Raymond and Beverly Sackler
Faculty of Exact Sciences
The Blavatnik School of Computer Science

Guaranteed Logarithmic-Time Point Location in General Two-Dimensional Subdivisions

New Bounds, Algorithms, and Implementation

Thesis submitted in partial fulfillment of the requirements for the M.Sc.

degree in the School of Computer Science, Tel-Aviv University

by

Michal Kleinbort

This work has been carried out at Tel-Aviv University
under the supervision of Prof. Dan Halperin

February 2013

Acknowledgements

I wish to express my sincere gratitude to my advisor, Prof. Dan Halperin, who offered me guidance, insight, and encouragement. His support and assistance were a tremendous help both for this research and for me.

I deeply thank Michael Hemmer from Tel Aviv University for the ongoing and fruitful collaboration, technical assistance, and many insights. I would also like to thank all other members of the Applied Computational Geometry group at the Computer Science school of Tel Aviv University for their support as colleagues and more importantly as friends.

Additionally, I would like to express my love and gratitude to my beloved families — to my parents and my parents-in-law for their tremendous help with my children, to my children Yonatan and Uri for being who they are and for reminding me the true joys in life, and, finally, to my loving husband Asaf for his endless support, encouragement, and understanding.

Clearly, without the support of these many people, this thesis would not have been possible.

This work has been supported in part by the 7th Framework Programme for Research of the European Commission, under FET-Open grant number 255827 (CGL—Computational Geometry Learning).

Abstract

This thesis studies certain theoretical aspects of planar point location using the randomized incremental construction of the trapezoidal-map. Moreover, it describes a major revision of the corresponding implementation in CGAL, the Computational Geometry Algorithms Library. We focus on a variant of the algorithm, which guarantees that the constructed search structure, a directed acyclic graph $\mathcal{G}$, is of linear size and provides logarithmic query time. A major challenge is to retain the expected $O(n \log n)$ preprocessing time while providing the above (deterministic) space and query-time guarantees. We describe two efficient preprocessing algorithms, which explicitly verify the length $\mathcal{L}$ of the longest query path. One runs in expected $O(n \log^2 n)$ time while the other runs in expected $O(n \log n)$ time only. The former, although slower, is simpler and in particular it does not require the construction of auxiliary structures.

Our revised CGAL implementation can now guarantee linear size and logarithmic query time. Another major innovation is the support of general unbounded subdivisions as well as subdivisions of two-dimensional parametric surfaces such as spheres, tori, cylinders. Like the previous implementation, it is exact, complete, and general, i.e., it can also handle non-linear subdivisions. The data structure also supports modifications of the subdivision, such as insertions and deletions of edges, after the initial preprocessing. However, instead of using $\mathcal{L}$, our implementation is based on the depth $\mathcal{D}$ of $\mathcal{G}$, which is a more reasonable choice in practice. Although we prove that the worst case ratio of $\mathcal{D}$ and $\mathcal{L}$ is $\Theta(n/\log n)$, we conjecture, based on our experimental results, that this solution achieves expected $O(n \log n)$ preprocessing time.

Via the construction of the Voronoi diagram for a given point set S of size n this also enables nearest-neighbor queries in guaranteed $O(\log n)$ time.

Contents

1 Introduction

Imagine the use of a maps application for mobile devices based on GPS coordinates. The latitude and longitude coordinates are used for determining the current location on earth easily. Often, in such maps applications, the street containing the given coordinates is returned as an output. One can define a planar subdivision (arrangement) based on a given map. The streets and the junctions defined in the underlying map correspond to the edges and vertices, respectively. The faces represent regions between different streets and roads. An example of such a planar subdivision is presented in Figure 1.1. However, since GPS is not exact, measuring errors should be taken into account, that is, it is likely that one would not hit the exact street at which the device is. Thus, within these maps, whose ambient dimension is two, the edges are treated as one-dimensional, i.e., have no width. Therefore, the search for a point, given in GPS coordinates, would locate the subdivision face containing the point and would then compare the point to all edges on the boundary of the face in order to return the closest one.

The maps application is a good example for point location with a rather static set of geometric objects, since the maps data is known in advance. However, the maps can be updated from time to time, therefore, having a dynamic method, which allows additional updates, such as deletions and further insertions, rather than reconstructing the whole structure, may be advantageous.

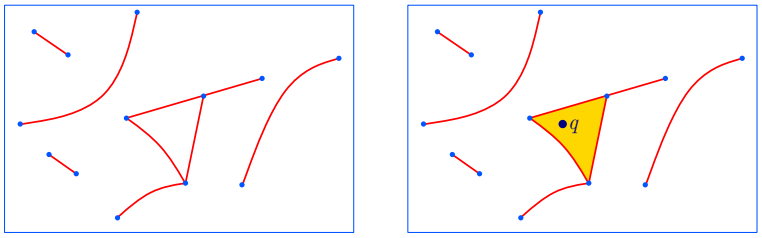


Figure 1.1: A planar point location example for a non-linear subdivision. (a) The planar subdivision (b) Marking the subdivision face containing a query point q

The planar point location problem for a set S of n pairwise interior disjoint x-monotone curves inducing a planar subdivision (or a planar arrangement) $\mathcal{A}(S)$ is defined as follows: given a query point q, locate the feature of $\mathcal{A}(S)$ containing q, i.e., the face, edge or vertex of $\mathcal{A}(S)$ that q lies in. In our mobile application example, the subdivision $\mathcal{A}(S)$ represents a map, and the query point is specified by GPS coordinates. However, the planar point location problem appears in many other fields as well. It is a fundamental problem in Computational Geometry and has numerous applications in a variety of domains, such as computer graphics, motion planning, computer aided design (CAD), geographic information systems (GIS), and many more.

This thesis concerns theoretical aspects of the planar point location problem as well as its implementation in CGAL, the Computational Geometry Algorithms Library [48]. CGAL is an open-source C++ library containing various efficient geometric algorithms. As CGAL, we follow the exact geometric computation paradigm [44,45] yielding robust implementation with exact results.

Previous Work

As a core problem in Computational Geometry, the planar point location problem has been well-studied for many years. Among the various solutions to the problem, some methods can only provide an *expected* query time of $O(\log n)$ but can not guarantee it for all possible scenarios. It is particularly true for solutions that only require $O(n)$ space. In addition, certain solutions may only support linear subdivisions, while others are applicable to non-linear ones as well. Triangulation-based point location methods, such as Kirkpatrick's approach [29] and Devillers's Delaunay Hierarchy [15] are restricted to linear subdivisions, since they build on a triangulation of the actual input. Kirkpatrick creates a hierarchy of $O(\log n)$ levels of triangulated faces (including the outer face), where at each level an independent-set of low-degree vertices is removed when creating the next level in the hierarchy. This approach guarantees that the data structure requires only $O(n)$ space and that a query takes only $O(\log n)$ time. The *Delaunay Hierarchy* of Devillers, on the other hand, does not guarantee logarithmic query time, and may have a linear query time in the worst-case. Nevertheless, it has an exact implementation in CGAL, which performs well for random data. The different behavior of the two approaches is due to the different methods for selecting the next-level vertices. It should be noted that large constants are involved in the complexity bounds of the hierarchical triangulation methods.

The *Landmarks* point location strategy [26] is a method maintaining good running times in practice, which is available for CGAL arrangements. The Landmarks strategy combines a nearest-neighbor search to find the nearest landmark, and a hopefully short walk in the full subdivision from the landmark to the query point. It is a heuristics, and therefore does not guarantee a logarithmic query time.

Most of the other methods can be summarized under the *trapezoidal search graph* model of computation, as pointed out by Seidel and Adamy [39]. The fundamental search structure used by this model is a directed acyclic graph G (which may even be just a tree for some methods) with one root and many leaves. Internal nodes in G have two outgoing edges each,

and are either labeled with a vertical line and are therefore left-right nodes, or labeled by an input curve and in such a case are top-bottom nodes. A search for a query point q starts from the root of G and at each encountered internal node the query point q is compared to the geometric entity represented by the node. In principal, all these solutions can be generalized to support input curves that are decomposable into a finite number of x-monotone pieces.

One of the earliest solutions that can be subsumed under this model is known as the *slabs method* introduced by Dobkin and Lipton [17]. Every endpoint induces a vertical wall giving rise to $2n + 1$ vertical slabs. Within each slab, the curves are efficiently stored according to their vertical order. A point location query is performed by a binary search to locate the correct slab and another search within the slab in $O(\log n)$ time. Preparata [35] introduced the *Trapezoid Graph method* based on the slabs method. His method, reduces the space bounds from $O(n^2)$, as required by Dobkin and Lipton's slabs method, to $O(n \log n)$ only, by uniquely decomposing each edge into $O(\log n)$ fragments. Sarnak and Tarjan [37] achieved a significant improvement in memory usage for a slabs-based method by using a *persistent* data structure. Their key observation is that the sequence of search structures in all slabs can be interpreted as one structure that changes over time, which can be stored as a persistent data structure requiring only $O(n)$ size. Another example for this model is the separating chains method by Lee and Preparata [30]. Their algorithm expects a monotone subdivision and uses horizontal monotone chains to separate faces. It is based on the idea that faces of any monotone subdivision can be totally ordered preserving the above-below relation. Each chain is a node in a binary search tree (each edge is kept only once). Querying the structure is essentially deciding whether the query point is above or below $O(\log n)$ chains. However, for each chain this test takes $O(\log n)$, using a binary search. Therefore, the total query time is $O(\log^2 n)$. Edelsbrunner et al. [18] used Fractional Cascading in order to create a layered chain tree as a search structure by copying every other x-value from a node to its parent and maintaining pointers from parent list to child lists. Querying this structure takes $O(\log n)$ time.

The most relevant method in the context of this thesis is the trapezoidal map randomized incremental construction (RIC). This asymptotically optimal solution was introduced by Mulmuley [32] and Seidel [38]. It uses a Directed Acyclic Graph (DAG) recording the history of the randomized incremental construction (RIC) of the trapezoidal map for an arrangement of x-monotone curves. In static settings, where the subdivision is unchanged, it achieves expected $O(n \log n)$ preprocessing time, expected $O(\log n)$ query time and expected $O(n)$ space. As pointed out by De Berg et al. [14], the latter two can even be guaranteed, when applying minor modifications to the basic algorithm. It is also claimed there that one can achieve an expected preprocessing time of $O(n \log^2 n)$, but no concrete proof is given. This approach is able to handle dynamic scenes as well, namely, it is possible to add or delete edges later on. The entire method is discussed in more detail in Chapter 2 below. The above five methods are all applicable for arbitrary subdivisions of well-behaved curves [20, Section 1.3.3] as one can switch to monotone subdivisions by decomposing every curve into a set of x-monotone sub-curves in time which is proportional to the total number of x-monotone edges.

A variant of the latter adds weights and thus gives expected query time satisfying entropy bounds [5]. Arya et al. also stated that entropy preserving cuttings can be used to give a

method the query time of which approaches the optimal entropy bound, at the cost of increased space and programming complexity [4]. These methods guarantee a logarithmic query time, however maintaining the search structures requires a considerably large amount of memory and a significant increase in the preprocessing time. Therefore, these solutions are generally rather complicated to implement. For other methods and variants the reader is referred to a comprehensive overview given in [40].

Contribution

This thesis studies certain theoretical aspects of planar point location using the trapezoidal-map random incremental construction algorithm as well as describes a major revision of its implementation in CGAL. Part of the work in this thesis was presented in the European Symposium on Algorithms (ESA) 2012 [27]. Our implementation provides a data structure for arbitrary subdivisions that can handle static settings. Dynamic settings, where both insertions and deletions of edges are allowed, are supported as well. However, this method is best used when not too many deletions occur and the settings are mostly static. Like the previous implementation by Oren Nechushtan [19], it is part of the "2D Arrangements" package [42] of CGAL. As it supports arbitrary subdivisions, both linear and non-linear subdivisions represented in CGAL by the "2D Arrangements" package can be given as input. The implementation is exact, complete, and general. The main new feature is that we are now able to guarantee for any input both $O(\log n)$ query time and $O(n)$ space. Moreover, the trigger for this major revision is the added support for unbounded curves, as it was introduced for the "2D Arrangements" package in [11]. In particular, it is now possible to provide guaranteed logarithmic point location for subdivisions embedded on two-dimensional orientable parametric surfaces in three-dimensional space (e.g., spheres, tori, etc.) [10, 11, 42, 43]. We conjecture, based on our experimental results, that the expected preprocessing time is $O(n \log n)$.

Chapter 2 discusses in detail the fundamental algorithm by Mulmuley [32] and Seidel [38]. A variant by De Berg et al. [14], which is the implementation's basis, is thoroughly described there as well. The latter guarantees logarithmic query time by constructing the directed acyclic graph (DAG) using the basic algorithm. However, it rebuilds the DAG if the length of the longest search path $\mathcal{L}$ or the size $\mathcal{S}$ exceed some thresholds. Therefore, in order to bound the runtime of the guaranteed variant both $\mathcal{L}$ and $\mathcal{S}$ must be efficiently accessible. An early version of [14] did not make the distinction between $\mathcal{L}$ and the depth $\mathcal{D}$ of the DAG, which is the length of the longest DAG path. Chapter 3 discusses the fundamental difference between the two quantities, $\mathcal{D}$ and $\mathcal{L}$. In fact, we show there that the worst case ratio between $\mathcal{D}$ and $\mathcal{L}$ can be $\Theta(n/\log n)$. As argued there, $\mathcal{D}$ and $\mathcal{L}$ are not trivially interchangeable. In particular, determining the value of $\mathcal{L}$ is rather expensive while $\mathcal{D}$ is easy to maintain. In Chapter 4 two construction algorithms for static settings are presented. For the first one we can show an expected $O(n \log^2 n)$ time complexity. However, we believe that using a more refined analysis an expected $O(n \log n)$ bound is achievable. For the second algorithm we can prove an expected $O(n \log n)$ bound on the runtime. The technical details of the implementation in CGAL are given in Chapter 5 as well as the details about the

previous implementation by Oren Nechushtan. There we also compare our implementation to other point location methods available for CGAL arrangements. A possible application for the planar point location implementation is a guaranteed logarithmic planar nearest-neighbor search, which is obtained by using our point location implementation on top of the Voronoi diagram of the input points. This application is described in detail in Chapter 6 with appropriate experiments. Our conclusions and open problems are given in Chapter 7.

In summary, this thesis contains a theoretical solution for constructing in expected $O(n \log n)$ time a linear size data structure for point location with logarithmic query time for a subdivision of n pairwise disjoint x-monotone curves (this bound is true only for static settings). However, we decided to implement the variant that uses the depth $\mathcal{D}$ of the DAG for which we conjecture, based on our experiments, that the preprocessing time is expected $O(n \log n)$ as well. Our implementation is exact and general, supporting both linear and non-linear subdivisions, and can handle unbounded curves as well. It guarantees logarithmic query time and linear size for all possible inputs. To the best of our knowledge, this is the only existing implementation having such properties.

2 Background

This chapter describes the trapezoidal map random incremental construction for point location in detail. After some relevant general definitions in Section 2.1, the basic algorithm presented by Mulmuley [32] and Seidel [38] is provided in Section 2.2. In Section 2.3 a variant by De Berg et al. [14] is described, on which the implemented algorithm is based.

2.1 Definitions

Let S be a set of n pairwise interior disjoint x-monotone curves inducing a planar subdivision (or a planar arrangement) $\mathcal{A}(S)$. For ease of reading let us assume that the input curves are in general position, i.e., no two distinct endpoints have the same x-coordinate and no endpoint of one curve lies in the interior of another curve. The arrangement $\mathcal{A}(S)$ is composed of vertices, and faces, in addition to its n edges.

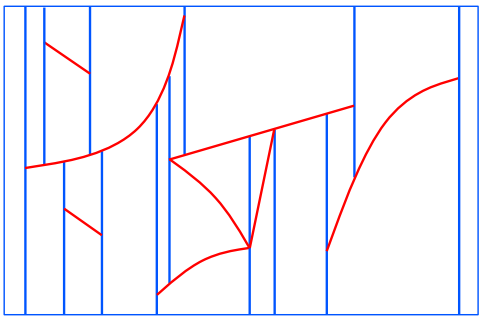


Figure 2.1: The trapezoidal map $\mathcal{T}(S)$ for an arrangement $\mathcal{A}(S)$. The edges of $\mathcal{A}(S)$ are in red while the vertical walls are in blue.

The *Trapezoidal Map* of an arrangement $\mathcal{A}(S)$, denoted by $\mathcal{T}(S)$, is obtained by extending vertical walls from each endpoint upwards and downwards until an input curve is reached or the wall extends to infinity. In general position, each trapezoid of $\mathcal{T}(S)$ is defined by at most two edges of $\mathcal{A}(S)$ and has one or two vertical edges (trapezoid bases)[1]. Each vertical edge can be represented by the one endpoint inducing the wall it lies on. Therefore, each trapezoid Δ in $\mathcal{T}(S)$ can be defined by a

[1] We use the term *trapezoid* even when the side edges are not linear segments.

unique quadruplet: $\langle left(\Delta), right(\Delta), bottom(\Delta), top(\Delta)\rangle$, in which $left(\Delta), right(\Delta)$ represent the left and right endpoints (inducing the left and right vertical edges), respectively, and $bottom(\Delta), top(\Delta)$ represent the bottom and top edges, respectively. Figure 2.2 illustrates a basic trapezoid. In a degenerate trapezoid, having only one vertical edge (a triangle), the two non-vertical edges are adjacent. The trapezoidal map $\mathcal{T}(S)$ is unique and does not depend on the order of insertion. As shown in [14], $\mathcal{T}(S)$ of an arrangement consisting of n edges in general position, has at most $3n+1$ trapezoids and at most $6n+4$ vertices. In other words, the trapezoidal map of an arrangement of linear size has a linear size as well.

The trapezoids in $\mathcal{T}(S)$ are neighbors if they share a vertical wall. In general position, each trapezoid has at most four neighboring trapezoids: at most two along its left vertical edge, and the same along its right vertical edge. In order to avoid the general position assumption, a symbolic shear transformation should be used. This is done by employing lexicographical comparison, that is, comparing first by the x-coordinate and then by the y-coordinate. This implies that two covertical points produce a virtual trapezoid, which has zero width.

For simplicity of presentation, and w.l.o.g., the following figures contain horizontal line-segments and are clipped using a bounding rectangle. However, both our analysis and software are for general x-monotone curves and do not depend on the existence of bounding boxes.

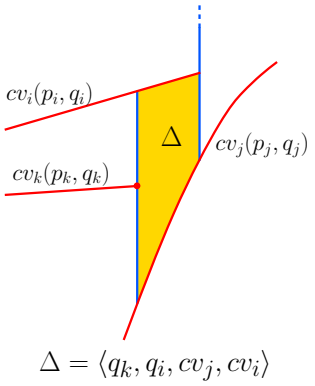


Figure 2.2: A trapezoid.

2.2 The Basic RIC Algorithm

We review here the random incremental construction (RIC) of an efficient point location structure, as introduced in [32, 38] and described in [14, 33]. Given an arrangement $\mathcal{A}(S)$ of n pairwise interior disjoint x-monotone curves, a random permutation of the curves is inserted incrementally, constructing the trapezoidal map $\mathcal{T}(S)$. During the incremental construction, an auxiliary search structure, a directed acyclic graph (DAG) G, is maintained. The DAG G has one root and many leaves, one for every trapezoid in the trapezoidal map $\mathcal{T}(S)$. Every internal node is a binary decision node, representing either an endpoint p, deciding whether a query point q lies to the left or to the right of the vertical line through p, or an x-monotone curve cv_i, deciding whether the query point q is above or below it. When we reach a curve-node representing a curve cv_i, we are guaranteed that the query point q lies in the x-range of cv_i. In Figure 2.3, the DAG root is an internal node representing an endpoint p_1 (in purple). This DAG also includes a curve-node representing cv_1, which is marked in orange. Finally, there are four leaves, representing the four trapezoids in the map. Note that this DAG is still a tree.

The trapezoids in the leaves are interconnected, such that each trapezoid knows its (at most) four neighboring trapezoids (two to the left and two to the right, as explained above). In particular, there are no common x-coordinates for two distinct endpoints, since we use a

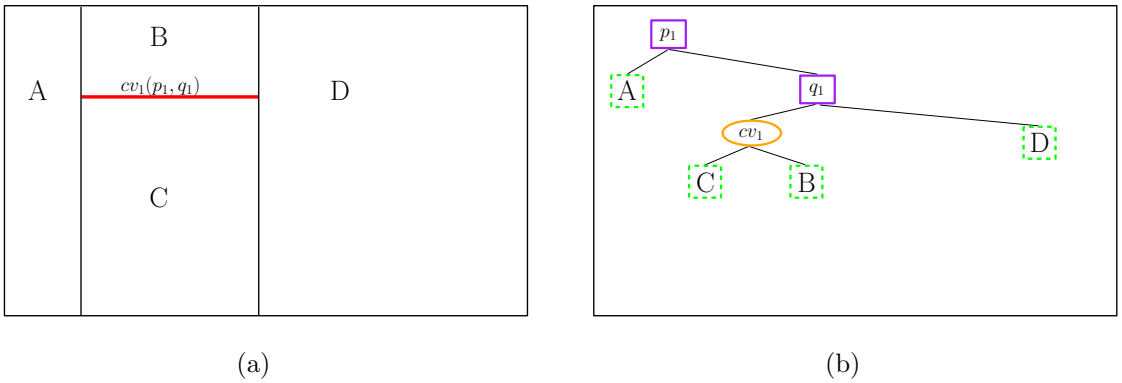


Figure 2.3: (a) The trapezoidal map $\mathcal{T}(S)$ of an arrangement $\mathcal{A}(S)$ containing one curve $cv_1(p_1, q_1)$ (marked in red). (b) The corresponding DAG G for $\mathcal{T}(S)$, which is currently still a tree.

symbolic shear transform. For example, trapezoid A in Figure 2.3, has only two neighbors, namely B and C, which are its top-right and bottom-right neighbors, respectively. Trapezoid C has also two different neighbors, which are trapezoid A to its left and trapezoid D to its right. In the underlying data structure, each trapezoid maintains four pointers to its potential neighbors.

2.2.1 Querying

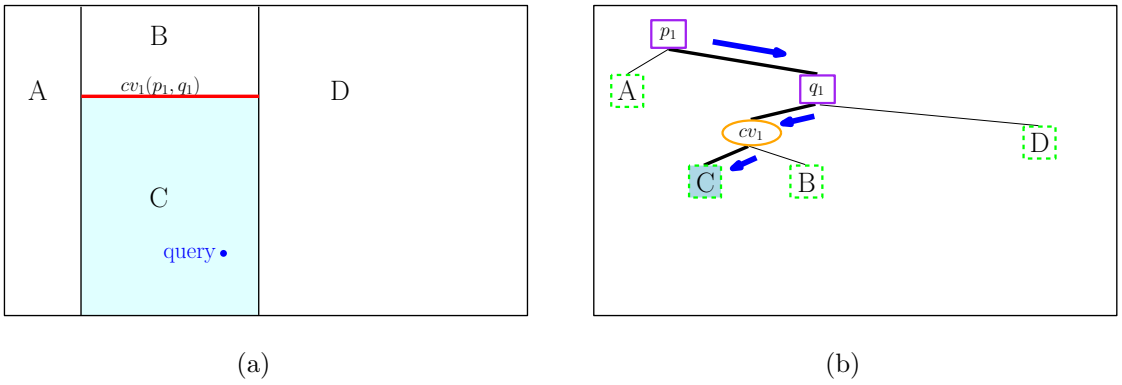


Figure 2.4: Locating query point q in the DAG. (a) The trapezoid containing the query point q is marked. (b) The query path for q along the history DAG is marked with arrows.

In order to locate the trapezoid containing a query point q one needs to perform a search in the history DAG. The search begins at the root of the DAG and ends in a DAG leaf. At each internal node a decision is made, depending on whether the node represents an endpoint or a curve, in order to find the next node in the path. In other words, the number of comparisons used by one query equals to the number of internal nodes along a search path to the leaf representing the trapezoid that contains the query point. Therefore, the query time is proportional to the length of the path.

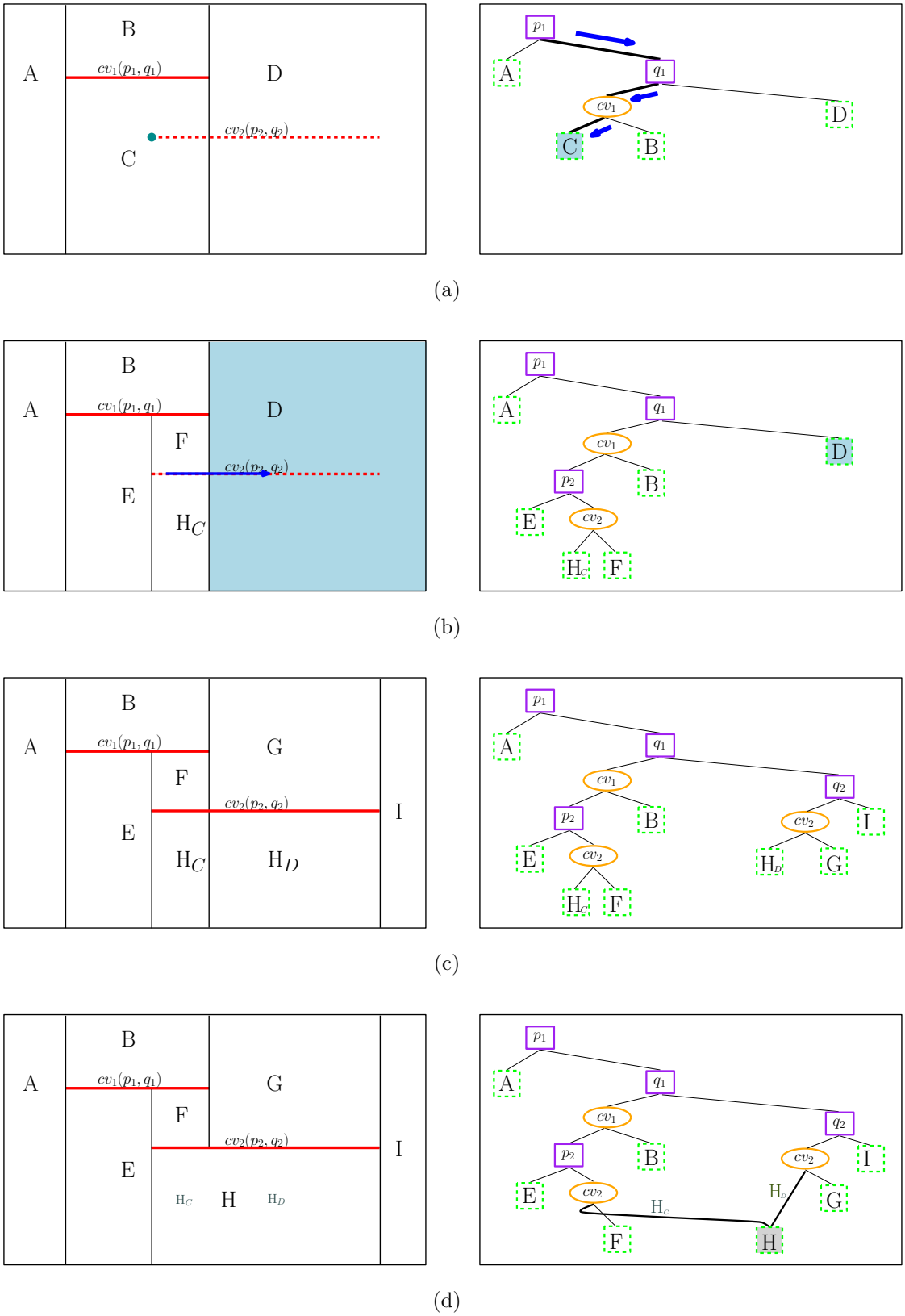
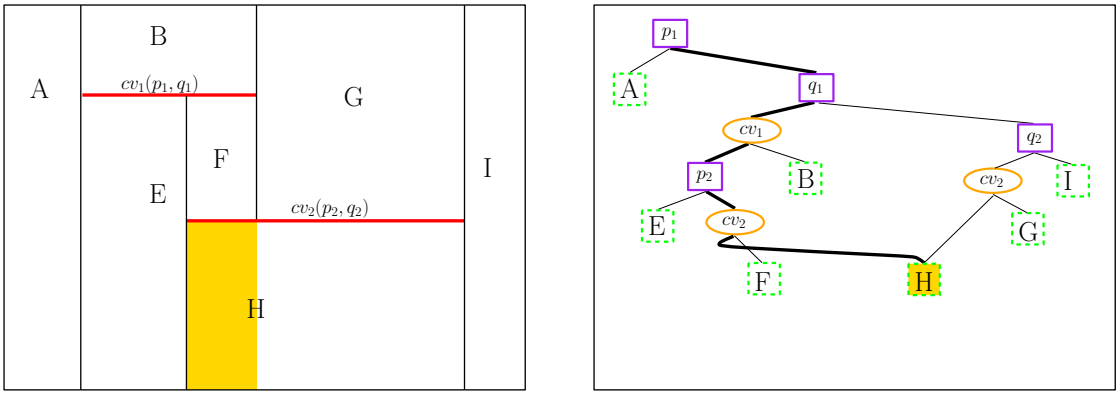


Figure 2.5: Updating the DAG with a second curve $cv_2(p_2, q_2)$. (a) Locating p_2 (the left endpoint of cv_2). (b) Splitting the trapezoid C, which contains p_2, into trapezoids E, F, H_C. The right neighbor of C, trapezoid D, is the next trapezoid to split. (c) Splitting trapezoid D into trapezoids H_D, G, I. (d) Newly created trapezoids H_C, H_D are merged into trapezoid H, since they share the same top and bottom curves, resulting in the updated DAG. Note that the search structure is no longer a tree, due to the merge.

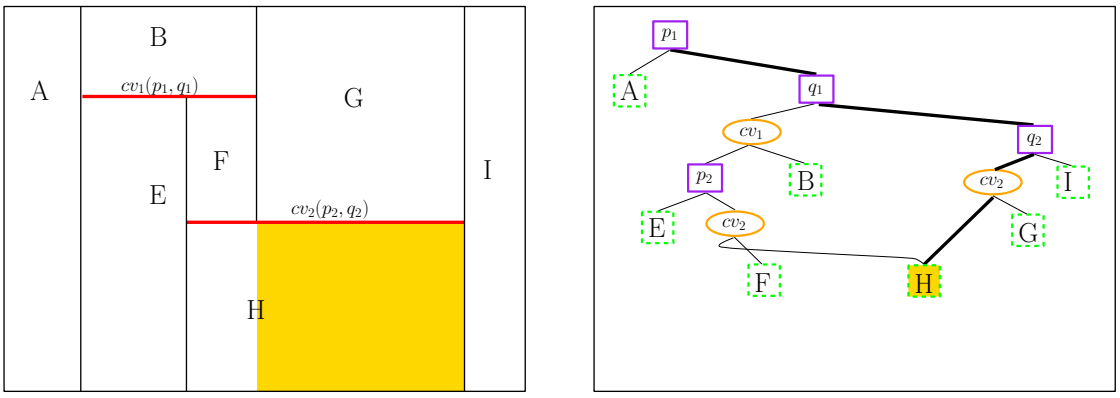
2.2.2 Insertion

When a new x-monotone curve is inserted, the trapezoid containing its left endpoint is located by a search from root to leaf. Then, using the connectivity information described above, the trapezoids intersected by the curve are gradually revealed and updated. Merging new trapezoids, if needed, takes time that is linear in the number of intersected trapezoids. The merge turns the data structure into a DAG with expected $O(n)$ size, instead of an $\Omega(n \log n)$ size binary tree [39]. The whole insertion process is illustrated in Figure 2.5. For an unlucky insertion order the size of the resulting data structure may be quadratic, and the longest search path may be linear. However, since the curves are inserted in a random order, one can expect $O(n)$ space, $O(\log n)$ query time, and $O(n \log n)$ preprocessing time. For proofs see [14, 32, 38].

As a result of the merge, the search structure in Figure 2.5 contains two directed paths from the root to the leaf labeled H. Figure 2.6 demonstrates the two paths, which are both valid search paths.



(a)



(b)

Figure 2.6: Two search paths to H. The paths illustrated in (a) and (b) are realized by queries in the left and right regions of trapezoid H, respectively. These regions are marked with yellow in the matching trapezoidal map figures (left-hand side).

2.3 A Guaranteed Logarithmic Query Time and Linear Size Variant

De Berg et al. [14] show that one can build a data structure, which guarantees $O(\log n)$ query time and $O(n)$ size, by monitoring the size $\mathcal{S}$ and the length of the longest search path $\mathcal{L}$ during the construction. The idea is that as soon as one of the values becomes too large, the structure is rebuilt using a different random insertion order. It is shown that only a small constant number of rebuilds is expected. We now analyze the algorithm presented in [14] in more details.

The basic algorithm, described in Section 2.2, bounds the query time with a logarithmic bound, in expectation. However, it can be shown that the probability that the maximum query path length $\mathcal{L}$ is bad (larger than some defined bound) is very small.

Lemma 2.1. *Let S be a set of n non-crossing x-monotone curves, let q be a query point, and let $\lambda > 0$ be a parameter. The probability that the search path for q in the DAG has more than $3\lambda \ln(n+1)$ nodes is at most $1/(n+1)^{\lambda \ln 1.25 - 1}$.*

The proof can be found in [14, Section 6.4]. The next lemma is used to bound the expected value for the length $\mathcal{L}$ of the longest query.

Lemma 2.2. *Let S be a set of n non-crossing x-monotone curves, and let $\lambda > 0$ be a parameter. The probability that $\mathcal{L}$, the maximum search path length, is greater than $3\lambda \ln(n+1)$ is at most $2/(n+1)^{\lambda \ln 1.25 - 3}$.*

Proof sketch: The idea is that by extending the vertical walls to full lines, the plane is decomposed into at most $2(n+1)^2$ regions. The search paths for two query points q, q' lying in the same region are identical. Therefore, it is sufficient to consider the search paths of representative queries for these regions. Using Lemma 2.1 the required result follows. Obviously, we can choose an appropriate value for λ such that the probability that the maximum query path length is "bad" is sufficiently small. For instance, choosing $\lambda = 20$ implies that the probability that for the search structure $\mathcal{L} \leq 60 \ln(n+1)$ is at least $3/4$.

Moreover, the following lemma, which does not appear in [14], shows that the probability that the size $\mathcal{S}$ of the search structure is too big is very small.

Lemma 2.3. *Let S be a set of n non-crossing x-monotone curves, and let $\rho \geq 1$ be a parameter. The probability that $\mathcal{S}$ is more than $15\rho n$ is at most $\frac{1}{\rho}$.*

Proof. Let $\mathcal{C}$ be a non-negative random variable representing the number of DAG nodes, i.e., the size $\mathcal{S}$ of the search structure. $\mathcal{C}$ is composed of the number of leaves in the final search structure and the sum of inner nodes created in all iterations. The number of leaves equals to $|\mathcal{T}(S)|$, the number of trapezoids in the final trapezoidal map. Hence, it is at most $3n+1$. Let k_i denote the number of trapezoids created by the insertion of the ith curve. Let $\mathcal{A}(S_i)$ denote the arrangement of the first i curves, and $\mathcal{T}(S_i)$ denote the matching trapezoidal map for this arrangement. The number of inner nodes created in iteration i is equal to the number of trapezoids that are split by the interior (not endpoints) of the inserted curve cv_i

plus the number of endpoints of cv_i that do not exist in the current structure $\mathcal{T}(S_{i-1})$. In fact, we can count the number of vertical walls that are blocked by the cv_i (which is equal to the number of trapezoids that are split by the interior of $cv_i - 1$), since the trapezoid lying on the left side of the blocked wall is new. If the left endpoint of cv_i is inserted to the structure (an inner node for this endpoint is added) then a new trapezoid to its left is added. Similarly, if the right endpoint of cv_i is inserted to the structure then a new trapezoid to its right is added. Therefore, the number of inner nodes created in iteration i is exactly the number of new trapezoids created in the same iteration minus 1, i.e., $k_i - 1$.

The expected number of DAG nodes, $\mathbb{E}[\mathcal{C}]$, can be bounded as follows:

$$\begin{aligned}
\mathbb{E}[\mathcal{C}] &= \mathbb{E}[|\mathcal{T}(S)| + \sum_{i=1}^{n}(k_i - 1)] \\
&= \mathbb{E}[|\mathcal{T}(S)|] + \mathbb{E}[\sum_{i=1}^{n}(k_i - 1)] \\
&= \mathbb{E}[|\mathcal{T}(S)|] + \mathbb{E}[\sum_{i=1}^{n} k_i] - n \\
&\leq (3n+1) + \mathbb{E}[\sum_{i=1}^{n} k_i] - n \\
&= 2n + 1 + \mathbb{E}[\sum_{i=1}^{n} k_i]\,.
\end{aligned}$$

Using linearity of expectation we get:

$$\mathbb{E}[\mathcal{C}] \leq 2n + 1 + \sum_{i=1}^{n} \mathbb{E}[k_i].$$

The expected number of new trapezoids created in the ith iteration can be bounded using the following backwards analysis. Each trapezoid is defined by at most four curves and by removing one of these curves the trapezoid is destroyed. If cv_i, the ith inserted curve, is removed from $\mathcal{T}(S_i)$, the probability that a given trapezoid Δ is destroyed is at most $4/i$. Since $|\mathcal{T}(S_i)|$ is the number of trapezoids in the trapezoidal map for the first i inserted curves, the expected number of trapezoids that are destroyed if cv_i is removed from $\mathcal{T}(S_i)$ is at most $|\mathcal{T}(S_i)| \cdot 4/i$, which is the expected number of new trapezoids created in the ith iteration. In addition, as we already mentioned, the number of trapezoids in a trapezoidal map of i curves in general position is at most $3i + 1$. Therefore, $\mathbb{E}[k_i]$ can be bounded as follows:

$$\mathbb{E}[k_i] \;\leq \frac{4 \cdot |\mathcal{T}(S_i)|}{i} \leq \frac{4(3i+1)}{i} = 12 + \frac{4}{i}\,.$$

Putting it all together:

$$\mathbb{E}[\mathcal{C}] \leq 2n + 1 + \sum_{i=1}^{n}(12 + \frac{4}{i}) = 14n + 1 + 4\sum_{i=1}^{n}\frac{1}{i} = 14n + 1 + 4H_n\,,$$

where H_n is the n-th harmonic number and $\ln n < H_n < \ln n + 1$, and therefore:

$$\mathbb{E}[\mathcal{C}] < 14n + 4\ln n + 2 < 15n, \text{ for } n \geq 12.$$

Since $\mathcal{C}$ is a non-negative random variable we can use Markov's inequality, according to which for any $\alpha > 0$ we have:

$$\Pr[\mathcal{C} \geq \alpha] \leq \frac{\mathbb{E}[\mathcal{C}]}{\alpha}.$$

Now, we choose $\alpha = 15\rho n$ and substitute

$$\Pr[\mathcal{C} \geq 15\rho n] \leq \frac{\mathbb{E}[\mathcal{C}]}{15\rho n} = \frac{15n}{15\rho n} = \frac{1}{\rho}.$$

□

Finally, we can choose an appropriate value for ρ such that the probability that the size is "bad" is small enough. If we choose $\lambda = 20$ and $\rho = 4$ then by Lemma 2.2 the probability that for the search structure $\mathcal{L} \leq 60\ln(n+1)$ occurs is at least $3/4$. Similarly, by Lemma 2.3, the probability that $\mathcal{S} \leq 60n$ is at least $3/4$. Therefore, the probability that both the size and the maximal query length are good is at least $1/2$.

The algorithm of the guaranteed logarithmic query time and linear size variant is as follows. We observe the size $\mathcal{S}$ and the maximum query path length $\mathcal{L}$ during the construction using the basic algorithm described in Section 2.2. If at some stage of the construction $\mathcal{S} \geq c_1 n$ or $\mathcal{L} \geq c_2 \log n$ occurs (for suitably chosen constants $c_1, c_2 > 0$), then we abort the construction and rebuild the structure using a different random permutation of the input curves. It follows from both Lemma 2.2 and Lemma 2.3 that only a constant number of rebuilds is expected. Since an insertion takes $O(\log n)$ time we observe the following:

Observation 2.4. *If the size $\mathcal{S}$ and the maximum query path length $\mathcal{L}$ are accessible in $O(\log n)$ time then the expected preprocessing time of the algorithm described above is $O(n \log n)$.*

The size $\mathcal{S}$ can be trivially made efficiently accessible (in fact, it can even be accessed in constant time). It is not clear, however, how to achieve this for $\mathcal{L}$. This topic will be thoroughly discussed both in Chapter 3 and Chapter 4. The proof of Lemma 2.2 above relies on the fact that there are at most $2(n+1)^2$ query paths in the DAG. In [14] it is mentioned that the number of representative points for different query paths can be reduced to $O(n \log n)$ only, setting the bound of the expected preprocessing time for static settings to be $O(n \log^2 n)$.

3

Depth vs. Maximum Query Path Length

An algorithm for constructing a point location data structure, which guarantees $O(\log n)$ query time and $O(n)$ size, was described in Section 2.3. This structure is constructed while monitoring the size $\mathcal{S}$ and the maximum search path length $\mathcal{L}$. However, in order to retain the expected construction time of $O(n \log n)$, both values, namely, $\mathcal{S}$ and $\mathcal{L}$, must be efficiently accessible. While this is trivial for the size $\mathcal{S}$, it is not clear how to achieve this for the maximum query path length $\mathcal{L}$. Hence, we resort to the depth $\mathcal{D}$ of the DAG, which is an upper bound on $\mathcal{L}$ as the set of all possible search paths is a subset of all paths in the DAG. Thus, the resulting data structure still guarantees a logarithmic query time.

The depth $\mathcal{D}$ can be made accessible in constant time, by storing the depth of each leaf in the leaf itself, and maintaining the maximum depth in a separate variable. The cost of maintaining the depth can be charged to new nodes, since existing nodes never change their depth value. This is not possible for $\mathcal{L}$ while retaining linear space, since each leaf would have to store a non-constant number of values, i.e., one for each valid search path that reaches it. In fact the memory consumption would be equivalent to the data structure that one would obtain without merging trapezoids, namely the trapezoidal search tree, which for certain scenarios requires $\Omega(n \log n)$ memory as shown in [39]. In particular, it is necessary to perform merges as the sizes of the resulting search tree and the resulting DAG considerably differ also in practice, as demonstrated in Table 3.1.

Section 3.1 shows that the depth $\mathcal{D}$ of a given DAG can be linear while its maximum query path length $\mathcal{L}$ is still logarithmic, that is, such a DAG would trigger an unnecessary rebuild. It is thus questionable whether we can still expect a constant number of rebuilds when relying on $\mathcal{D}$. Our experiments, reported in Section 3.2, show that in practice the two values hardly differ, indicating that it is sufficient to rely on $\mathcal{D}$. However, a theoretical proof to consolidate this is still missing.

Table 3.1: The table displays the number of trapezoidal search tree nodes vs. number of DAG nodes for the same input with the same insertion order. The trapezoidal search tree is obtained by inserting the curves in the same order as in the construction of the DAG, however, while avoiding merges. The last column presents the ratio between the number of tree nodes and the number of DAG nodes. As expected, its values correspond to the function $\log n$, where n is the number of edges.

# Arrangement Edges	# Tree nodes	# DAG nodes	ratio
22	125	101	1.23
138	1263	681	1.85
285	3167	1492	2.12
1483	23511	8019	2.93
2977	51551	16330	3.15
14975	350629	84576	4.14
29973	759075	169355	4.48

3.1 Theoretical Bounds on the Ratio

The difference between the DAG depth $\mathcal{D}$ and the maximum query path length $\mathcal{L}$ should first be clarified, and this is done in Subsection 3.1.1. Then, in Subsection 3.1.2, a construction in which the difference between $\mathcal{D}$ and $\mathcal{L}$ is significant is presented. However, the ratio between the two values can be even larger, as shown in Subsection 3.1.3. In particular, the $\Omega(n/\log n)$ ratio of $\mathcal{D}$ and $\mathcal{L}$ achieved in the latter is proven to be tight.

3.1.1 Definitions

To demonstrate the difference between the DAG depth $\mathcal{D}$ and the maximum query path length $\mathcal{L}$, a third curve cv_3 should be added to the subdivision of the two curves from Figure 2.5. Figure 3.1 shows the appropriate trapezoidal map and DAG after the insertion of cv_3. The leaf representing trapezoid H in the former structure (Figure 2.5) was replaced with a subtree rooted at p_3, the left endpoint of cv_3.

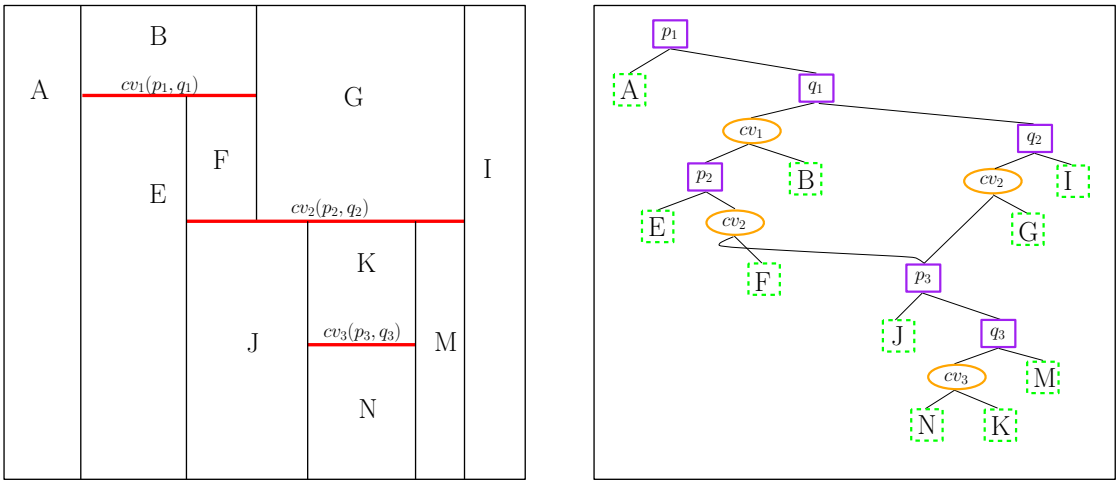
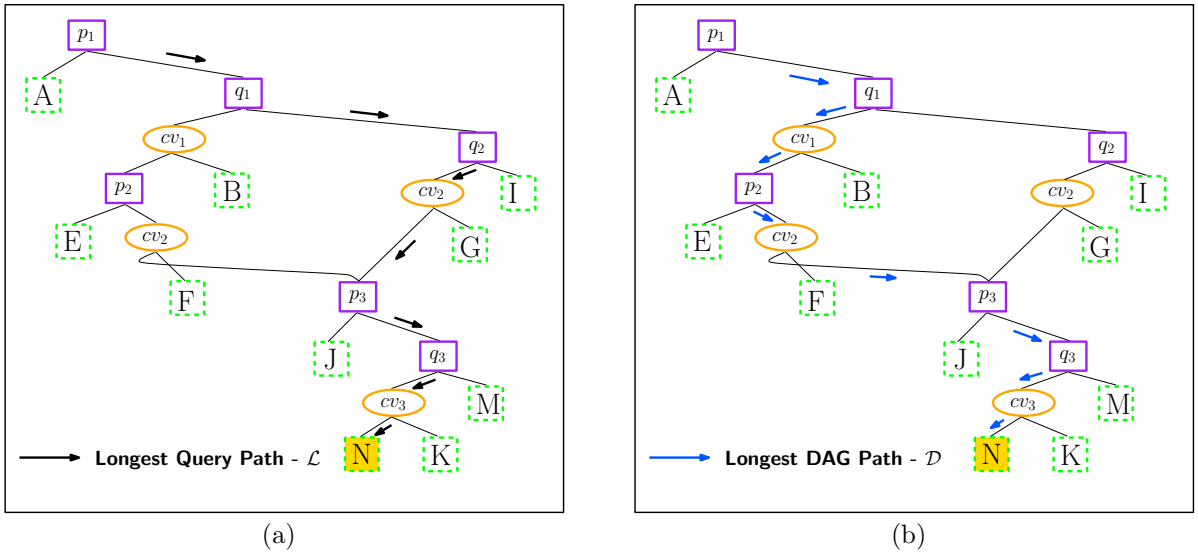


Figure 3.1: The trapezoidal map and appropriate DAG after inserting $cv_3(p_3, q_3)$.

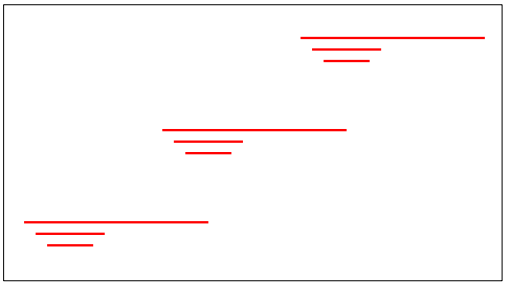
There are two directed paths starting at the root that reach trapezoid N, illustrated in Figure 3.2. The black path is a valid search path to all queries in trapezoid N. The blue path, on the other hand, is not a valid search path, since all points in N are to the right of q_1, that is, such a query would never visit the left child of q_1. This blue path, however, determines the depth of N, since it is the longer path of the two. This scenario occurs due to the merge that was part of the insertion of cv_2 (see Figure 2.6) creating two different paths to a leaf, which became an inner node in the updated structure. In the sequel we use this observation to construct an example that shows that the ratio between $\mathcal{D}$ and $\mathcal{L}$ can be as large as $\Omega(n/\log n)$. Moreover, we will show that this bound is tight.



(a) (b)

Figure 3.2: Two directed paths to N. The black path illustrated in (a) and the blue path illustrated in (b) represent the longest query path and the longest DAG path, respectively. While the black path in (a) is the search path for all queries that end up in trapezoid N, the blue path in (b) is not a valid search path and cannot be realized by any query.

3.1.2 Towards a Worst Case Bound



The figure to the right demonstrates a simple construction achieving an $\Omega(\sqrt{n})$ lower bound for the ratio between $\mathcal{D}$ and $\mathcal{L}$. Assuming that $n = k^2 \in \mathbb{N}$, the construction consists of k blocks, each containing k horizontal segments. The blocks are arranged as depicted in the figure. Segments are inserted from top to bottom. A block starts with a large segment at the top, which we call the *cover segment*, while the other segments successively shrink in size. Now the next block is placed to the left and below the previous block. Only the cover segment of this block extends below the previous block, which causes a merge as illustrated in Figure 3.3. All $k = \sqrt{n}$ blocks are placed in this fashion. This construction ensures that each newly inserted segment intersects the trapezoid

with the largest depth, which increases $\mathcal{D}$. The largest depth of $\Omega(n)$ is finally achieved in the trapezoid below the lowest segment. However, the actual search path into this trapezoid has only $O(\sqrt{n})$ length, since there are $O(\sqrt{n})$ blocks and a query is able to skip an entire block using only one comparison to the leftmost point of the cover segment. Within the relevant block there are at most $O(\sqrt{n})$ possible comparisons.

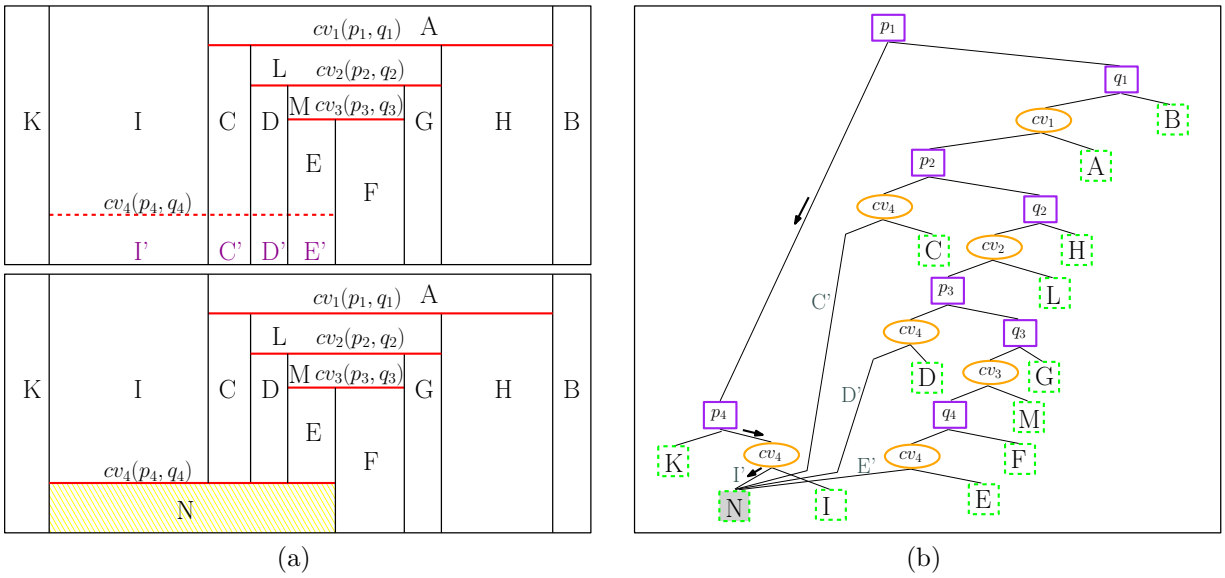
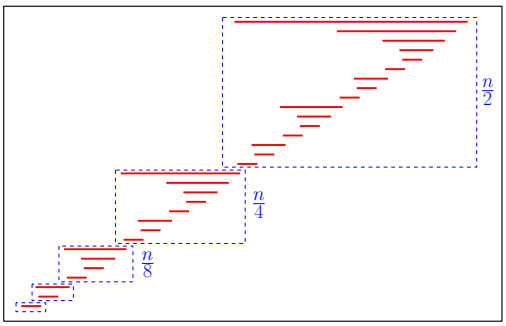


Figure 3.3: (a) The trapezoidal map after inserting cv_4. The map is displayed before and after the merge of I', C', D', and E' into N, in the top and bottom illustrations, respectively. (b) The DAG after merging. A query path to the region of I' in N will take 3 steps, while the depth of N in this example is 11.

3.1.3 Worst Case Ratio



The following construction, illustrated in the figure to the right, which uses a recursive scheme, establishes the lower bound $\Omega(n/\log n)$ for $\mathcal{D}/\mathcal{L}$. Blocks are constructed and arranged in a similar fashion as in the previous construction. However, this time we have $\log_2 n$ blocks, where block i contains $n/2^i$ segments. Within each block we then apply the same scheme recursively as depicted in the figure to the right. Again segments are inserted from top to bottom such that the depth of $\Omega(n)$ is achieved in the trapezoid below the lowest segment. The fact that the lengths of all search paths are logarithmic can be proven by the following argument. By induction we assume that the longest search path within a block of size $n/2^i$ is some constant times $(\log_2 n - i)$. Obviously this is true for a block containing only one segment. Now, in order to reach block i with $n/2^i$ segments, we require $i - 1$ comparisons to skip the $i - 1st$ preceding blocks. Thus in total the search path is of logarithmic length.

Theorem 3.1. *The $\Omega(n/\log n)$ worst-case lower bound on $\mathcal{D}/\mathcal{L}$ is tight.*

Proof. Obviously, $\mathcal{D}$ of $O(n)$ is the maximal achievable depth, since by construction each segment can only appear once along *any* path in the DAG. It remains to show that for any scenario with n segments there is no DAG for which $\mathcal{L}$ is smaller than $\Omega(\log n)$. Since there are n segments, there are at least n different trapezoids having these segments as their top boundary. Let T be a decision tree of the optimal search structure in the sense that its longest query path is the shortest possible. Each path in the decision tree corresponds to a valid search path in the DAG and vice versa. The depth of T must be at least $\log_2 n$, since it is only a binary tree. We conclude that the worst case ratio of $\mathcal{D}$ and $\mathcal{L}$ is $\Theta(n/\log n)$. □

3.2 Observed Ratio Experiments

Since $\mathcal{D}$ is an upper bound on $\mathcal{L}$ and since $\mathcal{D}$ is accessible in constant time, our implementation explores an alternative that monitors $\mathcal{D}$ instead of $\mathcal{L}$. Though this may cause some additional rebuilds, the experiments in this section give strong evidence that one can still expect an $O(n \log n)$ preprocessing time.

3.2.1 Experiments

We compared $\mathcal{D}$ and $\mathcal{L}$ in the following two scenarios: random non-intersecting line segments and Voronoi diagrams for random sites. In addition, the two special scenarios, which were constructed in order to achieve lower bounds on the worst case ratio of $\mathcal{D}$ and $\mathcal{L}$ as described in Subsection 3.1.2 and Subsection 3.1.3, were tested as well. Figure 3.4 illustrates the four different scenarios. It should be noted, however, that in all experiments here, in particular for the lower bound constructions, we chose a random insertion order for the input curves. Each scenario was tested with an increasing number of subdivision edges, with several runs for each input.

In the random line-segments scenario (Figure 3.4 (a)) each segment was created from two points chosen at random in $[-1, 1]^2$. The number of generated segments was $\lfloor 1.5^k \rfloor$, for $6 \leq k \leq 19$. The reported results are the average of 20 builds of the search structure for the same random scenario. In the scenarios of Voronoi diagrams of random sites (Figure 3.4 (b)) we used 2^k random sites for $6 \leq k \leq 15$. For each k we generated 10 different point sets and created the search structure 7 times, that is, the reported results are the average of 70 builds. Each run in the scenario achieving a worst-case $O(\sqrt{n})$ ratio (Figure 3.4 (c)) contains k^2 segments for $k \in \{10 \cdot 2^i | i \in \{1, \ldots, 6\}\}$. Finally, in the scenario achieving a worst-case $O(n/\log n)$ ratio (Figure 3.4 (d)) each run contains 2^k segments for $8 \leq k \leq 17$. As mentioned above all experiments used a different random permutation of the input curves for each construction.

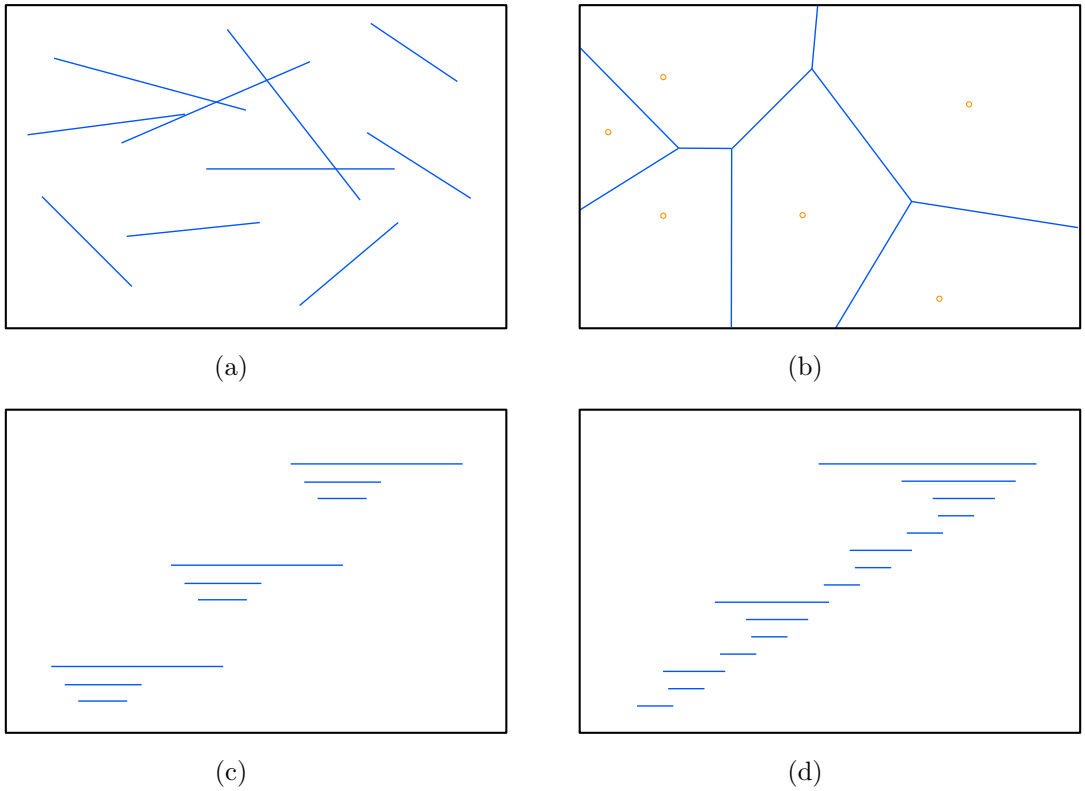


Figure 3.4: **Scenarios:** (a) random line-segments (b) Voronoi diagrams of random sites (c) construction for worst case $O(\sqrt{n})$ $\mathcal{D}/\mathcal{L}$ ratio (d) construction for worst case $O(n/\log n)$ $\mathcal{D}/\mathcal{L}$ ratio.

3.2.2 Results

The plots in Figure 3.5 display the average $\mathcal{D}/\mathcal{L}$ ratio with error bars. In all experiments the two values hardly differ, that is, the largest ratio that we were able to observe was strictly less than 1.3. In the special scenarios, this value was even lower and in many cases $\mathcal{D}$ and $\mathcal{L}$ actually had the same value. However, for very large scenarios of random curves, see Figure 3.5 (a), $\mathcal{D}$ was always a bit larger than $\mathcal{L}$, but on the other hand the largest observed ratio there went down to less than 1.2. Furthermore, even in the scenario of Figure 3.4 (d) that achieves a worst case ratio of $O(n/\log n)$, the results show that when taking a random permutation of the curves the chance of reaching an $O(n/\log n)$ ratio becomes very small.

This indicates that $\mathcal{D}$ and $\mathcal{L}$ behave sufficiently similarly in practice. Therefore, replacing the test for the length of the longest search path $\mathcal{L}$ by the depth $\mathcal{D}$ of the DAG in the randomized incremental construction should not harm the runtime. This led us to the following conjecture.

Conjecture 3.2. *There exists a constant $c > 0$ such that the runtime of the randomized incremental algorithm, modified such that it rebuilds in case the depth $\mathcal{D}$ of the DAG becomes larger than $c \log n$, is expected $O(n \log n)$, i.e., the number of expected rebuilds is still constant.*

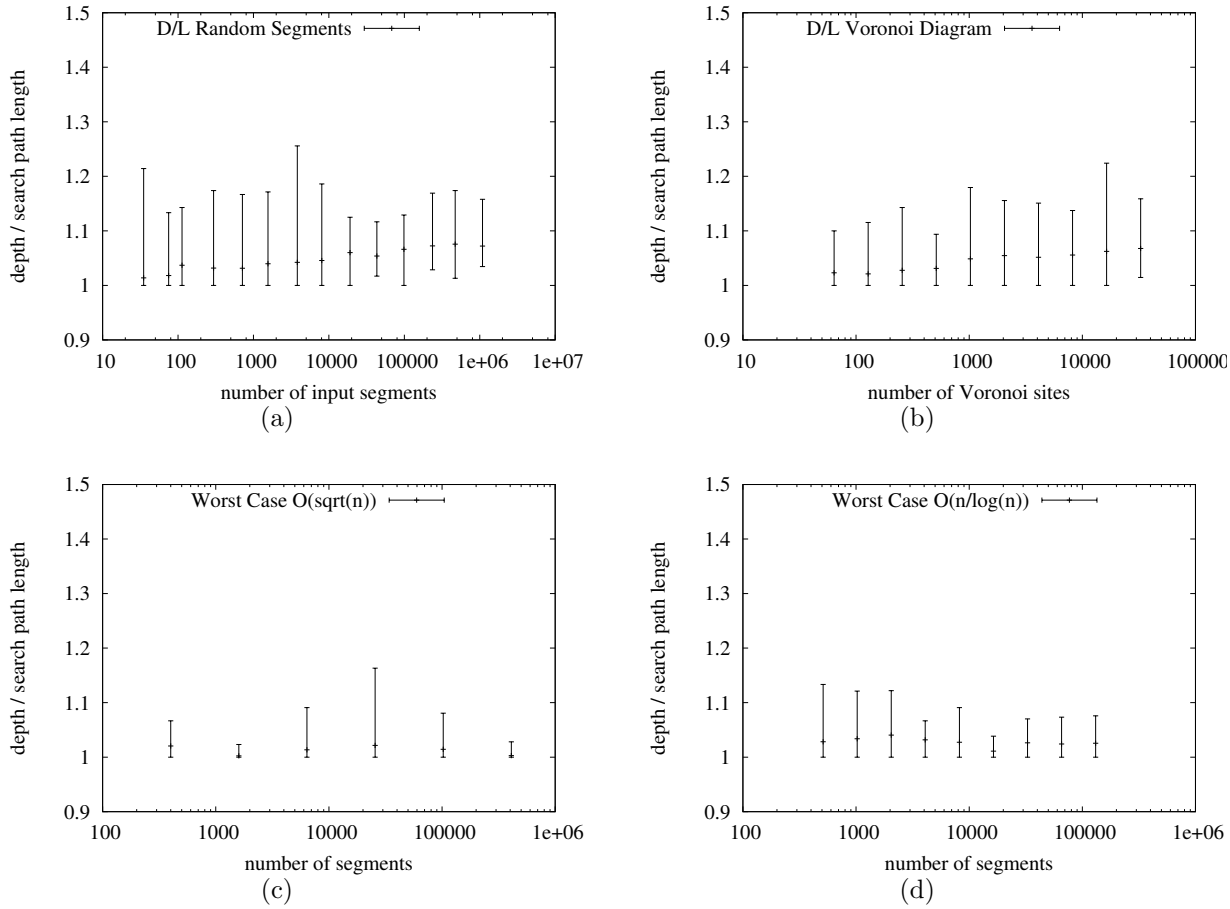


Figure 3.5: **Results:** (a) random line-segments (b) Voronoi diagrams of random sites (c) construction for worst case $O(\sqrt{n})$ $\mathcal{D}/\mathcal{L}$ ratio (d) construction for worst case $O(n/\log n)$ $\mathcal{D}/\mathcal{L}$ ratio.

4

Efficient Construction Algorithms for Static Settings

This chapter describes optional construction algorithms for static settings guaranteeing both logarithmic query time and linear size (similar to the algorithm reviewed in Section 2.3). Section 4.1 defines a basic scheme for such construction algorithms. This prototype uses as a "black box" an algorithm that verifies the maximum query path length $\mathcal{L}$. This "black box" algorithm has an effect on the total runtime of the whole construction algorithm. The next two sections describe two efficient verification algorithms for $\mathcal{L}$ that can be used by the general construction algorithm. The first verification algorithm, reviewed in Section 4.2, has expected $O(n \log^2 n)$ runtime. However, as discussed there, it is likely to have a tighter runtime bound. In Section 4.3 we present a second verification algorithm for $\mathcal{L}$ that has $O(n \log n)$ runtime.

4.1 Algorithmic Scheme for Static Settings

Given a set S of n pairwise interior disjoint x-monotone curves inducing a planar subdivision, one can define an efficient construction algorithm for static settings, when all input curves are given in advance. This can be done as follows:

Definition 4.1. *$f(n)$ denotes the time it takes to verify that, in a linear size DAG constructed over a set of n pairwise interior disjoint x-monotone curves, $\mathcal{L}$ is bounded by $c \log n$ for a constant c.*

Theorem 4.2. *Let S be a set of n pairwise interior disjoint x-monotone curves inducing a planar subdivision. A point location data structure for S, which has $O(n)$ size and $O(\log n)$*

query time in the worst case, can be built in $O(n \log n + f(n))$ expected time, where $f(n)$ is as defined above.

Proof. The construction of a DAG with some random insertion order takes expected $O(n \log n)$ time. The linear size can be verified trivially on the fly (as discussed in Section 2.3). After the construction an algorithm that verifies that the maximum query path length $\mathcal{L}$ is logarithmic is used. The verification of the size $\mathcal{S}$ and the maximum query path length $\mathcal{L}$ may trigger rebuilds with a new random insertion order. However, according to Lemma 2.2 and Lemma 2.3, one can expect only a constant number of rebuilds. Thus, the overall expected runtime remains $O(n \log n + f(n))$. □

As described in Section 2.3, with high probability the length of the longest query over all insertion orders (permutations of the input curves) is $O(\log n)$. Obviously part of these permutations will generate a linear size DAG while others will create DAGs with larger size complexity. Since we verify $\mathcal{L}$ only for permutations that construct a linear size DAG, it is not clear whether we can still expect a logarithmic query path length in this subpopulation of DAGs. Suppose that the expected value of $\mathcal{L}$ in the subpopulation of linear-size DAGs is larger than $O(\log n)$. Since at least three out of four DAGs have linear size (the subpopulation size is at least 3/4 of the population size), the expected value of $\mathcal{L}$ in the whole population would have also been larger than $O(\log n)$. Therefore, the expected value of $\mathcal{L}$ in the subpopulation of linear size DAGs is also $O(\log n)$. In other words, we can still expect a constant number of rebuilds when using the above algorithm.

4.2 An Expected $O(n \log^2 n)$ Verification Algorithm

The following algorithm verifies that the maximum query path length $\mathcal{L}$ in the search structure of linear size is bounded as desired. It is a recursive algorithm that performs a DFS-like (Depth First Search) traversal on the DAG in order to verify that $\mathcal{L}$ is logarithmic.

The algorithm essentially computes all possible search paths in the DAG by discarding those paths that are geometrically impossible. Starting at the root it descends towards the leaves in a recursive fashion. Taking the history of the current path into account, each recursion call maintains the interval of the x values that are still possible. More precisely, there are three cases: (i) if the recursion call reaches a point node whose x-coordinate is contained in the current interval, the recursion splits into two with updated intervals; (ii) if the recursion call reaches a point node that is not contained in the interval, the recursion does not split and continues to the proper child only. Such a node in a path that does not cause the recursion call to split is named a *bouncing-node* for this path; (iii) if the recursion call reaches a curve node, it always splits while the interval remains unchanged. Figure 4.1 illustrates a partial run of the algorithm.

In order to bound the expected time complexity of the algorithm a bound on the expected number of different search paths that are traced by the algorithm should be found. Obviously, in the DAG several different search path may reach the same leaf. In other words it is not obvious how to count the number of such search paths. Thus, finding a structure in which the number of different search paths can be computed more easily would be helpful. Such

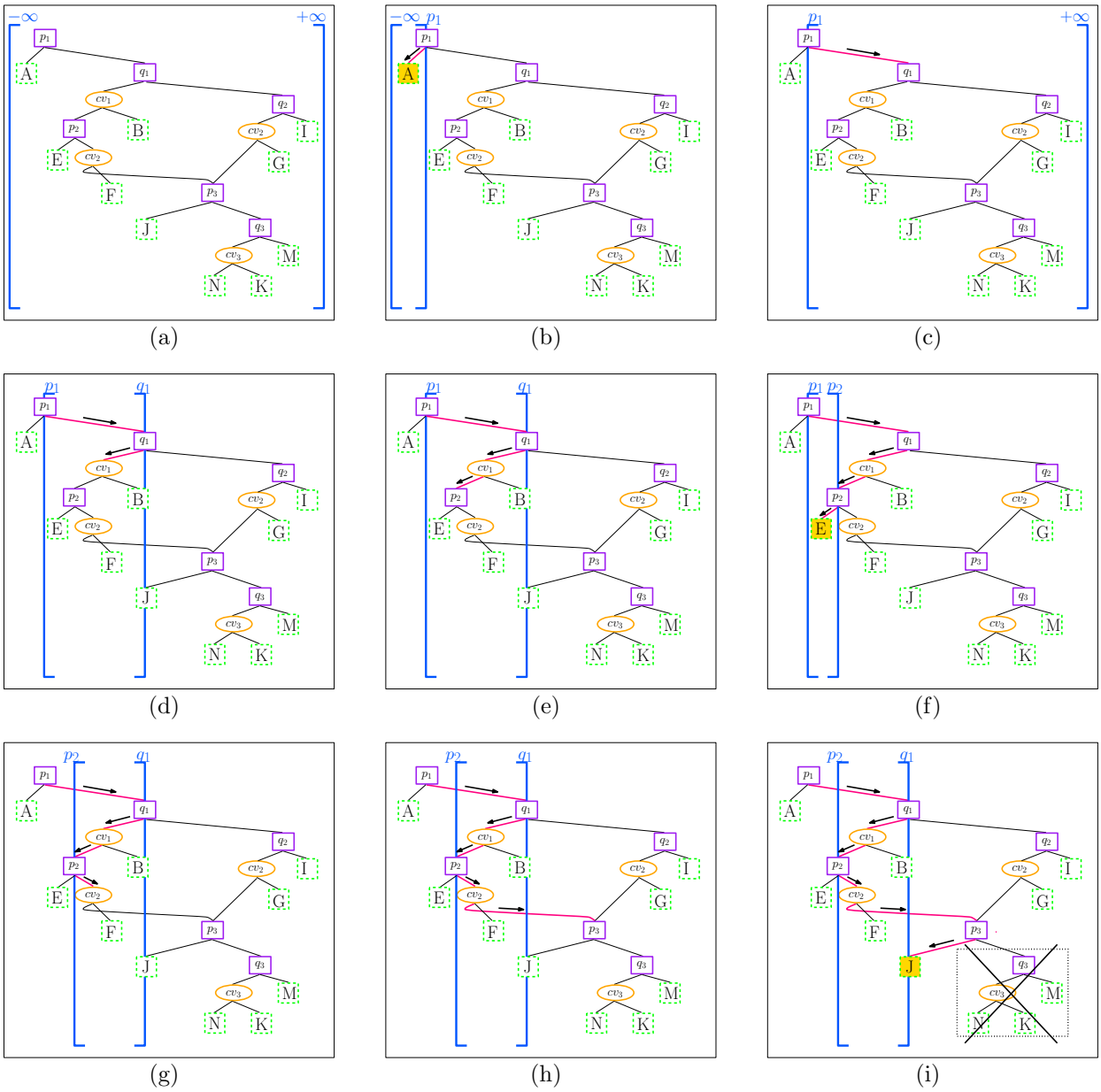


Figure 4.1: The first 9 steps of the recursive verification algorithm run on the search structure for 3 curves, as illustrated in Figure 3.1. A magnification of (a) is depicted in Figure 4.2. The interval of possible x-values is marked by the blue brackets. In each step the growing path so far is marked with arrows. In (i) the interval of possible x-values remains $[p_2, q_1]$ and does not shrink since p_3 is not contained in it. The subgraph rooted at the right child of p_3 is clearly not contained in $[p_2, q_1]$, since it represents regions that are completely to the right of p_3, and is, therefore, skipped.

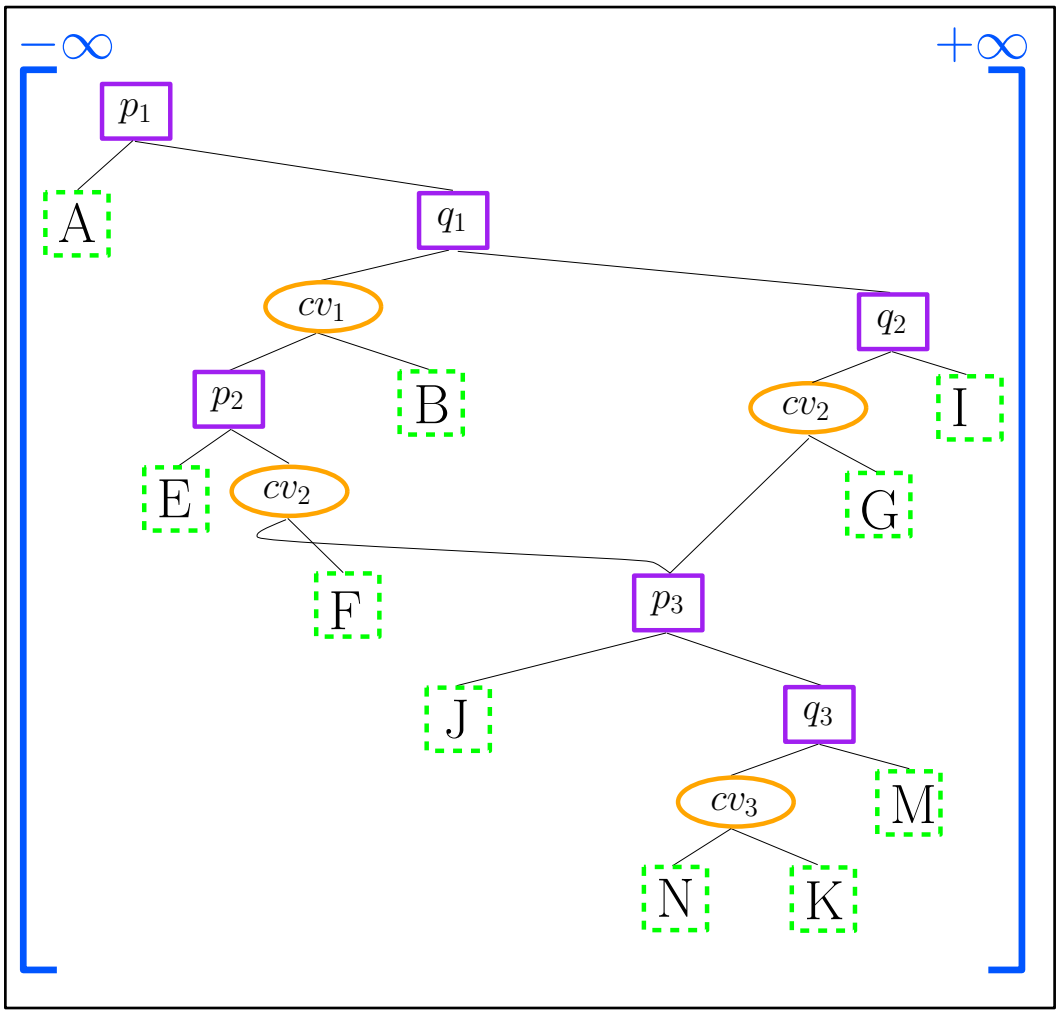


Figure 4.2: Magnification of Figure 4.1(a)

a structure is the trapezoidal search tree $\mathbb{T}$ which is a full binary tree constructed as the DAG using the same insertion order while skipping the merge step. In $\mathbb{T}$, as in any tree, the number of different search paths is exactly the number of leaves. Therefore, we would like to show that the number of search paths in the DAG is equal to the number of search paths in $\mathbb{T}$.

Lemma 4.3. *Let S be a set of n pairwise interior disjoint x-monotone curves inducing a planar subdivision. Let G and $\mathbb{T}$ be the DAG and the trapezoidal search tree created using the same permutation of the curves in S, respectively. The number of different search paths in G is equal to the number of search paths in $\mathbb{T}$.*

Proof. In order to prove this we can simply show that the following statement holds. Suppose that while searching in the DAG for some query point q we maintain the interval of possible x-values (similarly to what the recursive algorithm does). If the search in the DAG ends in trapezoid t and with the x-interval (a, b) (the x-range of trapezoid t contains (a, b) but may be even larger) then the search path for q in the trapezoidal search tree will end in trapezoid t' whose left and right x-values are a and b, respectively, and whose top and bottom curves are identical to the top and bottom curves of t. Moreover, we would like to show that the search paths for q in the DAG and the search path for q in the trapezoidal search tree are identical up to bouncing-nodes. This can be shown by induction on the set of inserted curves. We denote by G_i, $\mathbb{T}_i$ the DAG and the trapezoidal search tree after the first i curves were inserted, respectively.
Base case $k = 1$: $G_1 = \mathbb{T}_1$ since no merge has occurred. Therefore, the search paths for q in both structures are identical.
Suppose that the statement holds for $k = i - 1$. We now show that it holds for $k = i$, as well. The ith curve cv_i is inserted into both G_{i-1} and $\mathbb{T}_{i-1}$. Let t_{i-1} and t'_{i-1} denote the trapezoids containing the query point q in G_{i-1} and $\mathbb{T}_{i-1}$, respectively. Let (a_{i-1}, b_{i-1}) denote the x-interval of t'_{i-1}.

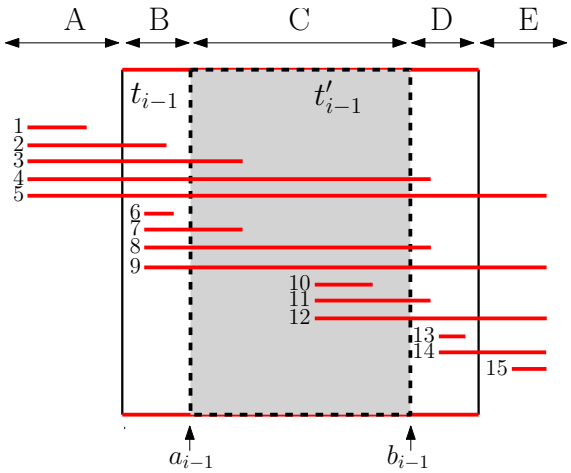


Figure 4.3: Possible positions for cv_i

We use the following notations for the five different vertical slabs (regions). We denote by A the region to the left of t_{i-1}, by B the region to the left of t'_{i-1} inside t_{i-1}. We denote

by C the region inside t'_{i-1} and t_{i-1}. D will denote the region to the right of t'_{i-1} inside t_{i-1}, and E represents the region to the right of t_{i-1}. There are 15 optional positions to insert curve $cv_i(p_i, q_i)$ presented in Figure 4.3 (note that p_i is always to the left of q_i). We will group them according to the region containing p_i as follows:

- p_i is located to the left of t_{i-1}, that is, in region A. In Figure 4.3 the relevant positions are 1-5. In such positions p_i will not be added to the search paths of a query point q that lies in t'_{i-1} both in the DAG G_i and in the trapezoidal search tree $\mathbb{T}_i$. We now distinguish the different cases depending on the position of q_i.

  - Position 1: q_i lies in region A as well. Therefore, will not be added to the search paths of a query point q that lies in t'_{i-1}. Clearly, $t_i = t_{i-1}$ and $t'_i = t'_{i-1}$. Therefore, since the statement holds for $k = i - 1$, it holds for $k = i$ as well.
  - Position 2: q_i lies in region B. In such a case q_i will be added to the query path to q as a bouncing node in G_i, but will not affect the x-interval maintained during the search since q_i is not contained in (a_{i-1}, b_{i-1}). The query path to q in $\mathbb{T}_i$ will not change, since cv_i does not intersect t'_{i-1}. Using the induction hypothesis and since the only new internal node in the search paths for q is a bouncing-node in G_i, then the statement holds for $k = i$ as well.
  - Position 3: q_i lies in region C. The search paths for q in both G_i and $\mathbb{T}_i$ will include q_i. If q is in the x-range of cv_i then an additional internal node representing cv_i will appear in the path for q in both structures. The x-interval (a_i, b_i) in such a case would change to (a_{i-1}, q_i). If, on the other hand, q is to the left of q_i then the new x-interval would be from q_i to b_{i-1}. Since the statement held for $k = i - 1$, it holds for $k = i$ as well.
  - Position 4: q_i lies in region D. Similar to the case where q_i lies in region B, that is, q_i will be added to the query path to q as a bouncing node in G_i, but will not appear in $\mathbb{T}_i$. In addition, since cv_i intersects t'_{i-1} completely, an internal node cv_i will be added to both structures. Using the induction hypothesis and since the only new internal node in the search paths for q that does not appear in both paths (i.e., in the search paths for q in both G_i and $\mathbb{T}_i$) is a bouncing-node in G_i, then the statement holds for $k = i$ as well.
  - Position 5: q_i lies in region E. Therefore, will not be added to the search paths of a query point q that lies in t'_{i-1}. Since cv_i intersects t'_{i-1} completely, an internal node cv_i will be added to both structures. Since the statement held for $k = i - 1$, it holds for $k = i$ as well.

- p_i is located inside t_{i-1} to the left of t'_{i-1}, that is, in region B. In Figure 4.3 the relevant positions are 6-9. In such positions p_i will be added to the search path of a query point q that lies in t'_{i-1} in the DAG G_i as a bouncing node for this path, since it is not contained in (a_{i-1}, b_{i-1}). On the other hand, it will not be added to the path of q in the trapezoidal search tree $\mathbb{T}_i$. We now distinguish several cases depending on the position of q_i.

- Position 6: q_i lies in region B. In such a case q_i will be added to the query path to q as a bouncing node in G_i, but will not affect the x-interval maintained during the search since q_i is not contained in (a_{i-1}, b_{i-1}). The query path to q in $\mathbb{T}_i$ will not change, since cv_i does not intersect t'_{i-1}. Using the induction hypothesis and since the only new internal nodes in the search paths for q are bouncing-nodes in G_i, then the statement holds for $k = i$ as well.
- Position 7: q_i lies in region C. The search paths for q in both G_i and $\mathbb{T}_i$ will include q_i. If q is in the x-range of cv_i then an additional internal node representing cv_i will appear in the path for q in both structures. The x-interval (a_i, b_i) in such a case would change to (a_{i-1}, q_i). If, on the other hand, q is to the left of q_i then the new x-interval would be from q_i to b_{i-1}. Using the induction hypothesis and since the only new internal node in the search paths for q that does not appear in both paths is a bouncing-node in G_i, then the statement holds for $k = i$ as well.
- Position 8: q_i lies in region D. Similar to the case where q_i lies in region B, that is, q_i will be added to the query path to q as a bouncing node in G_i, but will not appear in $\mathbb{T}_i$. In addition, since cv_i intersects t'_{i-1} completely, an internal node cv_i will be added to both structures. Using the induction hypothesis and since the only new internal node in the search paths for q that does not appear in both paths is a bouncing-node in G_i, then the statement holds for $k = i$ as well.
- Position 9: q_i lies in region E. Therefore, will not be added to the search paths of a query point q that lies in t'_{i-1}. Since cv_i intersects t'_{i-1} completely, an internal node cv_i will be added to both structures. Since the statement held for $k = i - 1$, it holds for $k = i$ as well.

- p_i is located inside t'_{i-1}, that is, in region C. In Figure 4.3 the relevant positions are 10-12. In such positions p_i will be added to the search path of a query point q that lies in t'_{i-1} both in G_i and in $\mathbb{T}_i$, since it is contained in (a_{i-1}, b_{i-1}). We now distinguish several cases depending on the position of q_i.
  - Position 10: q_i lies in region C. cv_i is contained completely in region C. The same internal nodes, depending on the position of q, will be added for both search structures. Using the induction hypothesis and since the search paths for q in both structures were added with the same nodes it holds for $k = i$ as well.
  - Position 11: q_i lies in region D. If the query point q is located to the left of p_i then no new node (other than p_i) will be added to the search paths of q in both G_i and $\mathbb{T}_i$. If, on the other hand, q is in the x-range of cv_i then q_i will be added to the path as a bouncing node in G_i, but will not appear in $\mathbb{T}_i$. In addition the paths in the two structures will be added with a node representing cv_i. Using the induction hypothesis and since the only new internal node in the search paths for q that does not appear in both paths is a bouncing-node in G_i, then the statement holds for $k = i$ as well.
  - Position 12: q_i lies in region E. Therefore, will not be added to the search paths of a query point q that lies in t'_{i-1}. Depending on the location of q, an internal

node cv_i may be added to the paths in both structures. Since the statement held for $k = i - 1$, it holds for $k = i$ as well.

- p_i is located inside t_{i-1} to the right of t'_{i-1}, that is, in region D. In Figure 4.3 the relevant positions are 13-14. In such positions p_i will be added to the search path of a query point q that lies in t'_{i-1} in the DAG G_i as a bouncing node for this path, since it is not contained in (a_{i-1}, b_{i-1}). On the other hand, it will not be added to the path of q in the trapezoidal search tree $\mathbb{T}_i$. In both positions q_i is to the right of p_i and is, therefore, blocked by p_i for query points that lie in t'_{i-1} and will not be added to the search paths of such points. Using the induction hypothesis and since the only new internal node in the search paths for q that does not appear in both paths is a bouncing-node in G_i, then the statement holds for $k = i$ as well.

- p_i is located to the right of t_{i-1}, that is, in region E. In Figure 4.3 the relevant position is 15. p_i will not be added to the search paths of a query point q that lies in t'_{i-1} both in the DAG G_i and in the trapezoidal search tree $\mathbb{T}_i$. q_i is located to the right of p_i (also in region E) Since the statement held for $k = i - 1$, it holds for $k = i$ as well.

□

We showed a bijection between the paths in the trapezoidal search tree $\mathbb{T}$ and the different search paths in the DAG. Therefore, the expected number of leaves in $\mathbb{T}$ should be bounded in order to bound the expected number of different search paths in the DAG.

Lemma 4.4. *Let S be a set of n pairwise interior disjoint x-monotone curves inducing a planar subdivision. The expected number of leaves in the trapezoidal search tree $\mathbb{T}$, which is constructed as the DAG but without merges, is $O(n \log n)$.*

Proof. We would like to bound the expected number of leaves in $\mathbb{T}$, namely, the expected number of trapezoids in the decomposition without merges. We can symbolically shorten every curve at its two endpoints by an arbitrarily small value, namely ϵ. In other words, if a curve cv_i has an x-range (a, b), then the shortened curve will have an x-range $(a+\epsilon, b-\epsilon)$. The curves of the updated subdivision are now completely disjoint. In addition, this operation gave rise to new artificial trapezoids. Clearly, for any point p in such an artificial trapezoid, we can decrease the value of ϵ such that p will not be covered by this trapezoid. Now we would like to bound the number of trapezoids in the set of shortened curves. It is clearly bounded by the number of vertical edges + 1. First, consider the vertical line W through one endpoint of the ith inserted curve. W is intersected by n curves, in the worst-case. The $i-1$ already inserted curves partition W into i intervals. However, we are only interested in the interval I containing the endpoint of the ith curve, as it will appear in the final structure. Curves inserted after the ith curve may split I. The expected number of intersections in I (including the endpoint of the ith curve) is $O((n-i)/i)$. Summing up over all vertical walls gives a total of expected $O(n \log n)$ intersections. Thus, the expected number of vertical edges is $O(n \log n)$ as well, and, clearly, this is also the expected number of leaves in the tree. □

By Lemma 4.4 the expected number of leaves of $\mathbb{T}$, and thus of search paths, is $O(n \log n)$. We verify that the length of each search path is at most $c \log n$ for some constant $c > 0$. Therefore, the time complexity bound for this algorithm is expected $O(n \log^2 n)$.

This is of course a trivial bound, since it is based on the argument that the algorithm follows expected $O(n \log n)$ paths of at most logarithmic length each. However, due to its recursive nature the algorithm does not treat every path separately. Thus, it might be possible to achieve a better bound. Lemma 4.3 essentially argues that for a given query point q its search path in the DAG is identical to its search path in the matching trapezoidal search tree up to bouncing-nodes. When running this algorithm on the trapezoidal search tree $\mathbb{T}$ the number of steps performed by the algorithm is equal to the number of edges in $\mathbb{T}$, namely: expected $O(n \log n)$ (since $\mathbb{T}$ is a full binary tree with expected $O(n \log n)$ leaves). However, since we allow merges in order to retain a linear-size structure, a search path may include additional nodes, which are not part of the matching search path in $\mathbb{T}$. These additional nodes are exactly the bouncing-nodes that may only appear in DAG paths. An example for such a bouncing-node is given in Figure 4.4. Hence, if we had a bound on the number of occurring bouncing nodes, then a better analysis which takes the recursive fashion of the algorithm into account would be available.

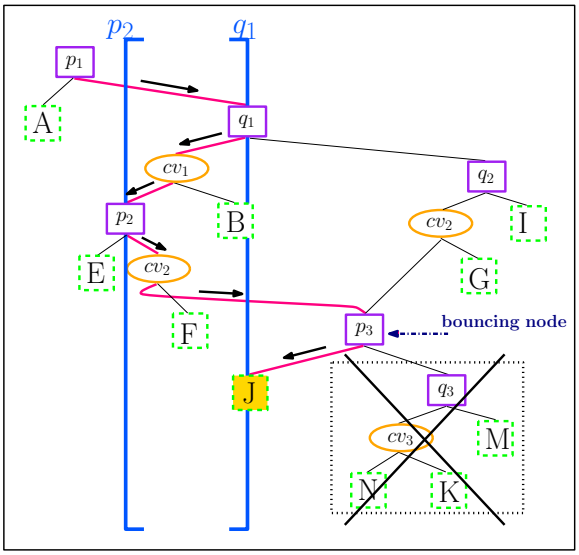


Figure 4.4: The node representing p_3 is a bouncing node, and is due to a merge.

Another way to observe this is that several edges of the matching trapezoidal search tree $\mathbb{T}$ are split by additional nodes, i.e., bouncing nodes. Therefore, each edge of $\mathbb{T}$ is split into several consecutive edges. A possible work plan for showing an expected $O(n \log n)$ bound on the runtime of the verification algorithm would be to show that each original edge of $\mathbb{T}$ can be charged with a constant number of bouncing nodes. Moreover, each bouncing node adds potential geometrically impossible DAG paths and having a bound on the number of bouncing nodes may help understanding the expected value of the depth $\mathcal{D}$ of the DAG.

4.3 An $O(n \log n)$ Verification Algorithm

Let $\mathcal{T}(S_i)$ denote the trapezoidal map for the first i insertions. We will also use this notation in order to identify the set of trapezoids of this map. We denote by $\mathcal{T}^*$ the collection of *all* the trapezoids created during the construction of the DAG, including intermediate trapezoids that are later killed by the insertion of later segments. More formally:

$$\mathcal{T}^* = \bigcup_{i=1}^{n} \mathcal{T}(S_i).$$

Let $\mathcal{A}(\mathcal{T}^*)$ denote the arrangement of all trapezoids in $\mathcal{T}^*$. Notice that each face of the arrangement can be covered by overlapping trapezoids. The *depth* of a point p in $\mathcal{A}(\mathcal{T}^*)$ is defined as the number of trapezoids in $\mathcal{T}^*$ that cover p. The key to the improved algorithm is the following observation by Har-Peled [25].

Observation 1. *The length of a path in the DAG for a query point q is at most three times the depth of q in $\mathcal{A}(\mathcal{T}^*)$.*

It follows that we need to verify that the maximum depth of a point in $\mathcal{A}(\mathcal{T}^*)$ is $c_1 \log n$ for some constant $c_1 > 0$. We remark that this depth is established in an interior of a face of $\mathcal{A}(\mathcal{T}^*)$, since the longest path will always end in a leaf of the DAG, which represents a trapezoid. Moreover, for any query point that falls on either a curve or an endpoint of the initial subdivision the search path will end in an internal node of the DAG. If, on the other hand, the query point q falls on a vertical edge of a trapezoid, the search path for q will be identical to a path for a query point in a neighboring trapezoid. Therefore, we consider the boundaries of the trapezoids as open.

Since the input curves are interior pairwise disjoint, according to the separation property stemming from [24], one can define a total order on the curves. This order allows us to apply a modified version (as described next in Subsection 4.3.1) of an algorithm by Alt and Scharf [2], which originally detects the maximum depth in an arrangement of n axis-parallel rectangles in $O(n \log n)$ time. Recall that we only apply this verification algorithm on DAGs of linear size.

4.3.1 Computing the Depth of $\mathcal{A}(\mathcal{T}^*)$

We would like to describe a linear space algorithm with $O(n \log n)$ runtime for computing the depth of an arrangement of open trapezoids with the following properties: their bases are y-axis parallel (vertical walls) and if the top or bottom curves of two different trapezoids intersect not only in a joint endpoint then the two curves overlap completely in their joint x-range. The depth of such an arrangement is the maximum number of trapezoids containing a common point, that is, we are only interested in points located in faces of this arrangement. In Subsection 4.3.1.1 we restate an algorithm by Alt & Scharf [2] such that the general position assumption can be dropped. The restated algorithm is more general than what we essentially need as it can handle rectangles with independently open or closed boundaries, while we are only interested in open rectangles. Subsection 4.3.1.2 defines a reduction from the collection of open trapezoids $\mathcal{T}^*$ to a collection $\mathcal{R}^*$ of open axis-parallel rectangles such that the maximum depth in $\mathcal{A}(\mathcal{R}^*)$ is the same as the maximum depth in $\mathcal{A}(\mathcal{T}^*)$. Finally, in Subsection 4.3.1.3 we describe a modification for the restated algorithm such that it can compute the depth of an arrangement of all trapezoids created during the construction of the DAG.

4.3.1.1 An Algorithm for Computing the Depth of an Arrangement of Axis-aligned Rectangles

The algorithm of Alt & Scharf [2] is an $O(n \log n)$ algorithm that computes the depth of an arrangement of axis-aligned rectangles in general position, using $O(n)$ space. We present here a minor modification, which does not assume general position, i.e., rectangles may share boundaries. Moreover, it can consider each of the four boundaries of a rectangle as either belonging to the rectangle or not; we call these closed or open boundaries, respectively.

Given a finite set of rectangles, the set of all x-coordinates of the vertical sides of the input rectangles is first sorted. Let $x_1, x_2, ..., x_m$, $m \leq 2n$ be the sorted set of x-coordinates. The ordered set of intervals $\mathcal{I}$, is defined as follows; for $i \in 1, 2, ..., m-1$, the $2(i-1)$th and $2(i-1)+1$st intervals in the set $\mathcal{I}$ are $[x_i, x_i]$ and (x_i, x_{i+1}), respectively. The last interval is $[x_m, x_m]$. A balanced binary tree T is then constructed, holding all intervals in $\mathcal{I}$ in its leaves, according to their order in $\mathcal{I}$. An internal node represents the union of the intervals of its two children, which is a contiguous interval. In addition, each internal node v stores in a variable $v.x$ the x-value of the merge point between the intervals of its two children. Since we extended the algorithm to support both open or closed boundaries, internal nodes also maintain a flag indicating whether the merge point is to the left or to the right of the x-value.

According to the description of the algorithm in [2], a sweep is performed using a horizontal line from $y = \infty$ to $y = -\infty$. The sweep-line events occur when a rectangle starts or ends, i.e., when top or bottom boundary of a rectangle is reached. Since the rectangles are not in general position, several events may share the same y-coordinate. In such a case, the order of event processing in each y-coordinate is as follows:

1. Closing rectangle with open bottom boundary events.
2. Opening rectangle with closed top boundary events.
3. Closing rectangle with closed bottom boundary events.
4. Opening rectangle with open top boundary events.

The order of event processing within each of these four groups in a specific y-coordinate is not important.

The basic idea of the algorithm is that each sweep event updates the leaves of the tree T that span the intervals that are covered by the event. Therefore, each leaf holds a counter c for the number of covering rectangles in the current position of the horizontal sweep line. In addition, each leaf maintains in a variable c_m the maximal number of covering rectangles for this leaf seen so far. Clearly, the maximal coverage of an interval is the maximal c_m of all leaves. The problem with this naïve approach is that one such update can already take $O(n)$ time. Therefore, the key idea of [2] is that when updating an event of a rectangle whose x-range is (a, b), one should follow only two paths; the path to a and the path to b. The nodes on the path should hold the information of how to update the interval spanned by their children. In the end of the update the union of intervals spanned by the updated nodes (internal nodes and only 2 leaves) is (a, b).

In order to hold the information in the internal nodes each internal node should maintain the following variables:

- l A counter storing the difference between the number of rectangles that were opened and that were closed since the last traversal of the left child of v and that cover the interval spanned by that child.
- r A counter storing the difference between the number of rectangles that were opened and that were closed since the last traversal of the right child of v and that cover the interval spanned by that child.
- l_m A counter storing the maximum value of l since the last traversal of that child.
- r_m A counter storing the maximum value of r since the last traversal of that child.

A leaf, on the other hand, holds two variables:

- c The coverage of the associated interval during the sweep at the point the leaf was traversed for the last time.
- c_m The maximum coverage of the associated interval during the sweep from the start until the leaf was traversed for the last time.

In relation to these values we define the following functions:

$$t(v) = \begin{cases} u.l + t(u) & \text{if } v \text{ is the left child of } u \\ u.r + t(u) & \text{if } v \text{ is the right child of } u \\ 0 & \text{if } v \text{ is the root} \end{cases},$$

$$t_m(v) = \begin{cases} max(u.l_m, u.l + t_m(u)) & \text{if } v \text{ is the left child of } u \\ max(u.r_m, u.r + t_m(u)) & \text{if } v \text{ is the right child of } u \\ 0 & \text{if } v \text{ is the root} \end{cases}.$$

At any point of the sweep the following two invariants hold for every leaf ℓ and its associated interval I:

- The current coverage of I is: $\ell.c + t(\ell)$.
- The maximum coverage of I that was seen so far is: $\max(\ell.c_m, \ell.c + t_m(\ell))$.

Updating the structure with an event is done as follows: Let I be the x-interval spanned by the processed rectangle creating the event. Depending on whether the rectangle starts or ends, we set a variable $d = 1$ or $d = -1$, respectively. We follow the two search paths to the leftmost leaf and the rightmost leaf that are covered by I. In the beginning the two paths are joined until they split, for every node w on this path (including the split node) we can ignore d and simply update the tuple $(w.l, w.r, w.l_m, w.r_m)$ using $t(w)$ and $t_m(w)$ according to the invariants stated above. Note that this process needs to clear the corresponding

values in the parent node as otherwise the invariants would be violated.[1] After the split the paths are processed separately. We discuss here the left path, the behavior for the right path is symmetric. Let v be a node on the left path. As long as v is not a leaf we update $(v.l, v.r, v.l_m, v.r_m)$ as usual. However, if the path continues to the left we also have to incorporate d into $v.r$ and $v.r_m$ as the subtree to the right is covered by I. If v is a leaf we simply update $v.c$ and $v.c_m$ using $t(v), t_m(v)$ and d. A more detailed description (including pseudo code) can be found in [2]. In total, this process takes $O(\log n)$ time.

Finally, in order to find the maximal number of rectangles covering an interval one last propagation from root to leaves is needed, such that all l, r, l_m, r_m values of internal nodes are cleared. This is done using one traversal on T. Now, the maximal number of rectangles covering an interval is the maximal c_m of all leaves of T.

Clearly, the running time of the algorithm is $O(n \log n)$, since constructing the tree and sorting the y-events takes $O(n \log n)$ time. Updating each of the $2n$ y-events takes $O(\log n)$ time, and the final propagation of values to the leaves takes $O(n)$ time. The algorithm uses $O(n)$ space.

We remark that the above algorithm is not optimal in memory usage in practice. A more efficient variant which stores less variables in the nodes of the tree can be easily implemented.

4.3.1.2 A Depth Preserving Reduction

Let $\mathcal{T}^c$ be a collection of open trapezoids with y-axis parallel bases with the following property: if the top or bottom curves of two different trapezoids intersect not only in joint endpoints then the two curves overlap completely in their joint x-range. Let $\mathcal{A}(\mathcal{T}^c)$ denote the arrangement of the trapezoids in $\mathcal{T}^c$. Notice that each arrangement face can be covered by overlapping trapezoids. We describe a reduction from $\mathcal{T}^c$ to $\mathcal{R}^c$, where $\mathcal{R}^c$ is a collection of axis-parallel rectangles, such that the maximum depth in $\mathcal{A}(\mathcal{R}^c)$ equals to the maximum depth in $\mathcal{A}(\mathcal{T}^c)$.

In order to define the reduction we need to have a total order $<$ on the non-vertical curves of the trapezoids in $\mathcal{T}^c$, such that one can translate the curves one by one according to this order to $y = -\infty$ without hitting other curves that have not been moved yet. Guibas & Yao [24] defined an acyclic relation $\prec$ on a set C of n interior disjoint x-monotone curves as follows:

Definition 4.5. *For two such curves $cv_i, cv_j \in C$, let the open interval (a, b) be the x-range of cv_i and the open interval (c, d) be the x-range of cv_j.*
If $x\text{-range}(cv_i) \bigcap x\text{-range}(cv_j) \neq \emptyset$ then:
$cv_i \prec cv_j \Leftrightarrow cv_i(x) < cv_j(x)$ *for some* $x \in x\text{-range}(cv_i) \bigcap x\text{-range}(cv_j)$.

As a matter of fact, their definition is more specific, in a way that the relation $cv_i \prec cv_j$ exists only if cv_i is the first curve encountered by cv_j in their joint x-range while translating cv_j to $y = -\infty$. In [24] it is also mentioned that $\prec^+$, which is the transitive closure of $\prec$, is a partial order (as it allows transitivity). This partial order $\prec^+$ can be extended to a total order $<$ in many ways. One possible extension is defined as follows:

[1] Notice that using $t(w)$ and $t_m(w)$ here takes constant time since we only need to access the parent node as all previous nodes on the path towards the root are already processed.

Definition 4.6. *Let C be a set of interior disjoint x-monotone curves. For two curves $cv_i, cv_j \in C$, let the open interval (a, b) be the x-range of cv_i and the open interval (c, d) be the x-range of cv_j.*
The total order $<$ on C is defined as follows:
$cv_i < cv_j \Leftrightarrow (cv_i \prec^+ cv_j)$ *or* $(\neg(cv_j \prec^+ cv_i)$ *and* $(cv_i$ left $cv_j))$
where $(cv_i$ left $cv_j)$ is true if the x-value of the left endpoint of cv_i is less than the x-value of the left endpoint of cv_j.

Clearly, if $cv_i \prec^+ cv_j$ is true then $cv_i < cv_j$ is true as well. If for two different curves cv_i, cv_j the expression $cv_j \prec^+ cv_i$ is true then obviously $cv_i \prec^+ cv_j$ is false and also the right expression in the "or" phrase is false, since $\neg(cv_j \prec^+ cv_i)$ is false. Therefore, $cv_i < cv_j$ is also false. If the partial order $\prec^+$ does not say anything about cv_i and cv_j then both $(cv_i \prec^+ cv_j)$ and $(cv_j \prec^+ cv_i)$ are false. Thus, $cv_i < cv_j$ will be true only if $(cv_i$ *left* $cv_j)$ is true.

Ottmann & Widmayer [34] presented a one-pass $O(n \log n)$ time algorithm for computing $<$, as in Definition 4.6, using linear space. Their algorithm performs a sweep using a horizontal line from bottom to top which stops at each endpoint of a curve. The data structure maintained by the algorithm represents the curves encountered so far in reverse order. When a bottom endpoint of a curve is met, the correct position is chosen and the curve is inserted into an auxiliary structure holding the active curves only. A curve is removed from the auxiliary structure when its top endpoint is met by the sweep line. Since we would like to translate the curves to $y = -\infty$, then we should only require the curves to be x-monotone. In addition, we can require the curves to be interior disjoint, rather than completely disjoint.

Definition 4.7. *Let Rank: $C \to \{1, ..., n\}$ denote a function returning the rank of a given x-monotone curve $cv \in C$ when sorting C according to the total order $<$.*

Definition 4.8. *We define a reduction from $\mathcal{T}^c$ to $\mathcal{R}^c$ as follows; Every trapezoid $t \in \mathcal{T}^c$ is reduced to a rectangle $r \in \mathcal{R}^c$, such that:*

- *t and r have the same x-range,*
  i.e., (left(t) = left(r)) and (right(t) = right(r)), where left and right denote the left x-value and the right x-value of t (or r), respectively.
- *top(r) and bottom(r) lie on $y =$Rank(top(t)) and $y =$Rank(bottom(t)), respectively.*

Definition 4.8 actually defines a mapping from $\mathcal{T}^c$ to $\mathcal{R}^c$, such that r is the corresponding rectangular region to t. We will now show that this mapping is bijective. One can partition the plane into vertical slabs by passing a vertical line through every endpoint of the subdivision, and then partition each slab into regions by intersecting it with all possible curves in the subdivision. This defines a decomposition of the plane into at most $2(n+1)^2$ regions (see [14], for example).

Lemma 4.9. *Let Regions(arr) denote the collection of regions of an arrangement arr, as defined above. For any region $a_t \in$ Regions($\mathcal{A}(\mathcal{T}^c)$) let $a_r \in$ Regions($\mathcal{A}(\mathcal{R}^c)$) be the corresponding rectangular region to a_t. The collection Regions($\mathcal{A}(\mathcal{R}^c)$) of all such rectangular regions spans the plane.*

Proof. Trivial. The slabs remain the same and within each slab the rectangular regions remain adjacent. □

Lemma 4.10. *Let $a_t \in$ Regions($\mathcal{A}(\mathcal{T}^c)$) be a region and let $a_r \in$ Regions($\mathcal{A}(\mathcal{R}^c)$) be the corresponding rectangular region to a_t. The number of rectangles in $\mathcal{R}^c$ that cover a_r is at least the number of trapezoids in $\mathcal{T}^c$ that cover a_t. In other words, for every $t \in \mathcal{T}^c$ that covers a_t its corresponding rectangle $r \in \mathcal{R}^c$ covers a_r.*

Proof. Let $\{t_1, t_2, ..., t_m\} \subseteq \mathcal{T}^c$ be the set of trapezoids, ordered by creation time, such that for every $i \in \{1, ..., m\}$, t_i covers a_t. Let $\{r_1, r_2, ..., r_m\} \subseteq \mathcal{R}^c$ be the set of corresponding rectangles, such that r_i corresponds to t_i for $i \in \{1, ..., m\}$. For any t_i, since t_i covers a_t we get that x-range(a_t) $\subseteq$ x-range(t_i). By Definition 4.8 the x-ranges remain the same after the reduction, and therefore x-range(a_r) $\subseteq$ x-range(r_i). Since t_i covers a_t then we also get that in the shared x-range top(t_i) is above or on top(a_t) and bottom(t_i) is below or on bottom(a_t). According to Definition 4.8, it immediately follows that Rank(top(t_i)) $\geq$ Rank(top(a_t)). In other words, top(r_i) is above or on top(a_r). Similarly, bottom(r_i) is below or on bottom(a_r). We conclude that r_i covers a_r. □

Lemma 4.11. *Let $a_r \in$ Regions($\mathcal{A}(\mathcal{R}^c)$) be a rectangular region, whose corresponding region is $a_t \in$ Regions($\mathcal{A}(\mathcal{T}^c)$). The number of trapezoids in $\mathcal{T}^c$ that cover a_t is at least the number of rectangles in $\mathcal{R}^c$ that cover a_r. In other words, for every $r \in \mathcal{R}^c$ that covers a_r its corresponding trapezoid $t \in \mathcal{T}^c$ covers a_t.*

Proof. Let $\{r_1, r_2, ..., r_m\} \subseteq \mathcal{R}^c$ be the set of rectangles, such that for every $i \in \{1, ..., m\}$, r_i covers a_r. Let $\{t_1, t_2, ..., t_m\} \subseteq \mathcal{T}^c$ be the set of corresponding trapezoids, such that t_i corresponds to r_i for $i \in \{1, ..., m\}$. Proving that for any $i \in \{1, ..., m\}$, t_i covers a_t, is done symmetrically to the proof of Lemma 4.10. □

Combining Lemma 4.10 and Lemma 4.11 we conclude that the number of trapezoids in $\mathcal{T}^c$ that cover a region a_t equals to the number of rectangles in $\mathcal{R}^c$ that cover a_r, which is the corresponding region to a_t. The covering rectangles are the reduced trapezoids in the set of trapezoids covering a_t. Since both Regions($\mathcal{A}(\mathcal{T}^c)$) and Regions($\mathcal{A}(\mathcal{R}^c)$) span the plane (Lemma 4.9), we get the following theorem.

Theorem 4.12. *Let $\mathcal{T}^c$ be a collection of open trapezoids with the following properties: their bases are y-axis parallel (vertical walls) and if the top or bottom curves of two different trapezoids intersect not only in joint endpoints then the two curves overlap completely in their joint x-range. Let $\mathcal{A}(\mathcal{T}^c)$ denote the arrangement of the trapezoids in $\mathcal{T}^c$. Notice that each arrangement face can be covered by overlapping trapezoids. $\mathcal{T}^c$ can be reduced to a collection of open axis-parallel rectangles $\mathcal{R}^c$, such that the maximum depth in $\mathcal{A}(\mathcal{R}^c)$ equals to the maximum depth in $\mathcal{A}(\mathcal{T}^c)$.*

4.3.1.3 Modification of Alt & Scharf

Based on the correctness of the reduction described in Subsection 4.3.1.2 we can extend the basic algorithm presented in Subsection 4.3.1.1 to support not only collections of axis-aligned rectangles but also collections of open trapezoids with y-axis parallel bases and non-intersecting top and bottom boundaries, if they intersect not only in joint endpoints then

they overlap completely in their joint x-range. The only part of the basic algorithm that should change is the top-to-bottom sweep. More precisely, the simple predicate that is used for sorting the y-events should be replaced with a new predicate that compares according to the reverse order of $<$, as given in Definition 4.6. The new predicate can be obtained in a preprocessing stage for computing the total order $<$ using the algorithm in [34].

Notice that for simplicity we assumed that no two distinct endpoints in the original subdivision have the same x-value. However, if this is not the case, lexicographical comparison can be used on the endpoints of the curves in order to define the order of the induced vertical walls.

4.4 Summary

The two algorithms described in Section 4.2 and Section 4.3 can be used for defining efficient construction algorithms for static settings, according to the scheme detailed in Section 4.1.

Using the verification algorithm from Section 4.2, a construction algorithm with expected $O(n \log^2 n)$ runtime is obtained. However, an asymptotically better construction algorithm can be generated using the deterministic $O(n \log n)$ verification algorithm described in Section 4.3, implying the following theorem.

Theorem 4.13. *Let S be a set of n pairwise interior disjoint x-monotone curves inducing a planar subdivision. A point location data structure for S, which has $O(n)$ size and $O(\log n)$ query time in the worst case, can be built in $O(n \log n)$ expected time.*

5

Revamp of the CGAL Trapezoidal-Map RIC for Planar Point-Location

In this chapter we present our revamp of CGAL's implementation of planar point location via the randomized incremental construction of the trapezoidal map. The new implementation is based on the previous code by Oren Nechushtan and is available as of CGAL release 4.1. In Section 5.1 we describe CGAL and its basic principals. Section 5.2 includes the necessary background regarding the Arrangement class required to understand how point location strategies for arrangements work in general. The implementation details of the class `Arr_trapezoid_ric_point_location` are discussed in Section 5.3. We assume here some familiarity of the reader with the C++ programming language and with the generic programming paradigm [7].

5.1 CGAL and Arrangements

CGAL, the Computational Geometry Algorithms Library, is a C++ library incorporating generic and robust implementations for many geometric algorithms and data structures. CGAL was launched in 1996 as a result of a collaboration between several research institutes in Europe and in Israel. It has continued evolving since, and by now contains the state-of-the-art implementations of computational geometry software in many areas. Among the algorithms and data structures provided by CGAL are: convex hull algorithms, Delaunay triangulations, Voronoi diagrams, arrangements of curves, and search structures [41, 48]. CGAL is used in various fields in industry and academia, such as computer graphics, computer aided design (CAD), scientific visualization, bioinformatics, motion planning, and more.

Two difficulties that usually appear when implementing computational geometry algorithms are providing a robust and a general implementation. Many computational geometry

algorithms are formulated assuming the "real RAM" computational model [36]. This model assumes that operations on real numbers yield accurate results and that any basic numerical operation takes constant time. Obviously, in practical applications these assumptions must be dropped, since numerical inaccuracies are inevitable while using standard computer arithmetic. Such inaccuracies may result in rounding errors leading to inconsistency in the results of topological predicates forming instable geometric algorithms [28]. One could use exact operations on real numbers in order to avoid numerical errors. Such operations, however, require more than constant time. For example, rational and algebraic numbers require additional arithmetic operations and more than constant time per operation. The second problematic issue is that many computational geometry algorithms do not handle degenerate cases. They assume that the given input is in general position, i.e., special inputs are discarded. For example, under the general position assumption, three lines do intersect at a single point and no three points lie on the same line. In real life these degenerate cases often occur. Therefore, discarding such inputs creates a considerable gap between theory and practice.

In addition, in order to allow flexibility in the design and programming of algorithms, CGAL follows the generic programming paradigm [7]. In the generic programming paradigm the description of algorithms abstracts from the concrete types and leaves them unspecified. It allows writing dynamic and general code that is not type-specific (using abstract types), at the expense of code tangibility. The concrete types are provided as parameters when the algorithm is instantiated. The specific types used to instantiate the algorithms (referred to as *models*) should follow the collections of requirements and predefined behaviors (referred to as *concepts*). Generic programming in C++ is obtained by templates. A code that uses templates has an improved running time comparing to a similar polymorphic object oriented code, since it performs static computations instead of dynamic ones. Moreover, such a code has maximal flexibility.

CGAL is composed of packages, where each package contains an efficient implementation of either a family of algorithms or a geometric data structure. CGAL packages are ascribed to three types of layers, each layer uses the layer beneath it. The geometric algorithms and geometric data structures define the topmost layer. They operate on geometric objects such as points or curves and use them in geometric predicates. The objects and predicates are grouped in "Kernels", producing the next layer. The bottom layer is the "Support Library" of CGAL consisting of fundamental utilities that are used throughout CGAL, such as extensions for the STL [52], and BOOST [47] libraries. Classes that represent special number types and operations on numbers of these types are also included in CGAL's support library. These number types are used as parameters to the kernel classes. A specific kernel can be instantiated with machine provided numbers (such as int or double) or with exact or multi-precision number types based on packages like GMP [50], Core [49], or LEDA [51]. Both the algorithm and the handled input determine which number types and kernels are appropriate providing a trade-off between efficiency and accuracy. An interesting case is when a kernel uses *Lazy exact computations*. In such a case a predicate would first try to use a floating-point evaluation, and will resort to an exact evaluation only if the floating-point evaluation could not give a clear result [13].

CGAL achieves genericity and flexibility by using *traits classes*. A certain traits class

encapsulates all data types and operations needed by a certain algorithm and is passed as an additional template parameter to the algorithm. This structure decouples the implementation of the algorithms contained in the packages from the specific geometric computation.

In CGAL planar subdivisions are referred to as arrangements, represented by the "2D Arrangements" package. The package supports the construction of 2D arrangements and various operations on them. The main class of the "2D Arrangements" package is the `Arrangement_2` class, supporting operations on planar arrangements of arbitrary bounded or unbounded curves [20, 21, 43]. An additional class, namely the `Arrangement_on_surface_2` class, which is the parent class from which the `Arrangement_2` class is derived, supports in addition arrangements embedded on certain two-dimensional orientable parametric surfaces [10]. The set of operations supported by the package includes constructing an arrangement, traversing an arrangement, performing point-location queries on an arrangement, and overlaying two arrangements.

The `Arrangement_2` class should be instantiated with the following two template classes: a *geometry traits* class and a *topology traits* class. The geometry traits class provides the geometric functionality. In other words it is responsible for defining the associated geometric types for the specific family of curves, i.e., points, curves, and x-monotone curves, and also includes the geometric operations and predicates required for these types, e.g., construction, computing the intersection of two given curves, or determining whether a point lies below or above a given curve. The traits class defines function objects (also referred to as "functors"), which often use an underlying kernel class. The topology traits class controls the topological representation of the arrangement. In other words, the topology traits class maintains the relations between the arrangement's cells (faces, edges, and vertices) and their neighboring cells while taking the embedding surface into account.

The "2D Arrangements" package provides various algorithms that operate on arrangements of different kinds. Such algorithms are the sweep line algorithm, several traversal algorithms and different point location algorithms. The next section describes the point location methods that are available when using CGAL arrangements.

5.2 Point Location in CGAL Arrangements

CGAL arrangements support point location algorithms for queries on an arrangement. Since the arrangement representation is separated from the different algorithms, it does not support the point location algorithm directly. In addition, often changes in the arrangement should notify data structures that are associated to this arrangement. For example, if an auxiliary structure is maintained by the algorithm then a change in the arrangement, e.g., a curve is inserted causing faces to split, should be followed by an update in the structure. Therefore, a notification mechanism was introduced in order to receive notifications about the occurring changes in the underlying arrangement. This mechanism is described in Subsection 5.2.1. The different point location algorithms for CGAL arrangements are later detailed in Subsection 5.2.2.

5.2.1 The Notification Mechanism

The notification mechanism offered by the arrangement package is based on *observers* [23]. Observers can be attached to an arrangement and as soon as certain structural changes occur in the arrangement these observers are notified. Any defined observer should inherit from the `Arr_observer<Arrangement>` base class. The base observer class includes a pointer for storing the observed arrangement. In addition, the base class includes a set of empty virtual notification methods, notifying before and after different possible structural changes. Each observer can decide which of these methods to implement in order to be notified before or after certain changes occur in the arrangement.

The notification methods can be grouped in three different categories, according to the type of the structural changes notified by the method. The first type of methods notifies about global changes in the arrangement, such as clearing the arrangement or assigning it with the content of a different arrangement. The second type notifies about local changes in the arrangement, such as creating a new edge, removing an edge, removing a vertex, and others. The last type notifies about global changes that are initiated by a global function such as aggregate insert. An auxiliary search structure, if needed, is efficiently constructed during such a change, and therefore no point location queries are allowed as long as the operation is not completed.

An arrangement is familiar with the list of observers observing it. Before either a global change (initiated by an arrangement method) or a local change takes place, the arrangement invokes the relevant before-methods of the observers in the list. After the event is completed the arrangement traverses the list in a reverse order, invoking the relevant after-methods. Global functions, such as aggregated insert, may also trigger notifications.

5.2.2 Point Location Strategies

The arrangement package does not support point location directly. Instead, it allows classes that are models of the *ArrangementPointLocation* concept to answer point location queries. In other words, any point location strategy used by CGAL arrangements must be a model of this concept, and as a consequence must include several predefined methods. A core method that should be defined is the `locate()` function that given a query point as an input returns the arrangement cell containing this point as a `CGAL:Object` (either a `Face_handle`, a `Halfedge_handle`, or a `Vertex_handle`).

The arrangement package includes the following point location strategies. All five classes are models of the concept *ArrangementPointLocation*.

- `Arr_naive_point_location` performs a naïve search by comparing the query point to all arrangements cells. It operates directly on the arrangement.

- `Arr_walk_along_a_line_point_location` starts at the unbounded face and continues towards the query point along a vertical ray emanating from it, until it locates the arrangement cell containing the query point. It operates directly on the arrangement.

- Arr_landmarks_point_location finds the nearest point from a set of predefined points (landmarks) and performs a walk from this landmark to the query point. There are various ways to generate the landmark set in the arrangement. By default, the arrangement vertices are the selected landmarks, but one may use other landmark generators that sample random points or choose points on a grid. It uses the notification mechanism since it requires an auxiliary data structure.

- Arr_trapezoid_ric_point_location implements the trapezoidal map random incremental construction for point location algorithm. It uses the notification mechanism since it requires an auxiliary data structure.

5.3 The Arr_trapezoid_ric_point_location Class

The main class of the trapezoidal map random incremental construction for point location strategy is the Arr_trapezoid_ric_point_location<Arrangement> class. As all other point location strategies, it is templated by the Arrangement parameter, representing the arrangement type. The Arr_trapezoid_ric_point_location class inherits from the class Arr_observer<Arrangement>, and implements (overloads) notification methods for various events. The following events are the only events affecting the Arr_trapezoid_ric_point_location class:

- Before and after assigning the arrangement with the content of another arrangement.
- Before and after clearing the contents of the arrangement.
- Before and after the observer is attached to the arrangement instance.
- Before detaching the observer from the arrangement instance.
- Before and after an edge is split into two edges.
- Before and after two edges are merged to form a single edge.
- After a new edge has been created.
- Before an edge is removed from the arrangement.

Being a model of the concept *ArrangementPointLocation* it includes a member function Object locate(Point_2 q), which implements the search using the trapezoidal RIC point location algorithm. It uses a private class which maintains the search structure and the search algorithm.

5.3.1 Representing the DAG

The underlying data structure used by the point location algorithm is a directed acyclic graph (DAG), every node of which has at most two children. The DAG is accessed through a pointer to its root. A DAG node is represented by the class Td_dag_node. Each DAG node is associated with a trapezoidal map item and in order to reduce memory usage it is reference counted. Therefore, the class Td_dag_node derives from Handle [16].

As the previous implementation, our new code can be easily applied to linear geometry but also to non-linear geometry such as algebraic curves or Bézier curves. This is possible since we follow the generic programming paradigm. However, supporting unbounded curves, as it was introduced for the "2D Arrangements" package in [10], was the main new feature that encouraged us to revise the code. In order to support unbounded curves an endpoint of a curve is represented by `Curve_end`, i.e., a reference of the original `X_monotone_curve_2` object and a flag indicating whether this is the minimal or maximal end of the curve. In addition, special predicates are used in order to compare a `Curve_end` and other geometric entities. The previous implementation, on the other hand, used the `Point_2` class for keeping the endpoint data which required every vertex to have an underlying `Point_2` object.

In the previous implementation all different types of map items were represented using a single class containing a set of fields. The major disadvantage in such a representation is that not all types of map items need to use the whole set of fields. In other words, certain types may hold spurious fields. This became even more relevant due to the additional support of unbounded curves, since endpoints are no longer represented using `Point_2` but using `Curve_end`, as described above. The chosen alternative is to define a dedicated class for every type, storing its required data only.

The eight different types of trapezoidal map items are detailed in Table 5.1, all derive from the `Handle` class. Types can be either active or inactive, that is, representing either an existing map item or an item that was already destroyed due to the insertion of later curves, respectively. There are two different types that represent an endpoint in the trapezoidal map: `Td_active_vertex` and `Td_active_fictitious_vertex`. This is due to the fact that there are two types of curve endpoints in an arrangement: an interior point and a fictitious point on the parameter space boundary. Similarly, there are two different types representing an inactive endpoint.

This separation clearly complicates other operations that were accessible using a single method in the old class and should now be available to the whole set of item classes. We wish that the set of item classes will behave as polymorphic classes, while avoiding additional class hierarchies and virtual methods. Therefore, we use the `boost::variant` class template. This class template offers a way for manipulating objects from a heterogeneous set of types in a uniform manner. It allows both a dynamic value retrieval (checked at run-time) and a type-safe value visitation, which is checked at compile-time. Thus, whenever a reference to a trapezoidal map item is needed, the `boost::variant` is used instead. For instance, it was already mentioned that the `Td_dag_node` class encapsulates a reference to the trapezoidal map item. In other words `Td_dag_node` class has a data member of type `boost::variant` templated by all the possible types for an item.

An additional data member that is contained by every object of the types mentioned in Table 5.1 is a pointer to the DAG node. The DAG node pointer allows constant time access from the map item to the correct position in the search structure.

It should be noted that in order to keep the memory consumption low all item classes avoid copying redundant data and store handled data whenever it is possible. Moreover, the data structure now operates directly on the entities of the arrangement. In particular, it avoids copying of geometric data, which can significantly reduce the amount of additional

Table 5.1: Types of trapezoidal map items and their required fields

Class Name	Stored Data	
	Description	Type
Td_active_trapezoid	Left vertex	Arrangement::Vertex_const_handle
	Right vertex	Arrangement::Vertex_const_handle
	Bottom edge	Arrangement::Halfedge_const_handle
	Top edge	Arrangement::Halfedge_const_handle
	References to the four neighboring entities	
Td_inactive_trapezoid	– None –	
Td_active_edge	Edge	Arrangement::Halfedge_const_handle
	Reference to the next edge fragment	
Td_inactive_edge	Curve	boost::shared_ptr<X_monotone_curve_2>
Td_active_vertex	Vertex	Arrangement::Vertex_const_handle
	Reference to the next edge around the vertex (clockwise)	
Td_active_fictitious_vertex	Fictitious vertex (no underlying point)	Arrangement::Vertex_const_handle
	Reference to the next edge around the vertex (clockwise)	
Td_inactive_vertex	Point	Point_2
Td_inactive_fictitious_vertex	Curve	X_monotone_curve_2
	Min/Max end	Arrangement::Arr_curve_end

memory that is used by the search structure. This is important, since due to the generic nature of the code it is not clear whether the geometric types, which are provided by the user, are referenced.

5.3.2 User Interface

In the following subsection the user interface of the `Arr_trapezoid_ric_point_location` class is described.

The `Arr_trapezoid_ric_point_location` class has two possible constructors.

- `Arr_trapezoid_ric_point_location<Arrangement> pl ( bool with_guarantees = true);`
- `Arr_trapezoid_ric_point_location<Arrangement> pl ( Arrangement arr, bool with_guarantees = true);`

The first method constructs an `Arr_trapezoid_ric_point_location` object but does not attach it to an arrangement instance. Therefore, the auxiliary search structure (DAG) remains empty until an arrangement is attached. The second method receives an arrangement instance to which the point location strategy is attached. After the object is initialized, the collection of all arrangement edges is processed and a random permutation is inserted into the DAG. Both constructor methods have an optional parameter, namely `bool with_guarantees`, whose default value is `true`. When `with_guarantees` is set to true, the construction performs rebuilds in order to guarantee a resulting structure with linear size and logarithmic query time. This is done by constantly verifying that the size is at most linear and that the depth $\mathcal{D}$ of the DAG is at most logarithmic. We can maintain both values, as described in Chapter 3, such that they are accessible in constant time. If, on the other hand, the `with_guarantees` parameter is set to false, no rebuilds will occur and therefore the structure will have expected linear size and expected logarithmic query time. In both cases the expected construction time (for static settings) would be $O(n \log n)$. When `with_guarantees` is set to `true` then we believe this bound still holds (see Conjecture in Section 3.2), but this still remain to be proven.

Since the class is a concept of the model *ArrangementPointLocation* it includes `attach` and `detach` methods. The former attaches the point location strategy to a given arrangement object and the latter detaches the strategy from the arrangement it is currently attached to. After attaching the arrangement object, the auxiliary search structure is constructed according to the current arrangement edges. The search structure is cleared before detaching the arrangement. Thus, in order to rebuild the structure, the user should call `detach` and `attach` with the same arrangement object as an argument.

The user interface includes a modifier for the `with_guarantees` parameter. The method receives a boolean value and when set to true, the structure may be reconstructed in order to guarantee linear size and logarithmic query time. Three other available methods are `depth()`, `longest_query_path()`, and `size()` returning the maximal depth, the length of the longest query path in the DAG, and the DAG size, respectively, according to the definitions provided in Chapter 3.

Table 5.2: Comparing the memory usage in the previous and the new implementation

# Generated Segments	# Arrangement Edges	Memory Usage (in MB)	
		Old Imp	New Imp
32	342	1.121	1.128
64	884	0.812	0.761
128	3798	3.382	3.375
256	17206	15.461	15.457
512	67212	60.059	60.086
1024	229652	206.109	206.047
2048	1001474	903.422	903.414

5.3.3 Memory Benchmarks

Since unbounded curves are supported as well, making the representation of an endpoint more complex, one may expect an increase in the memory consumption. We use the combination of the `Arrangement_2::Halfedge_const_handle` and a bit indicating whether this is the lexicographically minimal or maximal endpoint of the underlying curve, in order to avoid such an increase. This new design avoids copying of non-referenced geometric data, and accesses the curve through the halfedge object in the arrangement.

Another design decision for maintaining the memory usage as compact as possible in the extended implementation was the separation for different types of map items described in Subsection 5.3.1.

Table 5.2 shows that the new implementation, which is now able to support a much broader range of subdivisions, uses the same amount of memory compared to the previous implementation. This is thanks to the new design that uses `boost::variant` and avoids keeping redundant data. The table displays the memory usage (MB) for arrangements of increasing size both in the old implementation and in the new one. The input arrangements were generated using an increasing number of random segments. It should be noted that the geometric data (i.e., curves) was reference counted in all experiments of both the old and the new implementation. This implies that the space used for storing the geometry in a test with the old implementation was similar to the one in the same test with the new implementation.

5.3.4 Comparison to the Landmarks Point Location

We emphasize that the new implementation of the trapezoidal-map random incremental construction for point location (RIC) performs better than all other point location methods available for CGAL arrangements.

Figure 5.1 displays the difference in the total query time in different arrangements of random segments using the RIC vs. the Landmarks (LM) point location. The landmarks generator in this experiment created landmarks on a $\lceil\sqrt{V}\rceil \times \lceil\sqrt{V}\rceil$ grid (V is the number of vertices in the arrangement). In [26] it is shown that for subdivisions of random segments the

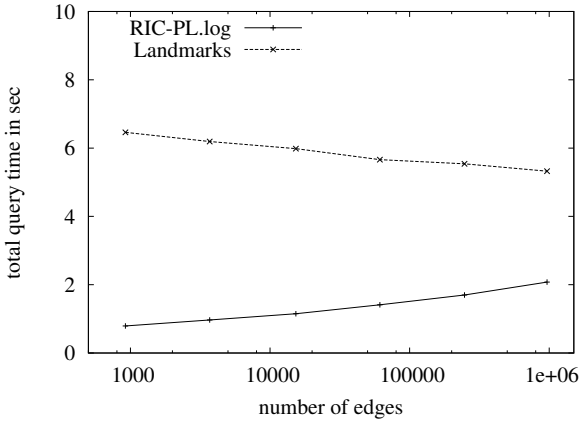


Figure 5.1: Comparing the total query time for 50k queries in random subdivision of a varying size using both the CGAL Landmarks and the RIC point location methods.

LM using the grid generator performs better than other point location methods implemented in CGAL, other than the RIC. As expected, the new RIC implementation outperforms the LM. Obviously, the RIC query time is logarithmic. The slight improvement of the query time of the LM can be explained by the fact that, at some point, while the number of input segments increases the average complexity of a face decreases, an effect similar to the one studied in [1].

6 Planar Nearest-Neighbor Search

Birn et al. [12] presented a structure for planar nearest-neighbor queries, based on Delaunay triangulations, named Full Delaunay Hierarchies (FDH). The FDH is a very simple and thus light data structure that is also very easy to construct. It outperforms many other methods in several scenarios, but it does not have a worst-case optimal behavior. However, Birn et al. [12] claim that methods that are optimal in the worst-case are too cumbersome to implement and thus not available. In particular, it was mentioned that guaranteed logarithmic nearest-neighbor search can be achieved via efficient point location on top of the Voronoi diagram of the input points, but that this approach *"does not seem to be used in practice"*. We got challenged by this claim.

This chapter emphasizes that such an approach is practically available. The definition of the nearest neighbor search problem is given in Section 6.1 along with some background. Section 6.2 and Section 6.3 discuss the nearest-neighbor search via Voronoi diagrams and via FDH, respectively. The main advantage indicated by our experiments, detailed in Section 6.4, is that using the RIC planar point location the query times are stable and independent of the actual scenario.

6.1 Definition and Background

The nearest-neighbor search problem is a fundamental problem in computer science, which has been extensively studied throughout the years. It concerns a wide range of domains, such as machine learning, geometric inference and others.

The problem is defined as follows: given a set P of n points in d-dimensional space, preprocess P into a data structure, which can efficiently report the point $p \in P$ that is closest to a given query point q, i.e., has minimal distance to q using a given distance function. This chapter focuses on two-dimensional problems, that is, planar nearest-neighbor search

problems, with Euclidean distance.

Obviously, preprocessing the points into a data structure consumes time. However, it allows a more efficient query time than with the brute-force algorithm that compares a query point to all points in P. Such approaches that do create a data structure in the preprocessing stage usually fit better to applications that require answering a large number of queries on the same set of input points.

Some approaches [6] show a significant acceleration in query time, by allowing a relatively small error in the search. In other words, such methods can return a point that may not be the nearest neighbor, but is not significantly further away from the query point than the true nearest neighbor. These methods are known as Approximate nearest-neighbor methods, as opposed to exact nearest-neighbor methods.

The following chapter compares only exact nearest-neighbor methods and not approximation methods, even though the exact version is computationally very hard. Both the FDH, described in Section 6.3, and the Voronoi-based nearest-neighbor method, described in Section 6.2, are exact methods. The two methods were compared to the kd-tree [9, 22], which is another exact method. The kd-tree is a data structure based on a recursive subdivision of the plane into disjoint cells which are rectangular regions. Each node in the kd-tree represents a region and holds the set of input points contained in this region. As soon as the number of points in a specific region exceeds some small predefined value, the region is split into two by an axis-orthogonal hyper-plane intersecting the region. Constructing the kd-tree takes $O(n \log n)$ time and requires $O(n)$ space. Query time is logarithmic when the input points are well distributed in the plane [8].

Another nearest-neighbor method that was compared is the Delaunay hierarchy [15]. Here a hierarchy of k triangulations is constructed, such that all points are contained in the triangulation of the first level, and random trials for each point decide at which level it will no longer be contained. In addition, the structure contains links between triangles in different levels and between neighboring triangles in the same level. The expected complexity of construction of the Delaunay hierarchy is $O(n \log n)$. A query may take $O(n)$ time in the worst case.

6.2 Nearest Neighbor Search via Voronoi Diagram

Suppose we are given a set P of n points which we wish to preprocess for efficient nearest-neighbor queries. We first create a Delaunay triangulation for the points in P in expected $O(n \log n)$ time using CGAL's "2D Delaunay triangulation" implementation [46]. The Voronoi diagram is then obtained by dualizing the constructed Delaunay triangulation. The arrangement representing the Voronoi diagram has at most $3n - 6$ edges, and can be constructed in $O(n \log n)$ time using a plane sweep. However, taking advantage of the spatial coherence of the edges, we use a more efficient method that directly inserts the Voronoi diagram edges while crawling over the Delaunay triangulation. We start from an arbitrary triangle in the Delaunay triangulation and insert the Voronoi edges, which are dual to the triangle edges. The algorithm continues processing the neighboring triangles in a Breadth First Search (BFS) fashion. The "2D Arrangements" package [42] is used with linear geom-

etry traits and unbounded planar topology traits in order to represent this arrangement of the Voronoi diagram.

The resulting arrangement is then further processed by our RIC implementation for point location, as described in Chapter 5. As was already mentioned, our implementation verifies the DAG depth $\mathcal{D}$ on-the-fly. Therefore, if Conjecture 3.2 is true then this takes expected $O(n \log n)$ time. Alternatively, it would have been possible to implement the solution presented in Section 4.3, for which we can prove an expected $O(n \log n)$ preprocessing time.

6.3 Nearest Neighbor Search via FDH

The Full Delaunay Hierarchies (FDH) structure, presented in [12], is based on the fact that one can find the nearest neighbor by performing a greedy walk on the edges of the Delaunay triangulation. The difference is that the FDH keeps all edges that appear during the randomized construction [3] of the Delaunay triangulation in a flattened n-level hierarchy structure, where level i contains the Delaunay triangulation of the first i points. In other words, the input points are processed in a random order such that for each point the Delaunay triangulation of the points processed so far is constructed and added to the data structure representing the hierarchy. The search for the nearest-neighbor of a query point q starts from the vertex representing the first processed point (the point with index 1). Then, at each vertex v continues to the neighboring vertex u which is closest to q of all adjacent vertices with larger index than v (the edges from v reaching these vertices are referred to as *downward edges*). The basic idea when using such an hierarchy is that a walk would be accelerated due to long edges that appeared at an early stage of the construction process while the Delaunay triangulation was still sparse.

The FDH is a very light, easy to implement, and fast data structure with expected $O(n)$ edges that can be constructed in expected $O(n \log n)$ time. The FDH achieves an expected $O(\log n)$ query path length. However, a query may take $\Theta(n)$ time since the degree of nodes in the traversed path can be linear.

Their paper [12] presents, in addition, several variants of the FDH. For instance, if we already know a "finger" f (a hint), which is supposedly close to the query point, then the search can start from f using edges to vertices with smaller indices (*upward edges*). Then, when it is no longer possible to make progress, the search continues using *downward edges* only. Other variants use exact arithmetics or plain floating point arithmetic. A usually faster exact variant first uses an inexact phase in order to accelerate the search and then continues from the point reached in the first phase with an exact walk using the "finger" variant. For the experiments in Section 6.4 we used the basic exact version (EFDH) and the usually faster variant (FFDH) that performs an inexact phase followed by an exact one. It should be noted that in the exact implementation of FDHs first an inexact distance computation using interval arithmetics is performed and the exact predicate of CGAL is then used only if the initial outcome is uncertain.

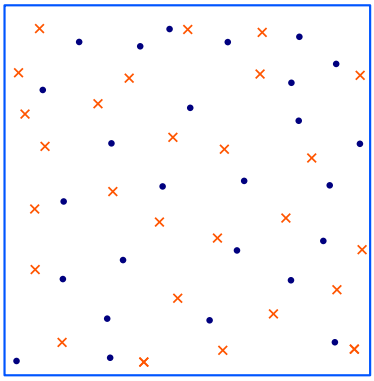
6.4 Experiments

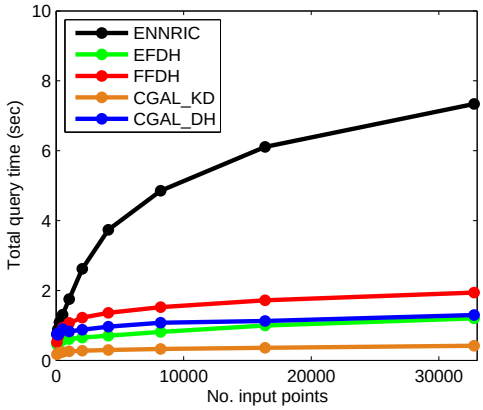
We compared our implementation for nearest-neighbor search using the RIC point location on the Voronoi-diagram (ENNRIC) to the following exact methods: EFDH, FFDH, CGAL's Delaunay hierarchy (CGAL_DH) [15], and CGAL's kd-tree (CGAL_KD).[1]

All experiments have been executed on a Intel(R) Core(TM) i5 CPU M 450 with 2.40GHz, 512 kB cache and 4GB RAM memory, running Ubuntu 10.10. Programs were compiled using `g++` version 4.4.5 optimized with `-O3` and `-DNDEBUG`. In all plots of Figure 6.1 the y-axis represents the total query time of 500k nearest-neighbor queries. Figure 6.1 (a) displays the results of the experiments in a random scenario, in which both input points and query points are uniformly sampled within the unit square. Clearly, all methods have logarithmic query time, however due to larger constants ENNRIC is slower. Figure 6.1 (b) presents a combined scenario of $(n - \lfloor \log_2 n \rfloor)$ equally spaced input points on the unit circle and $\lfloor \log_2 n \rfloor$ uniformly sampled outliers in the unit square. The queries are uniformly sampled points in the same region. In these experiments both the CGAL_KD and ENNRIC are significantly faster and maintain a stable query time. A similar scenario, presented in Figure 6.1 (c), contains equally spaced input points on a circle and a point in the center. The query points are sampled uniformly inside the circle. This scenario showed even more significant differences between the five methods, as compared to the circle with outliers scenario. In fact, it acts as a worst-case scenario for many methods. In particular, the query time for the FDH in such a scenario may be linear, and as indicated by the plot the runs for both EFDH and FFDH with 2^{16} input points could not complete. The query time of ENNRIC, on the other hand, was stable and similar to the query time in other scenarios of similar size, and in this scenario was always faster than the other methods.

The main drawback of the ENNRIC method, comparing to the other methods, is the considerable preprocessing time. In all tested scenarios ENNRIC required significantly longer preprocessing time in order to construct the efficient search structure for queries, and obviously could not compete with the fast construction time of the other methods.

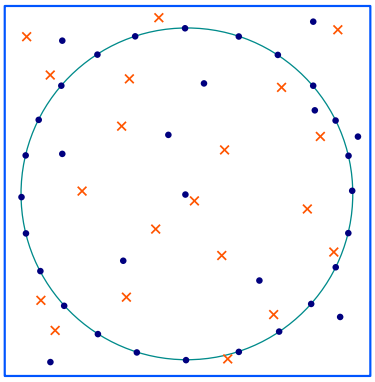
[1] Due to similar performance we elided the kd-tree implementation in ANN [31].

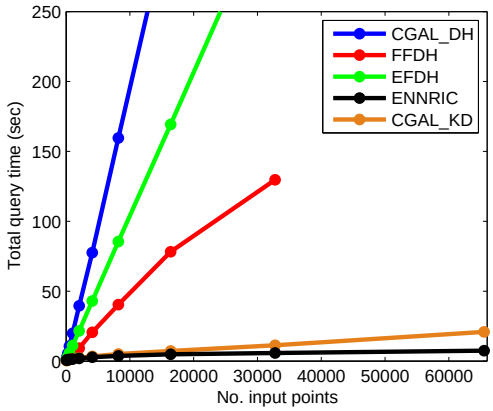




(a)

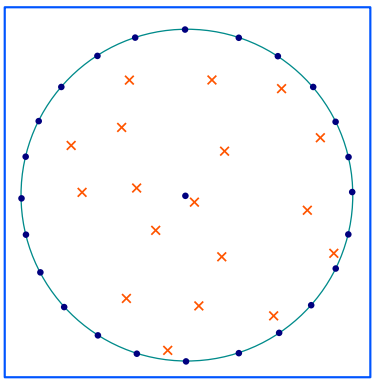
(a) Random points

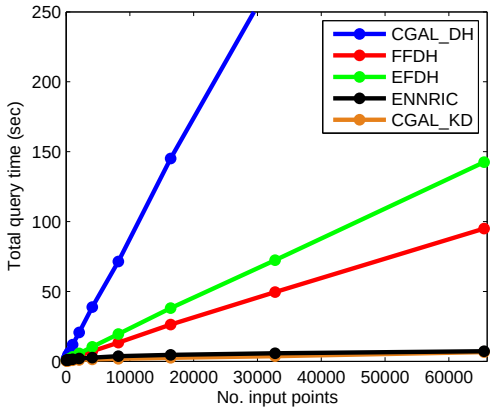




(b)

(b) Points on a circle with logarithmic number of outliers





(c)

(c) Points on a circle with a center

Figure 6.1: Performance of 500k nearest-neighbor queries for different methods on three scenarios: (a) random points; (b) circle and center with outliers; (c) circle and center.

7

Conclusions and Open Problems

In this thesis we described the details of the revised implementation of planar point location using the trapezoidal-map random incremental construction algorithm in CGAL. Our implementation provides a data structure for arbitrary curves (linear or non-linear) guaranteeing logarithmic query time and linear size for any input. It supports any subdivision that can be represented in CGAL by the "2D Arrangements" package [42]. The trigger for this major revision is the additional support of unbounded curves. Moreover, it can also be applied to arrangements of curves embedded on certain two-dimensional orientable parametric surfaces in three-dimensional space. Our implementation is exact, complete, and general. We conjecture, based on our experimental results, that its expected preprocessing time is $O(n \log n)$.

Another contribution of this thesis is the study of the fundamental difference between the length $\mathcal{L}$ of the longest search path and the DAG depth $\mathcal{D}$, which is the length of the longest path in the constructed DAG. Clearly, determining the value of $\mathcal{L}$ is rather expensive while $\mathcal{D}$ is easy to maintain. We clarified why the two entities are not trivially interchangeable and proved that the worst case ratio between $\mathcal{D}$ and $\mathcal{L}$ is bounded by $\Theta(n/\log n)$.

We also presented two construction algorithms for static settings, where all curves are given at the preprocessing stage. We showed that the time complexity of the first algorithm is expected $O(n \log^2 n)$. For the second algorithm we proved an asymptotically faster bound for the time complexity of only expected $O(n \log n)$. Therefore, we proved that given a subdivision of n pairwise interior disjoint x-monotone curves a point location data structure which has $O(n)$ size and $O(\log n)$ query time in the worst case can be built in expected $O(n \log n)$ time. The main advantage of the first algorithm is that it does not require construction of any other structures and only uses the already constructed DAG. Moreover, we conjecture that it is possible to refine the analysis of this algorithm and prove an expected $O(n \log n)$ bound by utilizing its recursive nature.

A possible application for the planar point location implementation that we provided here is a guaranteed logarithmic planar nearest-neighbor search. It is obtained by using our point location implementation on top of the Voronoi diagram of the input points, and as opposed to many other methods allows a stable logarithmic query time, which is independent of the scenario.

Further improvements of the CGAL point location implementation should be considered as well. For instance, a potential future work would be to reduce the actual preprocessing time. In addition, since the code requires a considerable amount of memory, making the implementation more compact in terms of space is another desired goal. Both issues are not trivial to solve. In fact, it seems that both the preprocessing time and memory usage could not be reduced significantly.

Our implementation treats isolated vertices in a naïve manner. That is, after the face containing the query point is found, the query is compared to all isolated vertices inside this face. This is clearly linear in the number of isolated vertices inside the face. A major improvement would be to treat the isolated vertices as we treat the input curves, that is, insert them into the DAG during the random incremental construction. However, in the DAG, each isolated vertex will be represented by an appropriate point-node.

One major open problem is to prove Conjecture 3.2, that is, prove that it is possible to rely on the depth $\mathcal{D}$ of the DAG and still expect only a constant number of rebuilds. This solution would not require any changes to the current implementation and is preferable than other solutions since computing the depth does not require geometric operations, which are practically rather costly.

Bibliography

[1] Noga Alon, Dan Halperin, Oren Nechushtan, and Micha Sharir. The complexity of the outer face in arrangements of random segments. In *Proceedings of the twenty-fourth Annual ACM Symposium on Computational Geometry (SoCG)*, pages 69–78, 2008.

[2] Helmut Alt and Ludmila Scharf. Computing the depth of an arrangement of axis-aligned rectangles in parallel. In *Proceedings of the twenty-sixth European Workshop on Computational Geometry*, pages 33–36, Dortmund, Germany, March 2010.

[3] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con BRIO. In *Proceedings of the nineteenth Annual ACM Symposium on Computational Geometry (SoCG)*, pages 211–219, 2003.

[4] Sunil Arya, Theocharis Malamatos, and David M. Mount. Entropy-preserving cuttings and space-efficient planar point location. In *Proceedings of the twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 256–261, 2001.

[5] Sunil Arya, Theocharis Malamatos, and David M. Mount. A simple entropy-based algorithm for planar point location. In *Proceedings of the twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 262–268, 2001.

[6] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 271–280, 1993.

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

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[9] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Proceedings of the sixth Annual ACM Symposium on Computational Geometry (SoCG)*, pages 187–197, 1990.

[10] Eric Berberich, Efi Fogel, Dan Halperin, Kurt Mehlhorn, and Ron Wein. Sweeping and maintaining two-dimensional arrangements on surfaces: A first step. In *Proceedings of the fifteenth Annual European Symposium on Algorithms (ESA)*, pages 645–656, 2007.

[11] Eric Berberich, Efi Fogel, Dan Halperin, Kurt Mehlhorn, and Ron Wein. Arrangements on parametric surfaces I: General framework and infrastructure. *Mathematics in Computer Science*, 4:67–91, 2010.

[12] Marcel Birn, Manuel Holtgrewe, Peter Sanders, and Johannes Singler. Simple and fast nearest neighbor search. In *Proceedings of the twelfth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 43–54, 2010.

[13] Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2D and 3D geometry kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012.

[14] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, third edition, 2008.

[15] Olivier Devillers. The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13(2):163–180, 2002.

[16] Olivier Devillers, Lutz Kettner, Michael Seel, and Mariette Yvinec. Handles and circulators. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012.

[17] David P. Dobkin and Richard J. Lipton. Multidimensional searching problems. *SIAM Journal on Computing*, 5(2):181–186, 1976.

[18] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[19] Eyal Flato, Dan Halperin, Iddo Hanniel, Oren Nechushtan, and Eti Ezra. The design and implementation of planar maps in CGAL. *The ACM Journal of Experimental Algorithmics*, 5:13, 2000.

[20] Efi Fogel, Dan Halperin, and Ron Wein. *CGAL Arrangements and Their Applications*. Springer, 2012.

[21] Efi Fogel, Ron Wein, and Dan Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA)*, pages 664–676, 2004.

[22] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, September 1977.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[24] Leonidas J. Guibas and F. Frances Yao. On translating a set of rectangles. In *Proceedings of the twelfth Annual ACM Symposium on Theory of Computing (STOC)*, pages 154–160, 1980.

[25] Sariel Har-Peled. Personal communication, 2012.

[26] Idit Haran and Dan Halperin. An experimental study of point location in planar arrangements in CGAL. *The ACM Journal of Experimental Algorithmics*, 13, 2008.

[27] Michael Hemmer, Michal Kleinbort, and Dan Halperin. Improved implementation of point location in general two-dimensional subdivisions. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA)*, pages 611–623, 2012.

[28] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee-Keng Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry: Theory and Applications*, 40(1):61–78, May 2008.

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

[30] D. T. Lee and Franco P. Preparata. Location of a point in a planar subdivision and its applications. In *Proceedings of the eighth Annual ACM Symposium on Theory of Computing (STOC)*, pages 231–235, 1976.

[31] David M. Mount and Sunil Arya. ANN: A library for approximate nearest neighbor searching. `http://www.cs.umd.edu/~mount/ANN/`.

[32] Ketan Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10(3/4):253–280, 1990.

[33] Ketan Mulmuley. *Computational geometry - an introduction through randomized algorithms*. Prentice Hall, 1994.

[34] Thomas Ottmann and Peter Widmayer. On translating a set of line segments. *Computer Vision, Graphics, and Image Processing*, 24(3):382–389, 1983.

[35] Franco P. Preparata. A new approach to planar point location. *SIAM Journal on Computing*, 10(3):473–482, 1981.

[36] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[37] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[38] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1:51–64, 1991.

[39] Raimund Seidel and Udo Adamy. On the exact worst case query complexity of planar point location. *Journal of Algorithms*, 37(1):189–217, 2000.

[40] Jack Snoeyink. Point location. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 34, pages 767–785. Chapman & Hall/CRC, 2nd edition, 2004.

[41] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012.

[42] Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman. 2D arrangements. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012.

[43] Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. Advanced programming techniques applied to CGAL's arrangement package. *Computational Geometry: Theory and Applications*, 38(1-2):37–63, September 2007.

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

[45] Chee-Keng Yap and Thomas Dubé. The exact computation paradigm. In *Computing in Euclidean Geometry*, volume 1, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

[46] Mariette Yvinec. 2D triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012.

Links

[47] Boost — portable C++ libraries.
http://www.boost.org.

[48] Cgal — computational geometry algorithms library.
http://www.cgal.org.

[49] Core number library.
http://cs.nyu.edu/exact/core_pages.

[50] Gmp — GNU multiple precision arithmetic library.
http://gmplib.org.

[51] Leda — library of efficient data types and algorithms.
http://www.algorithmic-solutions.com/leda/index.htm.

[52] Stl — C++ standard template library.
http://www.sgi.com/tech/stl.