triangulations, especially the Delaunay triangulation.

The "walk" stage is summarized in the diagram in Figure 2. First, the startup face must be determined. As explained in the previous section, certain types of landmarks (vertices, edges) are not associated with a single startup face. A virtual line segment s is then drawn from the landmark (whose

location in the arrangement is known) to the query point q. Based on the direction of s, the startup face f out of the faces incident to the landmark is associated with the landmark.

Then, a test whether the query point q lies inside f is applied. This operation requires a pass over all the edges on the face boundary. This pass is quick, since we only count the number of f's edges above q. We first check if the point is in the edge's x-range. If it is, we check the location of q with respect to the edge, and count the edge only if the point is below it. If the number of edges above q is odd, then q is found to be inside f, and the query is terminated.

Otherwise, we continue our walk along the virtual segment s toward q. In order to walk along s, we need to find the first edge e on f's boundary that intersects s. Since the arrangement's data-structure holds for each edge the information of both faces incident to this edge, all we need is to cross to the face on the other side of e.

Figure 3 shows two examples of walking from a vertex type landmark towards the query point.

As explained above, crossing to the next face requires finding the edge e on the boundary of f that intersects s. Actually, 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 boundaries of this range. If the vertical order changes, it implies that the curves intersect; see, e.g., Figure 4(a). In case several edges on f's boundary intersects s, 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 4(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 non-linear curves, shown in Figure 4(c), is when e and s intersect an even number of times (two in this case), and thus no crossing is needed.

3 Experimental Results

3.1 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.

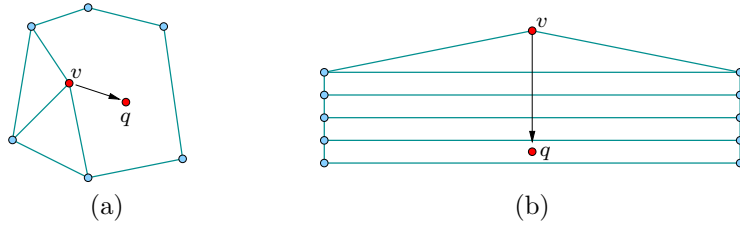The benchmark was conducted using four

(a)   (b)

Figure 3: Walking from a landmark located on a vertex v to a query point q: no crossing is needed (a), multiple crossings are required during the walk (b).

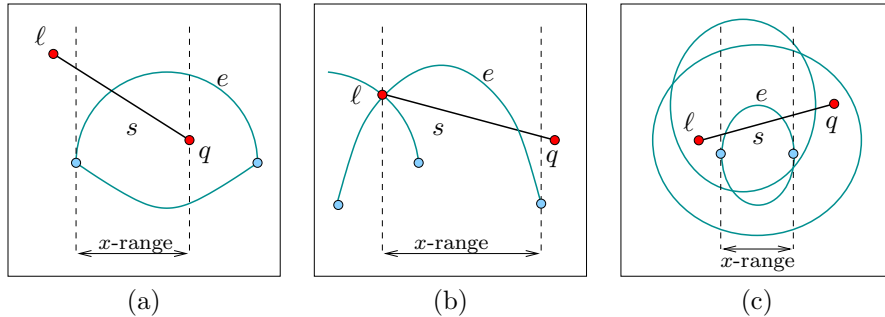(a)   (b)   (c)

Figure 4: Walk algorithms, crossing to the next face. In all cases the vertical order of the curves is compared on the left and right boundaries 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. (c) The y-order does not change, as s and e have an even number (two) of intersections.

types of arrangements: denotes as *random segments*, *random conics*, *robotics*, and *Norway*. Each arrangement in the first type was constructed by 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 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 in the range $[0, 250]$.

The third type, *robotics*, is a line-segment arrangement that was constructed by computing the Minkowski sum[4] of a star-shaped robot and a set of obstacles. This arrangement consists of 25,533 edges. The last type, *Norway*, is also a line-segment arrangement, that was constructed by computing the Minkowski sum of the border of

Norway and a polygon. The resulting arrangement consist of 42,786 edges.

For each arrangement we selected 1000 random query points 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 run 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). The number of landmarks used in the LM(vert), LM(rand), LM(grid), 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), use Cgal's kd-tree as their nearest neighbor search structure.

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 also not implemented on conic-arc arrangements, since finding the midpoint of a conic arc connecting two vertices of the arrangement, which

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

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, all calculations use exact number types, and result in the exact point location. The benchmark was conducted on a single 2.4GHz PC with 1GB of RAM, running under LINUX.

3.2 Results. Table 1 shows the average query time associated with point location 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.

Table 2 shows the preprocessing time for the same arrangements and same algorithms as in Table 1. The actual 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 3 shows the memory usage of the point-location strategies of the random line-segment arrangements from Tables 1 and 2.

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 due to the time for subdividing the faces of the arrangement using Constrained Delaunay Triangulation (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 Dalaunay hierarchy, although it is not as fast as the RIC or the Landmarks algorithm.

The RIC algorithm results with fast query time, but it consumes the largest amount of memory, and its preprocessing stage is very slow.

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 line-segments and conic-arcs. The size of the memory used by LM(vert) algorithm is the smallest of all algorithms.

The other two variants of landmarks that were examined but are not reported in the tables are (i) the LM(halton), which has similar results to that of the LM(rand), and (ii) the LM(mide) which yields similar results to those of the LM(vert), although since it uses more landmarks, it has a little longer query and preprocess, which makes it less efficient for these types of arrangement.

Figure 5 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:

$$cost(m) = \frac{\text{preprocessing time}}{m} + \text{average query time}$$
(3.1)

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 that 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.

3.3 Analysis. As mentioned in Sections 2 and 3, there are various parameters that effect the performance of the Landmarks algorithm, such as the number of landmarks, their distribution over the arrangement, 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 4 shows typical results, obtained for the last 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 in-

Arrang. Type	#Edges	Naïve	Walk	RIC	Triang.	LM (vert)	LM (rand)	LM (grid)
random segments	2112	2.2	0.8	0.06	0.86	0.16	0.13	0.13
	37046	36.7	3.6	0.09	1.17	0.20	0.16	0.15
	235446	241.4	9.7	0.12	1.96	0.38	0.35	0.18
	955866	1636.1	15.0	0.23	1.83	1.27	1.45	0.18
	1366364	2443.6	18.0	0.27	2.10	1.80	2.06	0.19
random conics	1001	1.4	0.2	0.05	N/A	0.31	0.08	0.07
	3418	5.6	0.5	0.07	N/A	0.32	0.07	0.06
	13743	21.7	1.1	0.09	N/A	0.38	0.07	0.07
robotics	25533	37.6	1.3	0.08	0.39	0.12	0.11	0.07
Norway	42786	65.7	0.9	0.10	0.52	0.15	0.15	0.08

Table 1: Average time (in milliseconds) for one point-location query.

Arrang. Type	#Edges	Construct. Arrangement	RIC	Triang.	LM (vert)	LM (rand)	LM (grid)
random segments	2112	0.07	0.5	11.2	0.01	0.12	0.13
	37046	1.26	29.7	360.2	0.05	2.97	2.95
	235446	8.90	115.0	3360.1	0.33	24.23	22.25
	955866	60.51	616.5	21172.2	2.25	141.88	100.79
	1366364	97.67	1302.3	33949.1	3.37	212.79	148.61
random conics	1001	8.24	2.20	N/A	0.01	0.17	0.22
	3418	29.22	6.09	N/A	0.03	0.61	0.80
	13743	127.04	28.26	N/A	0.13	2.72	3.57
robotics	25533	2.63	8.29	34.67	0.06	1.69	0.35
Norway	42786	5.28	20.06	70.33	0.10	3.23	2.37

Table 2: Preprocessing time (in seconds).

creases the preprocessing time of the algorithm. However, the query time decreases only until a certain minimum around 100,000 landmarks, and it is much larger for 1,000,000 landmarks. 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 (and consequently, the number of landmarks used), and with the Landmarks type used. Figure 6 shows the duration percentages of the various steps of the query operation, in the LM(vert) and LM(grid) algorithms. As can be seen in the LM(vert) diagram, the nearest-neighbor search part increases when more landmarks are present, and becomes the most time-consuming part in large arrangements. 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 is the major step of the LM(grid) algorithm.

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

4 Conclusions

We propose a new Landmarks algorithm for exact point location in general planar arrangements, and have integrated an 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 observation of our experiments is that the Landmarks algorithm is the best strategy considering the cost per query, which takes

Arrang. Type	#Edges	Arrangement Size	RIC	Triang.	LM (vert)	LM (rand)	LM (grid)
	2112	0.8	1.3	0.3	0.2	0.5	0.5
random	37046	9.5	21.5	7.7	2.6	8.1	6.8
segments	235446	57.3	136.5	46.4	17.0	51.9	44.4
	955866	231.3	555.0	206.1	55.8	208.5	178.1
	1366364	333.8	793.2	268.9	86.8	307.0	258.9

Table 3: Memory usage (in MBytes) by the point location data structure.

Number of Landmarks	Preprocessing Time [sec]	Query Time [msec]	% Queries with AD=0
100	61.7	4.93	3.4
1000	59.0	1.60	7.6
10000	60.8	0.58	19.2
100000	74.3	0.48	42.3
1000000	207.2	3.02	71.9

Table 4: LM(rand) algorithm performance for a fixed arrangement and a varying number of random landmarks.
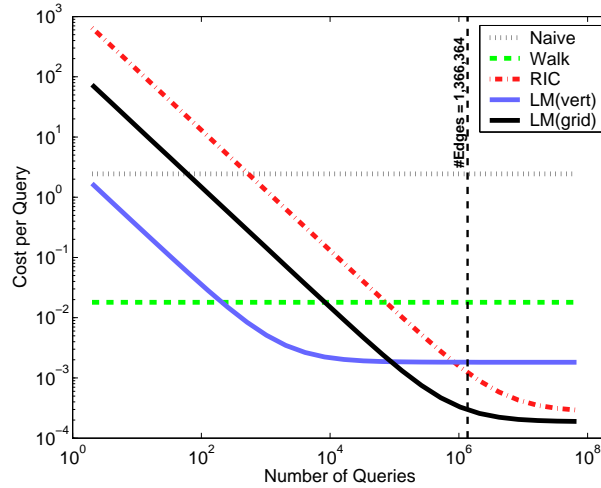

Figure 5: The average combined (amortized) cost per query in a large arrangement, with 1,366,384 edges.
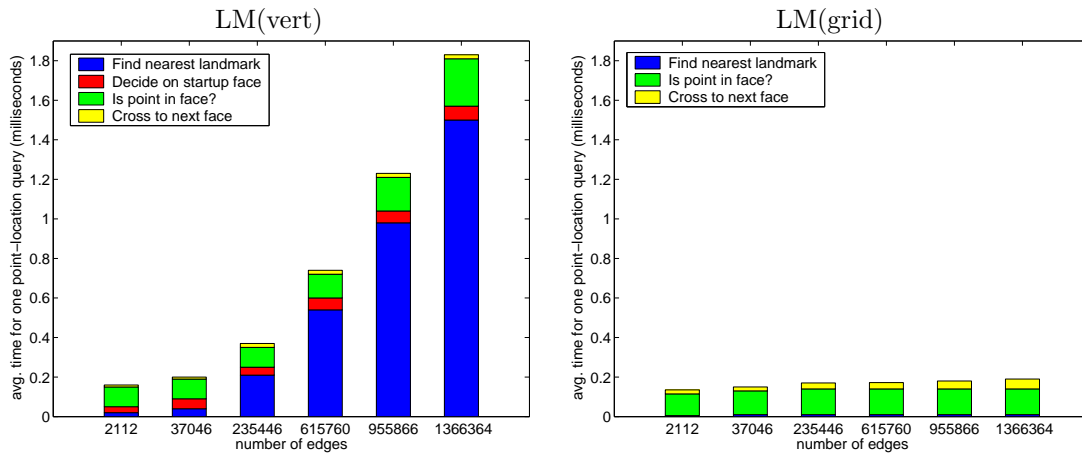

Figure 6: The average breakdown of the time required by the main steps of the Landmarks algorithms in a single point-location query, for arrangements of varying size.

into account both (amortized) preprocessing time and query time. Moreover, the memory space required by the algorithm is smaller compared to other algorithms that use auxiliary data structure for point location. The algorithm is easy to implement, maintain, and adjust for different needs using different kinds of landmarks and search structures.

It remains open to study the optimal number of landmarks required for arrangements of different sizes. This number should balance well between the time it takes to find the nearest landmark using the nearest-neighbor search structure, and the time it takes to walk from the landmark to the query point.

Acknowledgments

References

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

[2] The CORE library homepage. http://www.cs.nyu.edu/exact/core_pages/.

[3] The LEDA homepage. http://www.algorithmic-solutions.com/enleda.htm.

[4] The GNU MP bignum library. http://www.swox.com/gmp/.

[5] P. K. Agarwal and M. Sharir. Arrangements and their applications. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[6] S. Arya, T. Malamatos, and D. M. Mount. Entropy-preserving cutting and space-efficient planar point location. In *Proc. 12th ACM-SIAM Sympos. Disc. Alg.*, pages 256–261, 2001.

[7] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891–923, 1998.

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

[9] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22(1–3):5–19.

[10] O. Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.

[11] O. Devillers and P. Guigue. The shuffling buffer. *Internat. J. Comput. Geom. Appl.*, 11:555–572, 2001.

[12] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.

[13] L. Devroye, C. Lemaire, and J.-M. Moreau. Fast Delaunay point-location with search structures. In *Proc. 11th Canad. Conf. Comput. Geom.*, pages 136–141, 1999.

[14] L. Devroye, E. P. Mücke, and B. Zhu. A note on point location in Delaunay triangulations of random points. *Algorithmica*, 22:477–482, 1998.

[15] M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency — comparison with existing algorithms. *ACM Trans. Graph.*, 3:86–109, 1984.

[16] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *J. Exp. Algorithmics*, 5:13, 2000.

[17] E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving cgal's arrangements. In *Proc. 12th Annual European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 664–676. Springer-Verlag, 2004.

[18] D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.

[19] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[20] J. Matoušek. *Geometric Discrepancy — An Illustrated Guide*. Springer, 1999.

[21] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.

[22] E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 274–283, 1996.

[23] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3-4):253–280, 1990.

[24] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *Regional Conference Series in Applied Mathematics*. CBMS-NSF, 1992.

[25] F. P. Preparata and M. I. Shamos. *Computational Geometry — An Introduction*. Springer, 1985.

[26] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.

[27] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51–64, 1991.