A Space-Efficient Framework for Dynamic Point Location

Meng He[1], Patrick K. Nicholson[2], and Norbert Zeh[1]

[1] Faculty of Computer Science, Dalhousie University, Canada
{mhe,nzeh}@cs.dal.ca
[2] Cheriton School of Computer Science, University of Waterloo, Canada
p3nichol@uwaterloo.ca

Abstract. Let G be a planar subdivision with n vertices. A succinct geometric index for G is a data structure that occupies $o(n)$ bits beyond the space required to store the coordinates of the vertices of G, while supporting efficient queries. We describe a general framework for converting dynamic data structures for planar point location into succinct geometric indexes, provided that the subdivision G to be maintained has bounded face size. Using this framework, we obtain several succinct geometric indexes for dynamic planar point location on G with query times matching the currently best (non-succinct) data structures and polylogarithmic update times.

1 Introduction

Many fundamental problems in computational geometry involve constructing data structures over large geometric data sets to support efficient queries on the data. Such queries include point location, ray shooting, nearest neighbour searching, as well as a plethora of range searching variants (see [5, 1]). Data structures supporting these types of queries provide the building blocks of software in many important application areas, such as geographic information systems, network traffic monitoring, database systems, computer aided design (e.g., very-large-scale integration), and numerous graphical applications.

One of the most heavily studied queries of this type is that of planar point location: Given an n-vertex planar subdivision, the goal is to support queries of the form, "Which region of the subdivision contains the query point?" Starting with Kirkpatrick's work [14], many linear-space data structures have been proposed to support planar point location queries in $O(\lg n)$ time[3], which is the asymptotically optimal number of point-line comparisons. Further research has focused on determining the exact number of point-line comparisons [12, 17], developing data structures that bound the query time based on the entropy of the query distribution [13], and exploiting word-RAM parallelism [8, 7].

Recently, Bose et al. [6] presented a space-efficient framework for planar point location in the word-RAM model. In this framework, the coordinates of

[3] We use $\lg n$ to denote $\lceil \log_2 n \rceil$.

Source	Model	Restrictions	Query	Insert	Delete
CJ92 [9]	PM	General	$O(\lg^2 n)$	$O(\lg n)$	$O(\lg n)$
BJM94 [4]	PM	General	$O(\lg n \lg\lg n)$	$O(\lg n \lg\lg n)^{\mathrm{a}}$	$O(\lg^2 n)^{\mathrm{a}}$
ABG06 [3]	PM	General	$O(\lg n)$	$O(\lg^{1+\varepsilon} n)^{\mathrm{a}}$	$O(\lg^{2+\varepsilon})^{\mathrm{a}}$
ABG06 [3]	RAM	General	$O(\lg n)$	$O(\lg n \lg^{1+\varepsilon} \lg n)^{\mathrm{a,p}}$	$O(\frac{\lg^{2+\varepsilon} n}{\lg\lg n})^{\mathrm{a,p}}$
GK09 [11]	RAM	Horizontal	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
N10 [15]	RAM	Horizontal	$O(\frac{\lg n}{\lg\lg n})$	$O(\lg^{1+\varepsilon} n)^{\mathrm{a}}$	$O(\lg^{1+\varepsilon} n)^{\mathrm{a}}$

Table 1. Previous results for linear-space data structures for dynamic planar point location. In the model column, "PM" denotes the pointer machine, and "RAM" the word-RAM model. In the restrictions column, "General" denotes an arbitrary planar subdivision, and "Horizontal" denotes point location among horizontal segments (vertical ray shooting). The letters "a" and "p" in the columns showing query, insertion, and deletion bounds indicate amortized bounds and high-probability bounds (i.e., probability $1 - O(1)/n^c$, for some $c \geq 1$), respectively. The value ε is any positive constant.

the n vertices of the subdivision are permuted and stored along with an auxiliary data structure called a *succinct geometric index*. The succinct geometric index occupies only $o(n)$ *bits*, which is asymptotically negligible compared to the space occupied by the coordinates. With only this index and access to the permuted sequence, they showed how to match the efficiency of many of the previous data structures for planar point location. Specifically, they presented several succinct geometric indexes that can answer point location queries in $O(\lg n)$ time; $O(H + 1)$ time, where H is the entropy of the query distribution; $O(\min\{\lg n / \lg\lg n, \sqrt{\lg U}\})$ time, if the coordinates are integers in the range $[1, U]$; and, finally, $\lg n + 2\sqrt{\lg n} + O(\lg^{1/4} n)$ point-line comparisons. Furthermore, they showed how to make their data structure *implicit*. Their implicit data structure uses only $O(1)$ words of space beyond the permuted sequence of coordinates for the vertices and supports point location queries in $O(\lg^2 n)$ time.

But what about the *dynamic case*, where we are allowed to modify the structure of the planar subdivision? Many data structures have been proposed to solve this problem, though the best choice depends on whether we desire fast queries, fast updates, worst-case behaviour, deterministic behaviour, or are operating on a restricted class of subdivisions. Table 1 summarizes the skyline results for dynamic planar point location. Note that we consider only linear-space data structures that are fully dynamic, i.e., support insertions and deletions. So far, no point location data structures have been proposed that are dynamic *and* use only $o(n)$ bits of memory beyond the space required for the coordinates of the vertices. Developing such structures is the focus of this paper.

1.1 Our Results

Our main result is a framework for creating succinct geometric indexes for dynamic point location in planar subdivisions with *bounded face size*: i.e., the

maximum number of vertices defining a face is a fixed constant that does not depend on n. This framework allows us to convert any existing linear-space data structure for dynamic planar point location into a succinct geometric index, subject to the constraint that the subdivision to be maintained have bounded face size[4]. The query times of our data structures can be set to match any of the data structures for general subdivisions from Table 1, since our framework introduces only an additive $o(\lg n)$ term to the query cost. Updates are supported in polylogarithmic time, where the exact update time depends on the choice of the underlying data structure. Our *update operations* allow for insertion and deletion of vertices and edges into/from G, with some restrictions that we describe in Section 3. We note that the types of update operations are similar to previous work [9], and complete in the sense that they allow the assembly and disassembly of any planar subdivision of bounded face size. All our results hold in the word-RAM model with word size $\Theta(\lg n)$ bits. The following theorem, which is a consequence of our main theorem (Theorem 3 on page 9), summarizes our contributions.

Theorem 1. *Let G be an n-vertex planar subdivision with bounded face size and each of whose vertices has coordinates occupying $M = O(\lg n)$ bits. For any constant $\varepsilon > 0$, there exists a data structure for dynamic planar point location in G that occupies $nM + o(n)$ bits and*

- *Supports queries in $O(\lg n)$ time and updates in $O(\lg^{3+\varepsilon} n)$ amortized time with high probability (using [3]),*
- *Supports queries in $O(\lg n \lg\lg n)$ time and updates in $O(\lg^{2+\varepsilon} n)$ amortized time (using [4]), or*
- *Supports queries in $O(\lg^2 n)$ time and updates in $O(\lg^{2+\varepsilon} n)$ worst-case time (using [9]).*

Techniques and Overview: Our point location framework is based on the two-level decomposition used in Bose et al.'s framework for obtaining succinct indexes for *static* planar point location [6]. This framework uses a two-level partition of the subdivision using planar separators. The main challenge in obtaining a *dynamic* framework based on these ideas is to maintain the separator decomposition under updates of the subdivision. Aleksandrov and Djidjev [2] introduced the P-tree, a linear-space data structure for maintaining planar graph partitions under updates of the graph. Our main technical contribution is to develop a succinct version of this data structure that requires only $o(n)$ bits of space beyond the space required to store the coordinates of the vertices of the graph. Obtaining this structure requires a non-trivial combination of the original P-tree data structure with the labelling scheme by Bose et al. [6]. While our motivation to develop this data structure was its importance as part of our point location framework, we expect it to be of independent interest, as graph partitions find applications in a wide range of algorithms.

[4] Previous results for general subdivisions, e.g., [9], do not have this constraint.

2 Definitions and Preliminaries

We require the following definitions, closely following Aleksandrov and Djidjev [2], but making some slight modifications. For a graph G, let $V(G)$ and $E(G)$ denote the vertex and edge sets of G, respectively. A *planar graph* G is any graph that can be embedded (drawn) in the plane so that its edges intersect only at their endpoints. A *straight-line embedding* of G represents each edge as a line segment. A *planar subdivision* is a straight-line embedding of a 2-edge-connected planar graph. The *faces* of the subdivision are the connected components of $\mathbb{R}^2 \setminus (V(G) \cup E(G))$. We denote the set of faces by $F(G)$, the number of faces by N, and the number of vertices by n.

A *region* R is any set of faces, and we use $G(R)$ to denote the subgraph spanned by the edges on their boundaries. An edge $e \in E(G(R))$ is a *boundary edge* of R if only one of the faces in $F(G)$ incident to e is in $G(R)$. All other edges of $G(R)$ are referred to as *interior* edges. The boundary of R, denoted ∂R, is the subgraph induced by the boundary edges of R. We say a region is *connected* if the dual of $G(R)$ is connected.

A *partition* $\mathcal{R} = \{R_1, ..., R_r\}$ of G is a set of regions such that each face in $F(G)$ appears in exactly one region in $\mathcal{R}$. A partition is *weakly connected* if every region is either connected or adjacent to (i.e., shares a boundary edge with) at most two other regions, in which case these two adjacent regions are connected. The boundary of partition $\mathcal{R}$, denoted $\partial\mathcal{R}$, is the the union of the boundaries ∂R_i, $1 \leq i \leq r$, of its regions.

Let G be a planar graph with N faces, and $\varepsilon > 0$. A partition $\mathcal{R}$ is an ε-partition of G if no region of $\mathcal{R}$ contains more than εN faces. In the remainder of this paper we deal with graphs whose face size is bounded by a constant. We note that the number of vertices, n, is $\Theta(N)$ in this case.

We now review some preliminary lemmas that we will use extensively. The following lemma is an extension of the result of [10], and is used throughout our data structures to save space.

Lemma 1 ([6], Lemma 4.1). *Given a planar subdivision of n vertices, for a sufficiently large n, there exists an algorithm that can encode it as a permutation of its point set in $O(n)$ time and such that the subdivision can be decoded from this permutation in $O(n)$ time.*

For a sequence S of length n, let $S[i]$ denote the ith symbol in S. We use $\texttt{rank}_b(S, i)$ to denote the frequency of symbol b in the prefix $S[1], ..., S[i]$. Similarly, we use $\texttt{select}_b(S, i)$ to denote the index j containing the ith occurrence of symbol b in S. We make use of the following lemma, which can be used to support `rank` and `select` operations on a *bit sequence*, while also compressing the sequence if it sparse.

Lemma 2 ([16]). *Given a sequence S of n bits, with m one bits 1, there exists a data structure that represents S using $m \lg(n/m) + 1.92m + o(m)$ bits and supports* `rank` *and* `select` *operation in $O(\lg(n/m) + (\lg^4 m)/\lg n)$ and $O((\lg^4 m)/\lg n)$ time, respectively. Construction of the data structure takes $O(n)$ time.*

3 P-Trees

The *P-tree*, introduced by Aleksandrov and Djidjev [2], is a dynamic data structure for maintaining ε-partitions of a planar graph under the following operations, assuming the face size is bounded by a constant d:

- `insert_vertex`(v, e) : Create a new vertex v and replace the edge $e = (u, w)$ with two new edges $e_1 = (u, v)$ and $e_2 = (v, w)$.
- `insert_edge`(u, w, f) : Insert a new edge $e = (u, w)$ across face f. Vertices u and w have to be on the boundary of face f.
- `delete_vertex`(v): Assuming v is a degree-2 vertex with neighbours u and w, delete v and replace the edges (u, v) and (v, w) with a single edge (u, w).
- `delete_edge`(e): Delete the edge $e = (u, w)$, assuming both u and w are of degree greater than two.
- `list_partition`(ε): Return an ε-partition of G.

As noted by Aleksandrov and Djidjev [2], these operations can be used to transform any planar graph into any other planar graph, as long as neither contains a vertex of degree less than two. Next we give a brief summary of this data structure. For full details, refer to [2].

The P-tree represents a hierarchy of graphs $G_0, G_1, \ldots, G_\ell$, where $G_0 = G$ and $|G_\ell| = O(1)$. For $1 \leq i \leq \ell$, G_i is obtained from G_{i-1} by computing a weakly connected h/N_{i-1}-partition $\mathcal{R}_{i-1}$ of G_{i-1}, where N_{i-1} is the number of faces of G_{i-1} and h is an appropriate constant. The faces of G_i represent the regions in this partition. Every edge of G_i represents a maximal list of boundary edges on the boundaries of the two adjacent regions. Every face f of G_i has the faces of G_{i-1} in the corresponding region of $\mathcal{R}_{i-1}$ as its children in the P-tree and stores a pointer to the list of edges on its boundary. Every edge of G_i stores pointers to the edges of G_{i-1} it represents. By following these pointers recursively, every face f of G_i represents a collection of faces of G, a region $R(f)$, and every edge of G_i represents a collection of edges of G. The edges of G corresponding to the edges on the boundary of a face f of G_i are exactly the boundary edges of the region $R(f)$. We define the *cost* of an edge of G_i as the number of edges in G it represents. For a P-tree T, we use $\mathcal{R}(T, i)$ to denote the partition $\{R(f_1), ..., R(f_r)\}$, where $f_1, ..., f_r$ are the faces of G_i, that is, the faces represented by nodes in T at distance i from the leaf level.

We call a node z of a P-tree T with children $z_1, z_2, \ldots, z_q$ *balanced* if it satisfies three properties:

(B1) $z, z_1, z_2, \ldots, z_q$ have at most h children each, for an appropriate constant h.
(B2) If $\#\mathtt{c}(z)$ and $\#\mathtt{g}(z)$ respectively denote the number of children and the number of grandchildren of z, then $\#\mathtt{c}(z)/\#\mathtt{g}(z) \leq c/h$, for an appropriate constant $c \leq h/2$.
(B3) Let k be the level of z in the P-tree, i.e., its distance from the leaf level. Then the total cost of all edges on z_i's boundary, for every $1 \leq i \leq q$, is at most $dh^{(k-1)/2}$.

We note that all nodes of the P-tree are balanced after applying the described construction algorithm. Furthermore, using these balancing conditions, it is easy to prove the following lemma.

Lemma 3. *Let $1 \leq k \leq \ell$ and $\varepsilon_k = h^k/N$, where N is the number of faces in G. The partition $\mathcal{R}(T,k)$ is an ε_k-partition of G with boundary size not exceeding $d\sqrt{(Nc^{2k})/\varepsilon_k}$.*

By Lemma 3, a `list_partition`(ε) query amounts to finding the right level in the P-tree and reporting the regions corresponding to the nodes at this level and their boundaries.

The update operations supported by the P-tree may create or destroy a constant number of leaves (faces of G) and change the boundary of a constant number of leaves. For example, `insert_edge` increases the number of leaves by one. This may affect the balancing of the ancestors of these leaves. In order to rebalance the tree, the path from each such leaf to the root is traversed and every unbalanced node z is rebalanced using the following lemma.[5]

Lemma 4. *Let G be a planar graph with N faces, assume the edges of G have associated costs, and assume the total cost of the edges bounding each face is bounded by some parameter b. Then there exists a weakly connected h/N-partition $\mathcal{R} = \{R_1, ..., R_r\}$ such that the cost of each region's boundary is at most $b\sqrt{h}$ and $r \leq cN/h$, for some constant c. Furthermore, $\mathcal{R}$ can be constructed in $O(N \lg N)$ time.*

We apply this lemma to every unbalanced node z we encounter. Since we rebalance nodes in a bottom-up fashion, z can violate condition (B1) only if one of its children has too many children. While rebalancing, we maintain the invariant that every node, balanced or not, has at most $2ch$ children and every node below the current node is balanced. Thus, once we are done processing the root, the entire tree is balanced.

Now consider an unbalanced level-k node z with at most h children $z_1, \ldots, z_q$ that each satisfy the conditions for z to be balanced, except one which may have between h and $2ch$ children or boundary cost greater than $dh^{(k-1)/2}$. Moreover, z may violate condition (B2). To rebalance z, we consider the constant-size subgraph of G_{k-2} consisting of z's grandchildren and their boundaries, and partition it into subgraphs with at most h faces each. Each region in this partition becomes a new face of G_{k-1}, and these faces become the new children of z. By Lemma 4, this restores condition (B2), the part of condition (B1) bounding the number of children of each child of z and, since each grandchild of z has boundary cost at most $b \leq dh^{(k-2)/2}$, condition (B3). Since z has at most h children before rebalancing, one of which has up to $2ch$ children, while all others have at most h children, z has less than $h^2 + 2ch \leq 2h^2$ grandchildren. Thus,

[5] In [2, Theorem 1], this result was stated incorrectly, but apparently the remainder of the paper applied it correctly. The running time stated in the lemma can be reduced to $O(N)$, but it would have no effect on our data structure and would require a significantly more tedious analysis.

by Lemma 4, we partition the subgraph defined by these grandchildren into at most $2h^2 \cdot c/h \leq 2ch$ regions, each of which becomes a child of z. Thus, z satisfies the upper bound on its number of children necessary to proceed to rebalancing its parent. Since each rebalancing operation works with a graph of constant size and the height of the tree is $O(\lg N)$, each update takes $O(\lg N)$ time.

Based on Lemmas 3 and 4, we get the following corollary:

Corollary 1. *Let $h \geq c^\alpha$, for some $\alpha > 2$, and let $k = \log_h \lg^\lambda N$, for some $\lambda > 0$. The partition $\mathcal{R}(T, k)$ is a $(\lg^\lambda N/N)$-partition with boundary size that does not exceed $dN/\lg^{\lambda/2-\lambda/\alpha} N$. Furthermore, the number of regions in $\mathcal{R}(T, k)$ does not exceed $N/\lg^{\lambda-\lambda/\alpha} N$.*

4 Succinct P-Trees

In this section we introduce the *succinct P-tree*, which uses only $nM + o(n)$ bits of space to represent an n-vertex planar graph whose vertices have M-bit coordinates in the plane. The price we pay for this space reduction is a polylogarithmic slowdown in update time.

The main idea of the succinct P-tree is to prune all nodes below level $\log_h \lg^\lambda N$ in the tree, for some $\lambda > 0$, referred to as the *pruning level*. We call the nodes at the pruning level *pruned nodes* and the nodes above the pruning level *internal nodes*. By Corollary 1, there are $r \leq N/\lg^{\lambda-\lambda/\alpha} N$ pruned nodes. We refer to the regions defined by the pruned nodes as *pruned regions*.

Suppose we apply Lemma 1 to a pruned subgraph $G(R(z))$, obtaining a permutation of the coordinates of its vertices that uniquely identifies the structure of $G(R(z))$. Storing this permuted sequence of coordinates in the nodes at the pruning level allows us to perform operations as in the original P-tree, at the cost of decoding and re-encoding a subgraph of size $O(\lg^\lambda N)$ during each update. However, explicitly storing this permutation in each pruned node causes the coordinates of the boundary vertices to be duplicated in several pruned regions. To avoid this, we adapt the labelling scheme of Bose et al. [6] to the dynamic setting. The details are as follows.

Data structures of pruned nodes: Let π_z denote the permutation of vertices obtained by applying Lemma 1 to $G(R(z))$, for a fixed pruned node $z \in T$. We denote the ith vertex in the permutation as $\pi_z(i)$. We separate the vertices into two categories: vertices that are endpoints of edges on the boundary of the pruned region are *boundary vertices*; all other vertices are *interior vertices*. Every interior vertex belongs to exactly one pruned region. Every boundary vertex belongs to more than one pruned region. Each pruned node z now stores the following data structures, where $\partial R(z)$ and n_z denote the boundary of and the number of vertices in $R(z)$, respectively.

- A binary sequence B_z, where $B_z[j] = 0$ if $\pi_z(j)$ is an interior vertex, and $B_z[j] = 1$ if $\pi_z(j)$ is a boundary vertex of $R(z)$. We represent B_z using the data structure of Lemma 2.

– An array I_z that stores the coordinates for interior vertices. Entry $I_z[j]$ stores the coordinates for the vertex $\pi_z(\texttt{select}_0(B_z, j))$, for $1 \leq j \leq \texttt{rank}_0(B_z, n_z)$.
– An array X_z that stores pointers to *records* external to node z. For $1 \leq j \leq \texttt{rank}_1(B_z, n_z)$, the record pointed to by entry $X_z[j]$ stores the coordinates of vertex $\pi_z(\texttt{select}_1(B_z, j))$: the jth boundary vertex in $R(z)$.
– An array E_z that stores pointers to records representing the edges on the boundary of $\partial R(z)$. The ordering of E_z is any canonical ordering based on the permutation π_z. For example, we can order the edges lexicographically by the positions of their endpoints in π_z. Let $\mathcal{C}$ be the record representing the jth boundary edge e of $R(z)$ in this ordering, pointed to by entry $E_z[j]$. $\mathcal{C}$ stores the indices of e's endpoints in arrays X_z, as well as a pointer to z. Furthermore, $\mathcal{C}$ stores the symmetric information about the other pruned region $R(z')$ that has e on its boundary.

Each internal node stores the same information as in a standard P-tree. Using Corollary 1 and Lemma 2, we can bound the space occupied by the succinct P-tree data structure as follows.

Lemma 5. *The succinct P-tree occupies $nM + O(n/\lg^{\lambda/2-\beta-1} n)$ bits, for any constant $\beta > 0$.*

We next state the following lemma about supporting updates. Intuitively, the idea is to simulate the operation of a standard P-tree. Above the pruning level, each update operation proceeds identically as if it were run on a standard P-tree and, thus, takes $O(\lg n)$ time. In order to implement the portion of the update operation that operates on nodes below the pruning level, we reconstruct the affected pruned region from its succinct representation and build a P-tree from it. This takes $O(\lg^{\lambda} n \lg\lg n)$ time. Since each update affects only a constant number of pruned regions, the lemma follows.

Lemma 6. *A succinct P-tree supports the operations* `insert_vertex`, `insert_edge`, `delete_vertex`, *and* `delete_edge` *in $O(\lg^{\lambda} n \lg\lg n)$ time.*

Combining Lemmas 5 and 6, and setting $\lambda > 2$ leads us to our main theorem of this section.

Theorem 2. *Let G be a planar subdivision with n vertices and N faces, where each face has at most d vertices, for some constant $d \geq 3$. Each vertex is assumed to store M-bit coordinates. Let ε be any positive constant. There exists a data structure representing G in $nM + o(n)$ bits of space that can perform the operations* `insert_vertex`, `delete_vertex`, `insert_edge`, *and* `delete_edge` *in $O(\lg^{2+\varepsilon} n)$ time. The operation* `list_partition`(ε') *can be performed in time proportional to the partition's size, for $\lg^{2+\varepsilon} N/N \leq \varepsilon' \leq 1$. The boundary size of the partition returned by* `list_partition`(ε') *does not exceed $d\sqrt{(N^{1+\delta})/(\varepsilon'^{1-\delta})}$, where $\delta > 0$ is an arbitrarily small constant but depends on our choice of ε.*

5 Dynamic Planar Point Location

As an application of Theorem 2, we develop a succinct geometric index for dynamic planar point location. To do this, we add extra data structures to the pruned nodes of the succinct P-tree. These extra data structures are analogous to the *subregion* level data structures of the two-level index of Bose et al. [6].

Let γ be a positive constant in the range $(0, 2]$. We refer to the regions in $\mathcal{R}(T, \log_h \lg^\gamma n)$ as *γ-subregions*. Since $\lambda > 2$, each γ-subregion is contained in a pruned region and thus is not accessible without decoding this pruned region. The next lemma states that we can augment the succinct P-tree to provide efficient access to γ-subregions.

Lemma 7. *The succinct P-tree can be augmented to support extraction of the graph structure of an arbitrary γ-subregion (within a specified pruned region) in $O(\lg^\gamma(n)\texttt{polyloglog}(n))$ time, where $\gamma \in (0, 2]$ is fixed at construction time. The space bound becomes $nM + O((n \lg\lg n)/\lg^{\gamma/2-\gamma/\alpha} n)$, which is $o(n)$ for $\gamma \in (0, 2]$ and $\alpha > 2$, and the update costs remain as stated in Theorem 2.*

We now sketch how to use γ-subregion extraction to efficiently support point location queries, resulting in our main theorem:

Theorem 3. *Let $\mathcal{D}$ be a dynamic point location data structure that uses $O(n)$ words of space to store an n-vertex subdivision and supports queries and updates on this subdivision in $Q(n)$ and $U(n)$ time, respectively. We assume $Q(n) + U(n) = O(\texttt{polylog}(n))$. Let G be a planar subdivision, each of whose faces has a constant number of vertices, and assume the coordinates of each vertex can be stored in M bits. Let $\varepsilon > 0$ be any positive constant, and choose any constant $\gamma \in (0, 2]$. There exists a data structure for dynamic planar point location that occupies $nM + o(n)$ bits of space, supports queries in $O(\lg^\gamma(n)\texttt{polyloglog}(n) + Q(n))$ time, and supports the operations* `insert_vertex`, `delete_vertex`, `insert_edge`, *and* `delete_edge` *in $O(U(n) \lg^{1+\varepsilon/2} n + \lg^{2+\varepsilon} n)$ time.*

Proof (Sketch). We maintain G in a succinct P-tree T, augmented as in Lemma 7. The planar subdivision defined by the boundaries of the pruned regions of G is stored in the data structure $\mathcal{D}$. This is the *first level* point location structure. Inside each pruned node z, we also store another instance of $\mathcal{D}$, denoted $\mathcal{D}_z$. This *second-level* point location structure stores the planar subdivision defined by the boundaries of the γ-subregions contained in $R(z)$. Each face f of this subdivision corresponds to a connected component of a γ-subregion S_i. We store its index i with f. To answer a query for a point p, we first query $\mathcal{D}$ to identify the pruned region $R(z)$ that contains p. Next we query $\mathcal{D}_z$ to identify the γ-subregion S_i that contains p. Finally, we extract S_i and perform a brute-force search to find the face of S_i that contains p. The queries to $\mathcal{D}$ and $\mathcal{D}_z$ require $Q(n)$ time, and the γ-subregion extraction step requires $O(\lg^\gamma(n)\texttt{polyloglog}(n))$ time, by Lemma 7. The total query time is therefore $O(Q(n) + \lg^\gamma(n)\texttt{polyloglog}(n))$. The key idea of achieving the claimed update time is to use condition (B3) to bound the number of edges and vertices in D that are changed by an update. □

By choosing the data structures from Table 1 as $\mathcal{D}$ in the previous theorem, and setting γ appropriately, we get the result of Theorem 1.

References

1. Agarwal, P., Erickson, J.: Geometric range searching and its relatives. Contemporary Mathematics 223, 1–56 (1999)
2. Aleksandrov, L., Djidjev, H.N.: A dynamic algorithm for maintaining graph partitions. In: Proc. SWAT, LNCS, vol. 1851, pp. 3–30. Springer (2000)
3. Arge, L., Brodal, G., Georgiadis, L.: Improved dynamic planar point location. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science. pp. 305–314. IEEE Computer Society (2006)
4. Baumgarten, N., Jung, H., Mehlhorn, K.: Dynamic point location in general subdivisions. Journal of Algorithms 17(3), 342–380 (1994)
5. Berg, M.d., Cheong, O., Kreveld, M.v., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer, Santa Clara, CA, USA, 3rd edn. (2008)
6. Bose, P., Chen, E., He, M., Maheshwari, A., Morin, P.: Succinct geometric indexes supporting point location queries. ACM Trans. on Algorithms 8(2), 10 (2012)
7. Chan, T.M.: Persistent predecessor search and orthogonal point location on the word ram. In: SODA. pp. 1131–1145 (2011)
8. Chan, T.M., Patrascu, M.: Transdichotomous Results in Computational Geometry, I: Point Location in Sublogarithmic Time. SIAM J. Comput. 39(2), 703–729 (2009)
9. Cheng, S., Janardan, R.: New results on dynamic planar point location. SIAM Journal on Computing 21, 972 (1992)
10. Denny, M., Sohler, C.: Encoding a triangulation as a permutation of its point set. In: Proc. CCCG (1997)
11. Giora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. ACM Transactions on Algorithms (TALG) 5(3), 28 (2009)
12. Goodrich, M., Orletsky, M., Ramaiyer, K.: Methods for achieving fast query times in point location data structures. In: Proc. SODA. pp. 757–766. SIAM (1997)
13. Iacono, J.: A static optimality transformation with applications to planar point location. In: Symposium on Computational Geometry. pp. 21–26 (2011)
14. Kirkpatrick, D.: Optimal search in planar subdivisions. SIAM J. Comput. 12(1), 28–35 (1983)
15. Nekrich, Y.: Searching in dynamic catalogs on a tree. Arxiv preprint arXiv:1007.3415 (2010)
16. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: ALENEX (2007)
17. Seidel, R., Adamy, U.: On the exact worst case query complexity of planar point location. Journal of Algorithms 37(1), 189–217 (2000)