TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Efficient Point Location in General Planar Subdivisions using Landmarks

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Idit Haran

The research work for this thesis has been carried out at
Tel-Aviv University
under the supervision of Prof. Dan Halperin

April 2006

Acknowledgments

I would like to thank my supervisor Prof. Dan Halperin for his guidance and great help during this research.

I thank Zeev Rudnick for useful discussions and helpful comments concerning number theory.

I thank all the fellow students in the Robotics and Vision lab for their assistance and support. Each and every one of them helped me in his way.

I also thank my parents, and my parents-in-law for taking care of my children so that I could take the time to work.

Finally, I would like to thank my husband On for his great support and understanding, and my children, Uri and Inbal, for their existence in general, and for their ability to remind me of what is really important in life.

This work has been supported by the IST Programme of the EU as a Shared-corst RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

Abstract

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 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 a "walk" strategy is applied 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 (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 a highly effective point-location algorithm for most practical purposes.

A paper summarizing the main ideas and results of this thesis [25] has been recently presented in the eighth workshop on algorithm engineering and experiments (ALENEX06). A full version of this paper was invited for a special issue of JEA (ACM Journal of Experimental Algorithmics), dedicated to papers from ALENEX06.

Contents

1 Introduction **7**

2 Preliminaries **9**

2.1 CGAL 9

2.2 2D Arrangements in CGAL 9

2.3 KD-trees 11

2.3.1 ANN 12

3 Related Work **13**

3.1 Solving the Point Location Problem 13

3.2 Point Location in CGAL 15

4 Point Location with Landmarks **19**

4.1 Choosing the Landmarks 20

4.2 Nearest Neighbor Search Structure 22

4.3 Walking from the Landmark to the Query Point 24

5 Implementation Details **27**

5.1 The Main Arrangement Class 27

5.1.1 The Arrangement-Traits Concepts 28

5.2 The Notification Mechanism 30

5.3 Point-Location Queries 31

5.4 The class *Arr_landmarks_point_location* 32

5.4.1 The *Generator* 32

5.4.2 The Nearest Neighbor Class 34

5.4.3 Implementing the Walk 35

6 Experimental Results **37**

6.1 The Benchmark 37

6.2 Results 38

6.2.1 Comparing Point-Location Strategies 38

6.2.2 Analysis of the Landmarks Strategy 40

6.3 Comparison with Point-Location Algorithms in Triangulations 43

7 Conclusions and Future Work **45**

A The Importance of Being Rational **47**

Chapter 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-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 [1, 24]. Figure 1.1 shows two different types of arrangements, one induced by line segments and the other by conic arcs.[1]

The planar point-location problem is 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.

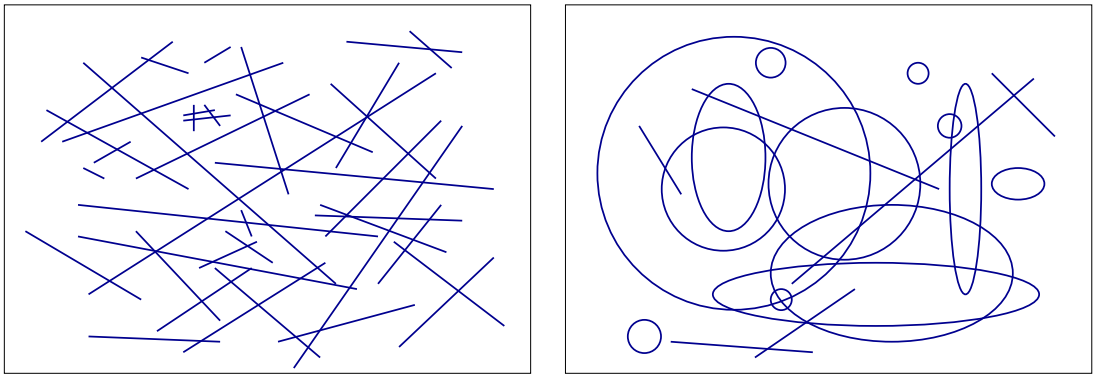
The planar point location problem may be solved naïvely by traversing all the edges and vertices in the arrangement and finding the geometric entity that is exactly on, or directly above, the query point. The time it takes to perform the query using this approach is proportional to the number of edges n, 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 real-time 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 the algorithms known for planar point location, and compare different implementations of these algorithms. We show that no existing strategy simultaneously addresses all the issues of preprocessing complexity, memory usage and query time.

[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)

Figure 1.1: Random arrangements of line segments (a) and of conic arcs (b).

We propose a new point location method, called *Landmarks*. In this 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.

We have implemented our algorithm in CGAL, the Computational Geometry Algorithms Library. 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.

Extensive 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 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.

Thesis Outline

The rest of the thesis is organized as follows: In Chapter 2 we introduce the terminology and notation used in the work. In Chapter 3 we review related work on other point location strategies, and the implementations of point-location algorithms in CGAL. Chapter 4 describes the Landmarks algorithm in details. Implementation details are given in Chapter 5. Chapter 6 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 Chapter 7. Appendix A provides further details regarding algebraic issues that arose while implementing the Landmarks algorithm on conic arc arrangements.

Chapter 2

Preliminaries

2.1 CGAL

CGAL, the Computational Geometry Algorithms Library[1] 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 [6]. Robustness of the algorithms is achieved by both handling all degenerate cases, and by using exact number types (see [37] and [42] for surveys on robustness issues in computational geometry). 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 [20, 21]. 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[2] for rational numbers, and CORE[3] [26] and LEDA[4] [30] for algebraic numbers.

2.2 2D Arrangements in CGAL

Given a set $\mathcal{C}$ of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane induced by the curves in $\mathcal{C}$ into zero-dimensional, one-dimensional and two-dimensional cells, called *vertices*, *edges* and *faces* respectively. The curves in $\mathcal{C}$ can intersect each other (a single curve may also be self-intersecting or may be comprised of several disconnected branches) and are not necessarily x-monotone.[5] A collection $\mathcal{C}''$ is constructed of x-monotone subcurves

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

[2] Gnu's multi-precision library: `http://www.swox.com/gmp/`

[3] `http://www.cs.nyu.edu/exact/core_pages/.`

[4] `http://www.algorithmic-solutions.com/enleda.htm`

[5] A continuous planar curve C is *x-monotone* if every vertical line intersects it at most once. For example, a non-vertical line segment is always x-monotone and so is the graph of any continuous function $y = f(x)$. For convenience, vertical line segments are treated as *weakly x-monotone*, as there exists a single vertical line that overlaps them.

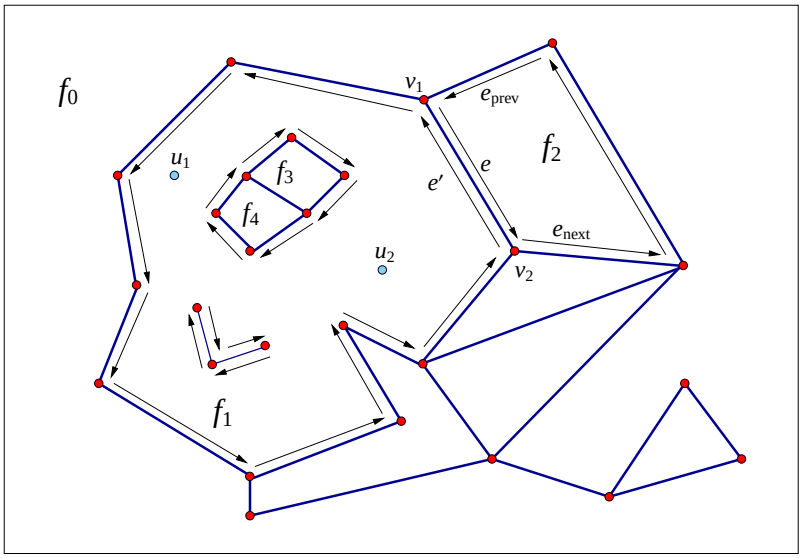


Figure 2.1: An arrangement of interior-disjoint line segments with some of the DCEL records that represent it. The unbounded face f_0 has a single connected component that forms a hole inside it, and this hole is comprised if several faces. The half-edge e is directed from its source vertex v_1 to its target vertex v_2. This edge, together with its twin e', corresponds to a line segment that connects the points associated with v_1 and v_2 and separates the face f_1 from f_2. The predecessor e_{prev} and successor e_{next} of e are part of the chain that forms the outer boundary of the face f_2. The face f_1 has a more complicated structure as it contains two holes in its interior: One hole consists of two adjacent faces f_3 and f_4, while the other hole is comprised of two edges. f_1 also contains two isolated vertices u_1 and u_2 in its interior.

that are pairwise disjoint in their interiors in two steps as follows. First, each curve in $\mathcal{C}$ is decomposed into maximal x-monotone subcurves (and possibly isolated points), obtaining the collection $\mathcal{C}'$. Note that an x-monotone curve cannot be self-intersecting. Then, each curve in $\mathcal{C}'$ is decomposed into maximal connected subcurves not intersecting any other curve (or point) in $\mathcal{C}'$. The arrangement induced by the collection $\mathcal{C}''$ can be conveniently embedded as a planar graph, whose vertices are associated with curve endpoints or with isolated points, and whose edges are associated with subcurves. (Evidently, $\mathcal{A}(\mathcal{C}) = \mathcal{A}(\mathcal{C}'')$.) This graph can be represented using a *doubly-connected edge list* data-structure (DCEL for short), which consists of containers of vertices, edges and faces and maintains the incidence relations among these objects.

The main idea behind the DCEL data-structure is to represent each edge using a pair of directed *halfedges*, one going from the xy-lexicographically smaller (left) endpoint of the curve toward its xy-lexicographically larger (right) endpoint, and the other, known as its *twin* halfedge, going in the opposite direction. As each halfedge is directed, we say it has a *source* vertex and a *target* vertex. Halfedges are used to separate faces, and to connect vertices.

If a vertex v is the target of a halfedge e, we say that v and e are *incident* to each other.

The halfedges incident to a vertex v form a circular list oriented in a clockwise order around this vertex.

Each halfedge e stores a pointer to its *incident face*, which is the face lying to its left. Moreover, every halfedge is followed by another halfedge sharing the same incident face, such that the target vertex of the halfedge is the same as the source vertex of the next halfedge. The halfedges are therefore connected in circular lists, and form chains, such that all edges of a chain are incident to the same face and wind along its boundary. We call such a chain a *connected component of the boundary* (or *CCB* for short).

The unique CCB of halfedges winding in a counterclockwise orientation along a face boundary is referred to as the *outer CCB* of the face. Exactly one unbounded face exists in every arrangement, as the arrangement package supports only bounded curves at this point. The unbounded face does not have an outer boundary. Any other connected component of the boundary of the face is called a *hole* (or *inner CCB*), and can be represented as a circular chain of halfedges winding in a clockwise orientation around it. Note that a hole does not necessarily correspond to a single face, as it may have no area, or alternatively it may consist of several faces. Every face can have several holes contained in its interior (or no holes at all). In addition, every face may contain isolated vertices in its interior. See Figure 2.1 for an illustration of the various Dcel features. For more details on the Dcel data structure see [12, Chapter 2].

2.3 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 k-nearest) points of P to q can be reported efficiently.

Answering nearest neighbor queries efficiently, especially in higher dimensions, seems to be a very difficult problem. It is always possible to answer any query by a simple brute-force process of computing the distances between the query point and each of the data points, but this may be too slow for many applications that require that a large number of queries be answered on the same data set. Instead the approach is to preprocess a set of data points into a data structure with which nearest neighbor queries can then be answered efficiently.

One common data structure that has been proposed for solving this problem is the Kd-tree [8, 22]. The Kd-tree data structure is based on a recursive subdivision of the space into disjoint hyper-rectangular regions called cells. Each node of the tree is associated with a region B, called the box, and the set of data points that lie within this box. The construction of the tree can be described as follows: The root node of the tree is associated with a bounding box that contains all the data points. For each node in the tree, if the number of data points associated with this node is greater than a small quantity, called the bucket size, the node's box is split into two boxes by an axis-orthogonal hyper-plane that intersects this box. (There are different methods to select this splitting hyperplane; for example, using the median point, splitting in the middle of the box, etc.) These two boxes are the cells associated with the two children of this node. The data points lying in the original box are split between these

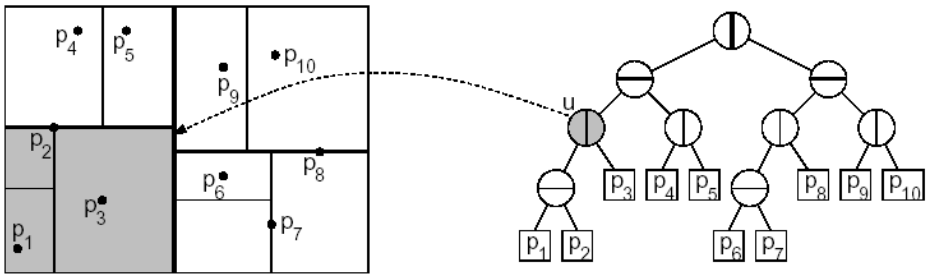


Figure 2.2: An example of a 2-dimensional Kd-tree.

two children, depending on the side of the splitting hyperplane in which they lie. Points lying on the hyperplane itself may be associated with either child (according to the dictates of the splitting rule). When the number of points that are associated with the current box falls below the bucket size, the resulting node is declared a leaf node, and these points are stored with the node. Figure 2.2 shows an example of a 2-dimensional Kd-tree with bucket size 1.

The Kd-tree can be built in $O(n \log n)$ time, and it uses linear space. If the input points are well distributed in space, it is known to answer nearest-neighbor queries in logarithmic time [7].

2.3.1 ANN

ANN 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 (pronounced like the name "Ann") stands for the Approximate Nearest Neighbor library. 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.

One difficulty with exact nearest neighbor searching is that for virtually all methods other than brute-force search, the running time or space grows exponentially as a function of dimension. Consequently these methods are often not significantly better than brute-force search, except in fairly small dimensions. However, it has been shown by Arya and Mount [4] and Arya et al. [5] 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.

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

Chapter 3

Related Work

3.1 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 $O(n)$ [40]. 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 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 [18]. 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 [35, 39]. To obtain linear storage space, Sarnack and Tarjan [36] used *persistent search trees*. In each node, a modification box is store. This box can hold a constant number of modifications to the node, each modification is stamped by a time stamp that indicates the relevant x-coordinate values of this node. Whenever we access a node, we check the modification box, and compare its time stamp against the access time. If the modification box is empty, or the access time is before the modification time, then we ignore the modification box and just deal with the normal part of the node. Otherwise, we use the value in the modification box, overriding the value in the node. (For example, if the modification box has a new left pointer, then we use it instead of the normal left pointer, but we still use the normal right pointer.)

Edahiro et al. [19] used the slabs 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. 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 calculated in advance. Instead, it is created on the fly during query time.

Another approach with an expected query time of $O(\log n)$ uses trapezoid graph as its search structure. A trapezoid graph is a directed, acyclic graph DAG in which each nonleaf node v is associated with a pseudo-trapezoid, T_v, whose parallel sides are vertical and whose top and bottom are either a single subdivision edge or are at infinity. Mulmuley [32] and Seidel [38] suggested to build this graph as the history graph of the random incremental construction (RIC) of an arrangement of segments. 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.

Point location in triangulations was extensively studied: Kirkpatrick [27] 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 the upper level in the hierarchy, these vertices are removed and the remaining area left from their removal is 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. [15] 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 the 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 towards the query point q, using four walk strategies: (1) A straight walk, which traverse 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 isometric path moving from the vertex v to q by changing one coordinate at a time. (3) The visibility walk: advance 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, check 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. Due to 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 [13], 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 level consists of a data structure that stores a triangulation of a small random sample of the vertices of the triangulation at the preceding 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 preceding 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 achieves fast

point location queries on real data. 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. [31] and by Devroye et al. [17]. The *Jump & Walk* proceeds as follows: Given a triangulation of n sites, pick $k \in [1, n]$ random sites in the data, select ζ, the one closest to the query point q, and traverse the triangulation from ζ to q, exploiting the adjacency relationship between the successive triangles crossed by segment ζq. The expected running 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. [16] 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 to start the walk to the query. They also developed a method called *2-d Search & Walk*, that uses a balanced 2-d tree in which the partition alternates directions between x and y, and member sets in the partition are rectangles (similar to a KD-tree). 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. [3] 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 $\sum_i -p_i - \log_2 p_i$. 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 [2], at the cost of increased space and programming complexity.

As the point-location problem is of practical importance, many works (some mentioned above) include an extensive experiments in their study, beside the theoretic analysis. Devillers et al. [15] 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. [16] have tested their *Jump & Walk* algorithm using different "jump" alternatives, and showed that the most efficient option would be to use a balanced 2-d tree to get to the nearest starting point.

3.2 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 the 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 Chapter 5 below) 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 that 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 case 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*[1] 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 non-linear 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 3.1 shows an example of the Walk algorithm.

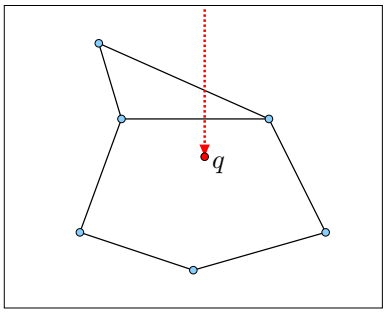


Figure 3.1: An example of walking along a vertical ray

3. A *Triangulation* strategy. This 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). A CDT is 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 [13],

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

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 [9].

4. The *RIC (Random Incremental Construction)* algorithm is an implementation of the dynamic algorithm introduced by Mulmuley [32] and Seidel [38]. The implementation consists of two structures: (i) a *trapezoidal map* and (ii) a *search structure* — the history DAG (Directed Acyclic Graph). The trapezoidal map is created by subdividing each arrangement face into pseudo-trapezoidal cells each of constant complexity. Each such cell is bounded above and below by curves and from the sides by vertical attachments. 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 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 non-intersecting edges in the arrangement). For detailed explanation of the algorithm see [12, chapter 6].

   Figure 3.2 shows an example of a trapezoidal map and the search structure associated with it.

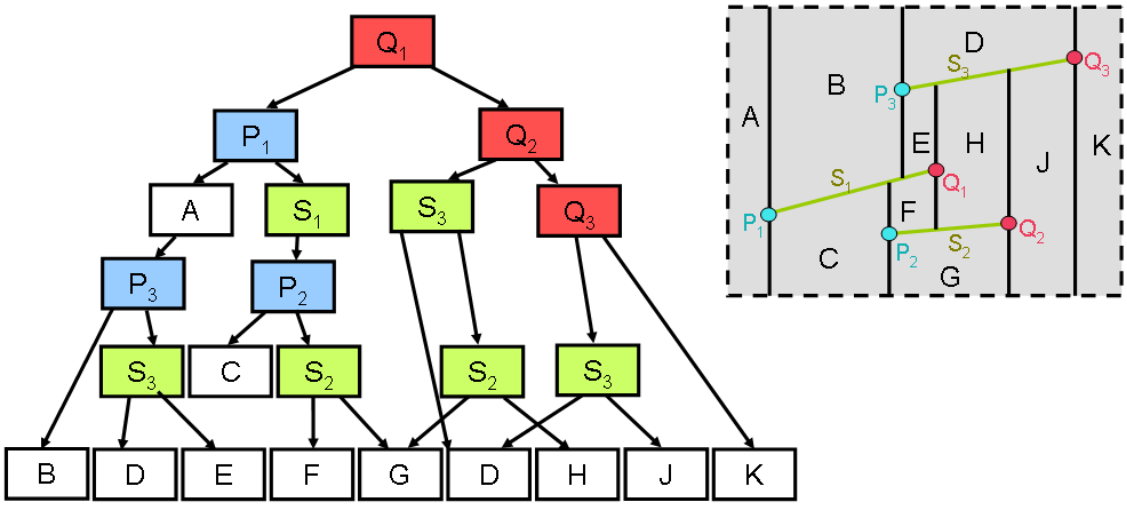


Figure 3.2: An example of a trapezoidal map and the search structure (DAG) associated with it. Each leaf of the tree represents the final trapezoid where the point is located. An inner node represents a segment or an end-point of a segment. During query time, these node are used to guide the search. A node marked S_i represents the question: "Is the query point q above the segment S_i?". The nodes marked Q_i or P_i represent the question: "Is the query point q on the left side of Q_i (or P_i, respectively)?". Now, given a query point q, start at the root of the tree. At each node, if the answer to the relevant question is "yes", continue the search from the left son of this node. Otherwise, continue from the right son. The search is over when a leaf node is reached.

Chapter 4

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 to or removed from the arrangement, 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 [14].

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 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 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 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.

4.1 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.

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 6.2.2 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 metric 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 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 4.1 shows the arrangement distance from the closest vertex v of the arrangement to a query point q: in 4.1(a) AD=0, and in 4.1(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 non-convex, for example using vertical decomposition or triangulation.[1] Such preprocessing 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 throughout a rectangle bounding the arrangement. Another is to use a uniform grid, or to use other

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

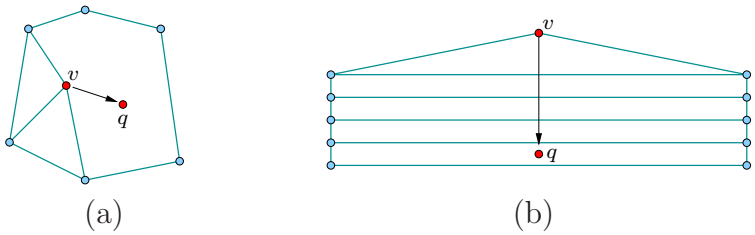


Figure 4.1: The arrangement distance from the closest vertex v to the query point q: (a) AD=0 and (b) AD=4.

structured point sets, such as Halton sequences or Hammersley points [29, 34]. 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 4.3 below).

- *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(grid), 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, 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.

- *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 the 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. 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 of the random points. 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 [28, Chapter 5]: We choose two prime integers (for two-dimensional 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:

$$r(i,p) = \frac{a_0}{p} + \frac{a_1}{p^2} + \frac{a_2}{p^3} + \frac{a_3}{p^4} + \cdots \tag{4.1}$$

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 $I = 1, 2, 3, ...$, we take each number I, write it in base 2, and reverse the digits, including the decimal sign, and convert back to base 10:

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

and so on.

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.

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.

Figure 4.2 shows the same random arrangement with different types of landmarks points. 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 5.4) 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).

4.2 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.

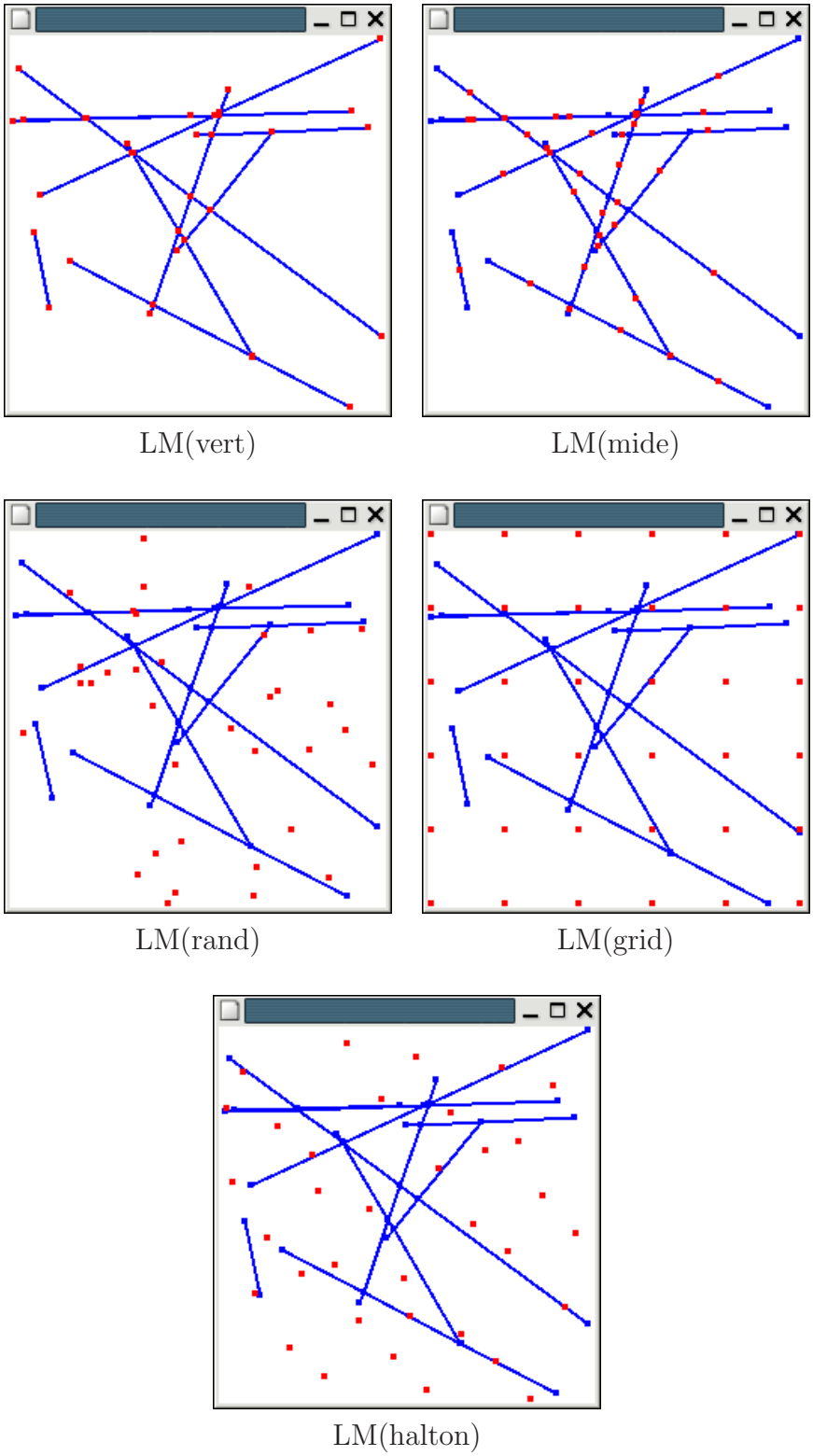


Figure 4.2: An arrangement of random line segments, with different landmarks. The number of landmarks in LM(vert), LM(rand), LM(grid) and LM(halton) is equal to the number of vertices in the arrangement. The number of landmarks in LM(mide) is equal to the number of edges in the arrangement.

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, 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 [5], 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.

4.3 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 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 here is arbitrary, rather than vertical.

The "walk" stage is summarized in the diagram in Figure 4.3.

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, than 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. And last, if the landmark is located on a vertex, we need to check all incident faces 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 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, return this element as the result of the query.

However, if this number is even, then the actual "walk" part begins. A virtual line

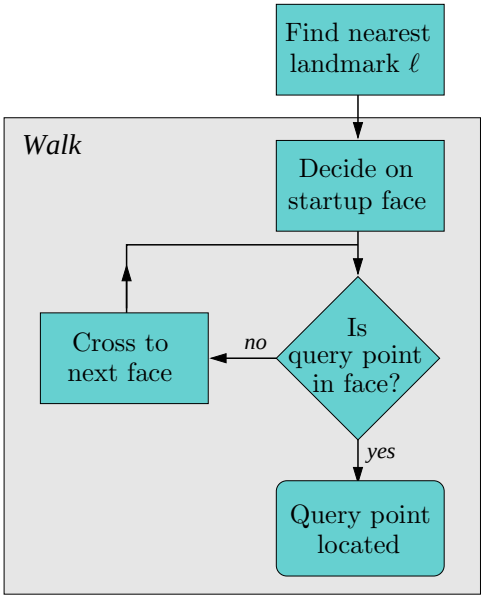


Figure 4.3: The "walk" part of the query algorithm.

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 4.4 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

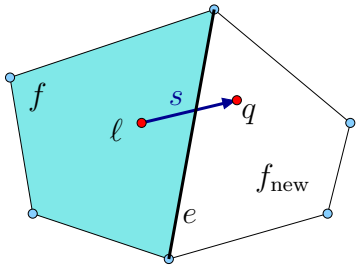


Figure 4.4: An example of walking from the landmark ℓ to the query point q. In this example ℓ is located in the face f, which is then set as the startup face. Since q is not inside f, we cross to f_{new} through the edge e, which is the edge of f's boundary that intersects the segment s connecting ℓ to q.

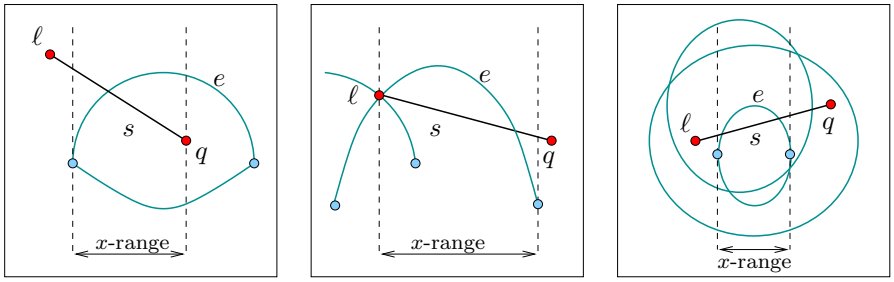


Figure 4.5: Walk algorithms, crossing to the next face. In all cases the vertical order of the curves is compared on the left and right boundaries of the marked x-range. (a) s and e swap their y-order, therefore we should use e to cross to the next face. (b) s and e share a common left endpoint, but e is above s immediately to the right of this point, and below s at the left boundaries 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.

boundaries of this range. If the vertical order changes, it implies that the curves intersect; see, e.g., Figure 4.5(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 4.5(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 4.5(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 boundaries of the common x-range will not find an intersection. This is the desired behavior of the predicate in this case, as we do not want to use this edge for crossing.

Chapter 5

Implementation Details

In this chapter 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 5.1). We then explain the notification mechanism (Section 5.2) supported by the arrangement and used by the landmarks point location. We also give details about the design of the point-location strategies in general (Section 5.3), and the implementation of the class `Arr_landmarks_point_location` (Section 5.4) in particular.

5.1 The Main Arrangement Class

The `Arrangement_2<Traits,Dcel>`[1] class-template represents the planar embedding of a set of weakly x-monotone[2] 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 two-dimensional subdivisions (see Section 2.2 for detailed explanation about the DCEL).

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

- A traits class, 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; see Section 5.1.1 for further details.
- 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

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

[2] 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 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. The separation is enabled by the modular design and conveniently implemented within the generic-programming paradigm [6].

The interface of `Arrangement_2` consists of various methods that enable the traversal of arrangement objects. For example, the class supplies iterators over its vertices, halfedges, or faces. These iterator types are `Vertex_iterator`, `Halfedge_iterator`, and `Face_iterator` respectively, and they are convertible to the handle types `Vertex_handle`, `Halfedge_handle`, and `Face_handle`, respectively. The handle classes serve as pointers to the arrangement features, which in turn supply methods for local traversal. For example, it is possible to visit all halfedges incident to a specific vertex using its `Vertex_handle`, or to iterate over all halfedges along the boundary of a face using its `Face_handle`.

An important guideline in the design is to decouple the arrangement representation from the various algorithms that operate on it. Thus, non-trivial algorithms that involve geometric operations are implemented as free (global) functions. For example, the package offers free `insert()` functions that insert general curves into the arrangement. For further details regarding the new design of the arrangement class information see [41].

5.1.1 The Arrangement-Traits Concepts

As mentioned in the previous section, the `Arrangement_2` class-template is parameterized with a geometric *traits* class that defines the abstract interface between the arrangement data-structure and the geometric primitives it uses. The name "traits" was given by Myers [33] for a concept, a model of which supports certain predefined methods that have a common denominator. In our case, a geometric traits class defines the family of curves handled. Moreover, details such as the number type used to represent coordinate values, the type of coordinate system used (i.e., Cartesian or homogeneous), the algebraic methods used, and auxiliary data stored with the geometric objects, if present, are all determined by the traits class and are encapsulated within it.

The traits concept is factored into a hierarchy of refined concepts. The refinement hierarchy is generated according to the identified minimal requirements imposed by different algorithms that operate on arrangements, thus alleviating the production of traits classes, and increasing the usability of the algorithms.

Every model of the traits concept must define two types of objects, namely `Point_2` and `X_monotone_curve_2`. The latter represents a planar x-monotone curve, and the former is the type of the endpoints of the curves, representing a point in the plane. The basic

concept *ArrangementBasicTraits_2* lists the minimal set of predicates on objects of these two types sufficient to enable the operations provided by the `Arrangement_2` class-template itself, namely a basic insertion of x-monotone curves that are interior disjoint from any vertex and edge in the arrangement. The predicates in this concept are:

1. Compare two points by their x-coordinates only, or lexicographically, by their x and then by their y-coordinates.
2. Return the lexicographically smaller (left), or the lexicographically larger (right), endpoint of a given x-monotone curve.
3. Determine whether a weakly x-monotone curve is a vertical segment.
4. Given an x-monotone curve C and a point $p = (x_0, y_0)$ such that x_0 is in the x-range of C (namely x_0 lies between the x-coordinates of C's endpoints), determine whether p is above, below, or lies on C.
5. Given two x-monotone curves C_1 and C_2 that share a common left endpoint p, determine the relative position of the two curves immediately to the right of p. The traits class can also provide a symmetric comparison method, namely to the left of a common right endpoint.

The set of predicates listed above is sufficient for answering point-location queries by various point-location strategies. However, using our Landmarks strategy requires that the arrangement is instantiated with a traits class that models the *ArrangementLandmarksTraits_2* concept, which adds a few requirements to the basic *ArrangementBasicTraits_2* concept. A model of this concept must define a fixed precision number type (typically `double`) and support the additional operations:

- Given a point p, approximate the x and y-coordinates of p using the given fixed precision number type.
  We use this operation for approximate computations — there are certain operations in the search for the location of the point that need not be exact and we can perform them faster than other operations. Such operations are finding the nearest landmarks to a given query point.
- Given two points p_1 and p_2, construct an x-monotone curve connecting p_1 and p_2.
  This curve is used for walking from the landmark ℓ to the query point q. It is usually a line segment, although this is not a requirement.

There are other traits in the hierarchy that refine the traits classes above, such as *ArrangementXMonotoneTraits_2* and *ArrangementTraits_2*, that support the incremental and aggregate insertion operation of x-monotone curve or general curves, respectively.

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 functions. Most of these traits support all operations listed above.

5.2 The Notification Mechanism

For some applications it is essential to know exactly what happens inside a specific arrangement-instance. For example, when a new curve is inserted into an arrangement, it might be desired to keep track of the faces that are split due to this insertion operation. Other important examples are the 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. The arrangement package offers a mechanism that uses *observers* [23], which can be attached to an arrangement instance and receive notifications about the changes this arrangement goes through.

The `Arr_observer<Arrangement>` class-template is parameterized with an arrangement type. It stores a pointer to an arrangement object, and is capable of receiving notifications just before a structural change occurs in the arrangement and immediately after such a change takes place. Hence, each notification is comprised of a pair of "before" and "after" functions (e.g., `before_split_face()` and `after_split_face()`). The `Arr_observer <Arrangement>` class-template serves as a base class for other observer classes and defines a set of virtual notification functions, giving them all a default empty implementation. The interface of the base class is designed to capture all possible changes that arrangements can undergo, with a minimal number of functions.

The set of functions can be subdivided into three categories as follows:

1. Notifiers of changes that affect the entire topological structure. Such changes occur when the arrangement is cleared or when it is assigned with the contents of another arrangement.

2. Notifiers of a *local* change to the topological structure. Among these changes are the creation of a new vertex or an edge, the splitting of an edge or a face, the formation of a new hole inside a face, the removal of an edge, etc.

3. Notifiers of a *global* change initiated by a free (global) function, and called by the free function (e.g., incremental or aggregate insert). This category consists of a single pair of notifiers, neither of them is called by methods of the `Arrangement_2` class-template itself. It is required that no point-location queries (or any other queries for that matter) are issued between the calls to the "before" and "after" functions of the global change. This constraint can improve the efficiency of the maintenance of auxiliary data structures for the relevant point-location strategies, which have to update their data structures according to the changes the arrangement undergoes (see the next section for more details). Since no point-location queries are issued between the invocation of `before_global_change()` and `after_global_change()`, it is not necessary to perform an update each time a local topological change occurs, and it is possible to postpone the updates until the global operation is completed.

Each arrangement object stores a list of pointers to `Arr_observer` objects, and whenever one of the structural changes listed in the first two categories above is about to take place,

the arrangement object performs a *forward* traversal of this list and invokes the appropriate function of each observer. After the change has taken place the observer list is traversed in a *backward* manner (from tail to head) and the appropriate notification function is invoked for each observer. This allows the nesting of observer objects. In addition, a free function may choose to trigger a similar notification, which falls under the third category above.

5.3 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 (a polymorphic `CGAL::Object` instance that can either be a `Face_handle`, a `Halfedge_handle`, or a `Vertex_handle`).

The following models for the concept *ArrangementPointLocation* are included in the arrangement package. Each employs a different point-location strategy, as explained in details in Section 3.2.

- `Arr_naive_point_location` locates the query point naïvely, by exhaustively scanning all arrangement cells.
- `Arr_walk_along_a_line_point_location` simulates a reverse traversal along an imaginary vertical ray emanating from the query point toward infinity. It starts from the unbounded face of the arrangement and moves downward towards the query point until it locates the arrangement cell containing it.
- `Arr_Triangulation_point_location` triangulates each face of the arrangement using constrained Delaunay triangulation. During query time the locate is performed using a triangulation hierarchy.
- `Arr_trapezoid_ric_point_location` implements the *RIC* algorithm, which is based on the vertical decomposition of the arrangement into pseudo-trapezoids.
- `Arr_landmarks_point_location` implements our Landmarks algorithm. See more details below.

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.

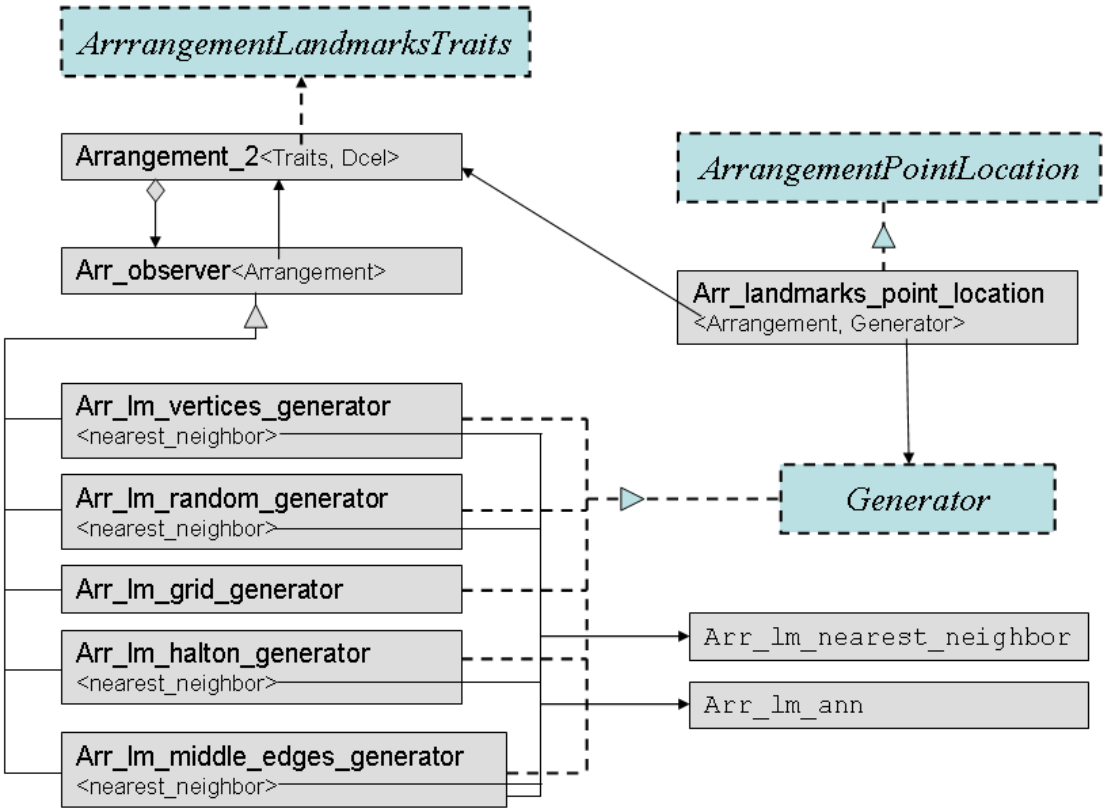


Figure 5.1: The classes and concepts involved in the implementation of the Landmarks point location. A rectangle with a solid frame designates a class, and a rectangle with a dashed frame designates a concept. A plain arrow designates a reference to an instance of a class or of a concept, solid lines 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.

5.4 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 5.1 shows the coorelation between all the classes and concepts that are involved in the Landmarks algorithm.

5.4.1 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. The generator class should support the following operations:

- Create the landmarks set, and save the landmarks in a nearest neighbor search structure.
- Clear the landmarks set and the search structure.
- Given a query point q, return the closest landmark point to q and the `object` which is the location of the landmark in the arrangement.

As mentioned, five types of Landmarks generators were implemented, with respect to the kind of landmarks they generate:

`Arr_landmarks_vertices_generator`	—	The landmarks are all the arrangement vertices.
`Arr_middle_edges_landmarks_generator`	—	The landmarks are the midpoints of all the arrangement edges.
`Arr_random_landmarks_generator`	—	Random points are selected.
`Arr_grid_landmark_generator`	—	The landmarks are chosen on a uniform grid.
`Arr_halton_landmarks_generator`	—	Halton sequence points are used.

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 local changes, such as when an edge is inserted, the generator implements only the functions that are called after the arrangement is updated with the change. For global changes, we implement both the function that is called before the change and after the change. The first one clears the arrangement and turns on a flag notifying not to make any updates in the data structure until the global change is finished. The second function, which is called after the global change is finished, rebuilds the set of landmarks and turns off the flag.

In most cases all the "after" functions rebuild the landmark set and the search structure for each change in the arrangement. The need to rebuild after all the changes 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 the effected landmarks, there should be "backward" pointers from each cell of the arrangement to the landmarks that are effected by it. 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, it saves a counter on 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 supports 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.

5.4.2 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 that 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, that might be represented with unlimited precision. The nearest neighbor search structure should support the following operations:

- Given a set of points, create their nearest neighbor search structure.
- Clean the nearest neighbor search structure.
- Given a query `Point_2 q`, find the closest `NN_Point_2` to `q`.

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 2.3.
- A nearest neighbor class that wraps the ANN package (see Section 2.3.1 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.

5.4.3 Implementing the Walk

As explained in Section 4.3, 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, intersect 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. We also show how we use the DCEL structure and the edge e that was found for crossing to the next face.

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 the same 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 done using the forth predicate in the traits class, which determines whether a given point is above, below or on a given curve. (see Section 5.1.1.) 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. The fifth predicate in the traits class enables this last test.

The first edge e on the boundary of f that meets the requirements listed above, is used for crossing to the next face.[3] The crossing is done by taking the twin halfedge e' of e, and the next face f' is the face that is to the left of e'. e and e' are then marked as flipped, and cannot be used again for flipping in the current walk. This restriction guarantees that the walk part will end.

[3] As explained in Section 4.3, although it may be better to use the edge that is the closest to the query point q, we have no way to check which edge is the closest, since we do not compute the exact intersection points of each edge on the boundary and s (as this is an expensive operation).

Chapter 6

Experimental Results

6.1 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 types of arrangements: *random segments*, *random conics*, *robotics*, and *Norway*. Each arrangement of 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 at random in the range $[0, 250]$.

The third type, *robotics*, is a line-segment arrangement that was constructed by computing the Minkowski sum[1] 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 flower-shaped polygon with 23 edges. The resulting arrangement consists of 42,786 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). The number of landmarks used in the LM(vert), LM(rand), LM(grid), LM(halton) is equal to

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

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 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, 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.

6.2 Results

6.2.1 Comparing Point-Location Strategies

Table 6.1 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 6.2 shows the preprocessing time for the same arrangements and same algorithms as in Table 6.1. 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 6.3 shows the memory usage of the point-location strategies for the same arrangements and same algorithms as in Tables 6.1 and 6.2. 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.

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 required 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,

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	37046	36.7	3.6	0.09	1.17	0.20	0.16	0.15
segments	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	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
	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 6.1: Average time (in milliseconds) for one point-location query.

Arrang. Type	**#Edges**	**Arrangement Construct.**	**RIC**	**Triang.**	**LM (vert)**	**LM (rand)**	**LM (grid)**
	2112	0.07	0.5	11.2	0.01	0.12	0.13
random	37046	1.26	29.7	360.2	0.05	2.97	2.95
segments	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	1001	8.24	2.20	N/A	0.01	0.17	0.22
conics	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 6.2: Preprocessing time (in seconds).

since it uses the Delaunay 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 yields similar results to those 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 preprocessing stages, which makes it

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
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
	13743	40.6	3.7	N/A	1.4	8.4	11.3
robotics	25533	7.2	14.9	3.7	1.3	3.9	3.1
Norway	42786	12.2	24.4	5.9	1.9	6.2	4.7

Table 6.3: Memory usage (in MBytes) by the point location data structure.

less efficient for these types of arrangements.

Figure 6.1 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:

$$cost(m) = \frac{\text{preprocessing time}}{m} + \text{average query time}. \tag{6.1}$$

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, $cost(m) = \textit{query time} = \textit{constant}$. 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.

6.2.2 Analysis of the Landmarks Strategy

As mentioned above, 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 6.4 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

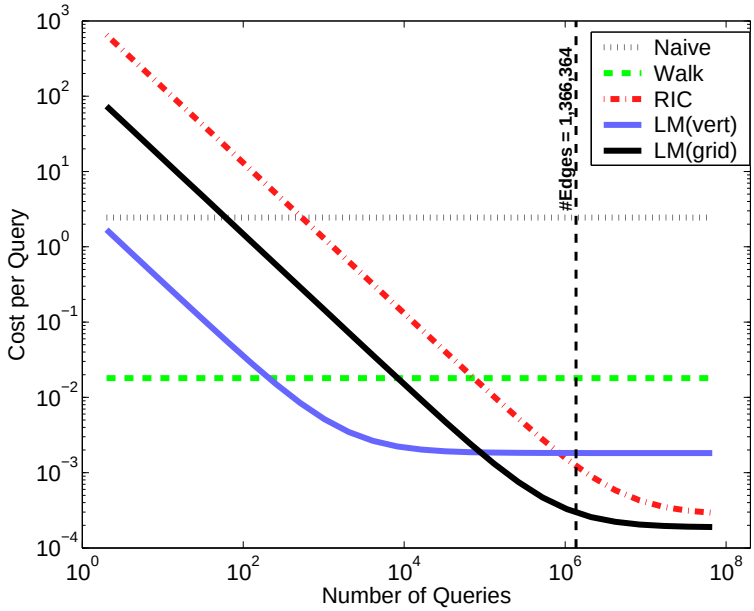


Figure 6.1: The average combined (amortized) cost per query in a large arrangement, with 1,366,384 edges.

landmarks increases 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 6.2 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

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

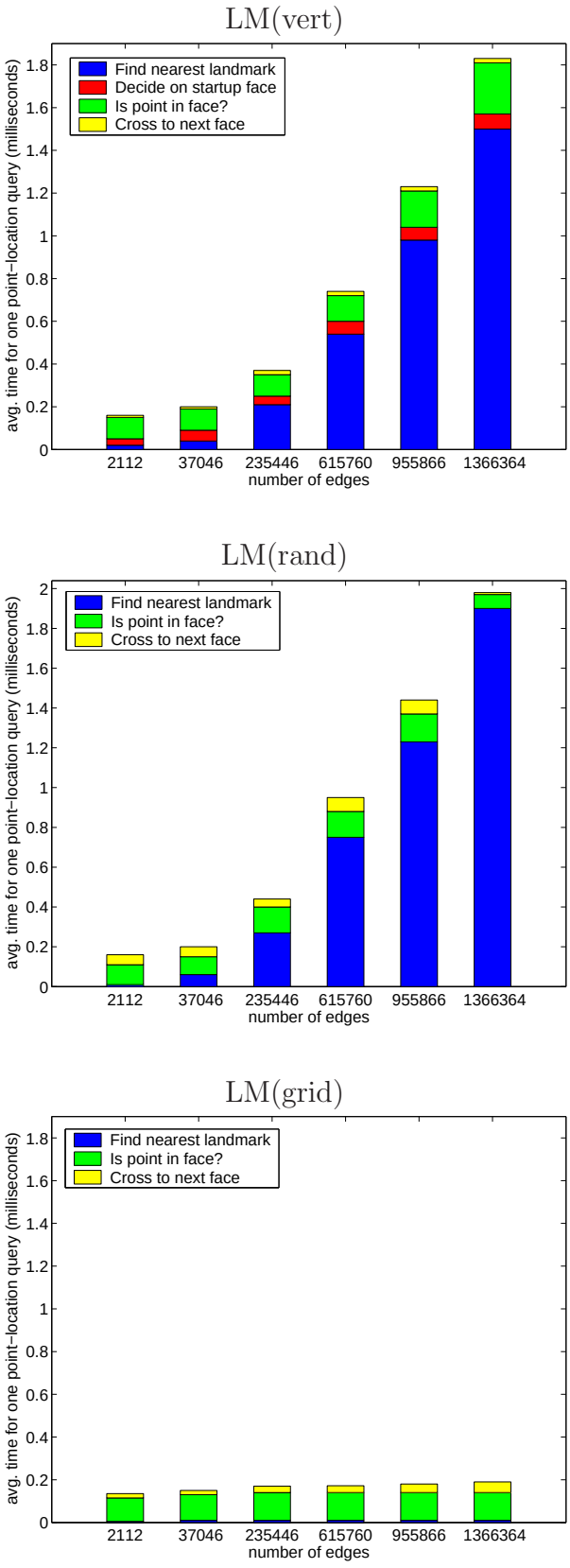


Figure 6.2: The average breakdown of the time required by the main steps of the Landmarks algorithms in a single point-location query, for arrangements of varying size.

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 consumes a major fraction 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 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 to be in the same face as the landmark point.

6.3 Comparison with Point-Location Algorithms in 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 work (see Chapter 3) have been done in order to speed 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 uses the Delaunay hierarchy (as explained in Chapter 3).

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 6.5 shows the results of this test. We can see that the location using LM(grid) is much faster than the location in the triangulation. 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 perform sometimes 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 represent the triangulation well, while LM(rand) and LM(halton) may require a longer "walk" part from the landmark to the query point. The LM(mide) contains more landmarks (equal to the number of the edges) and thus result with a longer nearest neighbor search.

In the second test, we chose random segments and created the Constrained Delaunay Triangulation (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 6.6 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

#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
100000	0.458	0.198	0.269	0.083	0.257	0.449
350000	0.597	0.577	0.752	0.078	0.710	1.580

Table 6.5: Average time (in milliseconds) for one point-location query in the Delaunay triangulation or the corresponding arrangement of random points.

#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 6.6: Average time (in milliseconds) for one point-location query in the Constrained Delaunay Triangulation or the corresponding arrangement of random segments.

#Random Site Points	Queries	Voronoi Diagram	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

Table 6.7: Average time (in milliseconds) for one point-location query in the Voronoi diagram or the corresponding arrangement of random points.

using the new CGAL package, which is an adaptor over the 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 set are random points that were selected uniformly at random inside the same bounding rectangle from which the original sites were selected.

Table 6.7 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)) 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.

Chapter 7

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 the Landmarks algorithm is the best strategy considering the cost per query, which takes into account both (amortized) preprocessing time and query time. Moreover, the memory space required by the algorithm is small 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 types of landmarks and search structures.

There are still many ways to improve the algorithm. In the following paragraphs we elaborate on some of them.

Dynamization. In the current implementation of the Landmarks algorithm, every time a new curves is added to or deleted from the arrangement, the data structure containing the landmarks is rebuilt (with a small exception in the LM(vert) strategy; see Section 5.4.1). The main reason for the need to rebuild is the fact that the nearest neighbor data structures that we use does not support insertions and deletions in the current implementations that we use (namely CGAL and ANN). In order to make the Landmarks algorithm dynamic, one will have to develop a dynamic nearest neighbor search structure that will support insertion and deletion of points.

Another extension that should be made in order to make the algorithm fully dynamic is to extend the arrangement data structure, or to maintain extra information. Given a certain change in the arrangement, the structure should be able to return the set of landmarks that were effected by this change. For example, when a new curve is added to the arrangement and a face is split because of this change, we would like to know which landmarks were inside this face, so that we can find their new locations.

Finding the Optimal Number of Landmarks. We have seen (Table 6.4) that the number of landmarks used by the Landmarks algorithm effects the location time. Obviously, 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 (except for LM(grid)), as seen in Figure 6.2. Thus, it may be instructive to look for the optimal number of landmarks versus the arrangement type and size.

Theoretic Analysis. It may be interesting to give a theoretical complexity analysis of the Landmarks algorithm. For example, to estimate 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 (or more generally, what is the average arrangement distance[1] from the closest landmark to the query point).

Additional Types of Landmark. In our work we have tested five types of landmarks: vertices, middle point of edges, random, Halton and grid points. However, many other types of landmarks can be used. For example, one can subdivide the faces of the arrangement into simple cells, such as triangles or trapezoids, and set a landmark in each of these cells. Another option may be to set the landmarks on very long edges. Landmarks can also be used on edges instead of vertices, when the vertex's coordinates have high algebraic degree (see Appendix A for more details).

Other options for the types of landmarks include combining different types of landmarks. It may be interesting to investigate the influence of combining, for example, landmarks that depend on the arrangement topology, such as vertices, with landmarks that are independent of the arrangement geometry, such as random points.

Improving Performance on 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 face 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 increasing preprocessing time and storage space.

[1] The definition of *Arrangement Distance* (AD) is given in Section 4.1

Appendix A

The Importance of Being Rational

Exact computation for non-linear curves requires calculations with irrational numbers, even if the input is given in rational coefficients. For example, the intersection point of two circles with a rational radius and rational coordinates of the center point may not be rational. In the general case, the intersection point of two conic arcs may be an algebraic number[1] of degree 4. In such cases, choosing this intersection point v as a landmark may considerably slow down the algorithm. In this appendix we elaborate on this problem, present the chord method giving a rational approximation for an irrational point on a circle, and show how this can be used in our algorithm.

When we started to implement the Landmarks algorithm, we first used the vertices of the arrangement as the only landmarks, as their location in the arrangement is known. We have implemented the walk stage from a vertex v to a query point q by finding an intersection point of an edge on the boundary of the current face with the segment s connecting v and q. This operation took much longer when the vertex v had high algebraic degree. In order to speed up the algorithm, we tried to avoid such operations (see Section 5.4.3 for detailed explanation of the method we are currently using).

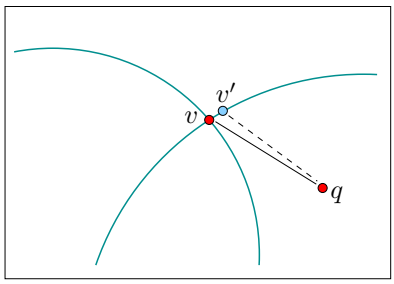


Figure A.1: Approximating the vertex v with v'.

One way we have considered in order to avoid such operations, was to find a point v' close to v that is still on the curve, but has lower algebraic degree (see Figure A.1). In this appendix we demonstrate the idea for circles, and compute a point v' on the circle with rational coordinates. We do this using the *Chord Method* [10]: Given a unit-circle equation:

[1] An algebraic number is any number that is a root of a non-zero polynomial with integer or rational coefficients. The degree of an algebraic number is the degree of this polynomial.

$x^2 + y^2 = 1$, the problem is to find a rational point v' close to the given point v on the circle. The solution relies on the fact that every line passing through the point $(-1, 0)$, and crosses the unit circle in another (distinct) point, can be written as: $y = t(x + 1)$, where t is the slope of the line. Now, we need to solve $x^2 + (t(x + 1))^2 = 1$. If t is rational, this equation has two rational solutions: (-1,0), that we already know, and $\left(\frac{1-t^2}{1+t^2}, \frac{2t}{1+t^2}\right)$; see Figure A.2. Thus, in order to approximate an irrational point $v = (x, y)$ on the circle centered at (x_0, y_0) with radius r, we have to approximate $t = \frac{y-y_0}{x-x_0+r}$ to any desired accuracy, and then calculate the rational point $\left(x_0 + r\frac{1-t^2}{1+t^2}, y_0 + r\frac{2t}{1+t^2}\right)$ as shown above. The chord method can also be applied to other conic curves that have at least one rational point on them.

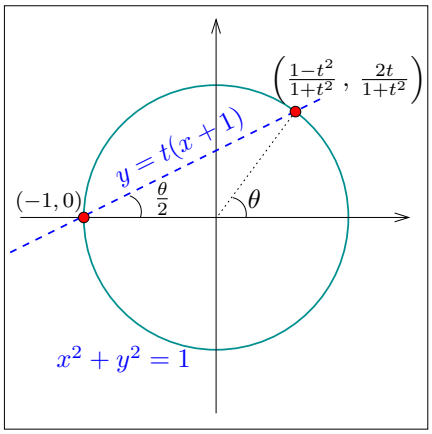


Figure A.2: The chord method on the unit circle.

In order to use this method in our algorithm, and since we typically deal with conics that have rational (and in many cases integer) coefficients, the problem that remains is to find one rational point on the conic. Apparently, this is not an easy question. Moreover, not all conics with rational coefficients have rational points on them. One nice example of a conic with no rational points is: $x^2 + y^2 = 3$. Cassels [11, Chapter 6] shows an algorithm to check if a conic equation has a rational solution or not, and if there is, a method to find such a point. This algorithm works in $O(F^2)$, where F is the sum of the absolute values of the coefficients of the conic equation.

In cases there is no rational point on the curve, the question is how to find a good rational approximation to the vertex, that is still in one of the faces incident to the vertex. What we actually need is a good small[2] rational approximation to the number that will be close up to ε to the original number. ε should be such that when we move ε away from the vertex v we only get into a face incident to v and not cross any other curve. Another way to solve the problem for curves that contain no rational points is to find another point on the curve that is not an intersection point, and thus has only algebraic degree 2.

In cases where a rational point is known, such as circles with rational center point and rational radius, this method can be applied. We leave this question for further study.

[2]A "small" rational number is such that both its enumerator and denominator can be represented with a small number of bits.

Bibliography

[1] 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.

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

[3] S. Arya, T. Malamatos, and D. M. Mount. A simple entropy-based algorithm for planar point location. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 262–268, 2001.

[4] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

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

[6] M. H. Austern. *Generic Programming and the STL.* Addison-Wesley, 1999.

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

[8] J. L. Bentley. Kd-trees for semidynamic point sets. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[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] J. Canny, B. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. 8th Annu. ACM Sympos. Comput. Geome.*, pages 251–260, 1992.

[11] J. W. S. Cassels. *Rational Quadratic Forms.* Academic Press, 1978.

[12] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

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

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

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

[16] 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.

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

[18] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5(2):181–186, 1976.

[19] 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.

[20] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *ACM Journal of Experimental Algorithmics*, 5, 2000. Special Issue, selected papers of the Workshop on Algorithm Engineering (WAE).

[21] 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.

[22] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[24] 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.

[25] I. Haran and D. Halperin. An experimental study of point location in general planar arrangements. In *Proc. ALENEX/ANALCO*, 2006.

[26] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 351–359, 1999.

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

[28] S. M. LaValle. *Planning Algorithms*. Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/), 2006.

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

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

[31] 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.

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

[33] N. Myers. A new and useful template technique: "Traits". In S. B. Lippman, editor, *C++ Gems*, volume **5** of *SIGS Reference Library*, pages 451–458. 1997.

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

[35] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10(3):473–482, 1981.

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

[37] S. Schirra. Robustness and precision issues in geometric computation. Research Report MPI-I-98-1-004, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1998.

[38] 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.

[39] M. I. Shamos. Geometric complexity. In *Proc. 7th ACM Sympos. Theory of Computing*, pages 224–233, 1975.

[40] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 34, pages 529–562. Chapman & Hall/CRC, 2nd edition, 2004.

[41] R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. In *Proc. Library-Centric Software Design Workshop (LCSD'05)*.

[42] C. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapman & Hall/CRC, 2nd edition, 2004.