38 POINT LOCATION

Jack Snoeyink

INTRODUCTION

"Where am I?" is a basic question for many computer applications that employ geometric structures (e.g., in computer graphics, geographic information systems, robotics, and databases). Given a set of disjoint geometric objects, the ***point-location problem*** asks for the object containing a query point that is specified by its coordinates. Instances of the problem vary in the dimension and type of objects and whether the set is static or dynamic. Classical solutions vary in preprocessing time, space used, and query time. Recent solutions also consider entropy of the query distribution, or exploit randomness, external memory, or capabilities of the word RAM machine model.

Point location has inspired several techniques for structuring geometric data, which we survey in this chapter. We begin with point location in one dimension (Section 38.1) or in one polygon (Section 38.2). In two dimensions, we look at how techniques of persistence, fractional cascading, trapezoid graphs, or hierarchical triangulations can lead to optimal comparison-based methods for point location in static subdivisions (Section 38.3), the current best methods for dynamic subdivisions (Section 38.4), and at methods not restricted to comparison-based models (Section 38.5). There are fewer results on point location in higher dimensions; these we mention in (Section 38.6).

The vision/robotics term ***localization*** refers to the opposite problem of determining (approximate) coordinates from the surrounding local geometry. This chapter deals exclusively with point location.

38.1 ONE-DIMENSIONAL POINT LOCATION

The simplest nontrivial instance of point location is list searching. The objects are points $x_1 \leq \cdots \leq x_n$ on the real line, presented in arbitrary order, and the intervals between them, (x_i, x_{i+1}) for $1 \leq i < n$. The answer to a query q is the name of the object containing q.

The list-searching problem already illustrates several aspects of general point location problems and several data structure innovations.

GLOSSARY

Decomposable problem: A problem whose answer can be obtained from the answers to the same problem on the sets of an arbitrary partition of the input [Ben79, BS80]. The one-dimensional point location as stated above—find

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry,
J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

the interval containing q—is not decomposable, since partitioning into subsets of points gives a very different set of intervals. "Find the lower endpoint of the containing interval" is decomposable, however; one can report the highest "lowest point" returned from all subsets of the partition.

Preprocessing/queries: If one assumes that many queries will search the same input, then resources can profitably be spent building data structures to facilitate the search. Three resources are commonly analyzed:

- ***Query time:*** Computation time to answer a single query, given a point location data structure. Usually a worst-case upper bound, expressed as a function of the number of objects in the structure, n.
- ***Preprocessing time:*** Time required to build a point location structure for n objects.
- ***Space:*** Memory used by the point location structure for n objects.

Dynamic point location: Maintaining a location data structure as points are inserted and deleted. The one-dimensional point location structures can be made dynamic without changing their asymptotic performances.

Randomized point location: Data structures whose preprocessing algorithms may make random choices in an attempt to avoid poor performance caused by pathological input data. Preprocessing and query times are reported as expectations over these random choices. Randomized algorithms make no assumptions on the input or query distributions. They often use a sample to obtain information about the input distribution, and can achieve good expected performance with simple algorithms.

Entropy bounds: If the probability of a query falling in region i is p_i, then Shannon entropy $H = \sum_i -p_i \log_2(p_i)$ is a lower bound for expected query time, where the expectation is over the query probability distribution.

Static optimality: A (self-adjusting) search structure has static optimality if, for any (infinite) sequence of searches, its cumulative search time is asymptotically bounded by cumulative time of the best static structure for those searches.

Transdichotomous: Machine models, such as the word RAM, that are not restricted to comparisons, are called transdichotomous if they support bit operations or other computations that allow algorithms to break through information-theoretic lower bounds that apply to comparison-based models, such as decision trees.

LIST SEARCH AS ONE-DIMENSIONAL POINT LOCATION

Table 38.1.1 reports query time, preprocessing time, and space for several search methods. Linear search requires no additional data structure if the problem is decomposable. Binary search trees or randomized search trees [SA96, Pug90] require a total order and an ability to do comparisons. An adversary argument shows that these comparison-based query algorithms require $\Omega(\log n)$ comparisons. If however, searches will be near each other, or near the ends, a finger search tree can find an element d intervals away in $O(\log d)$ time. If the probability distribution for queries is known, then the lower bound on expected query time is H, and expected $H + 2$ can be achieved by weight-balanced trees [Meh77]. Even if the distribution is not known, splay trees achieve static optimality [ST85].

Note that these one-dimensional structures can be built dynamically by operations that take the same amortized time as performing a query. So in theory, we need not report the preprocessing time; that will change in higher dimensions.

If we step away from comparison-based models, a useful method in practice is to partition the input range into b equal-sized buckets, and to answer a query by searching the bucket containing the query. If the points are restricted to integers $[1, \ldots, U]$, then van Emde Boas [EBKZ77] has shown how hashing techniques can be applied in stratified search trees to answer a query in $O(\log\log U)$ time. Combining these with Fredman and Willard's fusion trees [FW93] can achieve $O(\sqrt{\log n})$-time queries without the restriction to integers [CP09].

TABLE 38.1.1 List search as one-dimensional point location.

TECHNIQUE	QUERY	PREPROC	SPACE
Linear search	$O(n)$	none	data only
Binary search	$O(\log n)$	$O(n \log n)$	$O(n)$
Randomized tree	exp. $O(\log n)$	exp. $O(n \log n)$	$O(n)$
Finger search	$O(\log d)$	$O(n \log n)$	$O(n)$
Weight-balance tree	exp. H+2	$O(n \log n)$	$O(n)$
Splay tree	$O(OPT)$ in limit	$O(n)$	$O(n)$
Bucketing	$O(n)$	$O(n+b)$	$O(n+b)$
van Emde Boas tree	$O(\log\log U)$	exp. $O(n)$	$O(n)$
word RAM	$O(\sqrt{\log n})$	$O(n \log\log n)$	$O(n)$

38.2 POINT-IN-POLYGON

The second simplest form of point location is to determine whether a query point q lies inside a given n-sided polygon P [Hai94]. Without preprocessing the polygon, one may use parity of the winding or crossing numbers: count intersections of a ray from q with the boundary of polygon P. Point q is inside P iff the number is odd. A query takes $O(n)$ time.

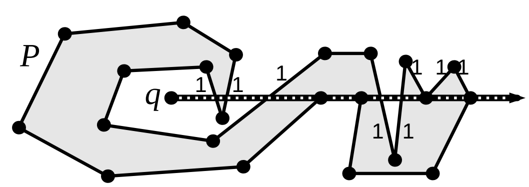


FIGURE 38.2.1
Counting degenerate crossings: eight crossings imply $q \notin P$.

One must count carefully in degenerate cases when the ray passes through a vertex or edge of P. When the ray is horizontal, as in Figure 38.2.1, then edges of P can be considered to contain their lower but not their upper endpoints. Edges inside the ray can be ignored. This is consistent with the count obtained by perturbing the ray infinitesimally upward. Schirra [Sch08] experimentally observes which points are incorrectly classified using inexact floating point arithmetic in various algorithms. Stewart [Ste91] considered point-in-polygon algorithm design when vertex and edge positions may be imprecise.

To obtain sublinear query times, preprocess the polygon P using the more general techniques of the next sections.

38.3 PLANAR POINT LOCATION: STATIC

Theoretical research has produced a number of planar point location methods that are optimal for comparison-based models: $O(n \log n)$ time to preprocess a planar subdivision with n vertices for $O(\log n)$ time queries using $O(n)$ space. Preprocessing time reduces to linear if the input is given in an appropriate format, and some preprocessing schemes have been parallelized (see Chapter 46).

We focus on the data structuring techniques used to reach optimality: persistence, fractional cascading, trapezoid graphs, and hierarchical triangulations.

In a planar subdivision, point location can be made decomposable by storing with each edge the name of the face immediately above. If one knows for each subproblem the edge below a query, then one can determine the edge directly below and report the containing face, even for an arbitrary partition into subproblems.

GLOSSARY

Planar subdivision: A partitioning of a region of the plane into point ***vertices,*** line segment ***edges,*** and polygonal ***faces.***

Size of a planar subdivision: The number of vertices, usually denoted by n. Euler's relation bounds the numbers of edges $e \leq 3n - 6$ and faces $f \leq 2n - 4$; often the constants are suppressed by saying that the number of vertices, edges, and faces are all $O(n)$.

Monotone subdivision: A planar subdivision whose faces are x-monotone polygons: i.e., the intersection of any face with any vertical line is connected.

Triangulation/trapezoidation: Planar subdivisions whose faces are triangles/ whose faces are trapezoids with parallel sides all in the same direction.

Dual graph: A planar subdivision can be viewed as a graph with vertices joined by edges. The dual graph has a node for each face and an arc joining two faces if they share a common edge.

SLABS AND PERSISTENCE

By drawing a vertical line through every vertex, as shown in Figure 38.3.1(a), we obtain vertical ***slabs*** in which point location is almost one-dimensional. Two binary searches suffice to answer a query: one on x-coordinates for the slab containing q, and one on edges that cross that slab. Query time is $O(\log n)$, but space may be quadratic if all edges are stored with the slabs that they cross.

The location structures for adjacent slabs are similar. We could sweep from left to right to construct balanced binary search trees on the edges for all slabs:

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

TABLE 38.3.1 A select few of the best static planar point location results known for subdivision with n edges. Expectations are over decisions made by the algorithm; averages are over a query distribution with entropy H. For distance sensitivity, scale the subdivision to have unit area and denote the distance from query q to the nearest boundary by Δ_q. The static optimality result is for regions of constant complexity.

TECHNIQUE	QUERY	PREPROC	SPACE
Slab + persistence [ST86]	$O(\log n)$	$O(n \log n)$	$O(n)$
Separating chain + fractional cascade [EGS86]	$O(\log n)$	$O(n \log n)$	$O(n)$
Randomized [HKH16]	$O(\log n)$	exp. $O(n \log n)$	$O(n)$
Weighted randomized [AMM07]	average $(5 \ln 2)H + O(1)$	exp. $O(n \log n)$	exp. $O(n)$
Optimal query [SA00]	$\log_2 n + \sqrt{\log_2 n} + \Theta(1)$	$O(2^{2\sqrt{\log n}})$	$O(2^{2\sqrt{\log n}})$
+ struct. sharing	$\log_2 n + \sqrt{\log_2 n} + O(\log_2^{1/4} n)$	exp. $O(n \log n)$	exp. $O(n)$
Optimal entropy [CDI+12]	avg. $H + O(H^{1/2} + 1)$	exp. $O(n \log n)$	$O(n)$
Distance sensitive [ABE+16]	$N \min\{\log n, -\log \Delta_q\}$	exp. $O(n \log n)$	$O(n)$
Static optimality [IM12]	$O(OPT)$ in limit	$O(n)$	$O(n)$

As we sweep over the right endpoint of an edge, we remove the corresponding tree node. As we sweep over the left endpoint of an edge, we add a node. This takes $O(n \log n)$ total time and a linear number of node updates. To store all slabs in linear space, Sarnak and Tarjan [ST86] add to this the idea of *persistence*.

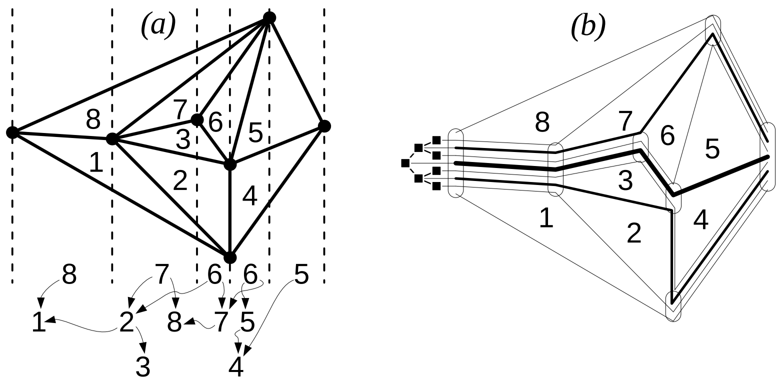
Rather than modifying a node to update the tree, copy the $O(\log n)$ nodes on the path from the root to this node, then modify the copies. This ***node-copying persistence*** preserves the former tree and gives access to a new tree (through the new root) that represents the adjacent slab. The total space for n trees is $O(n \log n)$. Figure 38.3.1(a) provides an illustration. The initial tree contains 8 and 1. (Recall that edges are named by the face immediately above.) Then 2, 3, and 7 are added, 8 is copied during rebalancing, but node 1 is not changed. When 6 is added, 7 is copied in the rebalancing, but the two subtrees holding 1, 2, 3, and 8 are not changed.

Limited node copying reduces the space to linear. Give each node spare left and right pointers and left and right time-stamps. Build a balanced tree for the initial slab. When a pointer is to be modified, use a spare and time-stamp it, if there is a spare available. Future searches can use the time-stamp to determine whether to follow the pointer or the spare. Otherwise, copy the node and modify its ancestor to point to the copy. If the slab location structures are maintained with $O(1)$ rotations per update, then the amortized cost of copying is also $O(1)$ per update.

Preprocessing takes $O(n \log n)$ time to sort by x coordinates and build either persistent data structure. To compare constants with other methods, the data structure has about 12 entries per edge because of extra pointers and copying. Searches take about $4 \log_2 N$ comparisons, where N is the number of edges that can intersect a vertical line; this is because there are two comparisons per node and "$O(1)$ rotation" tree-balancing routines are balanced only to within a factor of two.

We will see two other slab-based data structures in Section 38.4 on dynamic point location: interval trees and segment trees recursively merge slabs, and save

FIGURE 38.3.1
Optimal static methods: (a) *Slab (persistent);* (b) *separating chain (fractional cascading).*



space by choosing where to store segments in the resulting slab tree. The other comparison-based point location schemes in this section do not represent slabs explicitly. Nevertheless, Chan and Pătraşcu [CP09] have convincingly argued that point location in a slab is a fundamental operation in computational geometry; by using a word RAM to perform point location in a slab faster than the comparison-based lower bounds (Section 38.5), they are able to speed up many classical geometric computations.

SEPARATING CHAINS AND FRACTIONAL CASCADING

If a subdivision is monotone, then its faces can be totally ordered consistent with aboveness; in other words, we can number faces $1, \ldots, f$ so that any vertical line encounters lower numbers below higher numbers. The ***separating chain*** between the faces $< k$ and those $\geq k$ is a monotone chain of edges [LP77]. Figure 38.3.1(b) shows all separating chains for a subdivision; the middle chain, $k = 5$, is shown darkest.

A balanced binary tree of separating chains can be used for point location: if query point q is above chain i and below chain $i+1$, then q is in face i. To preserve linear space we need to avoid the duplication of edges in chains that can be seen in Figure 38.3.1(b).

Note that the separating chains that contain an edge are defined by consecutive integers; we can store the first and last with each edge. Then form a binary tree in which each subtree stores the separating chains from some interval: at each node, store the edges of the median chain that have not been stored higher in the tree, and recursively store the intervals below and above the median in the left and right subtrees respectively. The root, for example, stores all edges of the middle chain. Since no edge is stored twice, this data structure takes $O(n)$ space.

As we search the tree for a query point q, we keep track of the edges found so far that are immediately above and below q. (Initially, no edges have been found.) Now, the root of the subtree to search is associated with a separating chain. If that chain does not contain one of the edges that we know is above or below q, then we search the x-coordinates of edges stored at the node and find the one on the vertical line through q. We then compare against the separating chain and recursively search the left or right subtree. Thus, this separating chain method [LP77] inspects $O(\log n)$ tree nodes at a cost of $O(\log n)$ each, giving $O(\log^2 n)$ query time.

To reduce the query time, we can use fractional cascading [CG86, EGS86] for efficient search in multiple lists. As we traverse our search tree, at every node we search a list by x-coordinates. We can make all searches after the first take constant time, if we increase the total size of these lists by 50%. Pass every fourth x-coordinate from a child list to its parent, and establish connections so that knowing one's position in the parent list gives one's position in the child to within four nodes.

Preprocessing takes $O(n)$ time on a monotone subdivision; arbitrary planar subdivisions can be made monotone by plane sweep in $O(n \log n)$ time. One can trade off space and query time in fractional cascading, but typical constants are 8 entries per edge for a query time of $4 \log_2 n$.

TRAPEZOID GRAPH METHODS

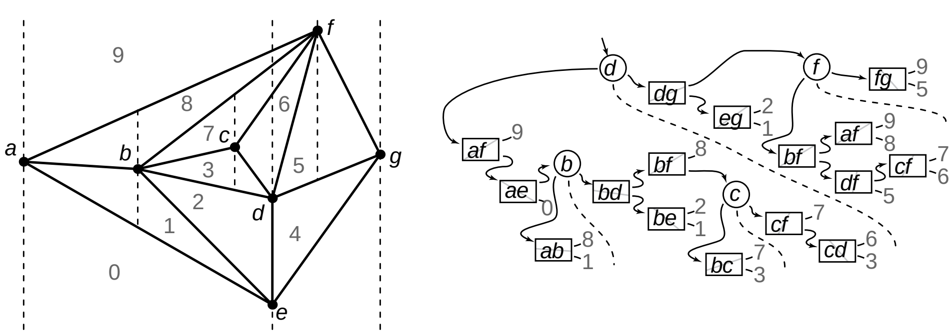
Preparata's [Pre81] trapezoid method is a simple, practical method that achieves $O(\log n)$ query time at the cost of $O(n \log n)$ space. Its underlying search structure, the ***trapezoid graph***, is the basis for important variations: randomized point location in optimal expected time and space, a recursive application giving exact worst-case optimal query time, and, in the next section, average time point location achieving the entropy bound.

A trapezoid graph is a directed, acyclic graph (DAG) in which each non-leaf node ν is associated with a trapezoid τ_ν whose parallel sides are vertical and whose top and bottom are either a single subdivision edge or are at infinity. Node ν splits τ_ν either by a vertical line through a subdivision vertex (a vertical node) or by a subdivision edge (a horizontal node). The root is associated with a trapezoid that contains the entire subdivision; each leaf reports the region that contains its implied trapezoid.

Most planar point location structures can be represented as trapezoid graphs, including the slab and separating chain methods. Bucketing and some triangulation methods cannot, since they may make comparisons with coordinates or segments that are not in the input.

FIGURE 38.3.2

An example subdivision with its trapezoid graph, using circles for vertical splits at vertices, rectangles for horizontal splits at edges, and numbered leaves. Edges af, bf, and cf are cut and duplicated.



In Preparata's trapezoid method of point location, the trapezoid graph is a tree constructed top-down from the root. Figure 38.3.2 shows an example. If trapezoid τ_ν does not contain a subdivision vertex or intersect an edge, then node ν is a leaf. If every subdivision edge intersecting τ_ν has at least one endpoint inside τ_ν, then make ν a vertical node and split τ_ν by a vertical line through the median vertex. Otherwise, make ν a horizontal node and split τ_ν by the median of all edges cutting through τ_ν, and call ν a *horizontal split node*. This tree has depth (and query time) $3\log n$ [SA00]. Experiments [EKA84] suggest that this method performs well, although its worst-case size and preprocessing time are $O(n\log n)$.

In a delightful paper, Seidel and Adamy [SA00] give the exact number of comparisons for point location in a planar subdivision of n edges by establishing a tight bound of $\log_2 n + \sqrt{\log_2 n} + \Theta(1)$ on the worst-case height of a trapezoid graph. (The paper has an extra factor of $O(\log_2 \log_2 n)$ that was removed by Seidel and Kirkpatrick [unpublished].) The lower bound uses a stack of $n/2$ horizontal lines that are each cut into two along a diagonal.

The upper bound divides a trapezoid into $t = 2\sqrt{\log_2 n}$ slabs and uses horizontal splits to define trapezoids with point location subproblems to be solved recursively. Each subproblem with a location structure of depth d is given weight 2^d, and a weight balanced trapezoid tree is constructed to determine the relevant subproblem for a query. Query time in this trapezoid tree is optimal. Preprocessing time is determined by the number of tree nodes, which is $O(n2^t)$.

They also show that $\Omega(n\log n)$ space is required for a trapezoid tree, but that space can be reduced to linear by using cuttings to make the trapezoid graph into a DAG.

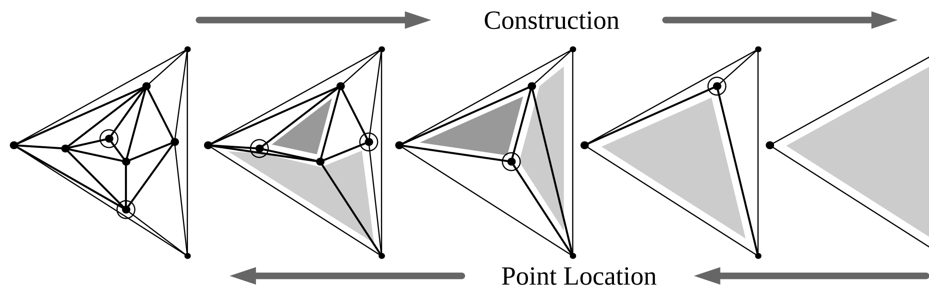
A space-efficient trapezoid graph can be most easily built as the history graph (a DAG) of the randomized incremental construction (RIC) of an arrangement of segments [Mul90, Sei91] (see Chapter 28 and Section 44.2). RIC gives an expected optimal point location scheme: $O(\log n)$ expected query time, $O(n\log n)$ expected preprocessing time, and $O(n)$ expected space, where the expectation is taken over random choices made by the construction algorithm. Hemmer et al. [HKH16] guarantee the query time and space bounds in the worst case: They develop efficient verification for space and query time of a structure by allowing the maximum path length, or depth of the DAG, to remain large as long as the longest query path remains logarithmic. By rerunning the randomized preprocessing if the space and query time bounds cannot be verified, the expectation remains only on the preprocessing time.

TRIANGULATIONS

Kirkpatrick [Kir83] developed the second optimal point-location method specifically for triangulations. This is not a restriction for subdivisions specified by vertex coordinates, since any planar subdivision can be triangulated, although it can be an added complication to do so. It can increase the required precision for subdivisions whose vertices are computed, such as Voronoi diagrams.

This scheme creates a hierarchy of subdivisions in which all faces, including the outer face, are triangles. Although point location based on hierarchical triangulations suffers from large constant factors, but the ideas are still of theoretical and practical importance. Hierarchical triangulations have become an important tool for algorithmic problems on convex polyhedra, terrain representation, and mesh

FIGURE 38.3.3
Hierarchical triangulation.



simplification.

In every planar triangulation, one can find (in linear time) an independent set of low-degree vertices that consists of a constant fraction of all vertices. In Figure 38.3.3 these are circled and, in the next picture, are removed and the shaded hole is retriangulated if necessary. Repeating this "coarsening" operation a logarithmic number of times gives a constant-size triangulation.

To locate the triangle containing a query point q, start by finding the triangle in the coarsest triangulation, at right in Figure 38.3.3. Knowing the hole (shaded) that this triangle came from, one need only replace the missing vertex and check the incident triangles to locate q in the previous, finer triangulation.

Given a triangulation, preprocessing takes $O(n)$ time, but the hidden constants on time and space are large. For example, choosing the independent set by greedily taking vertices in order of increasing degree up to 10 guarantees $1/6$th of the vertices [SK97], which leads to a data structure with $12n$ triangles in which a query could take $35 \log_2 n$ comparisons.

ENTROPY BOUNDS

The work of Malamotos with Arya, Mount, and co-authors initiated a fruitful exploration into how to modify the schemes above if we know something about the query distribution. By analogy to weights in a weighted binary search tree, suppose that we have a planar subdivision with regions of constant complexity (e.g., trapezoids or triangles) and that we know the probability p_i of a query falling in the ith region. The entropy is $H = \sum_i -p_i \log_2 p_i$. Arya et al. [AMM07] showed that a weighted randomized construction gives expected query times satisfying ***entropy bounds***. For a constant K, assign to a subdivision edge that is incident on regions with total probability P the weight $\lceil KPn \rceil$, and perform a randomized incremental construction. The use of integral weights ensures that ratios of weights are bounded by $O(n)$, which is important to achieve query time bounded by $O(H)$.

Entropy-preserving cuttings can be used to give a method whose query time of $H + O(1 + H^{1/2})$ approaches the optimal entropy bound [AMMW07], at the cost of increased space and programming complexity.

A subtlety related to decomposability has tripped up a few researchers: entropy is easy to work with only if the region descriptions have constant complexity, but triangulation or trapezoidation of complex regions can increase entropy. Collette et al. [CDI$^+$12] work with connected planar subdivision G and define an entropy $\hat{H}(G, D)$ as the expected cost of any linear decision tree that solves point location for query distribution D. They show three things: how to create a Steiner triangulation that nearly minimizes entropy over all triangulations of G, that the minimum entropy over triangulations is a lower bound for $\hat{H}$ (meaning that the increased entropy may be necessary), and that the entropy-preserving cuttings give a query structure that matches the leading term of the lower bound.

Entropy-bounded query structures are also used in two interesting applications that do not assume that the query distribution is known in advance.

Aronov et al. [ABE$^+$16] use them to give a ***distance-sensitive*** query algorithm that is faster for points far from the boundary. They show how to decompose a unit area subdivision into pieces of constant complexity such that any point q within distance Δ_q of the boundary is in a piece of area $\Omega(\Delta_p^2)$. (They construct a triangulation with this property for any convex polygon, and a decomposition into 7-gons for simple polygons.) Entropy bounds for a query in this decomposition give query time $O(\min\{\log n, -\log(\Delta_q)\})$.

Iacono and Mulzer [IM12] assume that regions have constant complexity and demonstrate how to prove ***static optimality***: They show that in the limit they answer queries in asymptotically the same time as the best (static) decision tree by simply rebuilding, after every n^α queries, an entropy-bounded query structure for the n^β regions that have been most frequently accessed. Cheng and Lau [CL15] show that the analysis can extend to convex subdivisions, at an additional $O(\log \log n)$ time per query, by simply using balanced hierarchical triangulations of each convex region.

PLANAR SEPARATOR THEOREM

The first optimal point location scheme was based on Lipton and Tarjan's ***planar separator theorem*** [LT80] that every planar graph of n nodes has a set of $O(\sqrt{n})$ nodes that partition it into roughly equal pieces. Goodrich [Goo95] gave a linear-time construction of a family of planar separators in his parallel triangulation algorithm. The fact that embedded graphs have small separators continues to be important in theoretical work.

When applied to the dual graph of a planar subdivision, the nodes are a small set of faces that partition the remainder of the faces: simple methods taking up to quadratic space can be used to determine which set of the partition needs to be searched recursively. Bose et al. [BCH$^+$12] combine separators with encodings of triangulations as permutations of points and bit-vector operations to build $o(n)$-size indices for point location in triangulations. (The bit-vectors are assumed to support rank and select operations, so their work implicitly assumes a RAM or cell-probe model of computation.) Their "succinct geometric indices" can be used to achieve the asymptotic bounds on the minimum number of comparisons, minimum entropy bounds, or $O(\log n)$-time query bounds for an implicit data structure that stores only a permutation of the input points.

38.4 PLANAR POINT LOCATION: DYNAMIC

In dynamic planar point location, the subdivision can be updated by adding or deleting vertices and edges. Unlike the static case, algorithms that match the performance of one-dimensional point location have not been found. (Except in special cases, like rectilinear subdivisions [GK09].) Like the static case, the search has produced interesting combinations of data structure ideas.

GLOSSARY

Updates: A dynamic planar subdivision is most commonly updated by inserting or deleting a vertex or edge. Update time usually refers to the worst-case time for a single insertion or deletion. Some methods support insertion or deletion of a chain of k vertices and edges faster than doing k individual updates.

Vertex expansion/contraction: Updating a planar subdivision by splitting a vertex into two vertices joined by an edge, or the inverse: contracting an edge and merging the two endpoints into one. This operation, supported by the "primal/dual spanning tree" (discussed below), is important for point location in three-dimensional subdivisions.

Amortized update time: When times are reported as amortized, then an individual operation may be expensive, but the total time for k operations, starting from an empty data structure, will take at most k times the amortized bound.

TABLE 38.4.1 Dynamic point location results.

TECHNIQUE	QUERY	UPDATE	SPACE	UPDATES SUPPORTED
Trapezoid method [CT92]	$O(\log n)$	$O(\log^2 n)$	$O(n \log n)$	ins/del vertex & edge
Separating chain [PT89]	$O(\log^2 n)$	$O(\log^2 n)$	$O(n)$	ins/del edge & edge
I/O-efficient [ABR12]	$O(\log_B^2 N)$	$O(\log_B N)$	$O(N/B)$	measures I/O blocks read
Pr/dual span tree [GT91]	$O(\log^2 n)$	$O(\log n)$	$O(n)$	ins/del edge & chain,
amortized	$O(\log n \log\log n)$	$O(1)$	$O(n)$	expand/contract vertex
Interval tree [CJ92]	$O(\log^2 n)$	$O(\log n)$	$O(n)$	ins/del edge & chain
with frac casc [BJM94]	$O(\log n \log\log n)$	$O(\log^2 n)$	$O(n)$	amort del, ins faster
Segment tree [CN15]	$O(\log n (\log\log n)^2)$	$O(\log n \log\log n)$	$O(n)$	many variants

Insertion/Deletion-only: When all updates are insertions or all are deletions, specialized structures can often be more efficient. Note that deletion-only structures for a decomposable problem support dynamic updates by this Bentley-Saxe transformation [BS80]: Maintain structures whose sizes are bounded by a geometric series, where only the smallest need support insertion. Whenever an update would make ith structure too large or small, rebuilding all structures through the $(i+1)$st. For k updates, an element participates in amortized $O(\log k)$ rebuilds.

Vertical ray shooting problem: Maintain a set of interior-disjoint segments in a structure that can report the segment directly below a query point q. Updates

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry,
J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

are insertion or deletion of segments. This decomposable problem does not require subdivisions to remain connected, but also does not maintain identity of faces.

I/O efficient algorithm: An algorithm whose asymptotic number of I/O operations is minimal. Model parameters are problem size N, disk block size B and memory size M, with typically $B \leq \sqrt{M}$. Sorting requires $O((N/B)\log_B N)$ time.

SEPARATING CHAIN AND TRAPEZOID GRAPH METHODS

The separating chain method of Section 38.3 was the first to be made fully dynamic [PT89]. Although both its asymptotics and its constant factors are larger than other methods, it has been made I/O-efficient [AV04]. This is an impressive theoretical accomplishment, but simpler algorithms that assume that the input is somewhat evenly distributed in the plane will be more practical.

Preparata's trapezoid graph method [Pre81] is one of the easiest to make dynamic. It preserves its optimal $O(\log n)$ query time, but also its suboptimal $O(n \log n)$ space. To support updates in $O(\log^2 n)$ time, Chiang and Tamassia [CT92, CPT96] store the binary tree on subdivision edges in a *link-cut* tree [ST83], which supports in $O(\log n)$ time the operation of linking two trees by adding an arc, and the inverse, cutting an arc to make two trees.

PRIMAL/DUAL SPANNING TREE

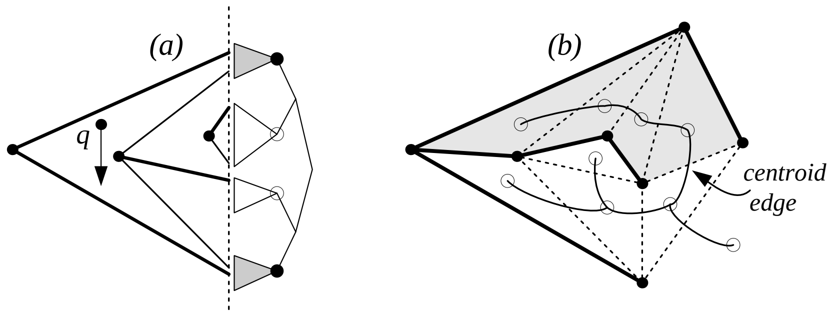
Goodrich and Tamassia [GT98] gave an elegant approach based on link-cut trees that takes linear space for the restricted case of dynamic point location in monotone subdivisions. A monotone subdivision has a ***monotone spanning tree*** in which all root-to-leaf paths are monotone. Each edge not in the tree closes a cycle and defines a monotone polygon.

In any planar graph whose faces are simple polygons, the duals of edges not in the spanning tree form a dual spanning tree of faces, as in Figure 38.4.1(b). Goodrich and Tamassia [GT98] use a centroid decomposition of the dual tree to guide comparisons with monotone polygons in the primal tree. The centroid edge, which breaks the dual tree into two nearly-equal pieces, is indicated in Figure 38.4.1(b). The primal edge creates the shaded monotone polygon; if the query is inside then we recursively explore the corresponding piece of the dual tree. Using link-cut trees, the centroid decomposition can be maintained in logarithmic time per update, giving a dynamic point-location structure with $O(\log^2 n)$ query time.

In the static setting, fractional cascading can turn this into an optimal point location method. Dynamic fractional cascading [MN90] can be used to reduce the dynamic query time and to obtain $O(1)$ amortized update time.

The dual nature of the structure supports insertion and deletion of dual edges, which correspond to expansion and contraction of vertices. These are needed to support static three-dimensional point location via persistence. Furthermore, a k-vertex monotone chain can be inserted/deleted in $O(\log n + k)$ time.

FIGURE 38.4.1
Dynamic methods: (a) *Priority search (interval tree);* (b) *primal/dual spanning tree.*



DYNAMIC INTERVAL OR SEGMENT TREES

The current best results solve the vertical ray shooting problem in a dynamic set of disjoint segments using interval or segment trees to store slabs. Let's consider the classic interval tree of Cheng and Janardan [CJ92], and the recent work of Chan and Nekrich [CN15], which presents many variants that reduce space and trade query and update times in a segment tree.

The key subproblem in both is vertical ray shooting for 1-sided segments: For a set of interior-disjoint segments S that intersect a common vertical line ℓ, report the segment of S directly below a query point q. Let's assume q is left of ℓ; we can make a separate structure for the right.

Cheng and Janardan [CJ92] solve this subproblem in linear space and $O(\log |S|)$ query and update time by a priority-tree search: they build a binary search tree on segments S in the vertical order along ℓ, and store in each subtree a pointer to the "priority segment" with endpoint farthest left of ℓ. At each level of this search tree, only two candidate subtrees may contain the segment below q—the ones whose priority segments are immediately above and below q. Figure 38.4.1(a) illustrates a case in which the search continues in the two shaded subtrees.

In their work, this subproblem arises naturally in a recursively defined ***interval tree***: The root stores segments that cross a vertical line ℓ through the median endpoint. Segments to the left (right) are stored in an interval tree that is the left (right) child of the root, with the corresponding ray-shooting structure. Space is $O(n)$, since each segment is stored once.

To locate a query point q, visit the $O(\log n)$ nodes on the path to the slab containing q, and return the closest of the segments found by ray shooting for 1-sided segments in each node. Total query time is $O(\log^2 n)$. Constants are moderate, with only 4 or 5 entries per edge and 6 comparisons per search step. Updates to the priority search tree take $O(\log n)$ time with larger constants; they must maintain tree balance and segment priorities. To handle changes in the number of slabs, use a BB[α] or weight-balanced B-tree [AV03] and rebuild the affected priority search tree structures in linear time when nodes split. This makes the update cost amortized $O(\log n)$.

To speed up the query time, one would like work done in one 1-sided subproblem to make the rest easier. The fractional cascading idea of sharing samples of segments between subproblems requires that pairs of segments can be locally ordered, but segments in different nodes of an interval tree may share no x-coordinates. (If all segments share the same slope, they can be ordered by y-intercept to enable dynamic fractional cascading [MN90]; more on this below.)

Several researchers [ABG06, BJM94, Ble08, CN15, GK09] have used ideas from *segment trees*, which store segments in the balanced slab recursively as follows: Starting from the root, store any segment that crosses the slab for that node (the union of the leaf slabs in its subtree). Pass unstored segments to the children whose slabs they intersect; segments that straddle the median are sent to both children. A segment is stored in at most $2\log_2 n$ nodes, since to be stored, an endpoint must be in the parent's slab. Thus, we have a structure with $O(n \log n)$ space that can perform updates and queries in $O(\log^2 n)$ time apiece. Thus, there are extra logs on space, query, and update.

Baumgarten et al. [BJM94] observed that segments stored in nodes whose slabs contain query point q all intersect a vertical line through q, so dynamic fractional cascading [MN90] from the bottom can reduce query and insertion time to $O(\log n \log\log n)$. They create a linear space data structure with this query time by combining segment and interval trees, using fractional cascading on blocks of $O(\log^2 n)$ segments in each interval tree node. Deletion time remains $O(\log^2 n)$.

Chan and Nekrich [CN15] carefully combine many ideas that come closest to removing all three logs. First, they point out that a deletion-only structure for horizontal segments can reduce space for any dynamic point location structure, including the trapezoid graph. They maintain a dynamic structure with up to $n/\log n$ segments, then use the Bentley-Saxe transformation [BS80] to put the rest into $O(\log\log n)$ groups whose size limits double. For each group they 1) build a static point location structure on the segments, 2) rank each segment in a total order consistent with aboveness, and 3) maintain a deletion-only structure for horizontal segments made by replacing each segment's original y coordinates with its rank. A query q in each static structure either returns the segment below, or its rank r if it has been deleted. The horizontal structure queried with (q_x, r) then returns the candidate segment from that group.

To reduce the query time, they use dynamic fractional cascading like Baumgarten et al.: using ideas from the segment tree to pass samples to speed up searches in the interval tree. (They describe the details using random sampling and finger search trees to more easily consider dynamic updates for several variants.) They trade query time for update time by coloring each segment's $O(\log n)$ fragments with $O(\log\log n)$ colors and building separate fractional cascading for each color. The colors for a segment are determined by the levels crossed by subtree of slabs spanned by the segment in a manner like the tree interpretation of van Emde Boas queues, which allows an inserted or deleted segment to be found and updated in $O(\log n \log\log n)$ time in all its colored lists. Query time increases to $O(\log n (\log\log n)^2)$ because of the extra lists that must be searched. Their work suggests other variations that can trade query and update times, so one is $O(\log n)$ while the other is $O(\log^{1+\varepsilon} n)$, or can use word RAM tricks to shave a factor of $\log\log n$ from the query.

As mentioned above, the special case of horizontal segments is easier, as the y-order can be used in fractional cascading. Giyora and Kaplan [GK09] achieve a linear space structures with $O(\log^{1+\varepsilon} n)$ query and $O(\log n)$ update on a pointer machine and $O(\log n)$ query and update times on a word RAM.

OPEN PROBLEMS

1. Improve dynamic planar point location to simultaneously attain $O(n)$ space and $O(\log n)$ query and update time, or establish a lower bound.
2. Can persistent data structures be made dynamic? The fact that data are copied seems to work against maintaining a data structure under insertions and deletions.
3. Create a dynamic data structure for subdivisions that need not remain connected (may have holes) that can report in sublinear time whether two points are in the same face.

38.5 PLANAR POINT LOCATION: OTHER MODELS

Programming complexity and non-negligible asymptotic constants mean that optimal point location techniques are used less than might be expected. See [TV01] for a study of geometric algorithm engineering that uses point location schemes as its example.

PICK HARDWARE

Graphic workstations employ special "pick hardware" that draws objects on the screen and returns a list of objects that intersect a query pixel. The hardware imposes a minimum time of about 1/30th of a second on a pick operation, but hundreds of thousands of polygons may be considered in this time.

BUCKETING AND SPATIAL INDEX STRUCTURES

Because data in practical applications tend to be evenly distributed, bucketing techniques are far more effective [AEI$^+$85, EKA84] than worst-case analysis would predict. For problems in two and three dimensions, a uniform grid will often trim data to a manageable size [MAFL16].

Adaptive data structures for more general spatial indexing, such as k-d trees, quadtrees, BANG files, R-trees, and their relatives [Sam90], can be used as filters for point location—these techniques are common in databases and geographic information systems.

Various definitions for "fat regions" have used to explore theoretical bounds on schemes that use spatial indexing structures. To give one example, Löffler, Simons, and Strash [LSS13] use dynamic quadtrees to store a representative points near the middle of each region, ensuring that cells for large regions are large and that a query point will have to do efficient point-in-region tests for only a constant number of regions. Thus, for disjoint fat regions, they achieve $O(\log n)$-time insert, delete, and query operations. They also can perform $O(\log\log n)$-time "local updates," which replace a region by another of similar diameter and separation.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry,
J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

Chan and Pătraşcu [CP09] combine sampling and bucketing ideas in their ***transdichotomous*** structures for point location in a slab. Given a slab with an ordered list of m crossing segments whose left and right y-coordinates are $O(w)$-bit rationals that lie in intervals of length L and R, they select b evenly spaced segments from the list and partition the endpoints into h equal length buckets on left and right. Iterate to select segments separating buckets: find the highest segment from the first non-empty bucket on the left, round up the coordinates of both ends, discard all segments with an endpoint below, and repeat.

Knowing the location of query point q among the selected segments, two comparisons of q with original segments allows a recursive query in either a set of m/b segments, or segments with left interval length L/h, or segments with right interval length R/h. Shrinking the intervals is progress because small y-coordinate offsets can be packed into words for parallel evaluation in the real RAM model. Thus, they can build, in $o(n \log n)$ time, an $O(n)$-space structure to answer point location in a slab queries in $O(\log n / \log \log n)$ time. They combine this with the point location techniques of Section 38.3 to give point location within the same bounds. They get even better bounds for off-line point location [CP10], where the queries are known in advance, by packing query points into words. They use these techniques to give transdichotomous bounds for many computational geometry problems.

SUBDIVISION WALKING

Applications that store planar subdivisions with their adjacency relations, such as geographic information systems, can walk through the regions of the subdivision from a known position p to the query q.

To walk a subdivision with $O(n)$ edges, compute the intersections of $\overline{pq}$ with the current region and determine if q is inside. If not, let q' denote the intersection point closest to q. Advance to the region incident to q' that contains a point in the interior of $\overline{q'q}$ and repeat. In the worst case, this walk takes $O(n)$ time. The application literature typically claims $O(\sqrt{n})$ time, which is the average number of intersections with a line under the assumption that vertices and edges of the subdivision are evenly distributed. When combined with bucketing or hierarchical data structures (for example, maintaining a regular grid or quadtree with known positions and starting from the closest to answer a query), walking is an effective, practical location method.

For triangulations, the algorithm walking $\overline{pq}$ is easy to implement. Guibas and Stolfi's [GS85] incremental Delaunay triangulation uses an even simpler walk from edge to edge, but this depends on an acyclicity theorem (Sections 19.4 and 26.1) that does not hold for arbitrary triangulations. A robust walk should remember its starting point and handle vertices on the traversed segment as if they had been perturbed consistently. Broutin, Devillers, and Hemsley prove nice bounds for their "cone walk" in random Delaunay triangulations [BDH16]

There have been several analyses of Jump & Walk schemes in triangulations, both analytically and experimentally. Devroye et al. [DLM04] show expected query times of $O(n^{1/4})$ for a scheme that keeps $n^{1/4}$ points with known locations, and walks from the nearest to find a query. In their experiments, the combination of a 2-d search tree with walking performed the best. De Castro and Devillers [CD13] survey, compare, and tune many variations, including those that save space by building a hierarchy formed from small samples (a technique implemented in the

CGAL library [BDP$^+$02, Dev02]) and those that are distribution sensitive by dynamically choosing points to keep. See also their Java Demo [DC11].

38.6 LOCATION IN HIGHER DIMENSIONS

In higher dimensions, known point location methods do not achieve both linear space and logarithmic query time. Linear space can be attained by relatively straightforward linear search, such as the point-in-polygon test.

Logarithmic time, or $O(d \log n)$ time, can be obtained by projection [DL76]: project the $(d-2)$-faces of a subdivision to an arrangement in $d-1$ dimensions and recursively build a point location structure for the arrangement in the projection. Knowing the cell in the projection gives a list of the possible faces that project to that cell, so an additional logarithmic search can return the answer. The worst-case space required is $O(n^{2^d})$.

Because point location is decomposable, batching can trade space for time: preprocessing n/k groups of k facets into structures with $S(k)$ space and $Q(k)$ time gives, in total, $O(nS(k)/k)$ space and $O(nQ(k)/k)$ query time.

Clever ways of batching can lead to better structures. Randomized methods can often reduce the dependence on dimension from doubly- to singly-exponential, since random samples can be good approximations to a set of geometric objects. They can also be used with objects that are implicitly defined.

We should mention that convex polyhedra can be preprocessed using the Dobkin-Kirkpatrick hierarchy (Section 38.3) so that the point-in-convex-polyhedron test does take $O(n)$ space and $O(\log n)$ query time.

THREE-DIMENSIONAL POINT LOCATION

Dynamic location structures can be used for static spatial point location in one higher dimension by employing persistence. If one swept a plane through a subdivision of three-space into polyhedra, one could see the intersection as a dynamic planar subdivision in which vertices (intersections of the sweep plane with edges) move along linear trajectories. Whenever the sweep plane passes through a vertex in space, vertices in the plane may join and split.

Goodrich and Tamassia's primal/dual method supports the necessary operations to maintain a point location structure for the sweeping plane. Using node-copying to make the structures persistent gives an $O(n \log n)$ space structure that can answer queries in $O(\log^2 n)$ time. Preprocessing takes $O(n \log n)$ time.

Devillers et al. [DPT02] tested several approaches to subdivision walking for Delaunay tetrahedralization, and established the practical effectiveness of the hierarchical Delaunay in three dimensions as well.

RECTILINEAR SUBDIVISIONS

Restricting attention to rectilinear (orthogonal) subdivisions permits better results via data structures for orthogonal range search. The ***skewer tree***, a multidimensional interval tree, gives static point location among n rectangular prisms with

$O(n)$ space and $O(\log^{d-1} n)$ query time after $O(n \log n)$ preprocessing [EHH86]. These can be made dynamic by using Giyora and Kaplan's [GK09] structure at the lowest level.

In dimensions two and three, stratified trees and perfect hashing [DKM+94] can be used to obtain $O((\log\log U)^{d-1})$ query time in a fixed universe $[1, \ldots, U]$, or $O(\log n)$ query time in general. Iacono and Langerman [IL00] use "justified hyperrectangles" to obtain $O(\log\log U)$ query times in every dimension d, but the space and preprocessing time, which are $O(fn \log\log U)$ and $O(fn \log U \log\log U)$, respectively, depend on a *fatness parameter* f that equals the average ratio of the dth power of smallest dimension to volume of all hyperrectangles in the subdivision.

POINT LOCATION AMONG ALGEBRAIC VARIETIES

Chazelle and Sharir [CS90] consider point location in a general setting, among n algebraic varieties of constant maximum degree b in d-dimensional Euclidean space. They augment Collins's cylindrical algebraic decomposition to obtain an $O(n^{2^{d-1}})$-space, $O(\log n)$-query time structure after $O(n^{2^{d+6}})$ preprocessing. Hidden constants depend on the degrees of projections and intersections, which can be b^{4^d}.

This method provides a general technique to obtain subquadratic solutions to optimization problems that minimize a function $\{F(a,b) \mid a \in A, b \in B\}$, where $F(a,b)$ has a constant-size algebraic description. For a fixed b, F is algebraic in a. Thus, small batches of points from B can be preprocessed in subquadratic time, and each a can be tested against each batch, again in subquadratic time.

OPEN PROBLEMS

1. Find an optimal method for static (or dynamic) point location in a three-dimensional subdivision with n vertices and $O(n)$ faces: $O(n)$ space and $O(\log n)$ query time.

RANDOMIZED POINT LOCATION

The techniques of Chapter 44 can lead to good point location methods when a random sample of a set of objects can be used to approximate the whole. Arrangements of hyperplanes in dimension d are a good example. A random sample of hyperplanes divides space into cells intersected by few hyperplanes; recursively sampling in each cell gives a point location structure for the arrangement. Table 38.6.1 lists the performance of some randomized point location methods for hyperplanes. Query time can be traded for space by choosing larger random samples.

The randomized incremental construction algorithms of Section 44.2 are simple because they naturally build randomized point location structures along with the objects that they aim to construct [Mul93, Sei93]. These have good "tail bounds" and work well as insertion-only location structures.

Randomized point location structures can be made fully dynamic by lazy deletion and randomized rebuild techniques [BDS95, MS91]; they maintain good ex-

TABLE 38.6.1 Randomized point location in arrangements.

TECHNIQUE	OBJECTS	QUERY	PREPROC	SPACE
Random sample [Cla87]	hyperplanes	$O(c^d \log n)$ exp	$O(n^{d+1+\epsilon})$ exp	$O(n^{d+\epsilon})$
Derandomized [CF94]	hyperplanes	$O(c^d \log n)$	$O(n^{2d+1})$	$O(n^d)$
Random sample [MS91]	dyn hpl $d \leq 4$	$O(\log n)$ exp	$O(n^{d+\epsilon})$ exp	$O(n^{d+\epsilon})$
Epsilon nets [Mei93]	hyperplanes	$O(d^5 \log n)$ exp	$O(n^{d+1+\epsilon})$ exp	$O(n^{d+\epsilon})$

pected performance if random elements are chosen for insertion and deletion. That is, the sequence of insertions and deletions may be specified, but the elements are to be chosen independently of their roles in the data structure.

IMPLICIT POINT LOCATION

In some applications of point location, the objects are not given explicitly. A planar motion planning problem may ask whether a start and a goal point are in the same cell of an arrangement of constraint segments or curves, without having explicit representations of all cells.

Consider a simple example: an arrangement of n lines, which defines nearly n^2 bounded cells. Without storing all cells, we can determine whether two points p and q are in the same cell by preprocessing $\sqrt{n}$ subarrangements of $\sqrt{n}$ lines ($O(n\sqrt{n})$ cells in all) and making sure that p and q are together in each subarrangement. If the lines are put into batches by slope, then within the same asymptotic time, an algorithm can return the pair of lines defining the lowest vertex as a unique cell name.

Implicit location methods are often seen as special cases of range queries (Chapter 40) or vertical ray shooting [Aga91]. Table 38.6.2 lists results on implicit location among line segments, which depend upon tools discussed in Chapters 40, 44, and 47, specifically random sampling, ϵ-net theory, and spanning trees with low stabbing number.

TABLE 38.6.2 Implicit point location results for arrangements of n line segments.

TECHNIQUE	QUERY	PREPROC	SPACE
Span tree lsn [Aga92]	$O(\sqrt{n}\log^2 n)$	$O(n^{3/2}\log^\omega n)$	$O(n\log^2 n)$
Batch sp tree [AK94]	$O\big((n/\sqrt{s})\log^2(n/\sqrt{s}) + \log n\big)$	$O\Big((sn(\log(n/\sqrt{s})+1)^{2/3}\Big)$	$n\sqrt{\log n} \leq s \leq n^2$

38.7 SOURCES AND RELATED MATERIAL

SURVEYS

Graphic Gems IV has code for point in polygon algorithms. These recent papers have good overviews of the literature or present variations of ideas on their topics.

[Hai94, Wei94]: Point-in-polygon algorithms in *Graphics Gems IV*, with code.

[IM12]: Nice history of entropy bounds for point location.

[CP09]: Speeding up point location speeds up many geometric algorithms.

[CN15]: Variations for dynamic point location.

RELATED CHAPTERS

Chapter 28: Arrangements
Chapter 29: Triangulations and mesh generation
Chapter 30: Polygons
Chapter 40: Range searching
Chapter 41: Ray shooting and lines in space
Chapter 44: Randomization and derandomization
Chapter 46: Parallel algorithms in geometry
Chapter 47: Epsilon-approximations and epsilon-nets
Chapter 52: Computer graphics

REFERENCES

[ABE+16] B. Aronov, M. de Berg, D. Eppstein, M. Roeloffzen, and B. Speckmann. Distance-sensitive planar point location. *Comput. Geom.*, 54:17–31, 2016.

[ABG06] L. Arge, G.S. Brodal, and L. Georgiadis. Improved dynamic planar point location. In *Proc. 47th IEEE Sympos. Found. Comp. Sci.*, pages 305–314, 2006.

[ABR12] L. Arge, G.S. Brodal, and S.S. Rao. External memory planar point location with logarithmic updates. *Algorithmica*, 63:457–475, 2012.

[AEI+85] Ta. Asano, M. Edahiro, H. Imai, M. Iri, and K. Murota. Practical use of bucketing techniques in computational geometry. In G.T. Toussaint, editor, *Computational Geometry*, pages 153–195, North-Holland, Amsterdam, 1985.

[Aga91] P.K. Agarwal. Geometric partitioning and its applications. In J.E. Goodman, R. Pollack, and W. Steiger, editors, *Computational Geometry*, vol. 6 of *DIMACS Ser. Discrete Math. Theor. Comp. Sci.*, AMS, Providence, 1991.

[Aga92] P.K. Agarwal. Ray shooting and other applications of spanning trees with low stabbing number. *SIAM J. Comput.*, 21:540–570, 1992.

[AK94] P.K. Agarwal and M. van Kreveld. Implicit point location in arrangements of line segments, with an application to motion planning. *Internat. J. Comput. Geom. Appl.*, 4:369–383, 1994.

[AMM07] S. Arya, T. Malamatos, and D.M. Mount. A simple entropy-based algorithm for planar point location. *ACM Trans. Algorithms*, 3:17, 2007.

[AMMW07] S. Arya, T. Malamatos, D.M. Mount, and K.C. Wong. Optimal expected-case planar point location. *SIAM J. Comput.*, 37:584–610, 2007.

[AV03] L. Arge and J.S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32:1488–1508, 2003.

[AV04] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Comput. Geom.*, 29:147–162, 2004.

[BCH+12] P. Bose, E.Y. Chen, M. He, A. Maheshwari, and P. Morin. Succinct geometric indexes supporting point location queries. *ACM Trans. Algorithms*, 8:10, 2012.

[BDH16] N. Broutin, O. Devillers, and R. Hemsley. Efficiently navigating a random Delaunay triangulation. *Random Structures Algorithms*, 49:95–136, 2016.

[BDP+02] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Comput. Geom.*, 22:5–19, 2002.

[BDS95] M. de Berg, K. Dobrindt, and O. Schwarzkopf. On lazy randomized incremental construction. *Discrete Comput. Geom.*, 14:261–286, 1995.

[Ben79] J.L. Bentley. Decomposable searching problems. *Inform. Process. Lett.*, 8:244–251, 1979.

[BJM94] N. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *J. Algorithms*, 17:342–380, 1994.

[Ble08] G.E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proc. ACM-SIAM Sympos. Discrete Algorithms*, pages 894–903, 2008.

[BS80] J.L. Bentley and J.B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *J. Algorithms*, 1:301–358, 1980.

[CD13] P.M.M. de Castro and O. Devillers. Practical distribution-sensitive point location in triangulations. *Comput. Aided Geom. Design*, 30:431–450, 2013.

[CDI+12] S. Collette, V. Dujmović, J. Iacono, S. Langerman, and P. Morin. Entropy, triangulation, and point location in planar subdivisions. *ACM Trans. Algorithms*, 8:29, 2012.

[CF94] B. Chazelle and J. Friedman. Point location among hyperplanes and unidirectional ray-shooting. *Comput. Geom.*, 4:53–62, 1994.

[CG86] B. Chazelle and L.J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[CJ92] S.-W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21:972–999, 1992.

[CL15] S.-W. Cheng and M.-K. Lau. Adaptive point location in planar convex subdivisions. In *Proc. 26th Int. Sympos. Algorithms Comput.*, vol. 9472 of *LNCS*, pages 14–22, Springer, Berlin, 2015.

[Cla87] K.L. Clarkson. New applications of random sampling in computational geometry. *Discrete Comput. Geom.*, 2:195–222, 1987.

[CN15] T.M. Chan and Y. Nekrich. Towards an optimal method for dynamic planar point location. In *Proc 56th IEEE Sympos. Found. Comp. Sci.*, pages 390–409, 2015.

[CP09] T.M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. *SIAM J. Comput.*, 39:703–729, 2009.

[CP10] T.M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, II: Offline search. Preprint, arXiv:1010.1948, 2010.

[CPT96] Y.-J. Chiang, F.P. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. *SIAM J. Comput.*, 25:207–233, 1996.

[CS90] B. Chazelle and M. Sharir. An algorithm for generalized point location and its application. *J. Symbolic Comput.*, 10:281–309, 1990.

[CT92] Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Internat. J. Comput. Geom. Appl.*, 2:311–333, 1992.

[DC11] O. Devillers and P.M.M. de Castro. A pedagogic JavaScript program for point location strategies. In *Proc. 27th Sympos. Comput. Geom.*, pages 295–296, ACM Press, 2011.

[Dev02] O. Devillers. The Delaunay hierarchy. *Int. J. Found. Comp. Sci.*, 13:163–180, 2002.

[DKM+94] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, 23:738–761, 1994.

[DL76] D.P. Dobkin and R.J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.

[DLM04] L. Devroye, C. Lemaire, and J.-M. Moreau. Expected time analysis for Delaunay point location. *Comput. Geom.*, 29:61–89, 2004.

[DPT02] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. *Int. J. Found. Comp. Sci.*, 13.:181–199, 2002.

[EBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[EGS86] H. Edelsbrunner, L.J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[EHH86] H. Edelsbrunner, G. Haring, and D. Hilbert. Rectangular point location in d dimensions with applications. *Comput. J.*, 29:76–82, 1986.

[EKA84] M. Edahiro, I. Kokubo, and Ta. Asano. A new point-location algorithm and its practical efficiency: Comparison with existing algorithms. *ACM Trans. Graph.*, 3:86–109, 1984.

[FW93] M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993.

[Goo95] M.T. Goodrich. Planar separators and parallel polygon triangulation. *J. Comput. Syst. Sci.*, 51:374–389, 1995.

[GK09] Y. Giora and H. Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Trans. Algorithms*, 5:28:1–28:51, 2009.

[GS85] L.J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[GT91] M.T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[GT98] M.T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. *SIAM J. Comput.*, 28:612–636, 1998.

[Hai94] E. Haines. Point in polygon strategies. In P. Heckbert, editor, *Graphics Gems IV*, pages 24–46, Academic Press, Boston, 1994.

[HKH16] M. Hemmer, M. Kleinbort, and D. Halperin. Optimal randomized incremental construction for guaranteed logarithmic planar point location. *Comput. Geom.*, 58:110–123, 2016.

[IL00] J. Iacono and S. Langerman. Dynamic point location in fat hyperrectangles with integer coordinates. In *Proc. 12th Canad. Conf. Comput. Geom.*, pages 181–186, 2000.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

[IM12] J. Iacono and W. Mulzer. A static optimality transformation with applications to planar point location. *Internat. J. Comput. Geom. Appl.*, 22:327–340, 2012.

[Kir83] D.G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[LP77] D.T. Lee and F.P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[LSS13] M. Löffler, J.A. Simons, and D. Strash. Dynamic planar point location with sub-logarithmic local updates. In *Proc. 13th Sympos. Algorithms Data Structures*, pages 499–511, vol. 8037 of *LNCS*, Springer, Berlin, 2013.

[LT80] R.J. Lipton and R.E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.

[MAFL16] S.V.G. Magalhães, M.V.A. Andrade, W.R. Franklin, and W. Li. PinMesh-fast and exact 3D point location queries using a uniform grid. *Computers and Graphics (Pergamon)*, 58:1–11, 2016.

[Meh77] K. Mehlhorn. Best possible bounds on the weighted path length of optimum binary search trees. *SIAM J. Comput.*, 6:235–239, 1977.

[Mei93] S. Meiser. Point location in arrangements of hyperplanes. *Inform. Comput.*, 106:286–303, 1993.

[MN90] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[MS91] K. Mulmuley and S. Sen. Dynamic point location in arrangements of hyperplanes. In *Proc. 7th Sympos. Comput. Geom.*, pages 132–141, ACM Press, 1991.

[Mul90] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10:253–280, 1990.

[Mul93] K. Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, 1993.

[Pre81] F.P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10:473–482, 1981.

[PT89] F.P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM J. Comput.*, 18:811–830, 1989.

[Pug90] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, 1990.

[SA96] R. Seidel and C.R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[SA00] R. Seidel and U. Adamy. On the exact worst case query complexity of planar point location. *J. Algorithms*, 37:189–217, 2000.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, 1990.

[Sch08] S. Schirra. How reliable are practical point-in-polygon strategies? In *Proc. 16th European Sympos. Algorithms*, vol. 5193 of *LNCS*, pages 744–755, Springer, Berlin, 2008.

[Sei91] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom.*, 1:51–64, 1991.

[Sei93] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, vol. 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, New York, 1993.

[SK97] J. Snoeyink and M. van Kreveld. Linear-time reconstruction of Delaunay triangulations with applications. In *Proc. 5th European Sympos. Algorithms*, vol. 1284 of *LNCS*, pages 459–471, Springer, Berlin, 1997.

[ST83] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26:362–381, 1983.

[ST85] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32:652–686, 1985.

[ST86] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[Ste91] A.J. Stewart. Robust point location in approximate polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 179–182, 1991.

[TV01] R. Tamassia and L. Vismara. A case study in algorithm engineering for geometric computing. *Internat. J. Comput. Geom. Appl.*, 11:15–70, 2001.

[Wei94] K. Weiler. An incremental angle point in polygon test. In P. Heckbert, editor, *Graphics Gems IV*, pages 16–23, Academic Press, Boston, 1994.