I/O-Efficient Dynamic Point Location in Monotone Planar Subdivisions

(Extended Abstract)

Pankaj K. Agarwal* Lars Arge† Gerth Stølting Brodal‡ Jeffrey S. Vitter§

Abstract

We present an efficient external-memory dynamic data structure for point location in monotone planar subdivisions. Our data structure uses $O(N/B)$ disk blocks to store a monotone subdivision of size N, where B is the size of a disk block. It supports queries in $O(\log_B^2 N)$ I/Os (worst-case) and updates in $O(\log_B^2 N)$ I/Os (amortized).

We also propose a new variant of B-trees, called *level-balanced B-trees*, which allow insert, delete, merge, and split operations in $O((1+\frac{b}{B}\log_{M/B}\frac{N}{B})\log_b N)$ I/Os (amortized), $2 \le b \le B/2$, even if each node stores a pointer to its parent. Here M is the size of main memory. Besides being essential to our point-location data structure, we believe that *level-balanced B-trees* are of significant independent interest. They can, for example, be used to dynamically maintain a planar st-graph using $O((1+\frac{b}{B}\log_{M/B}\frac{N}{B})\log_b N) = O(\log_B^2 N)$ I/Os (amortized) per update, so that reachability queries can be answered in $O(\log_B N)$ I/Os (worst case).

1 Introduction

Planar point location, a widely studied problem in computational geometry, is defined as follows: Given a planar subdivision Π with N vertices (i.e., a decomposition of the plane into polygonal regions induced by a straight-line planar graph), preprocess Π into a data structure so that the face of Π containing a query point can be reported quickly. This problem arises in several applications, including graphics, spatial databases, and geographic information systems. The planar subdivisions arising in many of these applications are too massive to fit in internal memory and must reside on disk. In such instances, the I/O communication is the bottleneck instead of the CPU running time. Most of the work to date, especially when we also allow to change the edges and vertices of Π dynamically, has focused on minimizing the CPU running time under the assumption that the subdivision fits in main memory [7, 9, 10, 14, 15, 22]. Only a few static (or batched dynamic) results are known for I/O-efficient point location when the subdivision is stored in external memory [5, 16, 25].

In this paper we develop the first space- and I/O-efficient dynamic data structure for planar point location in monotone subdivisions. We also propose a variant of B-trees in which each node stores a pointer to its parent and show that insert, delete, merge, and split operations can be performed efficiently. This structure is of independent interest and can, for example, be used to obtain I/O-efficient dynamic data structure for answering various queries in planar *st*-graphs.[1]

1.1 Previous results

A polygon is called *monotone* in direction θ if any line in direction $\pi/2+\theta$ intersects the polygon in a connected interval; a convex polygon is monotone in every direction. A planar subdivision Π is *monotone* if all faces of Π are monotone in a fixed direction. Every convex subdivision is a monotone subdivision. In the remainder of this paper we use the term a monotone subdivision to denote a subdivision that is monotone in the x-direction.

In internal memory, Edelsbrunner *et al.* [14] proposed an optimal data structure for point location in monotone subdivisions with $O(N)$ space, $O(N)$ preprocessing time, and $O(\log N)$ query time. For arbitrary planar subdivisions, the preprocessing time is $O(N \log N)$; see also [22]. If we allow the edges and

* Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708. Supported in part by National Science Foundation research grants CCR–93–01259 and EIA–9870724, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corporation, and by a grant from the U.S.-Israeli Binational Science Foundation. Email: pankaj@cs.duke.edu.

† Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708. Supported in part by the Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation ESS grant EIA–9870724. Email: large@cs.duke.edu.

‡ BRICS, University of Aarhus, DK-8000 Aarhus C., Denmark. Supported by the Carlsberg foundation under grant 96-0302/20. Partially supported by the ESPRIT Long Term Research Program of the EU under contract 20244 (project ALCOM-IT). Part of this work was done while at Max-Planck-Institut für Informatik, Saarbrücken, Germany. Email: gerth@brics.dk.

§ Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708. Supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation grants CCR–9522047 and EIA–9870724. Email: jsv@cs.duke.edu.

[1] A planar directed acyclic graph, with a given embedding, is called an *st-graph* if it has unique source and sink vertices lying on the boundary of the same face.

vertices to be changed dynamically, two linear-space structures are known for general subdivisions: one by Cheng and Janardan [9] that answers queries in $O(\log^2 N)$ time and supports updates in $O(\log N)$ time; the other by Baumgarten *et al.* [7] that supports queries in $O((\log N)\log\log N)$ time (worst-case), insertions in $O((\log N)\log\log N)$ time (amortized), and deletions in $O(\log^2 N)$ time (amortized). Both structures store the edges of the subdivision in an interval tree [13] constructed on their x-projection (as first suggested in [15]) and use this structure to answer *vertical ray-shooting queries*: for a query point p, find the first edge, if any, of Π hit by the ray emanating from p in the $(+y)$-direction. The face containing p can then be found in $O(\log N)$ time [20]. A summary of known results can be found in the recent survey [23].

In this paper we are interested in the problem of dynamically maintaining a monotone subdivision on disk, so that the number of I/O operations (or *I/Os*) used to perform a query or an update is minimized. We consider the problem in the standard two-level I/O model proposed by Aggarwal and Vitter [1]. In this model N denotes the number of elements in the problem instance, M is the number of elements fitting in internal memory, and B is the number of elements per disk block, where $M < N$ and $2 \leq B \leq M/2$. An I/O is the operation of reading (or writing) a disk block from (or into) external memory. Computations can only be done on elements present in internal memory. Our measures of performance are the number of I/Os used to solve a problem and the amount of space (disk blocks) used.

Aggarwal and Vitter [1] considered sorting and related problems in the I/O model and proved that sorting requires $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os. Subsequently, I/O-efficient algorithms and data structures have been developed for numerous problems—see recent surveys for a sample of these results [2, 3, 26]. All previous results on point location in external memory have been either static or batched dynamic: Goodrich *et al.* [16] designed a static data structure using $O(N/B)$ space to store a monotone subdivision of size N so that a query can be answered in optimal $O(\log_B N)$ I/Os. They also developed a structure for answering a batch of K point-location queries in optimal $O(\frac{N+K}{B}\log_{M/B} N)$ I/Os. Arge *et al.* [5] extended the batched result to general subdivisions and Arge *et al.* [4] to an off-line dynamic setting in which a sequence of queries and updates are given in advance and all the queries should be answered as the sequence of operations are performed. Vahrenhold and Hinrichs considered the problem under some practical assumptions about the input data [25].

1.2 Our results

In this paper we present the first provably I/O-efficient dynamic data structure for point location in a monotone planar subdivision Π. Our structure uses $O(N/B)$ disk blocks to store Π, answers queries in $O(\log_B^2 N)$ I/Os in the worst-case, and inserts/deletes edges and vertices in $O(\log_B^2 N)$ I/Os amortized per edge/vertex. Here we assume that an update operation is admissible only if the subdivision remains planar and monotone after the operation. Our algorithm first detects whether an update operation is admissible and carries it out only if it is.

In order to answer the queries efficiently, we introduce a total order $\prec_\Pi$ on the edges of Π, as in [21, 24]. As edges are inserted or deleted the order may change considerably. We maintain the order in a separate B-tree-like data structure using *split* and *merge* operations. Each node of this structure stores a pointer to its parent. Although merge and split operations on standard B-trees can be performed in $O(\log_B N)$ I/Os, updating the parent pointers requires $\Omega(B\log_B N)$ I/Os. We therefore introduce a new variant of B-trees called *level-balanced B-trees* in which parent pointers can be maintained efficiently. For $2 \leq b \leq B/2$, level-balanced B-trees use $O(N/B)$ blocks to store N elements, and support insert, delete, merge, and split operations in $O((1+\frac{b}{B}\log_{M/B}\frac{N}{B})\log_b N) = O(\log_B^2 N)$ I/Os amortized.

We believe that the level-balanced B-trees are of significant independent interest. They can, for example, be used to dynamically maintain a planar st–graph using $O((1+\frac{b}{B}\log_{M/B}\frac{N}{B})\log_b N) = O(\log_B^2 N)$ I/Os amortized per update, so that *reachability queries* (of the form "is there a path from x to y?") can be answered in $O(\log_B N)$ I/Os worst-case.

2 Static Point Location

In this section we present a static data structure for point location, which we will dynamize in the next section. Let Π be a monotone subdivision with N vertices. We assume that all vertices in Π have distinct x-coordinates, and that there are no unbounded edges in Π. Our structure can easily be extended to handle subdivisions in which these assumptions do not hold. We use s and t to denote the leftmost and the rightmost vertices of Π. We present a data structure for the vertical ray-shooting problem: Preprocess Π into a data structure so that the first edge of Π, if any, hit by a query ray in the $(+y)$-direction emanating from a query point p can be reported efficiently. As in internal memory, the face containing p can easily be found once the ray-shooting query is answered. Our data structure extends to an arbitrary set of disjoint segments, but we focus on monotone subdivisions because at present we do not know how to dynamize the structure for arbitrary segments.

2.1 Ordering the edges

We first define a total order on the edges of Π, originally introduced by Tamassia and Preparata [24], which will be crucial for our structure. We regard Π as a directed planar st-graph with the edges of Π directed from left to right. We define the dual graph of Π, denoted by Π^*, to be the directed planar graph in which there is a vertex f^* for every bounded face f of Π and two vertices s^* and t^* for the unbounded face. For every edge $e \in \Pi$, adjacent to two faces f_1 and f_2 with f_1 lying below f_2, we add the edge $e^* = (f_1^*, f_2^*)$ in Π^*; if f_1 (resp., f_2) is the unbounded face, then the edge in Π^* is (s^*, f_2^*) (resp., (f_1^*, t^*)). See Figure 1 a) and b).

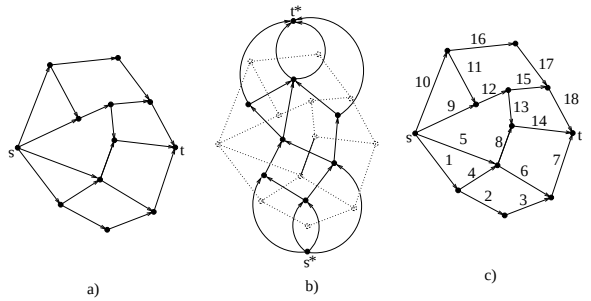

Figure 1: a) Π. b) Dual Π^*. c) Order of edges in Π.

We say that there is a *path* from an edge e to another edge e' in Π if there is a path $e = e_1, e_2, \ldots, e_l = e'$ in Π. It can be shown that there is a path from e_i to e_j in Π *if and only if* there is no path from e_i^* to e_j^* in Π^*; see e.g. [24]. For a pair $e_i, e_j \in \Pi$ of edges, we say $e_i \prec_\Pi e_j$ if either there is a path from e_i to e_j in Π or there is a path from e_i^* to e_j^* in Π^*. See Figure 1 c).

Lemma 1 (Tamassia-Preparata [24])

(i) $\prec_\Pi$ *defines a total order on the edges of* Π.

(ii) Let E *be a subset of the edges of* Π *so that all segments in* E *intersect a vertical line* ℓ. *If* E *is sorted according to* $\prec_\Pi$ *then* ℓ *intersects the edges in* E *in sorted order.*

2.2 Overall structure

In the following we make frequent use of (a, b)-trees [18]. In (a, b)-trees the leaves are all on the same level and they contain the elements stored in the structure. All internal nodes (except possibly the root) have between a and b children. In most of this paper, $a, b = \Theta(B^c)$ for some constant $0 < c \leq 1$, and each leaf contains $\Theta(B)$ data elements. Such a structure storing N elements is thus a $\Theta(B^c)$-ary tree over $O(N/B)$ leaves. Each leaf as well as internal node fits in one disk block, and thus the tree occupies $O(N/B)$ blocks. The height of the tree is $O(\log_{B^c} N) = O(\log_B N)$. Insert, delete, and search operations can be performed in $O(\log_B N)$ I/Os [18]. A normal B-tree [8, 11] is just such a structure with $c = 1$. For $c = 1/2$ we call the structure a $\sqrt{B}$-tree.

To simplify the presentation we assume without loss of generality that $N = B^{k/2}$ for some integer $k > 0$. Let S be the set of edges in Π. Our point-location structure is a two-level tree structure similar to the external interval tree, developed by Arge and Vitter [6]. The first level, called the *base tree*, is a $\sqrt{B}$-tree T over the x-coordinates of the endpoint of the segments in S. The segments in S are stored in secondary structures associated with the nodes of T. Each node v of T is associated with a vertical *slab* s_v; the root is associated with the whole plane. For each interior node v, s_v is partitioned into $\sqrt{B}$ vertical slabs $s_1, \ldots, s_{\sqrt{B}}$, separated by vertical lines which we call *slab boundaries* (the dotted lines in Figure 2), so that each slab contains the same number of vertices of Π. Here s_i is the slab associated with the i-th child of v. A segment e of S is stored at the highest node v of T at which it intersects a slab boundary associated with v. Let $S_v \subseteq S$ be the set of segments stored at v. A leaf z stores segments whose both endpoints lie in the interior of the slab s_z. The number of segments stored in a leaf is less than $3B - 6$ and they can thus be stored in at most three blocks.

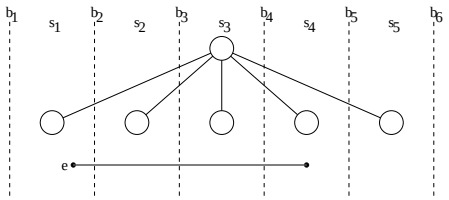

Figure 2: A node in the base tree. For segment e we have $l = 1$, $r = 4$. Its middle segment spans s_2 and s_3.

Let v be an internal node of T, let e be a segment of S_v, and suppose the left (resp., right) endpoint of e lies in the vertical slab s_l (resp., s_r) associated with v. We call the subsegment $e \cap s_l$ the *left* subsegment and $e \cap s_r$ the *right* subsegment of e. If $r > l + 1$, then the portion of e lying in $s_{l+1}, \ldots, s_{r-1}$ is called the *middle* subsegment; if $r = l + 1$, then the middle subsegment is the intersection point of e with the common boundary of s_l and s_r. See Figure 2. Let $\mathcal{M}$ denote the set of middle subsegments of segments in S_v. For each $1 \leq i \leq \sqrt{B}$, let L_i (resp., R_i) denote the set of left (resp., right) subsegments that lie in the slab s_i. We store the following secondary structures at v.

(i) A *multislab structure* Ψ_v on the set of middle segments $\mathcal{M}$ that requires $O(|\mathcal{M}|/B)$ disk blocks.

(ii) For each $i \leq \sqrt{B}$, we have the following two structures:

- A *left structure* $\mathcal{L}_i$ on all segments of L_i;

– A *right structure* $\mathcal{R}_i$ on all segments of R_i.

$\mathcal{L}_i$ and $\mathcal{R}_i$, over all slabs of v, require a total of $O(|S_v|/B)$ blocks.

A segment in S_v is thus stored in at most three secondary structures: the multislab structure, a left structure, and a right structure. For example the segment e in Figure 2 is stored in the multislab structure, the left structure of s_1, and in the right structure of s_4. Each node v requires $O(|S_v|/B)$ space, therefore the overall data structure requires $O(N/B)$ disk blocks.

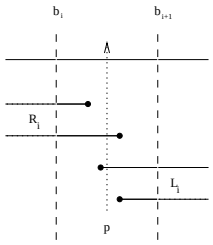

Figure 3: Answering a query.

Let ρ^+ be the ray emanating from a point p in the $(+y)$-direction. To find the first segment of S hit by ρ^+, we search T along a path from the root to a leaf z, such that S_z contains p, using $O(\log_B N)$ I/Os. The first segment of S_z hit by ρ^+ is computed by testing all segments of S_z explicitly. At each internal node v visited by the query procedure, we compute the first segment of S_v hit by ρ^+. In particular, we first search Ψ_v to find the first segment of $\mathcal{M}$ hit by ρ^+. Next, we find the vertical slab s_i that contains p and search $\mathcal{L}_i$ and $\mathcal{R}_i$ to find the first segments of L_i and R_i, respectively, that intersect ρ^+ (refer to Figure 3). We will show below that each of the three operations can be performed in $O(\log_B N)$ I/Os. By repeating this procedure for each node and choosing the first segment hit by ρ^+ among these $O(\log_B N)$ segments, we can answer a query in $O(\log_B^2 N)$ I/Os.

Theorem 1 *A monotone planar subdivision Π with N vertices can be stored in a data structure using $O(N/B)$ disk blocks, so that a vertical ray-shooting query can be answered in $O(\log_B^2 N)$ I/O operations.*

If S is an arbitrary set of disjoint segments, we can define another ordering on the segments of S, as in [17], that has essentially the same property as the $\prec_\Pi$-ordering, and prove the following (details omitted due to space constraints).

Theorem 2 *A set S of N disjoint segments can be stored in a data structure using $O(N/B)$ disk blocks, so that a vertical ray-shooting query can be answered in $O(\log_B^2 N)$ I/O operations.*

2.3 Left/right structures

In this subsection we sketch how to construct the left and right structures. These structures use ideas similar to the ones used by Cheng and Janardan [9]. We only describe how to construct and query a left structure; the right structure is analogous.

Let L be a set of K segments so that the right endpoints of all segments in L have the same x-coordinate, i.e., all of them lie on a vertical line. We will use L to denote the sequence of segments sorted by y-coordinate of their right endpoints. We construct a B-tree $\mathcal{L}$ on L. For each node v of $\mathcal{L}$, let $L_v \subseteq L$ be the set of segments stored in the subtree rooted at v. Let λ_v be the segment of L_v whose left endpoint has the minimum x-coordinate; λ_v is called the *minimal segment* of v. At each internal node v we store the minimal segments of all its children. Note that each node fits in $O(1)$ blocks and that the tree can be constructed (bottom-up) in $O(K/B)$ I/Os assuming the segments in L are sorted.

Let p be a query point, and let ρ^+ (resp., ρ^-) be the ray emanating from p in the $(+y)$-direction (resp., $(-y)$-direction). We actually answer the ray-shooting queries for both ρ^+ and ρ^-. To do so we explore $\mathcal{L}$ in a top-down fashion. In the ith step, the query procedure visits two nodes v_1 and v_2 at level i of $\mathcal{L}$, so that among the minimal segments stored at all level i nodes, v_1 (resp., v_2) contains the first segment hit by ρ^+ (resp. ρ^-). If ρ^+ (resp., ρ^-) does not intersect any of the minimal segments, v_1 (resp., v_2) is undefined. Note that v_1 and v_2 may be the same node. When we reach the leaf level, the leaf v_1 contains the desired segment.

In the ith step, among the $O(B)$ minimal segments stored at v_1 and v_2, we find the first segment λ_w (resp., λ_z) hit by ρ^+ (resp., ρ^-), and set v_1 to w and v_2 to z.

Lemma 2 *Among the minimal segments stored at level $(i+1)$ nodes, w and z contain the first segments hit by ρ^+ and ρ^-, respectively.*

The correctness of the query procedure follows from the above lemma, therefore we obtain the following.

Lemma 3 *A set L of K segments all of whose right endpoints lie on a single vertical line can be stored in a data structure using $O(K/B)$ blocks, so that a vertical ray-shooting query can be answered in $O(\log_B K)$ time. If L is sorted by the y-coordinates of the right endpoints, then the structure can be constructed in $O(K/B)$ I/Os.*

2.4 Multislab structure

We now describe the multislab structure. Let $\mathcal{M}$ be a set of K disjoint segments whose endpoints have $\sqrt{B}+1$ distinct x-coordinates, i.e., they lie on $\sqrt{B}+1$ vertical lines $b_1, \ldots b_{\sqrt{B}+1}$. Some of the segments in $\mathcal{M}$ may be

points. For $1 \leq i \leq \sqrt{B}$, let s_i be the vertical slab bounded by b_i and b_{i+1}. All vertical lines in the interior of a slab intersect the same subset of $\mathcal{M}$, so if we know the sorted (according to $\prec_\Pi$) set of lines $\mathcal{M}'$ intersecting the slab containing the query ray, we can easily answer the query in $O(\log_B N)$ I/Os using a B-tree on $\mathcal{M}'$. However, we cannot afford to construct a data structure for each slab separately, so we will construct a single data structure. Lemma 1 (ii) will be crucial for our construction.

Let $\mathcal{M}$ denote the sequence of segments sorted by $\prec_\Pi$-ordering. We first construct a $\sqrt{B}$-tree Ψ on $\mathcal{M}$. For a node $v \in \Psi$, let $\mathcal{M}_v$ denote the subsequence of $\mathcal{M}$ stored in the subtree rooted at v. We store a number of segments of $\mathcal{M}_v$ at each internal node v of Ψ to facilitate the query procedure. Let $w_1, \ldots, w_{\sqrt{B}}$ denote the children of an internal node v. For $1 \leq i, j \leq \sqrt{B}$, let μ_{ij} denote the maximal segment of $\mathcal{M}_{w_i}$ (in the $\prec_\Pi$-ordering) that intersects the vertical slab s_j. If no segment of $\mathcal{M}_{w_i}$ intersects s_j, μ_{ij} is undefined. For $1 \leq i \leq \sqrt{B}, 1 \leq j \leq \sqrt{B}+1$, let β_{ij} denote the maximal segment of $\mathcal{M}_{w_i}$ that intersects the vertical line b_j. If there is no such segment, β_{ij} is undefined. All less than $2B + \sqrt{B}$ segments μ_{ij} and β_{ij} are stored at v. The $\sqrt{B}$-tree Ψ requires $O(K/B)$ disk blocks and can be constructed (bottom-up) in $O(K/B)$ I/Os, assuming that $\mathcal{M}_v$ is sorted according to the $\prec_\Pi$-ordering.

Let p be a query point and ρ^+ the ray emanating from p in the $(+y)$-direction. To answer the query we follow a path from the root to a leaf z of Ψ so that $\mathcal{M}_z$ contains the first segment hit by ρ^+. At each node v visited by the procedure we do the following: If p lies in the interior of a slab s_r, let $E_v = \{\mu_{ir} \mid 1 \leq i \leq \sqrt{B}\}$. Since all segments in E_v intersect the vertical line containing p, by Lemma 1 and the definition of μ_{ij}, if μ_{ir} is the first segment of E_v intersected by ρ^+, then w_i contains the first segment of $\mathcal{M}$ hit by ρ^+. We therefore visit w_i next. If p lies on the slab boundary b_r, then we set $E_v = \{\beta_{ir} \mid 1 \leq i \leq \sqrt{B}\}$ and determine the child of v that we visit next, following a similar approach.

Lemma 4 *A set $\mathcal{M}$ of K disjoint segments whose endpoints have $\sqrt{B}+1$ distinct endpoints can be stored in a data structure using $O(N/B)$ blocks, so that a vertical ray-shooting query can be answered in $O(\log_B K)$ I/Os. If $\mathcal{M}$ is sorted according to $\prec_\Pi$ then the structure can be constructed in $O(N/B)$ I/Os.*

If we store both "maximal" and "minimal" segments in the internal nodes of Ψ, it is possible to find the first segment hit by ρ^- as well. Note that the above lemma can also be used to determine whether a segment $\gamma \in \mathcal{M}$: the query ray is the ray emanating from the left endpoint of γ; $\gamma \in \mathcal{M}$ if the query procedure returns γ itself.

3 Dynamic Point Location

In this section we show how to dynamize the data structure described in the previous section. Preparata and Tamassia [21] showed that the operations of inserting or deleting a chain of edges between two existing vertices are complete for monotone subdivisions (i.e., an arbitrary subdivision can be assembled or dissembled using $O(N)$ such operations). Because of space constraints, we only consider the insertion of a single edge in this extended abstract. Insertion (and deletion) of a chain of edges can be handled using a similar, though more involved, procedure. Recall that an update is admissible only if the subdivision remains monotone and planar after the update operation. It can be checked in $O(\log_B N)$ I/Os if an update is admissible using a modified version of the normal internal memory algorithm. Details will appear in the full paper.

In order to dynamize our data structure, we need efficient procedures for updating the base tree (when endpoints are inserted/deleted) and the secondary structures (when segments are inserted/deleted). Using the normal B-tree updating procedures the base tree can be updated in $O(\log_B N)$ I/Os. However, as rebalancing is done by splitting and fusing nodes we need to rebuild the corresponding secondary structures when performing such an operation. As in [6], we use a *weight balanced* B-tree to implement the base tree, in which a node v with ν elements in its subtrees can only be involved in a rebalance operation for every $\Omega(\nu)$ updates that access v. This allows us to rebalance the base tree I/O-efficiently while maintaining the secondary structures. Details will appear in the full paper.

To insert a new segment e we traverse down the base tree, using $O(\log_B N)$ I/Os, to find the first node v at which e intersects one or more slab boundaries. Then we insert the left, right, and middle subsegments of e in the left, right, and multislab structures stored at v. A segment can be inserted into a left or right structures in $O(\log_B N)$ I/Os using a slightly modified version of the standard B-tree insert procedure. The difficult part is updating the multislab structure. There are two main difficulties: First, insertion of an edge may change the $\prec_\Pi$ ordering considerably, so we may have to rearrange multislab structures at many nodes of the base tree. Second, it seems impossible to determine in $O(1)$ I/Os whether $e_i \prec_\Pi e_j$, for two edges $e_i, e_j \in \Pi$, and thus a segment cannot be inserted in a multislab structure using the standard B-tree insertion algorithm. The key to perform the update efficiently, is to maintain a second data structure on the $\prec_\Pi$-ordering of the whole subdivision. In Section 3.1 we first describe how to maintain this structure during insertions, and then in Section 3.2 we explain how to use it to update the multislab structures.

3.1 Updating $\prec_\Pi$

In order to update the $\prec_\Pi$-ordering as Π changes dynamically, we extend the $\prec_\Pi$-ordering to include vertices and faces of Π. Let Π^* be the dual graph of Π, as defined in the previous section. Recall that the dual of a vertex (resp., face) of Π is a face (resp., vertex) in Π^*. We refer to the vertices, edges, and faces of Π or Π^* as its *features*. For each feature ϕ of Π or Π^*, we define two vertices $l(\phi)$ and $r(\phi)$. If ϕ is a vertex, then $l(\phi) = r(\phi) = \phi$. If ϕ is a directed edge (α, β), then $l(\phi) = \alpha$ and $r(\phi) = \beta$. The boundary of a face ϕ of Π (or Π^*) consists of two paths from a vertex α to another vertex β. We set $l(\phi) = \alpha$ and $r(\phi) = \beta$. We say that there is a path from a feature ϕ_1 to another feature ϕ_2 of Π if there is a path in Π from $r(\phi_1)$ to $l(\phi_2)$. For two features $\phi_i, \phi_j \in \Pi$, we define $\phi_i \prec_\Pi \phi_j$ if either there is a path from ϕ_i to ϕ_j in Π or a path from ϕ_i^* to ϕ_j^* in the dual graph Π^*. As argued in [24], $\prec_\Pi$ is a total order on the features of Π. Let $\langle\Pi\rangle$ denote the sequence of all features of Π sorted by $\prec_\Pi$. The following lemma is due to Tamassia and Preparata [24] (refer to Figure 4 for case (iv)).

Lemma 5 (Tamassia-Preparata [24]) *Let Π be a monotone planar subdivision, and let Π' be the subdivision obtained after inserting an edge $e = (\alpha, \beta)$ that splits the face f of Π into two faces f_1, f_2, with f_1 lying to the left of e. Then $\langle\Pi'\rangle$ can be obtained from $\langle\Pi\rangle$ as follows.*

(i) $\alpha \prec_\Pi \beta \prec_\Pi f$:
$\langle\Pi\rangle = A\alpha B\beta C f D \Longrightarrow \langle\Pi'\rangle = A\alpha B f_1 e \beta C f_2 D.$

(ii) $f \prec_\Pi \alpha \prec_\Pi \beta$:
$\langle\Pi\rangle = A f B\alpha C\beta D \Longrightarrow \langle\Pi'\rangle = A f_1 B\alpha e f_2 C\beta D.$

(iii) $\alpha \prec_\Pi f \prec_\Pi \beta$:
$\langle\Pi\rangle = A\alpha B f C\beta D \Longrightarrow \langle\Pi'\rangle = A\alpha B f_1 e f_2 C\beta D.$

(iv) $\beta \prec_\Pi f \prec_\Pi \alpha$:
$\langle\Pi\rangle = A\beta B f C\alpha D \Longrightarrow \langle\Pi'\rangle = A f_1 C\alpha e \beta B f_2 D.$

In order to maintain $\langle\Pi\rangle$ efficiently, we store it in a level-balanced B-tree, denoted by Δ, in which each node also stores a pointer to its parent. In the most involved case (case (iv)) we need to swap two subsequences in $\langle\Pi\rangle$ and perform a constant number of insertions and deletions next to α, β, and f. Assuming that we have pointers to α, β, and f in the leaves of Δ, we can update $\langle\Pi\rangle$ using $O(1)$ insert, delete, split, and merge operations, provided that we can compute the paths from the root of Δ to the leaves containing α, β, and f. These paths can easily be found using the parent pointers. (It is not possible to do a root-leaf search for a given feature, as we cannot directly compare two features). As we will show in the next section, level-balanced B-trees support each of the above operations in $O(\log_B^2 N)$ I/Os amortized.

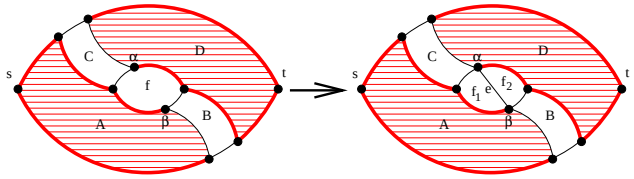

Figure 4: Inserting $e = (\alpha, \beta)$ when $\beta \prec_\Pi f \prec_\Pi \alpha$ $(A\beta B f C\alpha D \rightarrow A f_1 C\alpha e\beta B f_2 D)$. A, B, C, and D are defined by the topmost path from s to α and from s to β, and bottommost path from α to t and from β to t.

Theorem 3 *For a monotone planar subdivision Π, $\langle\Pi\rangle$ can be maintained after an insert operation in $O(\log_B^2 N)$ I/Os amortized*

We will perform two types of queries on Δ. First, given two features ϕ_1, ϕ_2 of Π, determine whether $\phi_1 \prec_\Pi \phi_2$. To do so we follow the parent pointers from the leaves storing ϕ_1 and ϕ_2, in $O(\log_B N)$ I/Os, until we reach their lowest common ancestor in Δ. We can then easily determine whether the leaf storing ϕ_1 lies to the left of that storing ϕ_2.

Lemma 6 *The relative order of two features of Π (according to $\prec_\Pi$) can be determined in $O(\log_B N)$ I/Os.*

Second, let α be a vertex of Π, and let ℓ be a vertical line that lies between α and t. Find the edge on the bottom-most path from α to t intersecting ℓ. Suppose at each node v of Δ, we also store the rightmost endpoint among all the edges stored in the subtree of Δ rooted at v. (As Π changes, this information can easily be updated without using additional I/Os.) We can then answer the query as follows: We follow the parent pointers, starting from the leaf of Δ storing α, until we reach a node w so that the right endpoint stored at w lies to the right of ℓ. We then follow the leftmost possible path in the subtree of Δ rooted at w so that at each node the rightmost endpoint lies to the right of ℓ. The leaf reached in this way stores the desired edge of Π.

Lemma 7 *For a vertex $\alpha \in \Pi$ and a vertical line ℓ, we we can find in $O(\log_B N)$ I/Os the edge of the bottommost path from α to t that intersects ℓ.*

3.2 Maintaining the multislab structure

We now describe how to maintain the multislab structures. We will see that although the segments in a multislab structure are ordered according to $\prec_\Pi$-ordering, we can still store them in a normal $\sqrt{B}$-tree, i.e., parent pointers are not required. Suppose we insert an edge

$e = (\alpha, \beta)$ into a face f of Π, which is to be stored at a node v of the base tree. We assume that the pointers to the leaves of Δ storing α, β, and f are given. In cases (i)–(iii) of Lemma 5, $\prec_\Pi$-ordering remains the same except that e is inserted, so we simply insert the middle subsegment of e into Ψ_v. In case (iv), the $\prec_\Pi$-ordering changes and thus we may have to modify the multislab structure at many nodes of the base tree. However, we can prove the following (proof omitted):

Lemma 8 *Let $e = (\alpha, \beta)$ be an edge to be inserted into a face f where $\beta \prec_\Pi f \prec_\Pi \alpha$, so that $\langle\Pi\rangle$ changes from $A\beta BfC\alpha D$ to $Af_1C\alpha e\beta Bf_2D$. Let v be the node in the base tree T at which e is to be stored. Insertion of e does not affect the $\prec_\Pi$ ordering for $\mathcal{M}_w$ if w is not an ancestor of v.*

Hence, in all cases, only the multislab structures at v and possibly its $O(\log_B N)$ ancestors are changed. At each such node w, we perform two steps: (i) re-organize the multislab structure Ψ_w to make it consistent with the new $\prec_\Pi$-ordering, and (ii) update the maximal segments μ_{ij} and β_{ij} at the nodes affected by the update procedure. The second step is relatively easy to handle, so we just describe step (i). Let A, B, C, and D be the same as in Lemma 5. Let A_w be the set of middle subsegments of segments in $A \cap S_w$; define B_w, C_w, and D_w similarly. $S_w = A_w \cup B_w \cup C_w \cup D_w$. We first split Ψ_w into four B-trees T_A, T_B, T_C, T_D, by performing three split operations, which store A_w, B_w, C_w, and D_w, respectively. Then we merge T_A with T_c and T_B with T_D, and merge the two resulting trees together to obtain the structure Ψ_w consistent with the new $\prec_\Pi$-ordering. Finally, if $w = v$, we insert the middle subsegment of e into the new Ψ_w. In order to perform the splits efficiently, we have to find the paths from the root to the leaves storing the first segments of B_w, C_w, and D_w. The following lemma suggests a way to find the first segment of B_w.

Lemma 9 *Let $e = (\alpha, \beta)$ be an edge to be inserted into a face f where $\beta \prec_\Pi f \prec_\Pi \alpha$, so that $\langle\Pi\rangle$ changes from $A\beta BfC\alpha D$ to $Af_1C\alpha e\beta Bf_2D$. Suppose e is to be stored at a node v of the base tree. Let w be an ancestor of the node v, and let s_i be the vertical slab s_i of w in which β lies. Let γ be the first segment on the path from β to t that intersects the right boundary b_{i+1} of s_i.*

Suppose $B_w \neq \emptyset$ and the first segment of B_w is the middle subsegment of $\xi \in S_w$. Then either $\xi = \gamma$ or ξ is the segment of S_w lying immediately above the point $\gamma \cap b_{i+1}$.

Proof: We first prove that ξ intersects b_{i+1}. Assume that this is not the case; then ξ must intersect some other boundary b_j to the right of b_{i+1}. Now all segments on the path from β to the left endpoint of ξ belong to B and are smaller than ξ in the $\prec_\Pi$-ordering. Since their middle subsegments are not in B_w, they must be stored at other nodes. One of these segments ξ' must cross b_{i+1} — its left endpoint lies to the right of (or on) β and it right endpoint lies to the left of (or on) the left endpoint of ξ. This contradicts the fact that ξ' is stored in another node.

Since ξ intersects b_{i+1}, if $\xi \neq \gamma$, then $\gamma \prec_\Pi \xi$ and therefore ξ lies above the point $\gamma \cap b_{i+1}$. Refer to Figure 5. This completes the proof of the lemma. □

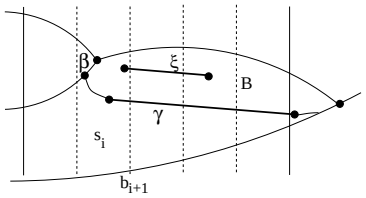

Figure 5: Proof of Lemma 9.

Using Lemma 9, we can compute the first segment of B_w as follows. Using Lemma 7, we can find in $O(\log_B N)$ I/Os the first segment γ of B that intersects the vertical line b_{i+1}. If $\gamma \notin B_w$, then we find the segment ξ in $\mathcal{M}_w$ immediately above the point $\gamma \cap b_{i+1}$ in $O(\log_B N)$ I/Os using Lemma 4. Next we check in $O(\log_B N)$ I/Os, using Lemma 6, whether $\xi \prec_\Pi f$. If the answer is "no," then $B_w = \emptyset$, otherwise ξ is the first segment of B_w. Note that since the query procedure for finding ξ traverses the path from the root of Ψ_w to the leaf storing ξ, we have actually found the path to split Ψ_w along (i.e., parent pointers are not needed). Similarly, we can find the first segments of C_w and D_w, as required, and split Ψ_w into four $\sqrt{B}$-trees. Details will appear in the full paper. Putting everything together, we conclude that we in total use $O(\log_B N)$ I/Os in each of $O(\log_B N)$ nodes to maintain the multislab structures. Thus we have obtained the following:

Theorem 4 *There exists a data structure using $O(N/B)$ blocks to store a monotone subdivision of size N, such that a vertical ray shooting query can be answered in $O(\log_B^2 N)$ I/Os worst case and such that updates can be performed in $O(\log_B^2 N)$ I/Os amortized.*

4 Level-Balanced B-Trees

Normal B-trees do not contain parent pointers and thus one cannot follow a leaf-root path in such trees as needed in our dynamic point-location structure. Augmenting B-trees with parent pointers leads to an undesirable $\Theta(B \log_B N)$ I/O bound for the split operation, the reason being that $\Theta(\log_B N)$ B-tree nodes are split and fused when performing a split and in each such operation $\Theta(B)$ parent pointers need to be updated.

In this section we describe level-balanced B-trees, which support insert, delete, split, and merge operations while maintaining parent pointers as needed in our dynamic point-location structure. The main new properties that allow us to perform all the above operations I/O-efficiently are that we (i) allow the degree of a node to be arbitrarily small, and (ii) allow disk blocks to contain several nodes. By doing so we avoid fusing nodes altogether and we are able to split a node in a constant number of I/Os. Note that when we allow nodes to have degree $o(B)$ we need some other way of assuring that the tree has height $O(\log_B N)$. We do so by imposing an invariant on the number of nodes on a given level of the tree, and rebuilding the nodes at a level whenever the invariant is violated. (Thus the name *level-balanced B-trees*.) In the next subsection we describe the general idea in level-balancing without discussing parent pointers and how the tree are layed out on disk blocks. In Section 4.2 we then sketch how to adopt the technique to external memory while maintaining parent pointers.

4.1 Level-balanced trees

A level-balanced tree of degree b is similar to a $(1, b)$-tree, i.e., all leaves are on the same level and all internal nodes have degree between one and b. To simplify our arguments, we assume that b is a power of 2 and 3 (that is, $b = 6^k$ for some integer $k \geq 0$). Since we allow split operations, we will show how to maintain a collection of trees (also called a *forest*) with a total of N leaves. As discussed, we impose an invariant on the number of nodes on a given level in order to bound the height of the trees. Set $N_i = N/(\frac{b}{3})^i$. We define the level of a node v to be the number of edges in a path from v to a leaf (i.e., leaves are on level zero), and maintain the following *level invariant* for all levels:

> *The number of non-root nodes at level i in the forest is at most $2N_i$.*

The invariant insures that the height of a tree is bounded by a function of the total number of leaves in the forest.

Lemma 10 *The height of all trees in the forest is $O(\log_b N)$.*

Update operations are now basically performed as in $(1, b)$-trees: To insert a new leaf e next to an existing leaf e', e is added to the child list of the parent p of e'. If p now has $b + 1$ children, p is split into two nodes of degree $b/2$ and $b/2 + 1$. The newly created node is recursively added to the parent of p. To delete a leaf z, we remove z from the tree. If the degree of the parent of z becomes zero, the parent is recursively deleted. (Note that no node fusions are performed).

To merge two trees of equal height h, we create a node at level $h+1$ and make the two roots the children of the new node. To merge two trees of height h_1 and h_2, with $h_1 < h_2$, we make the root of the tree of height h_1 a new child of the rightmost or leftmost node at level $h_1 + 1$ of the other tree. If this node now has degree $b + 1$, we recursively split the node as in insertions. To perform a split operation at a leaf e, we split the ancestor nodes of e so that e becomes the leftmost leaf of one of the resulting trees. The degree of ancestors of e may now be arbitrarily small, but we allow nodes to have small degrees, so no rebalancing (node fusion) is needed unless the level invariant is violated.

One main property of the operations above is summarized in the following lemma.

Lemma 11 *Insert, delete, split, and merge operations create at most one new node (perform one node split) at each level in the forest.*

After performing one of the above operations the resulting trees are $(1, b)$-trees, but the level invariant may be violated. If the level invariant is violated, we apply the following *level rebuilding* strategy: Let h be the lowest level for which the level invariant is violated (recall that the leaves are on level 0). For each tree T rooted at a level greater than h we simply replace level h and the subtree of T above level h with a new subtree in which all nodes, except possibly for the root, have degree at least $b/2$ and at most b.

Note that before the level rebuilding there are $2N_h+1$ non-root nodes at level h with a total of at most $2N_{h-1}$ children, i.e., the average degree of the nodes at level h is less than $\frac{b}{3}$. After the rebuilding, each non-root node has degree at least $b/2$ and the number of non-root nodes on level h is at most $2N_{h-1}/(\frac{b}{2}) \leq \frac{4}{3}N_h$. The rebuilding step reduces the number of non-root nodes at level h by increasing the average degree of the nodes at level h. In general, the number of non-root nodes at level $i \geq h$ in the resulting forest is at most $2N_{h-1}/(\frac{b}{2})^{i-h+1} \leq \frac{4}{3}N_i$. This means that $\Theta(N_i)$ nodes now need to be created at level $i \geq h$ before the level invariant can be violated for level i. Since each tree operation creates $O(\log_b N)$ nodes (Lemma 11) and $\Omega(N_i)$ nodes are rebuilt before the level invariant is violated for level i again, we obtain the following.

Lemma 12 *The operations insert, delete, split, and merge each rebuild $O(\log_b N)$ nodes in the amortized sense.*

4.2 External-memory representation

We now sketch how to lay out level-balanced trees on disk blocks and how to maintain parent pointers during

insert, delete, split, and merge operations. As mentioned, the key property that allows us to maintain parent pointers I/O-efficiently is that we allow each disk block to store several nodes. When splitting a node we can thus keep both resulting nodes in the same original block and avoid updating the parent pointers of the affected children.

We represent each node v in a level-balanced B-tree by up to $b+1$ constant-size records: One *node record* and between 1 and b *child records*. For each level i, node records of all nodes at level i are stored in a sequence of disk blocks such that each disk block contains $\Theta(B)$ records. All child records at level i are stored in another sequence of blocks such that each disk block contains $\Theta(B)$ records and such that all child records of a node are stored in the same block. (We require that all records in a block are from the same level). Note that the node and the child records of a node v are stored in different disk blocks.

Each child record of a node v stores a pointer to the node record of v and a pointer to the node record of a child node. The child records are kept in left-to-right order in a double linked list. The node record of v stores a pointer to the left-most child record of v, as well a pointer to a child record of the parent of v (which stores a pointer to the node record of v). A pointer is represented as a pair consisting of a disk block identifier and a record offset within the block (for example the pointer $(27, 95)$ would point to record 95 in disk block number 27). Note that because of the way records are stored in blocks, it is possible to update the parent pointer in the node record of v without having to load the child records of v, which in turn means that all the parent pointers of a level with X nodes can be updated in $O(X/B)$ I/Os. This will be crucial when rebuilding a level.

It is easy to realize that except for node splitting and level rebuilding, each operation as described in Section 4.1 can be performed in $O(\log_b N)$ I/Os (the height of the tree). Thus we only sketch how to split a node I/O-efficiently and how to rebuild a level I/O-efficiently. Details will appear in the full paper.

Assume that $2b \leq B$. A node can be split as follows: First the double linked list of child records are updated to reflect the split. Then a new node record is created and the pointers between the node record and the child records are set up. Finally, we have to create a new child record at the parent. There are two cases to consider. If the disk block storing the child list of the parent has space left for an additional child record, we create the new child record in the block. Otherwise, we first move between $\frac{1}{4}B$ and $\frac{3}{4}B$ of the child records to a new disk block and update the affected pointers with $O(B)$ I/Os before creating the new child record.

Except for the creation of the new child record, a split can be done in $O(1)$ I/Os as only records in a constant number of blocks are updated. It can be shown that a new child record can be created with $O(1)$ I/Os amortized.

Lemma 13 *A node in a level-balanced B-tree can be split in $O(1)$ I/Os amortized.*

The level i of the forest is rebuilt as follows: First, we visit the level-i nodes of all trees rooted at level greater than i and generate a list of pointers to the children at level $i-1$, using $O(N_i)$ I/Os. Using another $O(N_i)$ I/Os we then reconstruct the nodes at levels $i, i+1, \ldots$, as previously described, so that all new nodes have degree between $b/2$ and b and so that each disk block stores at most $B/2$ records. Finally, we need to update the parent pointers of the node records at level $i-1$. In order to do so we first scan the child records of level-i nodes. For each child record r visited, we construct the pair (p, q), where p is the pointer (to the node record at level $i-1$) stored at r, and q is the pointer to r itself. We then sort these pairs using p as the key, in $O(\frac{N_{i-1}}{B} \log_{M/B} \frac{N_{i-1}}{B})$ I/Os; let L be the sorted list. Finally, we scan L and the sequence of node records at level $i-1$ simultaneously, and for each pair $(p, e) \in L$, we set the parent pointer in the node record p at level $i-1$ to e. This step requires $O(N_{i-1}/B)$ I/Os. The total number of I/Os used to rebuild level i is $O(N_i + \frac{N_{i-1}}{B} \log_{M/B} \frac{N_{i-1}}{B} + \frac{N_{i-1}}{B}) = O(N_i(1 + \frac{b}{B} \log_{M/B} \frac{N_i}{B}))$.

Lemma 14 *Level i of a forest of level-balanced B-trees can be rebuilt in $O(N_i(1 + \frac{b}{B} \log_{M/B} \frac{N}{B})$ I/Os.*

Since tree operations create at most one node at each of the $O(\log_b N)$ levels (Lemma 11), level i is only rebuilt once every $\Omega(N_i)$ tree operations. Thus the amortized cost of rebuilding level i is $O(1 + \frac{b}{B} \log_{M/B} \frac{N_i}{B})$ I/Os per tree operation. The number of I/Os charged to each tree operation for rebuilding steps is thus $O((1 + \frac{b}{B} \log_{M/B} \frac{N}{B}) \log_b N)$. Since all other steps require $O(\log_b N)$ I/Os, we obtain the following.

Theorem 5 *A set of N elements can be stored in a forest of level-balanced B-tree using $O(N/B)$ blocks, such that insert, delete, split, merge operations can be performed in $O((1 + \frac{b}{B} \log_{M/B} \frac{N}{B}) \log_b N)$ I/Os amortized, $2 \leq b \leq B/2$, while maintaining parent pointers.*

Choosing $b = \frac{B}{\log B}$ and using the fact $\frac{b}{B} \log_{M/B} \frac{N}{B} = \frac{\log_{M/B} \frac{N}{B}}{\log B} \leq \log_B N$, we obtain the following.

Corollary 1 *A set of N elements can be stored in a forest of level-balanced B-tree using $O(N/B)$ blocks, such that insert, delete, split, merge operations can be performed in $O(\log_B^2 N)$ I/Os amortized while maintaining parent pointers.*

5 Conclusions

In this paper we have developed an I/O-efficient dynamic data structure for point location in monotone subdivisions. Part of the data structure is a novel search tree rebalancing technique which allows for efficient maintenance of parent pointers in B-trees. The most challenging open problem is of cause to extend our technique to general planar subdivisions.

We believe that the techniques developed in this paper can be used to obtain I/O-efficient data structures for a number of other problems. For example, a planar st-graph of size N can be maintained in $O((1 + \frac{b}{B}\log_{M/B}\frac{N}{B})\log_b N)$ I/Os for $2 \leq b \leq B/2$, so that reachability queries in it can be answered in $O(\log_B N)$ I/Os [24]. We believe that our techniques can be used to develop external dynamic data structures for more general ray-shooting queries and to obtain an efficient external memory data structure for three dimensional point location.

References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD Thesis, University of Aarhus, 1996.

[3] L. Arge. External-memory algorithms with applications in geographic information systems. In *Algorithmic Foundations of GIS* (M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, eds.). *Lecture Notes in Computer Science*, 1340, Springer-Verlag, 1997.

[4] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 685–694, 1998.

[5] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica (to appear in special issues on Geographical Information Systems)*, 1998.

[6] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, 560–569, 1996.

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

[8] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

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

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

[11] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[12] G. Di Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoret. Comput. Sci.*, 61:175–198, 1988.

[13] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *Internat. J. Comput. Math.*, 13:209–219, 1983.

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

[15] H. Edelsbrunner and H. A. Maurer. A space-optimal solution of general region location. *Theoretical Computer Science*, 16:329–336, 1981.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, 714–723, 1993.

[17] L. Guibas, M. Overmars and M. Sharir. Ray shooting, implicit point location, and related queries in arrangements of segments. Tech. Rept. 433, Dept. Computer Science, New York University, 1989.

[18] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[19] M. H. Overmars. *The Design of Dynamic Data Structures, Lecture Notes Comput. Sci.*, vol. 156, Springer-Verlag, Germany, 1983.

[20] M. H. Overmars. Range searching in a set of line segments. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, 177–185, 1985.

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

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

[23] J. Snoeyink. Point location. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, 559–574. CRC Press, 1997.

[24] R. Tamassia and F. P. Preparata. Dynamic maintenance of planar digraphs, with applications. *Algorithmica*, 5:509–527, 1990.

[25] J. Vahrenhold and K. S. Hinrichs. Fast and simple external-memory planar point-location. Second CGC Workshop on Computational Geometry, October 1997.

[26] J. S. Vitter. External memory algorithms. In *Proc. of the 1998 ACM Symposium on Principles of Database Systems*, 119–128, 1998.