Point Location in Incremental Planar Subdivisions

Eunjin Oh
Max Planck Institute for Informatics, Saarbrücken, Germany
eoh@mpi-inf.mpg.de

Abstract

We study the point location problem in incremental (possibly disconnected) planar subdivisions, that is, dynamic subdivisions allowing insertions of edges and vertices only. Specifically, we present an $O(n \log n)$-space data structure for this problem that supports queries in $O(\log^2 n)$ time and updates in $O(\log n \log\log n)$ amortized time. This is the first result that achieves polylogarithmic query and update times simultaneously in incremental planar subdivisions. Its update time is significantly faster than the update time of the best known data structure for fully-dynamic (possibly disconnected) planar subdivisions.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Dynamic point location, general incremental planar subdivisions

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2018.51

Related Version A full version of this paper is available at `http://arxiv.org/abs/1809.10495`.

1 Introduction

Given a planar subdivision, a point location query asks for finding the face of the subdivision containing a given query point. The planar subdivisions for point location queries are induced by planar embeddings of graphs. A planar subdivision consists of faces, edges and vertices whose union coincides with the whole plane. An edge of a subdivision is considered to be open, that is, it does not include its endpoints (vertices). A face of a subdivision is a maximal connected subset of the plane that does not contain any point on an edge or a vertex. The boundary of a face of a subdivision may consist of several connected components. Imagine that we give a direction to each edge on the boundary of a face F so that F lies to the left of it. (If an edge is incident to F only, we consider it as two edges with opposite directions.) We call a boundary component of F the *outer boundary* of F if it is traversed in counterclockwise order around F. Every bounded face has exactly one outer boundary. We call a connected component other than the outer boundary an *inner boundary* of F.

We say a planar subdivision is *dynamic* if the subdivision changes dynamically by insertions and deletions of edges and vertices. A dynamic planar subdivision is *connected* if the underlying graph is connected at any time. In other words, the boundary of each face is connected. We say a dynamic planar subdivision is *general* if it is not necessarily connected. There are three versions of dynamic planar subdivisions with respect to the update operations they support: incremental, decremental and fully-dynamic. An incremental subdivision allows only insertions of edges and vertices, and a decremental subdivision allows only deletions of edges and vertices. A fully-dynamic subdivision allows both of them.

The dynamic point location problem is closely related to the dynamic vertical ray shooting problem in the case of connected subdivisions [6]. In this problem, we are asked to find the edge of a dynamic planar subdivision that lies immediately above a query point. The boundary of each face in a dynamic connected subdivision is connected, so one can maintain the boundary of each face efficiently using a concatenable queue. Then one can answer a point location query without increasing the space and time complexities using a data structure for the dynamic vertical ray shooting problem [6].

However, it is not the case in general planar subdivisions. Although the dynamic vertical ray shooting data structures presented in [1, 2, 4, 6] work for general subdivisions, it is unclear how one can use them to support point location queries efficiently. As pointed out in previous works [4, 6], a main issue concerns how to test for any two edges if they belong to the boundary of the same face in the subdivision. This is because the boundary of a face may consist of more than one connected component.

Previous work. There are several data structures for the point location problem in *fully-dynamic* planar *connected* subdivisions [1, 2, 4, 6, 7, 8, 10, 14]. The latest result was given by Chan and Nekrich [4]. The linear-size data structure by Chan and Nekrich [4] supports $O(\log n(\log\log n)^2)$ query time and $O(\log n \log\log n)$ update time in the pointer machine model, where n is the number of the edges of the current subdivision. Some of them [1, 2, 4, 6] including the result by Chan and Neckrich can be used for answering vertical ray shooting queries without increasing the running time.

There are data structures for answering point location queries more efficiently in *incremental* planar *connected* subdivisions in the pointer machine model [1, 10, 11]. The best known data structure supports $O(\log n \log^* n)$ query time and $O(\log n)$ amortized update time, and it has size of $O(n)$ [1]. This data structure can be modified to support $O(\log n)$ query time and $O(\log^{1+\epsilon} n)$ amortized update time for any $\epsilon > 0$. In the case that every cell is monotone at any time, there is a linear-size data structure supporting $O(\log n \log\log n)$ query time and $O(1)$ amortized update time [10].

On the other hand, little has been known about this problem in *fully-dynamic* planar *general* subdivisions, which was recently mentioned by Snoeyink [15]. Very recently, Oh and Ahn [13] presented a linear-size data structure for answering point location queries in $O(\log n(\log\log n)^2)$ time with $O(\sqrt{n}\log n(\log\log n)^{3/2})$ amortized update time. In fact, this is the only data structure known for answering point location queries in general dynamic planar subdivisions. In the same paper, the authors also considered the point location problem in decremental general subdivisions. They presented a linear-size data structure supporting $O(\log n)$ query time and $O(\alpha(n))$ update time, where n is the number of edges in the current subdivision and $\alpha(n)$ is the inverse Ackermann function.

Our result. In this paper, we present a data structure for answering point location queries in incremental general planar subdivisions in the pointer machine model. The data structure supports $O(\log^2 n)$ query time and $O(\log n \log\log n)$ amortized update time. This is the first result on the point location problem specialized in incremental general planar subdivisions. The update time of this data structure is significantly faster than the update time of the data structure in fully-dynamic general planar subdivisions in [13].

Comparison to the decremental case. In decremental general subdivisions, there is a simple and efficient data structure for point location queries [13]. This data structure maintains the decremental subdivision explicitly: for each face F of the subdivision, it maintains a number of (concatenable) queues each of which stores the edges of each connected component of the boundary of F. When an edge is removed, two faces might be merged into one face, but no face is subdivided into two faces. Using this property, they maintain a disjoint-set data structure for each face such that an element of the disjoint-set data structure is the name of a queue representing a connected component of the boundary of this face.

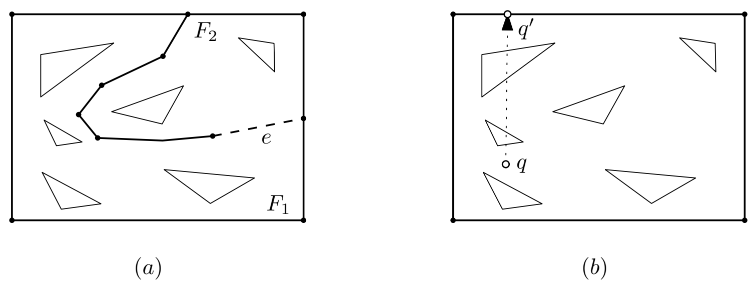

Figure 1 (a) The insertion of e makes the face subdivided into two subfaces F_1 and F_2. (b) Given a query point q, we shoot the upward vertical ray from q which penetrates inner boundaries not containing q until it hits the outer boundary of a face at q'.

In contrast to decremental subdivisions, it is unclear how to maintain incremental subdivisions explicitly. Suppose that a face F is subdivided into F_1 and F_2 by the insertion of an edge e. An inner boundary of F becomes an inner boundary of either F_1 or F_2 after e is inserted. See Figure 1(a). It is unclear how to update the set of the inner boundaries of F_i for $i = 1, 2$ without accessing every queue representing an inner boundary of F. If we access all such queues, the total insertion time for n insertions is $\Omega(n^2)$ in the worst case. Therefore it does not seem that the approach in [13] works for incremental subdivisions.

2 Preliminaries

Consider an incremental planar subdivision Π. We use $\overline{\Pi}$ to denote the union of the edges and vertices of Π. We require that every edge of Π be a straight line segment. For a set A of elements (points or edges), we use $|A|$ to denote the number of the elements in A. For a planar subdivision Π', we use $|\Pi'|$ to denote the complexity of Π', i.e., the number of the edges of Π'. We use n to denote the number of the edges of Π at the moment. Also, for a connected component γ of $\overline{\Pi}$, we use Π_γ to denote the subdivision induced by γ. Notice that Π_γ is connected. Due to lack of space, proofs and details are omitted. Missing proofs and details can be found in the full version of this paper.

In this problem, we are to process a mixed sequence of n edge insertions and vertex insertions so that given a query point q the face of the current subdivision containing q can be computed efficiently. More specifically, each face in the subdivision is assigned a distinct name, and given a query point the name of the face containing the point is to be reported. For the insertion of an edge e, we require e to intersect no edge or vertex in the current subdivision. Also, an endpoint of e is required to lie on a face or a vertex of the subdivision. We insert the endpoints of e in the subdivision as vertices if they were not vertices of the subdivision. For the insertion of a vertex v, it lies on an edge or a face of the current subdivision. If it lies on an edge, the edge is split into two (sub)edges whose common endpoint is v.

2.1 Tools

In this subsection, we introduce tools we use. A *concatenable queue* represents a sequence of N elements, and allows five operations: insert an element, delete an element, search an element, split a queue into two queues, and concatenate two queues into one. By implementing them with 2-3 trees, we can support each operation in $O(\log N)$ time.

The *vertical decomposition* of a (static) planar subdivision Π_s is a finer subdivision of Π_s induced by vertical line segments. For each vertex v of Π_s, consider two vertical extensions from v, one going upwards and one going downwards. The extensions stop when they meet an edge of Π_s other than the edges incident to v. The vertical decomposition of Π_s is the subdivision induced by the vertical extensions contained in the bounded faces of Π_s together with the edges of Π_s. Note that the unbounded face of Π_s remains the same. In this paper, we do not consider the unbounded face of Π_s as a cell of the vertical decomposition. Therefore, every cell is a trapezoid or a triangle (a degenerate trapezoid). There are $O(|\Pi_s|)$ trapezoids in the vertical decomposition of Π_s. We treat each trapezoid as a closed set. We can compute the vertical decomposition in $O(|\Pi_s|)$ time [5] since we do not decompose the unbounded face of Π_s.

We use segment trees, interval trees and priority search trees as basic building blocks. In the following, we briefly review those trees. But we use priority search trees and interval trees of larger fan-out only in the part omitted in the main text, so we also omit their description. Their description can be found in the full version of this paper. For more information, refer to [9, Section 10].

We first introduce the segment tree and the interval tree on a set $\mathcal{I}$ of n intervals on the x-axis. Let $\mathcal{I}_p$ be the set of the endpoints of the intervals of $\mathcal{I}$. The base tree is a binary search tree on $\mathcal{I}_p$ of height $O(\log n)$ such that each leaf node corresponds to exactly one point of $\mathcal{I}_p$. Each internal node v corresponds to a point $\ell(v)$ on the x-axis and an interval $\mathsf{region}(v)$ on the x-axis such that $\ell(v)$ is the midpoint of $\mathcal{I}_p \cap \mathsf{region}(v)$. For the root v, $\mathsf{region}(v)$ is defined as the x-axis. Suppose that $\ell(v)$ and $\mathsf{region}(v)$ are defined for a node v. For its two children v_ℓ and v_r, $\mathsf{region}(v_\ell)$ and $\mathsf{region}(v_r)$ are the left and right regions of $\mathsf{region}(v)$, respectively, in the subdivision of $\mathsf{region}(v)$ induced by $\ell(v)$.

For the interval tree, each interval $I \in \mathcal{I}$ is stored in exactly one node: the node v of maximum depth with $\mathsf{region}(v) \subseteq I$, that is, the lowest common ancestor of two leaf nodes corresponding to the endpoints of I. For the segment tree, each interval I is stored in $O(\log n)$ nodes: the nodes v with $\mathsf{region}(v) \subseteq I$, but $\mathsf{region}(u) \not\subseteq I$ for the parent u of v. For any point $p \in \mathbb{R}$, let $\pi(p)$ be the search path of p. The intervals of $\mathcal{I}$ containing p are stored in some nodes of $\pi(p)$ in both trees. However, not every interval stored in such nodes contains p in the interval tree while every interval stored in such nodes contains p in the segment tree.

Similarly, the segment tree and the interval tree on a set $\mathcal{S}$ of n line segments in the plane are defined as follows. Let $\mathcal{S}_x$ be the set of the projections of the line segments of $\mathcal{S}$ onto the x-axis. The segment and interval trees of $\mathcal{S}$ are basically the segment and interval trees on $\mathcal{S}_x$, respectively. The only difference is that instead of storing the projections, we store a line segment of $\mathcal{S}$ in the nodes where its projection is stored in the case of $\mathcal{S}_x$. As a result, $\ell_x(v)$ and $\mathsf{region}_x(v)$ for the trees of $\mathcal{S}$ are naturally defined as the vertical line containing $\ell(v)$ and the smallest vertical slab containing $\mathsf{region}(v)$ for the trees of $\mathcal{S}_x$, respectively. If it is clear in context, we use $\ell(v)$ and $\mathsf{region}(v)$ to denote $\ell_x(v)$ and $\mathsf{region}_x(v)$, respectively.

2.2 Subproblem: Stabbing-Lowest Query Problem for Trapezoids

The trapezoids we consider have two sides parallel to the y-axis. We consider the *stabbing-lowest query problem* for trapezoids as a subproblem. In this problem, we are given a set $\mathcal{T}$ of trapezoids which is initially empty and changes dynamically by insertions of trapezoids. Here, the trapezoids we are given satisfy that no two upper or lower sides of the trapezoids cross each other. But it is possible that the upper (or lower) side of one trapezoid crosses a vertical side of some other trapezoid. We process a sequence of updates so that given a query point q, the trapezoid with lowest upper side can be found efficiently among all trapezoids of $\mathcal{T}$ containing q. Here, we say a trapezoid has the *lowest* upper side if its upper side is intersected first by the vertical upward ray from q among all upper sides of the trapezoids of $\mathcal{T}$ containing q. We call such a trapezoid the *lowest trapezoid stabbed by* q.

In Section 4, we present a data structure for this problem . The worst case query time is $O(\log^2 n)$, the amortized update time is $O(\log n \log\log n)$, and the size of the data structures is $O(n \log n)$. We will use this data structure as a black box in Section 3.

3 Point Location in Incremental General Planar Subdivisions

Compared to connected subdivisions, a main difficulty for handling dynamic general planar subdivisions lies in finding the faces incident to the edge e lying immediately above a query point [6]. If e is contained in the outer boundary of a face, we can find such a face as the algorithm in [6] for connected planar subdivisions does. However, this approach does not work if e lies on an inner boundary of a face. To overcome this difficulty, instead of finding the edge in Π lying immediately above a query point q, we find an outer boundary edge of the face F of Π containing q. See Figure 1(b). To do this, we answer a point location query in two steps.

First, we find the (maximal) connected component γ of $\overline{\Pi}$ containing the outer boundary of the face F containing the query point q. We use FindCC(Π) to denote this data structure. Observe that the boundary of the face of Π_γ containing q coincides with the outer boundary of F. We maintain the boundary of each face of Π_γ using a concatenable queue. Thus given an outer boundary edge of F, we can return the name of F by defining the name of each face of Π as the name of the concatenable queue representing its outer boundary.

Second, we apply a point location query on Π_γ. More specifically, we find the face F_γ in Π_γ containing q, find the concatenable queue representing the boundary of F_γ, and return its name. Since Π_γ is connected, we can maintain an efficient data structure for point location queries on Π_γ. We use LocateCC(γ) to denote this data structure. Each of Sections 3.1 and 3.2 describes each of the two data structures together with query and update algorithms.

In addition to them, we maintain the following data structures: one for checking if a new edge is incident to $\overline{\Pi}$, one for maintaining the connected components of $\overline{\Pi}$, and one for maintaining the concatenable queue for the outer boundary of each face of Π. Details can be found in the full version.

3.1 FindCC(Π): Finding One Connected Component for a Query Point

We construct a data structure for finding the (maximal) connected component γ_q of $\overline{\Pi}$ containing the outer boundary of the face of Π containing a query point q. To do this, we compute a set $\mathcal{T}$ of $O(n)$ trapezoids each of which *belongs* to exactly one edge of Π such that the edge to which the lowest trapezoid stabbed by q belongs is contained in γ_q. Then we construct the stabbing-lowest data structure on $\mathcal{T}$ described in Section 4.

Data structure and query algorithm. For each connected component γ of $\overline{\Pi}$, consider the subdivision Π_γ induced by γ. Notice that Π_γ is connected. Let $U(\gamma)$ be the union of the closures of all bounded faces of Π_γ. Note that it might be disconnected. Imagine that we have the cells (trapezoids) of the vertical decomposition of $U(\gamma)$. Note that an edge of γ might intersect a cell. We say that a cell of the decomposition *belongs* to the edge of γ containing the upper side of the cell. Let $\mathcal{T}_\gamma$ be the set of such cells (trapezoids) for γ, and $\mathcal{T}$ be the union of $\mathcal{T}_\gamma$ for every connected component γ of $\overline{\Pi}$. See Figure 2. In the full version, we show that the lowest trapezoid in $\mathcal{T}$ stabbed by a query point q belongs to an edge in γ_q. If no trapezoid in $\mathcal{T}$ contains q, we conclude that q is contained in the unbounded face of Π.

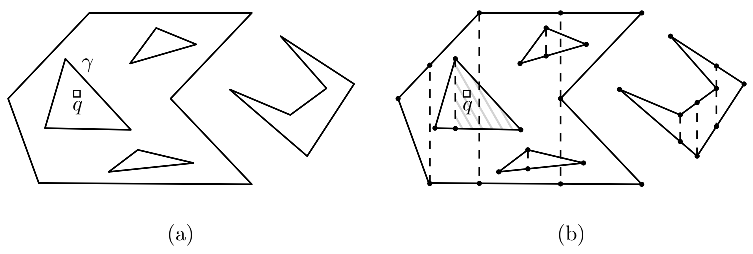

Figure 2 (a) The component γ contains the outer boundary of the face containing q. (b) Using the vertical decomposition, we obtain $O(n)$ (possibly intersecting) trapezoids. Their corners are marked with disks. The lowest trapezoid stabbed by q is the dashed one, which comes from γ.

However, each edge insertion may induce $\Omega(n)$ changes on $\mathcal{T}$ in the worst case. For an efficient update procedure, we define and construct the trapezoid set $\mathcal{T}_\gamma$ in a slightly different way by allowing some edges lying inside $U(\gamma)$ to define trapezoids in $\mathcal{T}_\gamma$. For a connected component γ of $\overline{\Pi}$, we say a set of connected subdivisions induced by edges of γ *covers* γ if an edge of γ is contained in at most two subdivisions, and one of the subdivisions contains all edges of the boundary of $U(\gamma)$. Let $\mathcal{F}_\gamma$ be a set of connected subdivisions covering γ. See Figure 3. Notice that $\mathcal{F}_\gamma$ is not necessarily unique. For a technical reason, if the union of some edges (including their endpoints) in a subdivision of $\mathcal{F}_\gamma$ forms a line segment, we treat them as one edge. Then we let $\mathcal{T}_\gamma$ be the set of the cells of the vertical decompositions of the subdivisions in $\mathcal{F}_\gamma$. Note that a cell of $\mathcal{T}_\gamma$ might intersect another cell of $\mathcal{T}_\gamma$. See Figure 3(b). We say that a cell (trapezoid) of $\mathcal{T}_\gamma$ *belongs* to the edge of γ containing the upper side of the cell. Let $\mathcal{T}$ be the union of all such sets $\mathcal{T}_\gamma$.

Due to the following lemma, we can maintain $\mathcal{T}$ efficiently. In the update algorithm, we insert trapezoids to $\mathcal{T}$ only.

▶ **Lemma 1.** *The size of $\mathcal{T}$ is $O(n)$, where n is the complexity of the current subdivision.*

The following lemma shows that the lowest trapezoid in $\mathcal{T}$ stabbed by q belongs to an edge of γ_q. Thus by constructing a stabbing-lowest data structure on $\mathcal{T}$, we can find γ_q in $O(Q(n))$ time, where $Q(n)$ is the query time for answering a stabbing-lowest query. The query time of the structure on n trapezoids described in Section 4 is $O(\log^2 n)$.

▶ **Lemma 2.** *The lowest trapezoid in $\mathcal{T}$ stabbed by a query point q belongs to an edge of the connected component of $\overline{\Pi}$ containing the outer boundary of the face of Π containing q. If the face of Π containing q is unbounded, no trapezoid in $\mathcal{T}$ contains q.*

▶ **Lemma 3.** *Given* FINDCC(Π) *of size $O(n)$, we can find the connected component of $\overline{\Pi}$ containing the outer boundary of the face of Π containing a query point in $O(\log^2 n)$ time.*

Update algorithm. We maintain a stabbing-lowest data structure on $\mathcal{T}$. Let $\mathcal{T}_\gamma$ be the set of the trapezoids of $\mathcal{T}$ which belong to edges of γ. Notice that we do not maintain the sets $\mathcal{F}_\gamma$ and $\mathcal{T}_\gamma$ for a connected component γ of $\overline{\Pi}$. We use them only for description purpose. Here, we describe the update algorithm for the insertion of an edge only. The update algorithm for the insertion of a vertex can be found in the full version.

We process the insertion of an edge e by inserting a number of trapezoids to $\mathcal{T}$. Here, we use Π to denote the subdivision of complexity n *before* e is inserted. There are four cases: e is not incident to $\overline{\Pi}$, only one endpoint of e is contained in $\overline{\Pi}$, the endpoints of e are contained in distinct connected components of $\overline{\Pi}$, and the endpoints of e are contained in the same

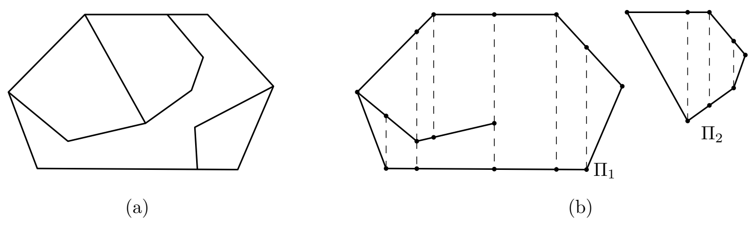

Figure 3 (a) A connected component γ. (b) A set of two subdivisions covering γ. The union of the edge sets of the two subdivisions is γ. The set $\mathcal{T}_\gamma$ consists of the trapezoids in the vertical decompositions of Π_1 and Π_2.

connected component of $\overline{\Pi}$. We can check if e belongs to each case in $O(\log n)$ time using the data structure described at the beginning of Section 3 in the full version. For the first three cases, we do not need to update $\mathcal{T}$. This is because no new face appears in the current subdivision. Thus the conditions on the definition of $\mathcal{F}_\gamma$ are not violated in these cases. (We will see this in more detail in the proof of Lemma 4.)

Now consider the remaining case: the endpoints of e are contained in the same connected component, say γ, of $\overline{\Pi}$. Recall that $U(\gamma)$ is closed. If e is contained in the interior of $U(\gamma)$, we do nothing since $\mathcal{F}_\gamma$ covers $\gamma \cup e$. We can check this in constant time. Details can be found in the full version. If e is not contained in the interior of $U(\gamma)$, we trace the edges of the new face in time linear in the complexity of the new face using the data structures presented at the beginning of Section 3 in the full version. Then we compute the vertical decomposition of the face in the same time [5], and insert them to $\mathcal{T}$. This takes time linear in the number of the new trapezoids inserted to $\mathcal{T}$, which is $O(n)$ in total over all updates by Lemmas 1 and 4, and the fact that no trapezoid is removed from $\mathcal{T}$. As new trapezoids are inserted to $\mathcal{T}$, we update the stabbing-lowest data structure on $\mathcal{T}$.

For the correctness, we have the following lemma. A proof can be found in the full version.

▶ **Lemma 4.** *For each connected component γ of $\overline{\Pi}$, there is a set $\mathcal{F}_\gamma$ of connected subdivisions covering γ such that $\mathcal{T}_\gamma$ consists of the cells of the vertical decompositions of the subdivisions of $\mathcal{F}_\gamma$ at any moment.*

Let $S(n), Q(n)$ and $U(n)$ be the size, the query time and the update time of an insertion-only stabbing-lowest data structure for n trapezoids, respectively. In the case of the data structure described in Section 4, we have $S(n) = O(n \log n)$, $Q(n) = O(\log^2 n)$ and $U(n) = O(\log n \log \log n)$. Recall that the total number of trapezoids inserted to $\mathcal{T}$ is $O(n)$. We have the following lemma.

▶ **Lemma 5.** *We can construct a data structure of size $O(S(n))$ so that the connected component of $\overline{\Pi}$ containing the outer boundary of the face containing q can be found in $O(Q(n))$ worst case time for any point q in the plane, where n is the number of edges at the moment. Each update takes $O(U(n))$ amortized time.*

3.2 LocateCC(γ): Find the Face Containing a Query Point in Π_γ

For each connected component γ of $\overline{\Pi}$, we maintain a data structure, which is denoted by LocateCC(γ), for finding the face of Π_γ containing a query point. Here, we need two update operations for LocateCC($\cdot$): inserting a new edge to LocateCC($\cdot$) and merging two data structures LocateCC(γ_1) and LocateCC(γ_2) for two connected components γ_1 and γ_2 of $\overline{\Pi}$. Notice that we do not need to support edge deletion since Π is incremental.

No known point location data structure supports the merging operation explicitly. Instead, one simple way is to make use of the edge insertion operation which is supported by most of the known point location data structures. For merging two data structures, we simply insert every edge in the connected component of smaller size to the data structure for the other connected component. By using a simple charging argument, we can show that the amortized update time (insertion and merging) is $O(U'(n)\log n)$, where $U'(n)$ is the insertion time of the dynamic point location data structure we use. If we use the data structure by Arge et al. [1], the query time is $O(\log n \log^* n)$ and the amortized update time is $O(\log^2 n)$.

In this section, we improve the update time at the expense of increasing the query time. Because FINDCC(Π) requires $O(\log^2 n)$ query time, we are allowed to spend more time on a point location query on γ. The data structure proposed in this section supports $O(\log^2 n)$ query time. The amortized update time is $O(\log n \log\log n)$.

Data structure and query algorithm. LOCATECC(γ) allows us to find the face of Π_γ containing a query point. Since γ is connected and we maintain the outer boundary of each face of Π, it suffices to construct a vertical ray shooting structure for the edges of γ. Recall that the boundary of a face of Π_γ coincides with the outer boundary of a face of Π. The vertical ray shooting problem is *decomposable* in the sense that we can answer a query on $\mathcal{S}_1 \cup \mathcal{S}_2$ in constant time once we have the answers to queries on $\mathcal{S}_1$ and $\mathcal{S}_2$ for any two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ of line segments in the plane. Thus we can use an approach by Bentley and Saxe [3].

We decompose the edge set of γ into subsets of distinct sizes such that each subset consists of exactly 2^i edges for some index $i \le \lceil \log n \rceil$. Note that there are $O(\log n)$ subsets in the decomposition. We use $\mathcal{B}(\gamma)$ to denote the set of such subsets, and $\mathcal{B}$ to denote the union of $\mathcal{B}(\gamma)$ for all connected components γ of $\overline{\Pi}$. LOCATECC(γ) consists of $O(\log n)$ static vertical ray shooting data structures, one for each subset in $\mathcal{B}(\gamma)$. To answer a query on γ, we apply a vertical ray shooting query on each subset of $\mathcal{B}(\gamma)$, and choose the one lying immediately above the query point. This takes $O(Q_s(n)\log n)$ time, where $Q_s(n)$ denotes the query time of the static vertical ray shooting data structure we use.

For a static vertical ray shooting data structure $\mathcal{D}_s(\beta)$ for $\beta \in \mathcal{B}$, we present a variant of the (dynamic) vertical ray shooting data structure of Arge et al. [1]. It supports $O(\log n)$ query time, and an efficient merging operation. In the update procedure, we merge two subsets β_1 and β_2 in $\mathcal{B}$ into one, and merge their static vertical ray shooting data structures. If we construct $\mathcal{D}_s(\beta_1 \cup \beta_2)$ from scratch, the total update time is $\Omega(n \log^2 n)$ because the construction of a vertical ray shooting data structure on N segments takes $\Omega(N \log N)$ time for any data structure. To improve this update time, we maintain a set of sorted lists of edges, which we call the *backbone tree*, so that we can merge two static ray shooting data structures more efficiently. Notice that the edges of Π cannot be consistently sorted with respect to the y-axis in advance. This happens if no vertical line crosses two edges of Π. The y-order of the two edges depends on the edges to be inserted. In our case, we maintain sets of edges which can be consistently sorted (i.e., edges intersecting a common vertical line), and maintain their sorted lists. Details can be found in the full version. Proofs of the following lemmas can also be found in the full version.

▶ **Lemma 6.** *Given $\mathcal{D}_s(\beta)$ for every subset $\beta \in \mathcal{B}$, we can find the edge lying immediately above a query point among the edges of a connected component γ of $\overline{\Pi}$ in $O(\log^2 n)$ time.*

▶ **Lemma 7.** *Given $\mathcal{D}_s(\beta_1)$ and $\mathcal{D}_s(\beta_2)$ for two subsets β_1 and β_2 of $\mathcal{B}$, we can construct $\mathcal{D}_s(\beta)$ in $O(|\beta| \log\log n)$ time, where $\beta = \beta_1 \cup \beta_2$.*

Update algorithm. We have two update operations, the insertion of edges and vertices. We do not need to update LocateCC($\cdot$) in the case of a vertex insertion. Details can be found in the full version. We use Π to denote the subdivision of complexity n *before* e is inserted.

Suppose that we are given an edge e and we are to update LocateCC($\cdot$). Specifically, we update the static vertical ray shooting data structures for some subsets of $\mathcal{B}$ and the backbone tree. We find the connected components of $\overline{\Pi}$ incident to e in $O(\log n)$ time. There are three cases: there is no such connected component, there is only one such connected component, or there are two such connected components. We show how to update the data structure only for the last case. Details for the other cases can be found in the full version.

For the last case, let γ_1 and γ_2 be two connected components incident to e. They are merged into one connected component together with e. If every subset in $\mathcal{B}(\gamma_1)$ and $\mathcal{B}(\gamma_2)$ has distinct size, we just collect every static vertical ray shooting data structure constructed on a subset in $\mathcal{B}(\gamma_1) \cup \mathcal{B}(\gamma_2)$, and insert e to the data structure. Then we are done. If not, we first choose the largest subsets, one from $\mathcal{B}(\gamma_1)$ and one from $\mathcal{B}(\gamma_2)$, of the same size, say 2^i. Then we construct a new vertical ray shooting data structure on the union β' of the two subsets in $O(2^{i+1} \log\log n)$ time. If there is a subset in $\mathcal{B}(\gamma_1)$ or $\mathcal{B}(\gamma_2)$ of size 2^{i+1} other than β', we again merge them together to form a subset of size 2^{i+2}. We repeat this until every subset in $\mathcal{B}(\gamma_1)$ and $\mathcal{B}(\gamma_2)$ of size at least 2^i has distinct size. Then we consider the largest subsets, one from $\mathcal{B}(\gamma_1)$ and one from $\mathcal{B}(\gamma_2)$, of the same size again. Note that the size of the two subsets is less than 2^i. We merge them, and repeat the merge procedure. We do this for every pair of subsets in $\mathcal{B}(\gamma_1)$ and $\mathcal{B}(\gamma_2)$ of the same size. Finally, we have the set $\mathcal{B}(\gamma_1 \cup \gamma_2)$ of subsets of the edges of $\gamma_1 \cup \gamma_2$ of distinct sizes, and the static vertical ray shooting data structure for each subset in $\mathcal{B}(\gamma_1 \cup \gamma_2)$. Then we insert e to the data structure. Details can be found in the full version.

▶ **Lemma 8.** *The total time for updating every vertical ray shooting data structure in the course of n edge insertions is $O(n \log n \log\log n)$.*

▶ **Lemma 9.** *We can maintain a data structure of size $O(n \log\log n)$ in an incremental planar subdivision Π so that the edge of γ lying immediately above q can be found in $O(\log^2 n)$ time for any edge e and any connected component γ of $\overline{\Pi}$. The amortized update time of this data structure is $O(\log n \log\log n)$.*

4 Incremental Stabbing-Lowest Data Structure for Trapezoids

In this section, we are given a set $\mathcal{T}$ of trapezoids which is initially empty. Then we are to process the insertions of trapezoids to $\mathcal{T}$ so that the lowest trapezoid in $\mathcal{T}$ stabbed by a query point can be found efficiently. Recall that the upper and lower sides of the trapezoids we consider in this paper do not cross each other. To make the description easier, we present a simplified version of our data structure supporting $O(\log^2 n \log\log n)$ query time and $O(\log n \log\log n)$ insertion time in the main text. By using an interval tree of fan-out $\log^\epsilon n$, we can improve the query time by a factor of $\log\log n$. Details can be found in the full version.

Data structure. The base tree is an interval tree of the upper and lower sides of the trapezoids of $\mathcal{T}$. Since the left and right sides of the trapezoids are parallel to the y-axis, a node of the interval tree stores the upper side of a trapezoid of $\mathcal{T}$ if and only if it stores the lower side of the trapezoid. Here, instead of storing the upper and lower sides of a trapezoid, we store the trapezoid itself in such a node. In this way, a trapezoid $\square$ of $\mathcal{T}$ is stored in at most one node of the interval tree. For details, refer to Section 2.

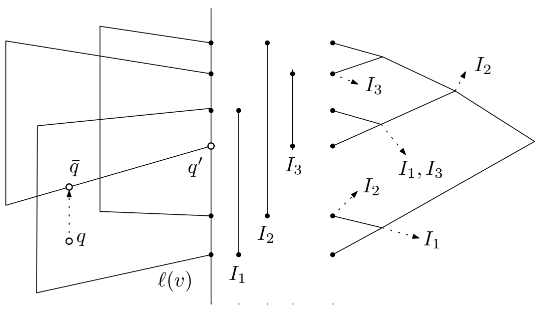

Figure 4 The segment tree constructed on the intersections of the trapezoids of $S(v)$ with $\ell(v)$.

We construct a secondary structure associated with a node v of the base tree as follows. Let $S(v)$ be the set of the trapezoids stored in v. Every trapezoid of $S(v)$ intersects a common vertical line $\ell(v)$. Thus, their upper and lower sides can be sorted in their y-order. See Figure 4. Let $\mathcal{I}(v)$ be the set of the intersections of the trapezoids of $S(v)$ with $\ell(v)$. Note that it is a set of intervals of $\ell(v)$. We construct a segment tree $T(v)$ of $\mathcal{I}(v)$. A node u of $T(v)$ corresponds to an interval $\mathsf{region}(u)$ contained in $\ell(v)$. Every interval of $\mathcal{I}(v)$ stored in u contains $\mathsf{region}(u)$. An interval $I \in \mathcal{I}(v)$ has its corresponding trapezoid $\square$ in $S(v)$ such that $\square \cap \ell(v) = I$. We let I have the *key* which is the x-coordinate of the left side of $\square$.

For each node u of $T(v)$, we construct a tertiary data structure so that given a query value x the interval with lowest upper endpoint can be found efficiently among the intervals stored in u and having their keys less than x. Imagine that we sort the intervals of $\mathcal{I}(v)$ stored in u with respect to their keys, and denote them by $\langle I_1, \ldots, I_k \rangle$. And we use $\square_i \in \mathcal{T}$ to denote the trapezoid corresponding to the interval I_i (i.e., $\ell(v) \cap \square_i = I_i$) for $i = 1, \ldots, k$. The tertiary data structure is just a sublist of $\langle I_1, \ldots, I_k \rangle$. Specifically, suppose x is at least the key of I_i and at most the key of I_{i+1} for some i. Then every interval in $\langle I_1, I_2, \ldots, I_i \rangle$ has its key at most x. Thus the answer to the query is the one with lowest upper endpoint among $\langle I_1, I_2, \ldots, I_i \rangle$. Using this observation, we construct a sublist of $\langle I_1, \ldots, I_k \rangle$ as follows. We choose the interval, say I_i, if its upper endpoint is the lowest among the upper endpoints of the intervals in $\langle I_1, \ldots I_i \rangle$. We maintain the sublist consisting of the chosen intervals. Notice that the sublist has *monotonicity* with respect to their upper endpoints. That is, the upper endpoint of I_i lies lower than the upper endpoint of $I_{i'}$ if I_i comes before $I_{i'}$ in the sublist. This property makes the update procedure efficient.

By applying binary search on the sublist with respect to the keys, we can find the interval with lowest endpoint among the intervals stored in u and having the keys less than x. For each node of the base tree, we maintain a structure for dynamic fractional cascading [12] on the segment tree so that the binary search on the sublist associated with each node of the segment tree can be done in $O(\log n \log\log n)$ time in total. Then we also do this for the right sides of the trapezoids of $S(v)$.

A tricky problem here is that a query point q and the upper or lower side of a trapezoid in $S(v)$ cannot be ordered with respect to the y-axis in general. This happens if the left side of the trapezoid lies to the right of q. See Figure 4. This makes it difficult to follow a search path in the segment tree associated with v. To resolve this, we find the side e lying immediately above q among the upper and lower sides of the trapezoids in $S(v)$, and then follow the search path of $q' = e \cap \ell(v)$. To do this, we construct a vertical ray shooting data structure on the upper and lower sides of the trapezoids in $S(v)$. Details can be found in the full version.

Query algorithm. Using this data structure, we can find the lowest trapezoid in $\mathcal{T}$ stabbed by a query point q as follows. We follow the base tree (interval tree) along the search path π of q of length $O(\log n)$. For each node of π, we consider its associated secondary structures, and we find the lowest trapezoid stabbed by q among the trapezoids stored in the node. And we return the lowest one among all trapezoids we obtained from the nodes of π. We spend $O(\log n \log\log n)$ time on each node in π, which leads to the total query time of $O(\log^2 n \log\log n)$.

We have a segment tree on the intersections of the trapezoids of $S(v)$ with $\ell(v)$ for a node v in π. We first find the upper or lower side e of a trapezoid of $S(v)$ immediately lying above q among them in $O(\log n)$ time using the vertical ray shooting data structure associated with v, and let q' be the intersection point between e and $\ell(v)$. See Figure 4. We show that the lowest trapezoid stabbed by q is stored in a node in the search path of q'. A proof can be found in the full version. Thus, it suffices to consider $O(\log n)$ nodes w in the segment tree with $q' \in \mathsf{region}(w)$. Then we find the successor of the x-coordinate of q on the sublist associated with each such node. By construction, the trapezoid corresponding to the successor is the lowest trapezoid stabbed by q among all trapezoids stored in w. Using dynamic fractional cascading, we can find it in $O(\log\log n)$ time for each node after spending $O(\log n)$ time for the initial binary search of only one node in the segment tree. Thus we can find all successors in $O(\log n \log\log n)$ time.

▶ **Lemma 10.** *Using the data structure described in this section, we can find the lowest trapezoid stabbed by a query point in $O(\log^2 n \log\log n)$ time.*

Update algorithm. We assume that the trapezoids to be inserted are known in advance so that we can keep the base tree and all segment trees balanced. We can get rid of this assumption with standard technique using weight-balanced B-trees. We show how to do this in the full version. Let $\square$ be a trapezoid to be inserted to the data structure. We find the node v of maximum depth in the base tree such that $\mathsf{region}(v)$ contains $\square$ in $O(\log n)$ time. The trapezoid $\square$ is to be stored only in this node.

We update the secondary structure (segment tree) for $S(v)$ by inserting $\square$. We find the set W of $O(\log n)$ nodes in the segment tree where $\square$ is to be inserted. Each node $w \in W$ is associated with a sorted list $L(w)$ of intervals stored in w. We decide if we store $\square \cap \ell(v)$ in $L(w)$. To do this, we find the position for $\square$ in $L(w)$ by applying binary search on $L(w)$ with respect to the key. Here we do this for every node in W, and thus we can apply fractional cascading. The key of each interval in the sorted lists is in $\mathbb{R}$. Thus we can apply (dynamic) fractional cascading so that each binary search takes $O(\log\log n)$ time after spending $O(\log n)$ time on the initial binary search on a node of W [12].

Let $\langle I_1, \ldots, I_k \rangle$ be the sorted list of the intervals stored in w. The list $L(w)$ is a sublist of this list, say $\langle I_{i_1}, \ldots, I_{i_t} \rangle$. Let I_{i_j} be the predecessor of $\square \cap \ell(v)$. We determine if $\square$ is inserted to the list in constant time: if the upper side of $\square$ lies below the upper side of the trapezoid $\square_{i_{j+1}}$ with $\square_{i_{j+1}} = I_{i_{j+1}} \cap \ell(v)$, we insert $\square \cap \ell(v)$ to the list. Otherwise, the list stored in w remains the same. If we insert $\square \cap \ell(v)$ to the list, we check if it violates the monotonicity of $L(w)$. To do this, we consider the trapezoid $\square'$ whose corresponding interval lies before $\square$ one by one from $\square_{i_j}$. If the upper side of $\square'$ lies above $\square$, we remove $\square'$ from the list. Each insertion into and deletion from $L(w)$ takes $O(\log\log n)$ time [12]. We do this until the upper side of $\square'$ lies below the upper side of $\square$. The total update time for the insertion of $\square$ is $O(\log n + N \log\log n)$, where N is the number of the trapezoids deleted due to $\square$. We show that the sum of N over all n insertions is $O(n \log n)$ in the full version. Thus the amortized update time is $O(\log n \log\log n)$ time.

In the full version, we show how to improve the query time by a factor of $O(\log\log n)$. Therefore, we have the following lemma.

▶ **Lemma 11.** *We can maintain an $O(n \log n)$-size data structure on an incremental set of n trapezoids supporting $O(\log n \log\log n)$ amortized update time so that given a query point q, the lowest trapezoid stabbed by q can be computed in $O(\log^2 n)$ time.*

References

1 Lars Arge, Gerth Stølting Brodal, and Loukas Georgiadis. Improved Dynamic Planar Point Location. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006)*, pages 305–314, 2006.

2 Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic Point Location in General Subdivisions. *Journal of Algorithms*, 17(3):342–380, 1994.

3 Jon Louis Bentley and James B. Saxe. Decomposable Searching Problems 1: Static-to-Dynamic Transformations. *Journal of Algorithms*, 1(4):297–396, 1980.

4 Timothy M. Chan and Yakov Nekrich. Towards an Optimal Method for Dynamic Planar Point Location. In *Proceedings of the 56th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2015)*, pages 390–409, 2015.

5 Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(3):485–524, 1991.

6 Siu-Wing Cheng and Ravi Janardan. New Results on Dynamic Planar Point Location. *SIAM Journal on Computing*, 21(5):972–999, 1992.

7 Yi-Jen Chiang, Franco P. Preparata, and Roberto 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.

8 Yi-Jen Chiang and Roberto Tamassia. Dynamization of the trapezoid method for planar point location in monotone subdivisions. *International Journal of Computational Geometry & Applications*, 2(3):311–333, 1992.

9 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 2008.

10 Michael T. Goodrich and Roberto Tamassia. Dynamic Trees and Dynamic Point Location. *SIAM Journal on Computing*, 28(2):612–636, 1998.

11 Hiroshi Imai and Takao Asano. Dynamic orthogonal segment intersection search. *Journal of Algorithms*, 8(1):1–18, 1987.

12 Kurt Mehlhorn and Stefan Näher. Dynamic fractional cascading. *Algorithmica*, 5(1):215–241, 1990.

13 Eunjin Oh and Hee-Kap Ahn. Point Location in Dynamic Planar Subdivision. In *Proceedings of the 34th International Symposium on Computational Geometry (SOCG 2018)*, volume 99 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 63:1–63:14, 2018. `doi:10.4230/LIPIcs.SoCG.2018.63`.

14 Franco P. Preparata and Roberto Tamassia. Fully Dynamic Point Location in a Monotone Subdivision. *SIAM Journal on Computing*, 18(4):811–830, 1989.

15 Jack Snoeyink. Point Location. In *Handbook of Discrete and Computational Geometry, Third Edition*, pages 1005–1023. Chapman and Hall/CRC, 2017.