Point Location in Dynamic Planar Subdivisions

Eunjin Oh
Max Planck Institute for Informatics
Saarbrücken, Germany
eoh@mpi-inf.mpg.de
 https://orcid.org/0000-0003-0798-2580

Hee-Kap Ahn
POSTECH
Pohang, Korea
heekap@postech.ac.kr
 https://orcid.org/0000-0001-7177-1679

Abstract

We study the point location problem on dynamic planar subdivisions that allows insertions and deletions of edges. In our problem, the underlying graph of a subdivision is not necessarily connected. We present a data structure of linear size for such a dynamic planar subdivision that supports sublinear-time update and polylogarithmic-time query. Precisely, the amortized update time is $O(\sqrt{n}\log n(\log\log n)^{3/2})$ and the query time is $O(\log n(\log\log n)^2)$, where n is the number of edges in the subdivision. This answers a question posed by Snoeyink in the Handbook of Computational Geometry. When only deletions of edges are allowed, the update time and query time are just $O(\alpha(n))$ and $O(\log n)$, respectively.

1 Introduction

Given a planar subdivision, a point location query asks with a query point specified by its coordinates to find the face of the subdivision containing the query point. In many situations such point location queries are made frequently, and therefore it is desirable to preprocess the subdivision and to store it in a data structure that supports point location queries fast.

The planar subdivisions for point location queries are usually induced by planar embeddings of graphs. A planar subdivision is connected if the underlying graph is connected. The vertices and edges of the subdivision are the embeddings of the nodes and arcs of the underlying graph, respectively. An edge of the subdivision is considered to be open, that is, it does not include its endpoints (vertices). A face of the subdivision is a maximal connected subset of the plane that does not contain any point on an edge or a vertex.
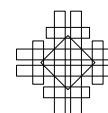
We say a planar subdivision *dynamic* if the subdivision allows two types of operations, the insertion of an edge to the subdivision and the deletion of an edge from the subdivision. The subdivision changes over insertions and deletions of edges accordingly. For an insertion of an edge e, we require e to intersect no edge or vertex in the subdivision and the endpoints of e to lie on no edge in the subdivision. We insert the endpoints of e in the subdivision as vertices if they were not vertices of the subdivision. In fact, the insertion with this restriction is general enough. The insertion of an edge e with an endpoint u lying on an edge e' of the subdivision can be done by a sequence of four operations: deletion of e', insertion of e, and insertions of two subedges of e' partitioned by u.

The dynamic point location problem is closely related to the dynamic vertical ray shooting problem [6]. For this problem, we are asked to find the edge of a dynamic planar subdivision that lies immediately above (or below) a query point. In the case that the subdivision is connected at any time, we can answer a point location query without increasing the space and time complexities using a data structure for the dynamic vertical ray shooting problem by maintaining the list of the edges incident to each face in a concatenable queue [6].

However, it is not the case in a general (possibly disconnected) planar subdivision. Although the dynamic vertical ray shooting algorithms presented in [2, 3, 5, 6] work for general (possibly disconnected) subdivisions, it is unclear how one can use them to support point location queries efficiently. As pointed out in some previous works [5, 6], a main issue concerns how to test whether the two edges lying immediately above two query points belong to the boundary of the same face in a dynamic planar subdivision. Notice that the boundary of a face may consist of more than one connected component.

In this paper, we consider a point location query on dynamic planar subdivisions. The subdivisions we consider are not necessarily connected, that is, the underlying graphs may consist of one or more connected components. We also require that every edge is a straight line segment. We present a data structure for a dynamic planar subdivision which answers point location queries efficiently.

Previous work. The dynamic vertical ray shooting problem has been studied extensively [2, 3, 5, 6]. These data structures do not require that the subdivision is connected, but they require that the subdivision is planar. None of the known algorithms for this problem is superior to the others. Moreover, optimal update and query times (or their optimal trade-offs) are not known. The update time or the query time (or both) is worse than $O(\log^2 n)$, except the data structures by Arge et al. [2] and by Chan and Nekrich [5]. The data structure by Arge at al. [2] supports expected $O(\log n)$ query time and expected $O(\log^2 n / \log \log n)$ update time under Las Vegas randomization in the RAM model. The data structure by Chan and Nekrich [5] supports $O(\log n (\log \log n)^2)$ query time and $O(\log n \log \log n)$ update time in the pointer machine model. Their algorithm can also be modified to reduce the query time at the expense of increasing the update time. As pointed out by Cheng and Janardan [6], all these data structures [2, 3, 5, 6] can be used for answering point location queries if the underlying graph of the subdivision is connected without increasing any resource.

Little has been known for the dynamic point location in general planar subdivisions. In fact, no nontrivial data structure is known for this problem.[1] Cheng and Janardan asked whether such a data structure can be maintained for a general planar subdivision [6], but

---

[1] The paper [2] claims that their data structure supports a point location query for a general subdivision. They present a vertical ray shooting data structure and claim that this structure supports a point location query for a general subdivision using the paper [9]. However, the paper [9] mentions that it works only for a subdivision such that every face in the subdivision has a constant complexity. Therefore, the point location problem for a general subdivision is still open.

this question has not been resolved until now. Very recently, it was asked again by Chan and Nekrich [5] and by Snoeyink [12]. Specifically, Snoeyink asked whether it is possible to construct a dynamic data structure for a general (possibly disconnected) planar subdivision supporting *sublinear query time* of determining if two query points lie in the same face of the subdivision.

Our result. In this paper, we present a data structure and its update and query algorithms for the dynamic point location in general planar subdivisions under the pointer machine model. This is the first result supporting sublinear update and query times, and answers the question posed in [5, 6, 12]. Precisely, the amortized update time is $O(\sqrt{n} \log n (\log \log n)^{3/2})$ and the query time is $O(\log n (\log \log n)^2)$, where n is the number of edges in the current subdivision. When only deletions of edges are allowed, the update and query times are just $O(\alpha(n))$ and $O(\log n)$, respectively. Here, we assume that a deletion operation is given with the *pointer* to an edge to be deleted in the current edge set.

Our approach itself does not require that every edge in the subdivision is a line segment, and can handle arbitrary curves of constant description. However, the data structures for dynamic vertical ray shooting queries require that every edge is a straight line segment, which we use as a black box. Once we have a data structure for answering vertical ray shooting queries for general curves, we can also extend our results to general curves. For instance, the result by Chan and Nekrich [5] is directly extended to x-monotone curves, and so is ours.

One may wonder if the problem is *decomposable* in the sense that a query over $D_1 \cup D_2$ can be answered in constant time from the answers from D_1 and D_2 for any pair of disjoint data sets D_1 and D_2 [8]. If a problem is decomposable, we can obtain a dynamic data structure from a static data structure of this problem using the framework of Bentley and Saxe [4], or Overmars and Leeuwen [10]. However, the dynamic point location problem in a general planar subdivision is not decomposable. To see this, consider a subdivision D consisting of a square face and one unbounded face. Let D_1 be the subdivision consisting of three edges of the square face and D_2 be the subdivision consisting of the remaining edge of the square face. There is only one face in D_1 (and D_2). Any two points in the plane are contained in the same face in D_1 (and D_2). But it is not the case for D. Therefore, the answers from D_1 and D_2 do not help to answer point location queries on D.

Outline. Consider any two query points in the plane. Our goal is to check whether they are in the same face of the current subdivision. To do this, we use the data structures for answering dynamic vertical ray shooting queries [2, 3, 5, 6], and find the edges lying immediately above the two query points. Then we are to check whether the two edges are on the boundary of the same face. In general subdivisions, the boundary of each face may consist of more than one connected components. This makes $\Theta(n)$ changes to the boundaries of the faces in the worst case, where n is the number of edges in the current subdivision. Therefore, we cannot maintain the explicit description of the subdivision.

To resolve this problem, we consider two different subdivisions, $\mathsf{M_o}$ and $\mathsf{M_n}$, such that the current subdivision consists of the edges of $\mathsf{M_o}$ and $\mathsf{M_n}$, and construct data structures on the subdivisions, $\mathsf{D_o}$ and $\mathsf{D_n}$, respectively. Recall that the dynamic point location problem is not decomposable. Thus the two subdivisions must be defined carefully. We set each edge in the current subdivision to be one of the three states: old, communal, and new. Then let $\mathsf{M_o}$ be the subdivision induced by all old and communal edges, and $\mathsf{M_n}$ be the subdivision induced by all new and communal edges. Note that every communal edge belongs to both subdivisions.

The state of each edge is defined as follows. The data structures are reconstructed periodically. In specific, they are rebuilt after processing $f(n)$ updates since the latest reconstruction, where n is the number of edges in the current subdivision. Here, $f(n)$ is called a reconstruction period, which is set to $\sqrt{n}$ roughly. When an edge e is inserted, we find the face F in $\mathsf{M_o}$ intersecting e and set the old edges on the outer boundary of F to communal. If one endpoint of e lies on the outer boundary of F and the other lies on an inner boundary of F, we set the old edges on this inner boundary to communal. Also, we set e to new. When an edge e is deleted, we find the faces in $\mathsf{M_o}$ incident to e and set the old edges of the outer boundaries of the faces to communal.

We show that the current subdivision has the following property: no face in the current subdivision contains both new and old edges on its outer boundary. In other words, for every face in the current subdivision, either every edge is classified as new or communal, or every edge is classified as old or communal. Due to this property, for any two query points, they are in the same face in the current subdivision if and only if they are in the same face in both $\mathsf{M_o}$ and $\mathsf{M_n}$. Therefore, we can represent the name of a face in the current subdivision as a pair of faces, one in $\mathsf{M_o}$ and one in $\mathsf{M_n}$. To answer a point location query on the current subdivision, it suffices to find the faces containing the query point in $\mathsf{M_o}$ and in $\mathsf{M_n}$.

To answer point location queries on $\mathsf{M_o}$, we observe that no edge is inserted to $\mathsf{M_o}$ unless it is rebuilt. Therefore, it suffices to construct a semi-dynamic point location data structure on $\mathsf{M_o}$ supporting only deletion operations. If only deletion operations are allowed, two faces are merged into one face, but no new face appears. Using this property, we provide a data structure supporting $O(\alpha(n))$ update time and $O(\log n)$ query time.

To answer point location queries on $\mathsf{M_n}$, we make use of the following property: the boundary of each face of $\mathsf{M_n}$ consists of $O(f(n))$ connected components while the number of edges of $\mathsf{M_n}$ is $\Theta(n)$ in the worst case, where n is the number of all edges in the current subdivision. Due to this property, the amount of the change on the subdivision $\mathsf{M_n}$ is $O(f(n))$ at any time. Therefore, we can maintain the explicit description of $\mathsf{M_n}$. In specific, we maintain a data structure on $\mathsf{M_n}$ supporting point location queries, which is indeed a doubly connected linked list of $\mathsf{M_n}$.

Due to lack of space, some proofs and details are omitted. The missing proofs and missing details can be found in the full version of the paper.

2   Preliminaries

Consider a planar subdivision M that consists of n straight line segment edges. Since the subdivision is planar, there are $O(n)$ vertices and faces. One of the faces of M is unbounded and all other faces are bounded. Notice that the boundary of a face is not necessarily connected. For the definitions of the faces and their boundaries, refer to [7, Chapter 2].

We consider each edge of the subdivision as two directed *half-edges*. The two half-edges are oriented in opposite directions so that the face incident to a half-edge lies to the left of it. In this way, each half-edge is incident to exactly one face, and the orientation of each connected component of the boundary of F is defined consistently. We call a boundary component of F the *outer boundary* of F if it is traversed along its half-edges incident to F in counterclockwise order around F. Except for the unbounded face, every face has a unique outer boundary. We call each connected component other than the outer boundary an *inner boundary* of F. Consider the outer boundary γ of a face. Since γ is a noncrossing closed curve, it subdivides the plane into regions exactly one of which contains F. We say a face F *encloses* a set C in the plane if C is contained in the (open) region containing F

of the planar subdivision induced by the outer boundary of F. Note that if F encloses F', the outer boundary of F does not intersect the boundary of F'. For more details on planar subdivisions, refer to the computational geometry book [7].

Our results are under the pointer machine model, which is more restrictive than the random access model. Under the pointer machine model, a memory cell can be accessed only through a series of pointers while any memory cell can be accessed in constant time under the random access model. Most of the results in [2, 3, 5, 6] are under the pointer machine model, and the others are under the random access model.

Updates: insertion and deletion of edges.   We allow two types of update operations: INSERTEDGE(e) and DELETEEDGE(e). In the course of updates, we maintain a current edge set E, which is initially empty. INSERTEDGE(e) is given with an edge e such that no endpoints of e lies on an edge of the current subdivision. This operation adds e to E, and thus update the current subdivision accordingly. Recall that an edge of the subdivision is a line segment excluding its endpoints. If an endpoint of e does not lie on a vertex of the current subdivision, we also add the endpoint of e to the current subdivision as a vertex. DELETEEDGE(e) is given with an edge e in the current subdivision. Specifically, it is given with a pointer to e in the set E. This operation removes e from E, and updates the subdivision accordingly. If an endpoint of e is not incident to any other edge of the subdivision, we also remove the vertex which is the endpoint of e from the subdivision.

Queries.   Our goal is to process update operations on the data structure so that given a query point q the face of the current subdivision containing q can be computed from the data structure efficiently. Specifically, each face is assigned a distinct name in the subdivision, and given a query point the name of the face containing the point is to be reported. A query of this type is called a *location query*, denoted by LOCATE(x) for a query point x in the plane.

2.1   Data structures

In this paper, we show how to process updates and queries efficiently by maintaining a few data structures for dynamic planar subdivisions. In specific, we use disjoint-set data structures and concatenable queues. Before we continue with algorithms for updates and queries, we provide brief descriptions on these structures in the following. Throughout this paper, we use $S(n), U(n)$ and $Q(n)$ to denote the size, the update and query time of the data structures we use for the dynamic vertical ray shooting in a general subdivision. Notice that $U(n) = \Omega(\log n)$, $U(n) = o(n)$, and $Q(n) = \Omega(\log n)$ for any nontirivial data structure for the dynamic vertical ray shooting problem under the pointer machine model. Also, $U(n)$ is increasing. Thus in the following, we assume that $U(n)$ and $Q(n)$ satisfy these properties.

A disjoint-set data structure keeps track of a set of elements partitioned into a number of disjoint subsets [13]. Each subset is represented by a rooted tree in this data structure. The data structure has size linear in the total number of elements, and can be used to check whether two elements are in the same partition and to merge two partitions into one. Both operations can be done in $O(\alpha(N))$ time, where N is the number of elements at the moment and $\alpha(\cdot)$ is the inverse Ackermann function.

A concatenable queue represents a sequence of elements, and allows four operations: insert an element, delete an element, split the sequence into two subsequences, and concatenate two concatenable queues into one. By implementing them with 2-3 trees [1], we can support each operation in $O(\log N)$ time, where N is the number of elements at the moment. We can search any element in the queue in $O(\log N)$ time.

3   Deletion-only point location

In this section, we present a semi-dynamic data structure for point location queries that allows only DELETEEDGE operations. Initially, we are given a planar subdivision consisting of n edges. Then we are given update operations DELETEEDGE(e) for edges e in the subdivision one by one, and process them accordingly. In the course of updates, we answer point location queries. We maintain static data structures on the initial subdivision and a disjoint-set data structure that changes dynamically as we process DELETEEDGE operations.

Static data structures.   We construct the static point location data structure on the initial subdivision of size $O(n)$ in $O(n \log n)$ time [11]. Due to this data structure, we can find the face in the initial subdivision containing a query point in $O(\log n)$ time. We assign a name to each face, for instance, the integers from 1 to m for m faces. Also, we compute the doubly connected edge list of the initial subdivision, and make each edge in the current edge set to point to its counterparts in the doubly connected edge list. These data structures are static, so they do not change in the course of updates.

History structure of faces over updates.   Consider an edge e to be deleted from the subdivision. If e is incident to two distinct faces in the current subdivision, the faces are merged into one and the subdivision changes accordingly. To apply such a change and keep track of the face information of the subdivision, we use a disjoint-set data structure S on the names of the faces in the initial subdivision. Initially, each face forms a singleton subset in S. In the course of updates, subsets in S are merged. Two elements in S are in the same subset of S if and only if the two faces corresponding to the two elements are merged into one face.

Deletion.   We are given DELETEEDGE(e) for an edge e of the subdivision. Since only history structure changes dynamically, it suffices to update the history structure only. We first compute the two faces in the initial subdivision that are incident to e in $O(1)$ time by using the doubly connected edge list. Then we check if the faces belong to the same subset or not in S. If they belong to two different subsets, we merge the subsets into one in $O(\alpha(n))$ time. The label of the root node in the merged subset becomes the name of the merged face. If the faces belong to the same subset, e is incident to the same face F in the current subdivision, and therefore there is no change to the faces in the subdivision, except the removal of e from the boundary of F. Since we do not maintain the boundary information of faces, there is nothing to do with the removal and we do not do anything on S. Thus, there is a bijection between faces in the current subdivision and subsets in the disjoint-set data structure S. We say that the face corresponding to the root of a subset *represents* the subset.

Location queries.   To answer LOCATE(x) for a query point x in the plane, we find the face F in the initial subdivision in $O(\log n)$ time. Then we return the subset in the disjoint-set data structure S that contains F in $O(\alpha(n))$ time. Precisely, we return the root of the subset containing F whose label is the name of the face containing x in the current subdivision. The argument in this section implies the correctness of the query algorithm.

▶ **Theorem 1.** *Given a planar subdivision consisting of n edges, we can construct data structures of size $O(n)$ in $O(n \log n)$ time so that* LOCATE(x) *can be answered in $O(\log n)$ time for any point x in the plane and the data structures can be updated in $O(\alpha(n))$ time for a deletion of an edge from the subdivision.*

4   Data structures for fully dynamic point location

In Section 4 and Section 5, we present a data structure and its corresponding update and query algorithms for the dynamic point location in fully dynamic planar subdivisions. Initially, the subdivision is the whole plane. While we process a mixed sequence of insertions and deletions of edges, we maintain two data structures, one containing *old and communal edges* and one containing *new and communal edges*. We consider each edge of the subdivision to have one of three states, "new", "communal", and "old". The first data structure, denoted by D_o, is the point location data structure on old and communal edges that supports only DELETEEDGE operations described in Section 3. The second data structure, denoted by D_n, is a fully dynamic point location data structure on new and communal edges.

Three states: old, communal and new.   We rebuild both data structures periodically. When they are rebuilt, every edge in the current subdivision is set to old. As we process updates, some of them are set to communal as follows. For INSERTEDGE(e), there is exactly one face F of M_o whose interior is intersected by e as e does not intersect any edges or vertices. We set all edges on the outer boundary of F to communal. If e connects the outer boundary of F with one inner boundary of F, we set all edges on the inner boundary to communal. As a result, the outer boundary edges in the faces incident to e in M_o are communal or new after e is inserted. For DELETEEDGE(e), there are at most two faces of M_o whose boundaries contain e. We set all edges on the outer boundary of these faces to communal. Here, we do not maintain the explicit description (the doubly connected edge list) of M_o, but maintain the semi-dynamic data structure on M_o described in Section 3.

   Also, the edges inserted after the latest reconstruction are set to new. The subdivision M_n of the new and communal edges has complexity of $\Theta(n)$ in the worst case. We maintain a fully dynamic point location data structure on M_n, which is indeed the explicit description (the doubly connected edge list) of M_n.
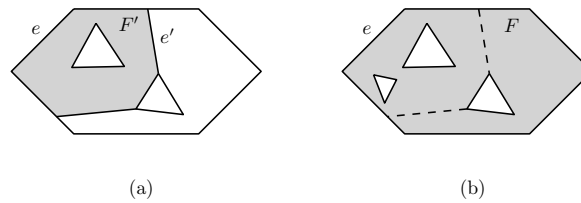
4.1   Reconstruction

Let $f : \mathbb{N} \to \mathbb{N}$ be an increasing function satisfying that $f(n)/2 \le f(n/2) \le n/4$ for every n larger than a constant, which will be specified later. We call the function a *reconstruction period*. We reconstruct D_o and D_n if we have processed $f(n)$ updates since the latest reconstruction time, where n is the number of the edges in the current subdivision.

   The following lemma is a key to achieve an efficient update time. Each face of M_n has $O(f(n))$ boundary components while the number of edge in M_n is $\Theta(n)$ in the worst case.

▶ **Lemma 2.** *Each face of M_n has $O(f(n))$ inner boundaries.*

Proof. Consider a face F of M_n. There are two types of the inner boundaries of F: either all edges on an inner boundary are communal or at least one edge on an inner boundary is new. For an inner boundary of the second type, all edges other than the new edges are communal. By the construction, the number of new edges in M_n is at most $f(n)$. Recall that when we rebuild the data structures, M_o and M_n, all edges are set to old.

   We pick an arbitrary edge on each inner boundary of F of the first type, and call it the *representative* of the inner boundary. Each representative is inserted before the latest reconstruction, and is set to communal later. It becomes communal due to a pair (F', e'), where F' is a face of M_o and e' is an edge inserted or deleted after the latest reconstruction, such that the insertion or deletion of e' makes the edges on a boundary component of F' communal, and e was on this boundary component of F' at that moment. See Figure 1. If

Figure 1 (a) Subdivision M_o. All edges are old. (b) Subdivision M_n. The two dashed edges are deleted after the reconstruction, which makes all outer boundary edges become communal. Then the edges on the leftmost triangle (hole) are inserted.

the representatives of all first-type inner boundaries of F are induced by distinct pairs, it is clear that the number of the first-type inner boundaries of F is $O(f(n))$. But it is possible that the representative of some inner boundaries of F are induced by the same pair (F', e').

Consider the representatives of some inner boundaries of F that are induced by the same pair (F', e'). The insertion of deletion of e' makes the outer boundary of F' become communal. In the case that e' connects the outer boundary of F' and an inner boundary of F', let γ be the cycle consisting of the outer boundary of F', the inner boundary of F' and e'. Let γ be the outer boundary of F', otherwise. All representatives induced by (F', e') are on γ, and therefore they are connected after e' is inserted or before e' is deleted. Notice that any two of such representatives are disconnected later. This means that γ becomes at least t connected components due to the removal of t edges on it, where t is the number of the representatives induced by (F', e'). The total number of edges that are removed after the latest reconstruction is $O(f(n))$, and each edge that are removed after the latest reconstruction can be a representative of at most two first-type inner boundaries of F. Therefore, the total number of the representatives of the first-type inner boundaries of F is also $O(f(n))$.

Consider an inner boundary of the second type. Since each edge is incident to at most one inner boundary of F, the number of the second-type inner boundaries is at most the number of new edges. Therefore, there are $O(f(n))$ second-type inner boundaries of M_n.   ◀

4.2   Two data structures

We maintain two data structures: D_o, a semi-dynamic point location for old and communal edges, and D_n, a fully dynamic point location for new and communal edges. In this subsection, we describe the data structures D_o and D_n. The update procedures are described in Section 5.

Semi-dynamic point location for old and communal edges.   After each reconstruction, we construct the point location data structure D_o supporting only DELETEEDGE described in Section 3 for all edges in the current subdivision, which takes $O(n \log n)$ time. Recall that all edges in the current subdivision are old at this moment. In Section 5, we will see that the amortized time for reconstructing D_o is $O(n \log n / f(n))$ at any moment, where n is the number of all edges in the subdivision at the moment. As update operations are processed, some old or communal edges are deleted, and thus we remove them from D_o. Notice that no edge is inserted to D_o by the definition of old and communal edges.

In addition to this, we store the old edges on each boundary component of the faces of M_o in a concatenable queue. Notice that such edges are not necessarily contiguous on the boundary component. In spite of this fact, we can traverse the old edges along a boundary component of each face of M_o in time linear in the number of the old edges due to the concatenable queu for the old edges.

Fully dynamic point location for new and communal edges. Let $\mathsf{E_n}$ be the set of all new and communal edges and $\mathsf{M_n}$ be the subdivision induced by $\mathsf{E_n}$. Also, let $\mathsf{E_o}$ denote the set of all old and communal edges and $\mathsf{M_o}$ be the subdivision induced by $\mathsf{E_o}$.

We maintain a dynamic data structure that supports vertical ray-shooting queries for $\mathsf{E_n}$. The update time $U(n)$ is $O(\log n \log \log n)$ and the query time $Q(n)$ is $O(\log n (\log \log n)^2)$ if we use the data structure by Chan and Nekrich [5]. Or, there are alternative data structures with different update and query times [2, 3, 6].

We also maintain the boundary of each face F of $\mathsf{M_n}$. We store each connected component of the boundary of F in a concatenable queue. More specifically, a concatenable queue represents a cyclic sequence of the edges in a connected component of the boundary of F. Since e is incident to at most two faces of $\mathsf{M_n}$, there are at most two such elements in the queues. We implement the concatenable queues using the 2-3 trees. We choose an element in each queue and call it the *root* of the queue. For a concatenable queue implemented by a 2-3 tree, we choose the root of the 2-3 tree as the root of the queue. Given any element of a queue, we can access the root of the queue in $O(\log n)$ time. For an inner boundary of a face F of $\mathsf{M_n}$, we let the root of the queue for this inner boundary point to the root of the queue for the outer boundary of F. We also make the root of the queue for the outer boundary of F point to the root of the queue for all inner boundaries of F. Also, we let each edge of $\mathsf{E_n}$ point to its corresponding elements in the queues.

We maintain a balanced binary search tree on the vertices of $\mathsf{M_n}$ sorted in a lexicographical order so that we can check whether a point in the plane is a vertex of $\mathsf{M_n}$ in $O(\log n)$ time. Also, for each vertex of $\mathsf{M_n}$, we maintain a balanced binary search tree on the edges incident to it in $\mathsf{M_n}$ in clockwise order around it. The update procedure of this data structure is straightforward, and the update time is subsumed by the time for maintaining the boundaries of the faces of $\mathsf{M_n}$. Thus, in the following, we do not mention the update of this structure.

▶ **Lemma 3.** *The data structures* $\mathsf{D_o}$ *and* $\mathsf{D_n}$ *have size* $O(n)$.

5 Update procedures for fully dynamic point location

We have two update operations: INSERTEDGE(e) and DELETEEDGE(e). Recall that we rebuild the data structures periodically. More precisely, we reconstruct the data structures if we have processed $f(n)$ updates since the latest reconstruction time, where n is the number of the edges we have at the moment. After the reconstruction, the data structure $\mathsf{D_n}$ becomes empty. This is simply because the reconstruction resets all edges to old. For the data structure $\mathsf{D_o}$, we will show that the amortized time for reconstruction is $O(n \log n / f(n))$. Also, this data structure is updated as some old or communal edges are deleted.

In this section, we present a procedure for updates of the two data structures. Recall that we use $\mathsf{M_o}$ to denote the subdivision induced by the old and communal edges, and $\mathsf{M_n}$ to denote the subdivision induced by the new and communal edges. We use the subdivisions, $\mathsf{M_o}$ and $\mathsf{M_n}$, only for description purpose, and we do not maintain them.

5.1 Common procedure for edge insertions and edge deletions

We are given operation INSERTEDGE(e) or DELETEEDGE(e) for an edge e. Recall that we construct $\mathsf{D_o}$ and $\mathsf{D_n}$ periodically. The reconstruction period $f : \mathbb{N} \to \mathbb{N}$ is an increasing function satisfying that $f(n)/2 \le f(n/2) \le n/4$ for every n larger than a constant.

▶ **Lemma 4.** *The amortized reconstruction time of* $\mathsf{D_o}$ *is* $O(n \log n / f(n))$, *where* n *is the number of all edges at the moment.*

The insertion or deletion sets some old edges to communal. By applying a point location query for an endpoint of e in M_o, we find the faces F of M_o such that the boundary of F contains an endpoint of e or the interior of F is intersected by e. All edges lying on the outer boundary and at most one inner boundary of F become communal. We insert them and e to the data structure D_n for vertical ray shooting queries. This takes $O(N \cdot U(n))$ time, where N denotes the number of all edges inserted to the data structure. It is possible that some edges of the faces are already communal. In this case, we avoid removing (also accessing) such edges by using the concatenable queue representing the cyclic sequence of the old edges on each boundary component of F.

▶ **Lemma 5.** *The average number of old edges which are set to communal is $O(n/f(n))$ at any moment, where n is all edges at the moment.*

▶ **Corollary 6.** *The amortized time for inserting new and communal edges to the vertical ray shooting data structure in D_n is $O(n \cdot U(n)/f(n))$.*

5.2   Edge insertions

We are given operation INSERTEDGE(e) for an edge e. For the data structure D_o, we do nothing since the set of the old and communal edges remains the same. For the data structure D_n, we are required to insert one new edge e and several communal edges. In other words, we are required to update the ray shooting data structure, the concatenable queues and the pointers associated to each edge of E_n. We first process the update due to the communal edges, and then process the update due to the new edge e. The process for the new edge e is the same as the process for the communal edges, except that there is only one new edge e, but there are $O(n/f(n))$ communal edges (amortized). In the following, we describe the process for the communal edges only.

Let $\overline{E}_n$ be the union of the closures of all edges of E_n, where E_n is the set of the new and communal edges before INSERTEDGE(e) is processed. Recall that it is not necessarily connected. Recall that the old edges on the outer boundary of the face intersected by e become communal. If the outer boundary is connected to an inner boundary, we also set the edges of the inner boundary to communal. In this case, let γ be the cycle consisting of these two boundary components and e. Otherwise, let γ be the outer boundary of the face in M_o intersecting e. If γ consists of only communal edges, we do nothing. Thus we assume that it contains at least one old edge. We insert the old edges of γ to D_n in $O(n \cdot U(n)/f(n))$ amortized time by Corollary 6. Recall that the average number of such edges is $O(n/f(n))$ by Corollary 6.

Now we update the concatenable queues and the pointers made by the communal edges on γ in $O(f(n)Q(n) + n \log n/f(n))$. Let F be the face of M_n intersected by γ.

▶ **Lemma 7.** *The curve γ intersects no connected component of $\overline{E}_n$ enclosed by γ assuming that γ contains at least one old edge.*

By Lemma 7, there is a unique face F_γ in the subdivision M_n after the communal edges are inserted such that the outer boundary of F_γ is γ. See Figure 2. The boundaries of F change due to the communal edges, but the boundaries of the other faces remain the same. More precisely, F is subdivided into subfaces, one of which is F_γ. We compute the concatenable queues for each boundary component of the subfaces. We show how to do this for F_γ. While computing the boundary of F_γ, we can compute all boundaries of every subface in the same time. Details can be found in the full version of the paper.
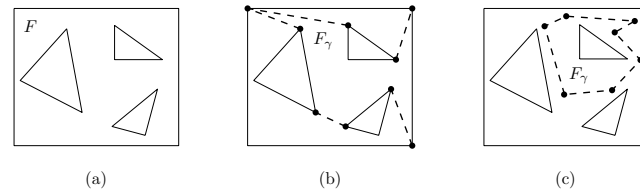
Figure 2 (a) Subdivision $\mathsf{M_n}$ before the communal edges are inserted. (b) The dashed edges are communal edges made by INSERTEDGE. They subdivide F into three faces one of which is F_γ. The boundary of F_γ is γ. (c) All edges of γ are communal edges set by INSERTEDGE.

Concatenable queue for the outer boundary γ of F_γ. We walk along the old edges of γ which become communal one by one using the concatenable queue for the old edges of γ. We make an empty concatenable queue for γ, and insert such edges one by one. If two consecutive old edges g_1 and g_2 of γ share no endpoints, there is a polygonal chain between g_1 and g_2 of γ consisting of communal edges only. Notice that this chain is a part of a boundary component of F. We find the boundary component of F in constant time. We split it with respect to g_1 and g_2, and combine one subchain with the concatenable queue for γ. We keep the other subchain for updating the boundary of F. In this way, we can obtain the concatenable queue for the outer boundary γ of F_γ. This takes $O(N \log n)$ time, where N is the number of old edges of γ which become communal.

Concatenable queues for the inner boundaries of F_γ. A inner boundary β of F might be enclosed by F_γ in the subdivision after the communal edges are inserted. For each inner boundary β of F, we check if it is enclosed by F_γ. To do this, we compute the edge e' immediately lying above the topmost vertex of β using the vertical ray shooting data structure on all new and communal edges, which include the edges of γ. Using the pointer for each edge e' pointing to the elements in the concatenable queues, we can find the boundary component $\beta_{e'}$ containing e' in constant time. If $\beta_{e'}$ is γ, we can determine if β is enclosed by F_γ immediately. Otherwise, β is enclosed by F_γ if and only if $\beta_{e'}$ is enclosed by F_γ. Therefore, we can determine for each inner boundary β of F whether β is enclosed by F_γ in $O(f(n)Q(n))$ time in total since there are $O(f(n))$ inner boundaries of F by Lemma 2.

Since each inner boundary of F_γ was an inner boundary of F before the communal edges are inserted, there is a concatenable queue for each inner boundary of F_γ whose root node points to the outer boundary of F. We make the root of the concatenable queue to point to F_γ, which takes $O(f(n))$ time in total for all inner boundaries of γ.

Pointers for edges. Finally, we update the pointers associated to each edge of $\mathsf{E_n}$ and each old edge of γ which become communal. Recall that each edge of $\mathsf{E_n}$ points to the element in the concatenable queues corresponding to it. For the update of the concatenable queues, we do not remove the elements of them. We just make their pointers to point to other elements of the queues. Therefore, we do not need to do anything for $\mathsf{E_n}$. The only thing we do is to make each old edge of γ to point to the elements in the queues, one representing the outer boundary of F_γ and one representing a boundary component of a subface of F. This takes $O(N \log n)$ time, where N is the number of old edges of γ which become communal.

Therefore, the overall update time for inserting the communal edges is $O(f(n)Q(n) + N \log n)$. Since the average value of N is $O(n/f(n))$ by Lemma 5, the amortized update time is $O(f(n)Q(n) + n \log n/f(n))$. Similarly, the update time for the insertion of e is $O(f(n)Q(n) + \log n)$. The amortized reconstruction time is $O(n \log n/f(n))$ by Lemma 4.

Also, the amortized time for inserting the communal edges to the vertical ray shooting data structure is $O(n \cdot U(n)/f(n))$ by Corollary 6. Therefore, the overall update time is $O(f(n)Q(n) + n \log n/f(n) + n \cdot U(n)/f(n))$, which is $O(f(n)Q(n) + n \cdot U(n)/f(n))$ since $U(n) = \Omega(\log n)$.

▶ **Lemma 8.** *We can process* INSERTEDGE(e) *in* $O(f(n)Q(n) + n \cdot U(n)/f(n))$ *amortized time.*

5.3 Edge deletions

We are given operation DELETEEDGE(e) for an edge e. For the data structure $\mathsf{D_o}$, we update the semi-dynamic point location data structure on the old and communal edges. We also update the concatenable queues for old edges on the boundary components of a face of $\mathsf{M_o}$.

For the data structure $\mathsf{D_n}$, we are required to update the concatenable queues and the pointers associated to each edge of $\mathsf{E_n}$. Here, we insert $O(n/f(n))$ communal edges and delete only one edge e from the data structure. The insertion of the communal edges is exactly the same as the case for edge insertions in the previous subsection. The deletion of e is also similar to the update procedure for edge insertions. The details can be found in the full version of the paper.

▶ **Lemma 9.** *We can process* DELETEEDGE(e) *in* $O(f(n)Q(n) + n \cdot U(n)/f(n))$ *amortized time.*

6 Query procedure

We call the subdivision induced by all old, communal, and new edges the *complete subdivision* and denote it by $\mathsf{M_c}$. Sometimes we mention a face without specifying the subdivision if the face is in the complete subdivision.

Given $\mathsf{D_o}$ and $\mathsf{D_n}$, we are to answer LOCATE(x), that is, to find the face containing the query point x in $\mathsf{M_c}$. Let F_o and F_n be the faces of $\mathsf{M_o}$ and $\mathsf{M_n}$ containing x, respectively. By Theorem 1, we can find F_o in $O(\log n)$ time. For F_n, we find the edge e of $\mathsf{M_n}$ immediately lying above x in $Q(n)$ time using the vertical ray shooting data structure. Then we find the faces containing e on their boundaries using the pointers e has. There are at most two such faces. Since the connected components of the boundary of each face are oriented consistently, we can decide which one contains x in constant time. Therefore, we can compute F_n in $O(Q(n))$ time in total. To answer LOCATE(x), we need the following lemmas.

▶ **Lemma 10.** *No face of* $\mathsf{M_c}$ *contains both an old edge and new edge in its outer boundary.*

Proof. Assume to the contrary that both a new edge e_n and an old edge e_o lie on the outer boundary of a face F of $\mathsf{M_c}$. Note that e_n is inserted after the latest reconstruction. When e_n was inserted, all outer boundary edges of the face F' that was intersected by e_n in $\mathsf{M_o}$ at the moment were set to communal. Since a communal edge is set to old only by a reconstruction, the only possibility for e_o to remain as old is that e_o was not on the outer boundary of F' but on the outer boundary of another face in $\mathsf{M_o}$, and after then it has become an outer boundary edge of F by a series of splits and merges of the faces that are incident to e_o. These splits and merges occur only by insertions and deletions of edges, and e_o is set to communal by such a change to the face that is incident to e_o, and remains communal afterwards. This contradicts that e_o is old, and this case never occurs. ◀

▶ **Lemma 11.** *For any face F in* $\mathsf{M_c}$*, there exists a face in* $\mathsf{M_o}$ *or in* $\mathsf{M_n}$ *whose outer boundary coincides with the outer boundary of F.*

Using the two lemmas above, we can obtain the following lemma.

▶ **Lemma 12.** *For any two points in the plane, they are in the same face in $\mathsf{M_c}$ if and only if they are in the same face in $\mathsf{M_o}$ and in the same face in $\mathsf{M_n}$.*

Proof. Assume that two points x and y are in the same face F in $\mathsf{M_c}$. There is a face F' in $\mathsf{M_o}$ or in $\mathsf{M_n}$ whose outer boundary coincides with the outer boundary of F by Lemma 11. Consider a face F'' in $\mathsf{M_o}$ or $\mathsf{M_n}$ enclosed by F'. Neither x nor y is enclosed by F'' because the edge set of $\mathsf{M_o}$ (and $\mathsf{M_n}$) is a subset of the edge set of $\mathsf{M_c}$. Since the complete subdivision $\mathsf{M_c}$ is planar, this implies that x and y are in the same face in both $\mathsf{M_o}$ and $\mathsf{M_n}$.

Now assume that two points x and y are in different faces in $\mathsf{M_c}$. Let F_x and F_y be the faces containing x and y in $\mathsf{M_c}$, respectively. This means that x is not enclosed by F_y or y is not enclosed by F_x. The outer boundaries of F_x and F_y are distinct. By Lemma 11, there are faces F'_x and F'_y in $\mathsf{M_o}$ or in $\mathsf{M_n}$ whose outer boundaries coincide with the outer boundaries of F_x and F_y, respectively. Note that x is enclosed by F'_x and y is enclosed by F'_y. However, either x is not enclosed by F'_y or y is not enclosed by F'_x. Therefore, x and y are in different faces in $\mathsf{M_o}$ or $\mathsf{M_n}$, which proves the lemma.   ◀

Lemma 12 immediately gives an $O(Q(n))$-time query algorithm. We represent the name of each face in $\mathsf{M_c}$ by the pair consisting of two faces, one from $\mathsf{M_o}$ and one from $\mathsf{M_n}$, corresponding to it. In this way, we have the following lemma.

▶ **Lemma 13.** *We can answer any query* LOCATE(x) *in $O(Q(n))$ time using $\mathsf{D_o}$ and $\mathsf{D_n}$.*

By setting $f(n) = \sqrt{n \cdot U(n)/Q(n)}$, we have the following theorem.

▶ **Theorem 14.** *We can construct a data structure of size $O(S(n))$ so that* LOCATE(x) *can be answered in $O(Q(n))$ time for any point x in the plane. Each update,* INSERTEDGE(e) *or* DELETEEDGE(e), *can be processed in $O(\sqrt{n \cdot U(n) \cdot Q(n)})$ amortized time, where n is the number of edges at the moment.*

Using the data structure by Chan and Nekrich [5], we set $S(n) = n$, $Q(n) = \log n(\log \log n)^2$ and $U(n) = \log n \log \log n$.

▶ **Corollary 15.** *We can construct a data structure of size $O(n)$ so that* LOCATE(x) *can be answered in $O(\log n(\log \log n)^2)$ time for any point x in the plane. Each update,* INSERTEDGE(e) *or* DELETEEDGE(e), *can be processed in $O(\sqrt{n} \log n(\log \log n)^{3/2})$ amortized time, where n is the number of edges at the moment.*

──── **References** ────

1   Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., 1974.
2   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.
3   Hanna Baumgarten, Hermann Jung, and Kurt Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17(3):342–380, 1994.
4   Jon Louis Bentley and James B Saxe. Decomposable searching problems 1: Static-to-dynamic transformations. *Journal of Algorithms*, 1(4):301–358, 1980.
5   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.

6   Siu-Wing Cheng and Ravi Janardan. New results on dynamic planar point location. *SIAM Journal on Computing*, 21(5):972–999, 1992.

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

8   Jiří Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8(3), 1992.

9   Mark H. Overmars. Range searching in a set of line segments. Technical report, Rijksuniversiteit Utrecht, 1983.

10 Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problem. *Information Processing Letters*, 12(4):168–173, 1981.

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

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

13 Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.