Dynamic Distribution-Sensitive Point Location

Siu-Wing Cheng
Department of Computer Science and Engineering, HKUST, Hong Kong, China
scheng@cse.ust.hk

Man-Kit Lau
Department of Computer Science and Engineering, HKUST, Hong Kong, China
lmkaa@connect.ust.hk

Abstract

We propose a dynamic data structure for the distribution-sensitive point location problem. Suppose that there is a fixed query distribution in $\mathbb{R}^2$, and we are given an oracle that can return in $O(1)$ time the probability of a query point falling into a polygonal region of constant complexity. We can maintain a convex subdivision $\mathcal{S}$ with n vertices such that each query is answered in $O(\mathrm{OPT})$ expected time, where OPT is the minimum expected time of the best linear decision tree for point location in $\mathcal{S}$. The space and construction time are $O(n \log^2 n)$. An update of $\mathcal{S}$ as a mixed sequence of k edge insertions and deletions takes $O(k \log^5 n)$ amortized time. As a corollary, the randomized incremental construction of the Voronoi diagram of n sites can be performed in $O(n \log^5 n)$ expected time so that, during the incremental construction, a nearest neighbor query at any time can be answered optimally with respect to the intermediate Voronoi diagram at that time.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases dynamic planar point location, convex subdivision, linear decision tree

Digital Object Identifier 10.4230/LIPIcs.SoCG.2020.30

Related Version `https://arxiv.org/abs/2003.08288`

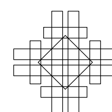
Funding Supported by Research Grants Council, Hong Kong, China (project no. 16201116).

1 Introduction

Planar point location is a classical problem in computational geometry. In the static case, a subdivision is preprocessed into a data structure so that, given a query point, the face containing it can be reported efficiently. In the dynamic case, the data structure needs to accommodate edge insertions and deletions. It is assumed that every new edge inserted does not cross any existing edge. There are well-known worst-case optimal results in the static case [1, 19, 25, 29]. There has been a long series of results in the dynamic case [3, 5, 8, 9, 14, 15, 21, 26, 27]. For a dynamic connected subdivision of n vertices, an $O(\log n)$ query time and an $O(\log^{1+\varepsilon} n)$ update time for any $\varepsilon > 0$ can be achieved [8].

When the faces have different probabilities of containing the query point, it is appropriate to minimize the expected query time. Assume that these probabilities are given or accessible via an oracle. Arya et al. [4] and Iacono [23] obtained optimal expected query time when the faces have constant complexities. Later, Collete et al. [16] obtained the same result for connected subdivisions. So did Afshani et al. [2] and Bose et al. [7] for general subdivisions.

In the case that no prior information about the queries is available, Iacono and Mulzer [24] designed a method for triangulations that can process an online query sequence σ in time proportional to n plus the entropy of σ. We developed solutions for convex and connected subdivisions in a series of work [11, 10, 12]. For convex subdivisions, the processing time is $O(T_{\mathrm{opt}} + n)$, where T_{opt} is the minimum time needed by a linear decision tree to process σ [10]. For connected subdivisions, the processing time is $O(T_{\mathrm{opt}} + n + |\sigma| \log(\log^* n))$ [12].

© Siu-Wing Cheng and Man-Kit Lau;
licensed under Creative Commons License CC-BY
36th International Symposium on Computational Geometry (SoCG 2020).
Editors: Sergio Cabello and Danny Z. Chen; Article No. 30; pp. 30:1–30:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


In this paper, we are interested in dynamic distribution-sensitive planar point location. Such a problem arises when there are online demands for servers that open and close over time, and a nearest server needs to be located for a demand. For example, walking tourists may look for a facility nearby (e.g. convenience store) and search on their mobile phones. The query distribution can be characterized using historical data. New convenience store may open and existing ones may go out of business. If we use the Euclidean metric, then we are locating a query point in a dynamic convex subdivision which is a Voronoi diagram. We are interested in solutions with optimal expected query time.

We assume that there is an oracle that can return in $O(1)$ time the probability of a query point falling inside a polygonal region of constant complexity. We propose a data structure for maintaining a convex subdivision $\mathcal{S}$ with n vertices such that each query is answered in $O(\mathrm{OPT})$ expected time, where OPT is the minimum expected time of the best *point location decision tree for* $\mathcal{S}$, i.e., the best linear decision tree for answering point location queries in $\mathcal{S}$. An update of $\mathcal{S}$ as a mixed sequence of k edge insertions and deletions can be performed in $O(k\log^5 n)$ amortized time. The space and construction time are $O(n\log^2 n)$. As a corollary, we can carry out the randomized incremental construction of the Voronoi diagram of n sites so that, during the incremental construction, a nearest neighbor query at any time can be answered optimally with respect to the intermediate Voronoi diagram at that time. The expected total construction time is $O(n\log^5 n)$ because each site insertion incurs $O(1)$ expected structural changes to the Voronoi diagram. A key ingredient in our solution is a new data structure, *slab tree*, for maintaining a triangulation with a nearly optimal expected point location time and polylogarithmic amortized update time. This data structuring technique may find other applications. Omitted proofs and details are in [13].

2 Dynamic convex subdivision

Let $\mathcal{S}$ be a convex subdivision. Let $\partial\mathcal{S}$ be the outer boundary of $\mathcal{S}$, which bounds a convex polygon. A *general-update sequence* Φ is a mixed sequence of edge insertions and deletions in $\mathcal{S}$ that produces a convex subdivision. The intermediate subdivision after each edge update is only required to be connected, not necessarily convex. Vertices may be inserted into or deleted from $\partial\mathcal{S}$, but the shape of $\partial\mathcal{S}$ is never altered. We will present in Sections 3-5 a dynamic point location structure for a DK-triangulation of $\mathcal{S}$ (to be defined below). Theorem 8 in Section 5 summarizes the performance of this data structure. We show how to apply Theorem 8 to obtain a dynamic distribution-sensitive point location structure for $\mathcal{S}$.

2.1 Dynamic DK-triangulation

Let P be a convex polygon. Find three vertices x, y and z that roughly trisect the boundary of P. This gives a triangle xyz. Next, find a vertex w that roughly bisects the chain delimited by x and y. This gives a triangle xyw adjacent to xyz. We recurse on the other chains to produce a DK-triangulation of P [17]. It has the property that any line segment inside P intersects $O(\log|P|)$ triangles. A DK-triangulation of $\mathcal{S}$ is obtained by computing the DK-triangulations of its bounded faces. Goodrich and Tamassia [20] proposed a method to maintain a *balanced geodesic triangulation* of a connected subdivision. We can use it to maintain a DK-triangulation of $\mathcal{S}$ because a DK-triangulation is a balanced geodesic triangulation. By their method, each edge insertion/deletion in $\mathcal{S}$ is transformed into $O(\log n)$ edge insertions and deletions in the DK-triangulation of $\mathcal{S}$, where n is the number of vertices of $\mathcal{S}$. Consequently, each edge insertion/deletion in $\mathcal{S}$ takes $O(\log^2 n)$ time.

2.2 Point location

We modify our adaptive point location structure for static convex subdivisions [10] to make it work for the distribution-sensitive setting. Compute a DK-triangulation Δ_1 of $\mathcal{S}$. For each triangle $t \in \Delta_1$, use the oracle to compute the probability $\Pr(t)$ of a query point falling into t. This probability is the weight of that triangle. We call the triangles in Δ_1 *non-dummy* because we will introduce some *dummy* triangles later.

Construct a data structure D_1 for Δ_1 with two parts. The first part of D_1 is a new dynamic distribution-sensitve point location structure for triangulations (Theorem 8). The query time of the first part of D_1 is $O(\mathrm{OPT}+\log\log n)$, where OPT is the minimum expected time of the best point location decision tree for Δ_1. The second part can be any dynamic point location structure with $O(\log n)$ query time, provided that its update time is $O(\log^2 n)$ and space is $O(n\log^2 n)$ [3, 8, 15, 28].

For $i \geq 2$, define $n_i = (\log_2 n_{i-1})^4$ inductively, where $n_1 = n$. To construct Δ_i from Δ_{i-1}, extract the non-dummy triangles in Δ_{i-1} whose probabilities of containing a query point are among the top $(\log_2 n_{i-1})^4$. For each subset of extracted triangles that lie inside the same bounded face of $\mathcal{S}$, compute their convex hull and its DK-triangulation. These convex hulls are holes in the polygon H_i with $\partial\mathcal{S}$ as its outer boundary. Triangulate H_i. We call the triangles used in triangulating H_i *dummy* and the triangles in the DK-triangulations of the holes of H_i *non-dummy*. The dummy and non-dummy triangles form the triangulation Δ_i. The size of Δ_i is $O(n_i)$. For each non-dummy triangle $t \in \Delta_i$, set its weight to be $\max\{\Pr(t), W_i^*/n_i\}$, where W_i^* is the sum of $\Pr(t)$ over the non-dummy triangles t in Δ_i. Dummy triangles are given weight W_i^*/n_i. The total weight W_i of all triangles in Δ_i is $\Theta(W_i^*)$. Construct D_i as the point location structure of Iacono [23] for Δ_i, which can answer a query in $O\big(\log \frac{W_i}{w_i}\big)$ time, where w_i is the weight of the triangle containing the query point. The query time of D_i is no worse than $O(\log n_i)$ in the worst case as $w_i \geq W_i^*/n_i = \Theta(W_i/n_i)$.

A hierarchy $(\Delta_1, D_1), \ldots, (\Delta_m, D_m)$ is obtained in the end, where the size of Δ_m is less than some predefined constant. So $m = O(\log^* n)$.

For $i \geq 2$, label every non-dummy triangle $t \in \Delta_i$ with the id of the bounded face of $\mathcal{S}$ that contains it. If t is located by a query, we can report the corresponding face of $\mathcal{S}$. The labelling of triangles in Δ_1 is done differently in order to allow updates in Δ_1 to be performed efficiently. For each vertex p of $\mathcal{S}$, its incident triangles in Δ_1 are divided into circularly consecutive groups by the incident edges of p in $\mathcal{S}$. Thus, each group lies in a distinct face of $\mathcal{S}$ incident to p. We store these groups in clockwise order in a biased search tree T_p [6] associated with p. Each group is labelled by the bounded face of $\mathcal{S}$ that contains it. The group weight is the maximum of $1/n$ and the total probability of a query point falling into triangles in that group. The threshold of $1/n$ prevents the group weight from being too small, allowing T_p to be updated in $O(\log n)$ time. The query time to locate a group is $O\big(\log \frac{W}{w}\big)$, where w is the weight of that group and W is the total weight in T_p. Suppose that D_1 returns a triangle $t \in \Delta_1$ incident to p. We find the group containing t which tells us the face of $\mathcal{S}$ that contains t. If p is a boundary vertex of $\mathcal{S}$, there are two edges in $\partial\mathcal{S}$ incident to p, so we can check in $O(1)$ time whether t lies in the exterior face. Otherwise, we search T_p to find the group containing t in $O\big(\log \frac{W}{w}\big) = O\big(\log \frac{1}{\Pr(t)}\big)$ time.

Given a query point q, we first query D_m with q. If a non-dummy triangle is reported by D_m, we are done. Otherwise, we query D_{m-1} and so on.

▶ **Lemma 1.** *Let $\mathcal{D} = ((\Delta_1, D_1), \ldots, (\Delta_m, D_m))$ be the data structure maintained for $\mathcal{S}$. The expected query time of $\mathcal{D}$ is $O(\mathrm{OPT})$, where OPT is the minimum expected time of the best point location decision tree for $\mathcal{S}$.*

2.3 General-update sequence

Let Φ be a general-update sequence with $k \leq n/2$ edge updates. We call k the *size* of Φ. As discussed in Section 2.1, each edge update in $\mathcal{S}$ is transformed into $O(\log n)$ edge updates in Δ_1. Updating Δ_1 takes $O(k \log^2 n)$ time. We also update the biased search tree T_p at each vertex p of $\mathcal{S}$ affected by the structural changes in Δ_1. This step also takes $O(k \log^2 n)$ time.

For $i \geq 2$, we recompute Δ_i from Δ_{i-1} and then D_i from Δ_i. By keeping the triangles of Δ_1 in a max-heap according to the triangle probabilities, which can be updated in $O(k \log^2 n)$ time, we can extract the $n_2 = \log_2^4 n$ triangles to form Δ_2 in $O(n_2 \log n_2)$ time. For $i \geq 3$, we scan Δ_{i-1} to extract the $n_i = \log_2^4 n_{i-1}$ triangles to form Δ_i in $O(n_{i-1} + n_i \log n_i)$ time. For $i \geq 2$, constructing D_i takes $O(n_i)$ time [23]. The total update time of Δ_i and D_i for $i \geq 2$ is $O\left(\sum_{i\geq 2} \log^4 n_{i-1} \log\log n_{i-1}\right)$, which telescopes to $O(\log^4 n \log\log n)$.

Consider D_1. The second part of D_1 is a dynamic point location structure that admits an edge insertion/deletion in Δ_1 in $O(\log^2 n)$ time, giving $O(k \log^3 n)$ total time. By Theorem 8 in Section 5, the update time of the first part of D_1 is $O(k \log^5 n)$ amortized.

In the biased search tree T_p's at the vertices p of $\mathcal{S}$, there are different weight thresholds of $1/n$ depending on when a threshold was computed. To keep these thresholds within a constant factor of each other, we rebuild the entire data structure periodically. Let n' be the number of vertices in the last rebuild. Let $c < 1/2$ be a constant. We rebuild when the total number of edge updates in $\mathcal{S}$ in all general-update sequences exceeds cn' since the last rebuild. Rebuilding the first part of D_1 takes $O(n \log^2 n)$ time by Theorem 8. The second part of D_1 can also be constructed in $O(n \log^2 n)$ time. This results in an extra $O(\log^2 n)$ amortized time per edge update in S.

▶ **Theorem 2.** *Suppose that there is a fixed but unknown query point distribution in $\mathbb{R}^2$, and there is an oracle that returns in $O(1)$ time the probability of a query point falling into a polygonal region of constant complexity. There exists a dynamic point location structure for maintaining a convex subdivision $\mathcal{S}$ of n vertices with the following guarantees.*

- *Any query can be answered in $O(\text{OPT})$ expected time, where OPT is the minimum expected query time of the best point location linear decision tree for $\mathcal{S}$.*
- *The data structure uses $O(n \log^2 n)$ space, and it can be constructed in $O(n \log^2 n)$ time.*
- *A general-update sequence with size $k \leq n/2$ takes $O(k \log^5 n)$ amortized time.*

3 Slab tree: fixed vertical lines

In this section, we present a static data structure for distribution-sensitive point location in a triangulation. Its dynamization will be discussed in Sections 4 and 5.

For any region $R \subset \mathbb{R}^2$, let $\Pr(R)$ denote the probability of a query point falling into R. Let Δ be a triangulation with a convex outer boundary. The vertices of Δ lie on a given set $\mathcal{L}$ of vertical lines, but some line in $\mathcal{L}$ may not pass through any vertex of Δ. For simplicity, we assume that no two vertices of Δ lie on the same vertical line at any time.

Enclose Δ with an axis-aligned bounding box B such that no vertex of Δ lies on the boundary of B. We assume that the left and right sides of B lie on the leftmost and rightmost lines in $\mathcal{L}$. Connect the highest vertex of Δ to the upper left and upper right corners of B, and then connect the lowest vertex of Δ to the lower left and lower right corners of B. This splits $B \setminus \Delta$ into two triangles and two simple polygons. The two simple polygons are triangulated using the method of Hershberger and Suri [22]. Let Δ_B denote the triangle tiling of B formed by Δ and the triangulation of $B \setminus \Delta$. Let n denote the number of triangles in Δ_B. Any line segment in $B \setminus \Delta$ intersects $O(\log n)$ triangles in Δ_B [22]. When we discuss updates in Δ later, the portion $\Delta_B \setminus \Delta$ of the tiling will not change although new vertices may be inserted into the outer boundary of Δ.

3.1 Structure definition

Let $(l_1, l_2, \cdots, l_{|\mathcal{L}|})$ be the vertical lines in $\mathcal{L}$ in left-to-right order. We build the *slab tree* $\mathcal{T}$ as follows. The root of $\mathcal{T}$ represents the slab bounded by l_1 and $l_{|\mathcal{L}|}$. The rest of $\mathcal{T}$ is recursively defined by constructing at most three children for every node v of $\mathcal{T}$.

We use *slab*(v) to denote the slab represented by v. Let $(l_i, \cdots, l_k)$ be the subsequence of lines that intersect *slab*(v). Choose $j \in [i, k)$ such that both the probabilties of a query point falling between l_i and l_j and between l_{j+1} and l_k are at most $\Pr(\mathit{slab}(v))/2$. Create the nodes v_L, v_M, and v_R as the left, middle, and right children of v, respectively, where *slab*(v_L) is bounded by l_i and l_j, *slab*(v_M) is bounded by l_j and l_{j+1}, and *slab*(v_R) is bounded by l_{j+1} and l_k. No vertex of Δ_B lies in the interior of v_M.

The recursive expansion of $\mathcal{T}$ bottoms out at a node v if v is at depth $\log_2 n$ or *slab*(v) contains no vertex of Δ_B in its interior. So the middle child of a node is always a leaf.

Every node v of $\mathcal{T}$ stores several secondary structures. A connected region $R \subset \mathbb{R}^2$ *spans* v if there is a path $\rho \subset R \cap \mathit{slab}(v)$ that intersects both bounding lines of *slab*(v). The triangulation Δ_B induces a partition of *slab*(v) into three types of regions:

- Free Gap: For all triangle t that spans v but not *parent*(v), $t \cap \mathit{slab}(v)$ is a *free gap* of v.
- Blocked Gap: Let E be the set of all edges and triangles in Δ_B that intersect *slab*(v) but do not span v. Every connected component in the intersection between *slab*(v) and the union of edges and triangles in E is a *blocked gap* of v.
- Shadow Gap: Take the union of the free gaps of all proper ancestors of v. Each connected component in the intersection between this union and *slab*(v) is a *shadow gap* of v.

The upper boundary of a blocked gap g has at most two edges, and so does the lower boundary of g. If not, there would be a triangle t outside g that touches g, intersects *slab*(v), and does not span v. But then t should have been included in g, a contradiction.[1]

Two gaps of v are *adjacent* if the lower boundary of one is the other's upper boundary.

The list of free and blocked gaps of v are stored in vertical order in a balanced search tree, denoted by *gaplist*(v). Group the gaps in *gaplist*(v) into maximal contiguous subsequences. Store each such subsequence in a biased search tree [6] which allows an item with weight w to be accessed in $O\big(\log \frac{W}{w}\big)$ time, where W is the total weight of all items. The weight of a gap g set to be $\Pr(g)$. We call each such biased search tree a *gap tree* of v.

For every internal node v of $\mathcal{T}$, we set up some pointers from the gaps of v to the gap trees of the children of v as follows. Let w be a child of v. The free gaps of v only give rise to shadow gaps of w, so they do not induce any item in *gaplist*(w). Every blocked gap g of v gives rise to a contiguous sequence σ of free and blocked gaps of w. Moreover, σ is maximal in *gaplist*(w) because g is not adjacent to any other blocked gap of v. Therefore, σ is stored as one gap tree T_σ of w. We keep a pointer from g to the root of T_σ.

Since we truncate the recursive expansion of the slab tree $\mathcal{T}$ at depth $\log_2 n$, we may not be able to answer every query using $\mathcal{T}$. We need a backup which is a dynamic point location structure $\mathcal{T}^*$ [3, 8, 15, 28]. Any worst-case dynamic point location structure with $O(\log n)$ query time suffices, provided that its update time is $O(\log^2 n)$ and its space is $O(n \log n)$.

[1] There is one exception: when a blocked gap boundary contains a boundary edge e of Δ, updates may insert new vertices in the interior of e, splitting e into collinear boundary edges. However, the portion $\Delta_B \setminus \Delta$ of the triangle tiling remains fixed. We ignore this exception to simplify the presentation.

3.2 Querying

Given a query point q, we start at the root r of $\mathcal{T}$, and q must lie in a gap stored in the only gap tree of r. In general, when we visit a node v of $\mathcal{T}$, we also know a gap tree T_v of v such that q lies in one of the gaps in T_v. We search T_v to locate the gap, say g, that contains q. If g is a free gap, the search terminates because we have located a triangle in Δ_B that contains q. Suppose that g is a blocked gap. Then, we check in $O(1)$ time which child w of v satisfies $q \in slab(w)$. By construction, g contains a pointer to the gap tree T_w of w that stores the free and blocked gaps of w in $g \cap slab(w)$. We jump to T_w to continue the search. If the search reaches a leaf of $\mathcal{T}$ without locating a triangle of Δ_B, we answer the query using $\mathcal{T}^*$.

▶ **Lemma 3.** *The expected query time of $\mathcal{T}$ is $O(OPT + \log\log n)$, where OPT is the expected query time of the best point location decision tree for Δ.*

3.3 Construction

The children of a node v of $\mathcal{T}$ can be created in time linear in the number of lines in $\mathcal{L}$ that intersect $slab(v)$. Thus, constructing the primary tree of $\mathcal{T}$ takes $O(|\mathcal{L}| \log n)$ time.

The gap lists and gap trees are constructed via a recursive traversal of $\mathcal{T}$. In general, when we come to a node v of $\mathcal{T}$ from $parent(v)$, we maintain the following preconditions.

- We have only those triangles in Δ_B such that each intersects $slab(v)$ and does not span $parent(v)$. These triangles form a directed acyclic graph G_v: triangles are graph vertices, and two triangles sharing a side are connected by a graph edge directed from the triangle above to the one below.[2]
- The connected components of G_v are sorted in order from top to bottom. Note that each connected component intersects both bounding lines of $slab(v)$.

Each connected component C in G_v corresponds to a maximum contiguous subsequence of free and blocked gaps in $gaplist(v)$ (to be computed), so for each C, we will construct a gap tree T_C. We will return the roots of all such T_C's to $parent(v)$ in order to set up pointers from the blocked gaps of $parent(v)$ to the corresponding T_C's.

Gap list. We construct $gaplist(v)$ first. Process the connected components of G_v in vertical order. Let C be the next one. The restriction of the upper boundary of C to $slab(v)$ is the upper gap boundary induced by C. Perform a topological sort of the triangles in C. We pause whenever we visit a triangle $t \in C$ that spans v. Let t' denote the last triangle in C encountered that spans v, or in the absence of such a triangle, the upper boundary of C. If $t \cap t' = \emptyset$ or $t \cap t'$ does not span v, the region in $slab(v)$ between t' and t is a blocked gap, and we append it to $gaplist(v)$. Then, we append $slab(v) \cap t$ as a newly discovered free gap to $gaplist(v)$. The construction of $gaplist(v)$ takes $O(|G_v|)$ time.

Recurse at the children. Let v_L, v_M and v_R denote the left, middle and right children of v. We scan the connected components of G_v in the vertical order to extract G_{v_L}. A connected component C in G_v may yield multiple components in G_{v_L} because the triangles that span v are omitted. The components in G_{v_L} are ordered vertically by a topological sort of C. Thus, G_{v_L} and the vertical ordering of its connected components are produced in $O(|G_v|)$ time. The generation of G_{v_M}, G_{v_R} and the vertical orderings of their connected components is similar. Then, we recurse at v_L, v_M and v_R.

[2] Refer to [19, Section 4] for a proof that this ordering is acyclic.

Gap trees. After we have recursively handled the children of v, we construct a gap tree for each maximal contiguous subsequence of gaps in $gaplist(v)$. The construction takes linear time [6]. The recursive call at v_L returns a list, say X, of the roots of gap trees at v_L, and X is sorted in vertical order. There is a one-to-one correspondence between X and the blocked gaps of v in vertical order. Therefore, in $O(|gaplist(v)|)$ time, we can set up pointers from the blocked gaps of v to the corresponding gap tree roots in X. The pointers from the blocked gaps of v to the gap tree roots at v_M and v_R are set up in the same manner. Afterwards, if v is not the root of $\mathcal{T}$, we return the list of gap tree roots at v in vertical order.

Running time. We spend $O(|G_v|)$ time at each node v. If a triangle t contributes to G_v for some node v, then either $slab(v) \cap t$ is a free gap of v, or $slab(v) \cap t$ is incident to the leftmost or rightmost vertex of t. Like storing segments in a segment tree, t contributes $O(\log n)$ free gaps. The nodes of $\mathcal{T}$ whose slabs contain the leftmost (resp. rightmost) vertex of t form a root-to-leaf path. Therefore, t contributes $O(\log n)$ triangles in the G_v's over all nodes v in $\mathcal{T}$. The sum of $|G_v|$ over all nodes v of $\mathcal{T}$ is $O(n \log n)$.

▶ **Lemma 4.** *Given Δ_B and $\mathcal{L}$, the slab tree and its auxiliary structures, including gap lists and gap trees, can be constructed in $O(|\mathcal{L}| \log n)$ time and $O(n \log n)$ space.*

4 Handling triangulation-updates: fixed vertical lines

We discuss how to update the slab tree when Δ_B is updated such that every new vertex lies on a vertical line in the given set $\mathcal{L}$. This restriction will be removed later in Section 5. A *triangulation-update* U has the following features:

- It specifies some triangles in Δ whose union is a polygon R_U possibly with holes.
- It specifies a new triangulation T_U of R_U. T_U may contain vertices in the interior of R_U. T_U does not have any new vertex in the boundary of R_U, except possibly for the boundary edges of R_U that lie on the outer boundary of Δ.
- The construction of T_U takes $O(|T_U| \log |T_U|)$ time.
- The *size* of U is the total number of triangles in $\Delta \cap R_U$ and T_U.

Our update algorithm is a localized version of the construction algorithm in Section 3.3. It is also based on a recursive traversal of the slab tree $\mathcal{T}$. When we visit a node v of $\mathcal{T}$, we have a directed acyclic graph H_v that represents *legal* and *illegal* regions in $T_U \cap slab(v)$:

- For each triangle $t \in T_U$ that intersects the interior of $slab(v)$ and does not span $parent(v)$, $t \cap slab(v)$ is a legal region in H_v.
- Take the triangles in T_U that span $parent(v)$. Intersect their union with $slab(v)$. Each resulting connected component that has a boundary vertex in the interior of $slab(v)$ is an *illegal region*. Its upper and lower boundaries contain at most two edges each. Requiring a boundary vertex inside $slab(v)$ keeps the complexity of illegal regions low.
- Store H_v as a directed acyclic graph: regions are graph vertices, and two regions sharing a side are connected by an edge directed from the region above to the one below.

To update $gaplist(v)$, we essentially merge it with a topologically sorted order of regions in each component of H_v.

▶ **Lemma 5.** *Updating $gaplist(v)$ and the gap trees of v takes $O(|H_v| \log n)$ amortized time.*

Let $c < 1/2$ be a constant. We rebuild $\mathcal{T}$ and its auxiliary structures with respect to $\mathcal{L}$ and the current Δ_B when the total size of triangulation-updates exceeds cn' since the initial construction or the last rebuild, where n' was the number of triangles in Δ_B then.

▶ **Lemma 6.** *Let n denote the number of triangles in Δ_B.*

- $n = \Theta(n')$.
- *Any query can be answered in $O(\mathrm{OPT} + \log\log n)$ expected time, where* OPT *is the minimum expected query time of the best point location decision tree for Δ.*
- *The data structure uses $O(n \log n)$ space and can be constructed in $O(|\mathcal{L}| \log n)$ time.*
- *A triangulation-update of size $k \le n/2$ takes $O(k \log^2 n + (|\mathcal{L}| \log n)/n)$ amortized time.*

5 Allowing arbitrary vertex location

In this section, we discuss how to allow a new vertex to appear anywhere instead of on one of the fixed lines in $\mathcal{L}$. This requires revising the slab tree structure. The main issue is how to preserve the geometric decrease in the probability of a query point falling into the slabs of internal nodes on every root-to-leaf path in $\mathcal{T}$.

Initialize $\mathcal{L}$ to be the set of vertical lines through the vertices of the initial Δ_B. Construct the initial slab tree $\mathcal{T}$ for Δ_B and $\mathcal{L}$ using the algorithm in Section 3.3. Whenever $\mathcal{T}$ is rebuilt, we also rebuild $\mathcal{L}$ to be the set of vertical lines through the vertices of the current Δ_B. Between two successive rebuilds, we grow $\mathcal{L}$ monotonically as triangulation-updates are processed. Although every vertex of Δ_B lies on a line in $\mathcal{L}$, some line in $\mathcal{L}$ may not pass through any vertex of Δ_B between two rebuilds.

The free, blocked, and shadow gaps of a slab tree node are defined as in Section 3. So are the gap trees of a slab tree node. However, gap weights are redefined in Section 5.1 in order that they are robust against small geometric changes.

When a triangulation-update U is processed, we first process the vertical lines through the vertices of T_U before we process T_U as specified in Section 4. For each vertical line ℓ through the vertices of T_U, if $\ell \notin \mathcal{L}$, we insert ℓ into $\mathcal{L}$ and then into $\mathcal{T}$. Sections 5.2 and 5.3 provide the details of this step. The processing of T_U is discussed in Section 5.4.

Querying is essentially the same as in Section 3.2 except that we need a fast way to descend the slab tree as some nodes have $O(\log n)$ children. This is described in Section 5.2.

5.1 Weights of gaps and more

Let n' be the number of triangles in Δ_B in the initial construction or the last rebuild, whichever is more recent. Let $N = 2(c+1)n'$, where c is the constant in the threshold cn' for triggering a rebuild of $\mathcal{T}$.

For every free gap g, let t_g denote the triangle in the current Δ_B that contains g, and the weight of g is $wt(g) = \max\{\Pr(t_g), 1/N\}$. The alternative $1/N$ makes the access time of g in a gap tree no worse than $O(\log N) = O(\log n)$.

For every blocked gap g, every vertex p of Δ_B, and every node v of $\mathcal{T}$, define:

- $wt(p)$ = sum of $\max\left\{\frac{1}{N}, \Pr(t)\right\}$ over all triangles $t \in \Delta_B$ incident to p.
- $vert(g) = \{\text{vertex } p \text{ lying in } g : \exists\, \text{triangle}\, pqr \in \Delta_B \text{ s.t. } interior(pqr) \cap interior(g) \ne \emptyset\}$.
- $wt(g) = \sum_{p \in vert(g)} wt(p)$.
- $blocked\text{-}gaps(p) = \{\text{blocked gap } g : p \in vert(g)\}$.
- $vert(v)$ = the subset of vertices of Δ_B that lie in $slab(v)$.
- $lines(v)$ = the subset of lines in $\mathcal{L}$ that intersect $slab(v)$.

The set $vert(g)$ is only used for notational convenience. The set $blocked\text{-}gaps(p)$ is not stored explicitly. We discuss how to retrieve $blocked\text{-}gaps(p)$ in Section 5.2. The sets $vert(v)$ and $lines(v)$ are stored as balanced search trees in increasing order of x-coordinates.

5.2 Revised slab tree structure

Node types. A vertical line *pierces* a slab if the line intersects the interior of that slab. An internal node v of $\mathcal{T}$ has children of two possible types.

- HEAVY-CHILD: A child w of v is a *heavy-child* if $\Pr(slab(w)) > \Pr(slab(v))/2$.
  - The heavy-child w may be labelled *active* or *inactive* upon its creation. This label will not change. If w was created in the initial construction or the last rebuild of $\mathcal{T}$, then w is inactive.
  - If w is inactive, $gaplist(w)$ and the gap trees of w are represented as before. If w is active, then w is a leaf, and $gaplist(w)$ and the gap trees of w are stored as persistent data structures using the technique of node copying [18].
- LIGHT-CHILD: There are two sequences of *light-children* of v, denoted by *left-light*(v) and *right-light*(v), which satisfy the following properties.
  - For each light child w of v, $\Pr(slab(w)) \leq \Pr(slab(v))/2$.
  - For each light child w of v, $gaplist(w)$ and the gap trees of w are represented as before.
  - Let *left-light*$(v) = (w_1, w_2, \cdots, w_k)$ and let *right-light*$(v) = (w_{k+1}, w_{k+2}, \cdots, w_m)$ in the left-to-right order of the nodes.
    * For $i \in [1, k-1] \cup [k+1, m-1]$, $slab(w_i)$ and $slab(w_{i+1})$ are interior-disjoint and share a boundary.
    * If v has an active heavy-child w, then $slab(w)$ is bounded by the right and left boundaries of $slab(w_k)$ and $slab(w_{k+1})$, respectively. Otherwise, the right boundary of $slab(w_k)$ is the left boundary of $slab(w_{k+1})$.
    * If v does not have an active heavy child, v has at most $2\log_2 N + 2$ children.
    * If v has an active heavy-child, the following properties are satisfied.
      (i) For $r \geq 1$, a light-child w of v has *rank* r if the number of lines in $\mathcal{L}$ that intersect $slab(w)$ is in the range $[2^r, 2^{r+1})$. So $r \leq \log_2 N$, where $N = 2(c+1)n'$. We denote r by $rank(w)$.
      (ii) We have $rank(w_1) > \cdots > rank(w_k)$ and $rank(w_{k+1}) < \cdots < rank(w_m)$. For $r \in [1, \log_2 N]$, there is at most one light-child of rank r in each of *left-light*(v) and *right-light*(v) .

Node access. Each node v keeps a biased search tree $children(v)$. The weight of a child w in $children(v)$ is $\max\{\frac{\Pr(slab(v))}{2\log_2 N+2}, \Pr(slab(w))\}$, where $N = 2(c+1)n'$. Since $n = \Theta(n')$, accessing w takes $O\big(\min\{\log \frac{\Pr(slab(v))}{\Pr(slab(w))}, \log\log n\}\big)$ time. For each blocked gap g of v, we use a biased search tree T_g to store pointers to the gap trees induced by g at the children of v. The weight of the node in T_g that represents a gap tree T at a child w is $\max\{\frac{\Pr(slab(v))}{2\log_2 N+2}, \Pr(slab(w))\}$. Accessing T via T_g takes $O\big(\min\{\log \frac{\Pr(slab(v))}{\Pr(slab(w))}, \log\log n\}\big)$ time. Given a vertex p of Δ_B, there are $O(\log n)$ blocked gaps in *blocked-gaps*(p) and we can find them as follows. Traverse the path from the root of $\mathcal{T}$ to the leaf whose slab contains p, and for each node v encountered, we search $gaplist(v)$ to find the blocked gap of v that contains p. The time needed is $O(\log^2 n)$.

5.3 Insertion of a vertical line into the slab tree

Let ℓ be a new vertical line. We first insert ℓ into $\mathcal{L}$ and then insert ℓ into $\mathcal{T}$ in a recursive traversal towards the leaf whose slab is pierced by ℓ.

Internal node. Suppose that we visit an internal node v. We first insert ℓ into $lines(v)$. We query $children(v)$ to find the child slab pierced by ℓ. If v does not have an active heavy-child, recursively insert ℓ at the child found. Otherwise, we work on $left\text{-}light(v)$ or $right\text{-}light(v)$.

- Case 1: ℓ pierces $slab(w_j)$ for some $w_j \in left\text{-}light(v)$. If $slab(w_j)$ intersects fewer than $2^{rank(w_j)+1}$ lines in $\mathcal{L}$, recursively insert ℓ into w_j and no further action is needed.[3] Otherwise, $slab(w_j)$ intersects $2^{rank(w_j)+1}$ lines in $\mathcal{L}$, violating the structural property of a light-child. In this case, we merge some nodes in $left\text{-}light(v)$ as follows.

  Let $left\text{-}light(v) = (w_1, \cdots, w_j, \cdots)$. Find the largest $i \leq j$ such that the number of lines in $\mathcal{L}$ that intersect $slab(w_i) \cup \cdots \cup slab(w_j)$ is in the range $[2^r, 2^{r+1})$ for some $rank(w_i) \leq r < rank(w_{i-1})$. Note that $r > rank(w_j)$. Let ℓ_L denote the left boundary of $slab(w_i)$. Let ℓ_R denote the right boundary of $slab(w_j)$. Let S denote the slab bounded by ℓ_L and ℓ_R. We rebuild the slab subtree rooted at w_i and its auxiliary structures to expand $slab(w_i)$ to S as follows. It also means that $rank(w_i)$ is updated to r. The children $w_{i+1}, \ldots, w_j$ and their old subtrees are deleted afterwards.

  Let $V = vert(w_i) \cup \cdots \cup vert(w_j)$. Let $\Lambda = lines(w_i) \cup \cdots \cup lines(w_j)$. First, we construct a new slab subtree rooted at w_i with respect to V and Λ as described in Section 3.1. No auxiliary structure is computed yet. We control the construction so that it does not produce any node at depth greater than $\log_2 N = O(\log n)$ with respect to the whole slab tree. The construction time is $O(|\Lambda| \log n)$. Afterwards, $slab(w_i)$ becomes S. *Label all heavy-children in the new slab subtree rooted at w_i as inactive.*

  Mark the triangles that are incident to the vertices in V, overlap with S, and do not span S. Let G_{w_i} be the set of marked triangles. This takes $O(|G_{w_i}|)$ time, assuming that each vertex p has pointers to its incident triangle(s) intersected by a vertical line through p. The old blocked gaps of w_i will be affected by the rebuild at w_i. The old free gaps of w_i contained in some triangles in G_{w_i} will be absorbed into some blocked gaps. The other old free gaps of w_i are not affected because their containing triangles span S.

  To update $gaplist(w_i)$, intersect G_{w_i} with S to generate the directed acyclic graph H_{w_i} and then update $gaplist(w_i)$ as in Section 4. This takes $O(|G_{w_i}| \log n)$ amortized time. Only the blocked gaps of w_i can induce gap lists and gap trees at the descendants of w_i. Therefore, as in the construction algorithm in Section 3.3, we can take the subset of G_{w_i} that induce the blocked gaps of w_i and recursively construct the gap lists and gap trees at the descendants of w_i. This takes $O(|G_{w_i}| \log^2 n)$ time by an analysis analogous to the one for Lemma 4.[4] For each blocked gap g of w_i, we create a biased search tree of pointers to the gap trees induced by g at the children of w_i.

  The update of $gaplist(w_i)$ preserves the old shadow gaps of w_i, and it does not generate any new shadow gap. Therefore, no two gap trees of w_i can be merged and no gap tree of w_i can be split, although the content of a gap tree may be updated. A gap tree of w_i is updated only when some free gaps in it are merged into some blocked gaps. Thus, updating the gap trees of w_i takes $O(|G_{w_i}| \log n)$ time.

  Finally, $vert(w_i) := V$, $lines(w_i) := \Lambda$, and the recursive insertion of ℓ terminates.
- Case 2: ℓ pierces $slab(w_j)$ for some $w_j \in right\text{-}light(v)$. Symmetric to Case 1.
- Case 3: ℓ pierces the active heavy-child of v. An active heavy-child is a leaf of the slab tree. We discuss how to insert a vertical line at a leaf next.

---

[3] The line ℓ has already been inserted into $\mathcal{L}$.
[4] Since w_i has $O(\log n)$ instead of $O(1)$ children, the construction time has an extra log factor.

Leaf node. Suppose that we come to a leaf v. If $\text{depth}(v) = \log_2 N$, do nothing and return. Otherwise, there are two cases. Note that $\mathit{gaplist}(v)$ consists of free gaps only. The line ℓ divides $\mathit{slab}(v)$ into slabs S_L and S_R on the left and right of ℓ, respectively.

- Case 1: v is not an active heavy-child of $\mathit{parent}(v)$. Turn v into an internal node by making two children w_L and w_R of v with $\mathit{slab}(w_L) = S_L$ and $\mathit{slab}(w_R) = S_R$. If $\Pr(\mathit{slab}(w_L)) > \Pr(\mathit{slab}(v))/2$, then w_L is the heavy-child of v, label w_L active, and set $\mathit{left\text{-}light}(v) := \emptyset$. If not, w_L is a light-child of rank one and set $\mathit{left\text{-}light}(v) := (w_L)$. The handling of w_R is symmetric. As $\mathit{gaplist}(v)$ consists of free gaps only, $\mathit{gaplist}(w_L)$ and $\mathit{gaplist}(w_R)$ are empty. So w_L and w_R have no gap tree. The initializations of $\mathit{vert}(w_L)$, $\mathit{vert}(w_R)$, $\mathit{lines}(w_L)$, and $\mathit{lines}(w_R)$ are trivial.
- Case 2: v is an active heavy-child of $\mathit{parent}(v)$. We expand $\mathit{left\text{-}light}(\mathit{parent}(v))$ and/or $\mathit{right\text{-}light}(\mathit{parent}(v))$ as follows. W.l.o.g., assume that $\Pr(S_L) \leq \Pr(\mathit{slab}(v))/2$. Update $\mathit{slab}(v) := S_R$, which does not change $\mathit{gaplist}(v)$ or any gap tree of v combinatorially. The weights of gaps in $\mathit{gaplist}(v)$ are also unaffected.
  - Case 2.1: $\Pr(S_R) \leq \Pr(\mathit{slab}(\mathit{parent}(v)))/2$. Then, $\mathit{parent}(v)$ has no heavy-child afterwards. Note that $\mathit{parent}(v)$ has at most $2\log_2 N + 1$ children before this update, where $N = 2(c+1)n'$. Create a light-child w_L of $\mathit{parent}(v)$ with $\mathit{slab}(w_L) = S_L$. Note that $\mathit{gaplist}(w_L)$ and the gap trees of w_L are combinatorially identical to those of v, which are stored as persistent search trees. We copy them to form $\mathit{gaplist}(w_L)$ and the gap trees of w_L, each taking $O(1)$ amortized space and time. Append w_L to $\mathit{left\text{-}light}(\mathit{parent}(v))$. Add v to $\mathit{right\text{-}light}(\mathit{parent}(v))$ as its leftmost element. Therefore, $\mathit{parent}(v)$ has at most $2\log_2 N + 2$ children afterwards.
  - Case 2.2: $\Pr(S_R) > \Pr(\mathit{slab}(\mathit{parent}(v)))/2$. Then, v remains the active heavy-child of $\mathit{parent}(v)$. We handle S_L as follows.
    - If $\mathit{left\text{-}light}(\mathit{parent}(v))$ contains no light-child of rank one, then create a light-child w_L with $\mathit{slab}(w_L) = S_L$ as in Case 2.1 above, and append w_L to $\mathit{left\text{-}light}(\mathit{parent}(v))$.
    - Otherwise, let $\mathit{left\text{-}light}(\mathit{parent}(v)) = (w_1, \cdots, w_k)$, i.e., $\mathit{rank}(w_k) = 1$. Find the largest $i \leq k$ such that the number of lines in $\mathcal{L}$ that intersect $\mathit{slab}(w_i) \cup \cdots \cup \mathit{slab}(w_k) \cup S_L$ is in the range $[2^r, 2^{r+1})$ for some $\mathit{rank}(w_i) \leq r < \mathit{rank}(w_{i-1})$. Expand $\mathit{slab}(w_i)$ to the slab bounded by the left boundary of $\mathit{slab}(w_i)$ and the right boundary of S_L as in Case 1 of the insertion of a vertical line at an internal node. Rebuild the slab subtree rooted at w_i and its auxiliary structures. *Label all heavy-children in the new slab subtree rooted at w_i as inactive.*

▶ **Lemma 7.** *Let $\mathcal{T}$ be the slab tree constructed for a set $\mathcal{L}'$ of vertical lines and Δ_B in the initial construction or the last rebuild, whichever is more recent. For any $\mathcal{L} \supset \mathcal{L}'$, the insertion time of lines in $\mathcal{L} \setminus \mathcal{L}'$ into $\mathcal{T}$ is $O(|\mathcal{L} \setminus \mathcal{L}'| \log^4 n)$ plus some charges on edges of Δ_B such that every edge gains at most $O(\log^4 n)$ charge since the initial construction or the last rebuild, whichever is more recent.*

5.4 Handling triangulation-updates

Let U be a triangulation-update of size $k \leq n/2$. Let n' be the number of triangles in Δ_B in the initial construction or the last rebuild, whichever is more recent. Let c be a constant less than $1/2$. If the threshold cn' has been exceeded by the total size of triangulation-updates (including U) since the initial construction or the last rebuild, we rebuild $\mathcal{T}$ and its auxiliary structures. It takes $O(n \log^2 n)$ time and space. If U does not trigger a rebuild, we proceed as follows instead.

Step 1. Check the $O(k)$ vertical lines through the vertices of T_U. For each line that does not appear in $\mathcal{L}$, we insert it into $\mathcal{L}$ and then into $\mathcal{T}$ as discussed in Section 5.3.

Step 2. The weights of $O(k)$ vertices may change and $O(k)$ vertices may be inserted or deleted. It is straightforward to update the weights of existing vertices, set the weights of new vertices, and delete vertices in $O(k)$ time. For every vertex p of the old triangulation, let $wt'(p)$ be its weight in the old triangulation. For every vertex p of the new triangulation, let $wt(p)$ be its weight in the new triangulation. We perform the following action.

Action-I: for every vertex p of the old triangulation that lies in R_U,
- for every gap $g \in \mathit{blocked\text{-}gaps}(p)$, update $wt(g) := wt(g) - wt'(p)$;
- if p does not lie in the boundary of R_U, then for each slab tree node v such that $p \in \mathit{vert}(v)$, delete p from $\mathit{vert}(v)$.

Action-I runs in $O(k \log^2 n)$ time.

Step 3. For every vertex p of T_U that lies strictly inside R_U, and every ancestor v of the leaf node of $\mathcal{T}$ whose slab contains p, insert p into $\mathit{vert}(v)$. This step takes $O(k \log^2 n)$ time.

Step 4. To update the gap lists and the gap trees, traverse $\mathcal{T}$ as in Section 4. For each node v of $\mathcal{T}$ visited, form a directed acyclic graph H_v of regions to update v as in Section 4. This step takes $O(\sum_v |H_v| \log n)$ amortized time.

Step 5. The weight of a free gap does not change as long as its defining triangle is preserved. The weights of some blocked gaps may not be updated completely yet, and we fix them by performing Action-II below. Assume that a zero weight is assigned initially to every blocked gap that is created by the triangulation-update and contains vertices in T_U only.

Action-II: for each vertex p of T_U and every gap $g \in \mathit{blocked\text{-}gaps}(p)$, update $wt(g) := wt(g) + wt(p)$.

Action-II runs in $O(k \log^2 n)$ time.

▶ **Theorem 8.** *Let n denote the number of triangles in Δ_B.*
- *Any query can be answered in $O(\mathrm{OPT} + \log\log n)$ expected time, where* OPT *is the minimum expected query time of the best point location decision tree for Δ.*
- *The data structure uses $O(n \log^2 n)$ space, and it can be constructed in $O(n \log^2 n)$ time.*
- *A triangulation-update of size $k \leq n/2$ takes $O(k \log^4 n)$ amortized time.*

References

1. U. Adamy and R. Seidel. On the exact worst case query complexity of planar point location. *Journal of Algorithms*, 27(1):189–217, 2000.
2. P. Afshani, J. Barbay, and T. Chan. Instance-optimal geometric algorithms. *Journal of the ACM*, 64(1):3:1–3:38, 2017.
3. L. Arge, G.S. Brodal, and L Georgiadis. Improved dynamic planar point location. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 305–314, 2006.
4. S. Arya, T. Malamatos, D. Mount, and K. Wong. Optimal expected-case planar point location. *SIAM Journal on Computing*, 37(2):584–610, 2007.
5. H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17(3):342–380, 1994.

6 S.W. Bent, D.D. Sleator, and R.E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.
7 Prosenjit Bose, Luc Devroye, Karim Douieb, Vida Dujmovic, James King, and Pat Morin. Odds-on trees, 2010. `arXiv:1002.1092`.
8 T. Chan and Y. Nekrich. Towards an optimal method for dynamic planar point location. *SIAM Journal on Computing*, 47(6):2337–2361, 2018.
9 S.-W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM Journal on Computing*, 21(5):972–999, 1992.
10 S.-W. Cheng and M.-K. Lau. Adaptive planar point location. In *Proceedings of the 33rd International Symposium of Computational Geometry*, pages 30:1–30:15, 2017.
11 S.-W. Cheng and M.-K. Lau. Adaptive point location in planar convex subdivisions. *International Journal of Computational Geometry and Applications*, 27(1–2):3–12, 2017.
12 S.-W. Cheng and M.-K. Lau. Adaptive planar point location, 2018. `arXiv:1810.00715`.
13 S.-W. Cheng and M.-K. Lau. Dynamic distribution-sensitive point location, 2020. `arXiv:2003.08288`.
14 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 Journal on Computing*, 25(1):207–233, 1996.
15 Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *Internatational Journal of Computational Geometry and Applications*, 2(3):311–333, 1992.
16 S. Collette, V. Dujmović, J. Iacono, S. Langerman, and P. Morin. Entropy, triangulation, and point location in planar subdivisions. *ACM Transactions on Algorithms*, 8(3):29:1–29:18, 2012.
17 D.P. Dobkin and D.G. Kirkpatrick. Determining the separation of preprocessed polyhedra—a unified approach. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 400–413, 1990.
18 J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
19 H. Edelsbrunner, L. J Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.
20 M.T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths in planar subdivisions via balanced geodesic triangulations. *Journal of Algorithms*, 23(1):51–73, 1997.
21 M.T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. *SIAM Journal on Computing*, 28(2):612–636, 1998.
22 J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18(3):403–431, 1995.
23 J. Iacono. Expected asymptotically optimal planar point location. *Computational Geometry: Theory and Applications*, 29(1):19–22, 2004.
24 J. Iacono and W. Mulzer. A static optimality transformation with applications to planar point location. *International Journal of Computational Geometry and Applications*, 22(4):327–340, 2012.
25 D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.
26 E. Oh. Point location in incremental planar subdivisions. In *Proceedings of the 29th International Symposium on Algorithms and Computation*, pages 51:1–51:12, 2018.
27 E. Oh and H.-K. Ahn. Point location in dynamic planar subdivision. In *Proceedings of the 34th International Symposium on Computational Geometry*, pages 63:1–53:14, 2018.
28 F.P. Preparata and R. Tamassia. Fully dynamic point location in a monotone subdivision. *SIAM Journal on Computing*, 18(4):811–830, 1989.
29 N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of ACM*, 29(7):669–679, 1986.