A Dynamic Data Structure for Efficient Bounded Line Range Search

by

Thuy T. T. Le and Bradford G. Nickerson

TR10 - 200, May 19, 2010

Faculty of Computer Science

University of New Brunswick

Fredericton, N.B. E3B 5A3

Canada

Phone: (506) 453-4566

Fax: (506) 453-3566

Email: fcs@unb.ca

www: http://www.cs.unb.ca

A Dynamic Data Structure for Efficient Bounded Line Range Search

Thuy Thi Thu Le and Bradford G. Nickerson

{m6839,bgn}@unb.ca
Faculty of Computer Science, University of New Brunswick
P.O. Box 4400, Fredericton, N.B. Canada E3B 5A3.

Abstract. A dynamic data structure for efficient axis-aligned orthogonal range search on a set of n lines in a bounded plane is presented. The algorithm requires $O(\log n + k)$ time in the worst case to find all lines intersecting an axis aligned query rectangle R, for k the number of lines in range. $O(n + \lambda)$ space is required for the data structure used by the algorithm, where λ is the number of intersection points among the lines. Insertion of a new rightmost line ℓ or deletion of a leftmost line ℓ requires $O(n)$ time in the worst case. For a sparse arrangement of lines (i.e., for $\lambda = O(n)$), insertion of a rightmost line ℓ or deletion of a leftmost line ℓ requires $O(\sqrt{n})$ expected time.

1 Introduction

Lines in a bounded plane can represent a large variety of natural phenomenon, including trajectories of moving objects, boundaries within the plane or linear constraints for optimization problems.

Range search among a set of geometric objects has been studied extensively for the last two decades (see e.g. [2], [9]). Data structures for searching an *arrangement* of n lines in the plane are presented in e.g. [5] and [6]. An arrangement stores the relationships among vertices, edges and convex regions arising from the $O(n^2)$ intersections of the lines. Arrangements arise naturally in point search as points in primal space become lines in dual space. Arrangements of lines are used to support a variety of geometric search problems, such as halfspace range search of points [1].

Line segment search is another important class of geometric search problem. Reporting the λ intersections among a set of n line segments was solved in optimal time $O(n \log n + \lambda)$ using $O(n + \lambda)$ space in [4]. The space was improved to optimal $O(n)$ in [3]. Reporting horizontal line segments intersecting a vertical query line segment was solved in $O(\log n + k)$ time and $O(n \frac{\log n}{\log \log n})$ space [10]. A well known data structure, the persistent search tree [11], can report k line segments crossing a vertical segment in $O(\log n + k)$ time using $O(n + \lambda)$ space to store n line segments. However, this data structure does not support insertion and deletion. We build a dynamic data structure to answer queries in $O(\log n + k)$ time using $O(n + \lambda)$ space.

We explore the problem of the 2-d axis aligned orthogonal range search of lines in a bounded plane using the pointer machine model. In a 2-d space with axes x and y, we are given a set of n lines in a plane whose bounds are $[0, x_{max}]$ and $[0, y_{max}]$, respectively. We propose a new algorithm using a data structure called the *ordered polyline tree* to efficiently index a set of n bounded lines. Given an axis aligned query rectangle R, our algorithm can report all lines intersecting R in $O(\log n + k)$ time in the worst case using $O(n + \lambda)$ space, where λ is the number of intersections among the lines. To our knowledge, this is the first dynamic data structure to match the persistent range search tree in space and range search time complexity. The algorithm we present is practical to implement. This paper improves on a previous result [7] requiring $O((\log n)^2 + \beta)$ time in the worst case, for β the number of segments (resulting from the arrangement of lines) intersecting R.

2 Our Approach

Given a set of lines having slopes $m \in (0, \infty]$. Searching for lines intersecting a query rectangle R with four vertices A, B, C, and D (in a clockwise direction, see Fig. 1) is to find lines intersecting the left vertical line segment AD and the bottom horizontal line segment DC. We divide a set L of lines on the plane into

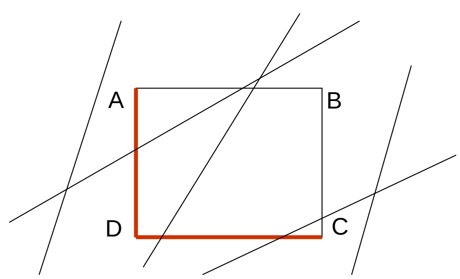

Fig. 1: Query rectangle R with four vertices A, B, C, and D. Lines with slopes $\in (0, \infty]$ intersect the rectangle R if and only if they intersect line segments AD or DC of the rectangle.

two subsets L_1 and L_2. L_1 contains lines oriented with slope $m \in (0, \infty]$ and L_2 has lines with slope $m \in (-\infty, 0]$. In the following discussion of the paper, we focus only on L_1, the subset of lines with slope $m \in (0, \infty]$. A similar algorithm and analysis applies to L_2. Ordered polyline trees for both L_1 and L_2 provide the basis for the complete search algorithm.

We use the notion x-level(i) to refer to the set of lines intersecting the line $x = i$ ordered top-to-bottom. Similarly, y-level(i) refers to a set of lines intersecting the line $y = i$ ordered left-to-right. Fig. 2 shows an example of two x-levels: x-level(15.0) and x-level(19.2), and two y-levels: y-level(3.6) and y-level(6.3). The order of lines changes where lines intersect. For the set of eight lines and query

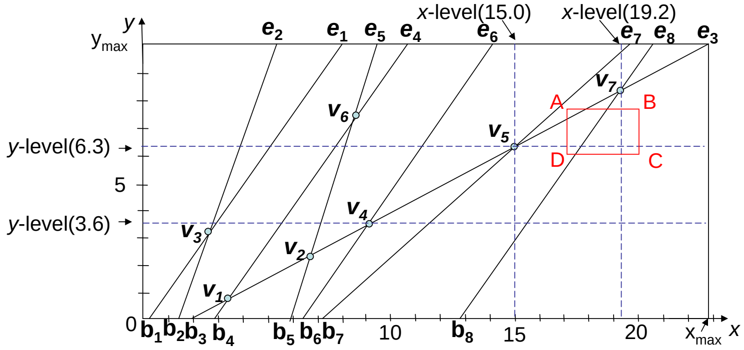

Fig. 2: Eight bounded lines having slopes $m \in (0, -\infty]$. Query rectangle $ABCD$ has points A=(17, 7.7) and C=(20,6). Dashed lines show x-levels and y-levels near AD and DC. Bounded line o_i has two endpoints b_i and e_i. v_1, .., v_7 are vertices at intersections. Lines o_3 and o_8 are in range.

rectangle $ABCD$ in Fig. 2, we only need to search for lines intersecting AD on x-level(15) and DC on y-level(3.6). We build a data structure for efficient search based on this idea. An ordered polyline p_i is created by connecting line segments at intersections (with each other and with the $x = 0$, $x = x_{max}$, $y = 0$, and $y = y_{max}$ boundaries). For example, the first three ordered polylines in Fig. 2 are $p_1 = \{b_1, v_3, e_2\}$, $p_2 = \{b_2, v_3, e_1\}$, and $p_3 = \{b_3, v_1, v_6, e_5\}$, ordered from left to right. Ordered polylines intersect each other only at intersection vertices. Points in an ordered polyline are monotonically increasing in both x and y. We connect points in an ordered polyline together into a list of entries, and arrange ordered polylines in a balanced search tree.

Each ordered polyline p_i divides the bounded plane into two disjoint parts. Points to the left of p_i are guaranteed to be in the left subtree of the node containing p_i. Similarly, points to the right of p_i are in the right subtree of the node containing p_i.

In the worst case, every line intersects all other lines (see Fig. 3). For n

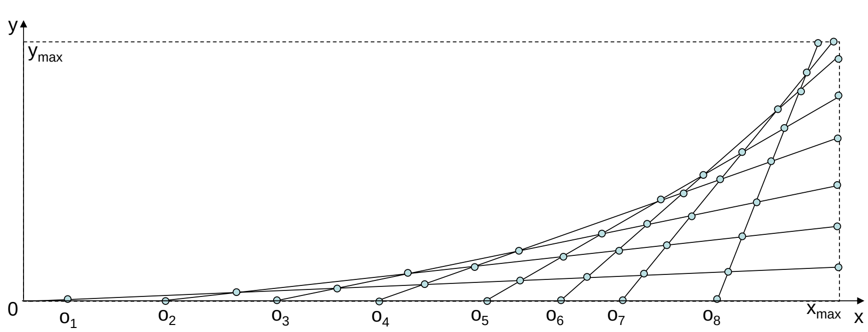

Fig. 3: Example of 8 lines o_1, ..., o_8 in the worst case, when each line intersects 7 others.

lines, this worst case results in at most $\frac{n(n-1)}{2}$, or $O(n^2)$ intersections, with each ordered polyline requiring at most $2(n-1)$ line segments, or each node of the tree storing at most $2(n-1)$ entries. The number of nodes of the tree and the number of ordered polylines is still precisely n.

3 Ordered Polyline Tree

Ordered polylines are arranged as a balanced binary search tree, called an *ordered polyline tree*, based on each p_i dividing space $(x \times y)$ into two parts. Each ordered polyline contains a list of entries. Each entry contains a point (x, y), a line ID, three (left, right, next) pointers on x, and one next pointer on y. We use the term x-entry (y-entry) to refer to the x value (y-value) at an entry. Fig. 4 shows the ordered polyline tree (for one entry on each node in x-entries) based on the ordered polyline tree in Fig. 2. A full ordered polyline tree has pointers on both

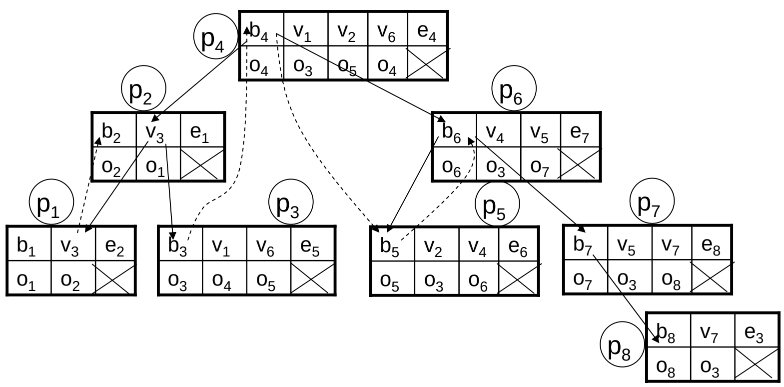

Fig. 4: Ordered polyline tree indexing the 8 lines from Fig. 2. A two-row rectangle represents an ordered polyline, where each column represents an entry containing a point and a line id o_i. A dashed line represents the next pointer of an entry to its adjacent entry on x-level.

x-entries and y-entries. For simplicity, Fig. 4 only shows pointers to the next x-entry.

For a polyline p_i with x-entry x_j, the (left, right, next) pointers point to the largest x-entry $\leq x_j$ in p_i's (left, right, next) nodes, respectively. If no x-entries in p_i's (left, right, next) nodes are $\leq x_j$, the (left, right, next) pointers point to the smallest x-entry $> x_j$. In this way, we record all line segments in the arrangement of bounded lines such that a traversal of the tree from root to leaf serves to find the polyline immediately to the left of a query point A. Following next pointers of x-entries finds segments of ordered polylines in downward order for a vertical query segment AD. Following next pointers of y-entries finds segments of ordered polylines in left-to-right order for a horizontal query segment DC.

3.1 Space Complexity

Theorem 1 *The ordered polyline tree use $O(n + \lambda)$ space to index a set of n lines with λ intersections among the lines.*

Proof. Without loss of generality, we assume that all n lines are oriented with slope $m \in (0, \infty]$. Assume each value stored in an entry of a node has size 1 (e.g., 1 for a coordinate x or y, a line ID, or a pointer). An entry of the ordered polyline tree is of size 7. There are $2n + 2\lambda$ entries among n nodes of the tree. The size of the tree is $7(2n + 2\lambda)$=$14(n + \lambda) = O(n + \lambda)$.□

Theorem 2 *For a set L of n lines in a bounded plane, the required space to index them using ordered polyline trees is $O(n + \lambda)$, where λ is the total number of intersection points among the lines.*

Proof. Since an ordered polyline tree is used to index lines having the same slope domain (i.e., slope $m \in (0, \infty]$, or $m \in (-\infty, 0]$), we need two ordered polyline trees to index all n lines of any slope. Assume that the set L of n lines in the bounded plane is divided into two subsets L_1 and L_2. L_1 contains n_1 lines oriented with slope $m \in (0, \infty]$ and L_2 has n_2 lines with slope $m \in (-\infty, 0]$. Let λ_1 and λ_2 be the number of intersection points among lines in L_1 and L_2, respectively. We need two ordered polyline trees, one for indexing L_1 and the other for indexing L_2. From Theorem 1, the required space for L_1 is $O(n_1 + \lambda_1)$, and the required space for L_2 is $O(n_2 + \lambda_2)$. The overall space of the both trees is $O(n_1 + n_2 + \lambda_1 + \lambda_2) = O(n + \lambda)$, since $\lambda \geq \lambda_1$+λ_2. □

4 Search Complexity

Given a query rectangle R with four vertices A, B, C, and $D = (t, r)$ in a clockwise direction. The search proceeds by finding the nearest polyline to the upper left of A, following x-entries to find lines intersecting AD (with $x = t$), then following y-entries to find lines intersecting DC (with $y = r$). The improvement is that (x, y)-entries point to the next adjacent segment in the next adjacent polyline. This reduces search time at each node from $O(log_2 w)$ to $O(1)$, where w is the number of entries stored in a node. The following shows the main steps of the search algorithm:

(1) Searching starts from the root node, choosing the largest entry $e_i = (x_i, y_i, id_i)$ whose $x_i \leq t$. If $t <$ smallest x_i, choose the smallest entry.
(2) Follow the entry's $left$ or $right$ pointer to the next entry by comparing line id_i to point A. If A is left of the line, follow the left pointer; otherwise follow the right pointer.
(3) We arrive at entry $e_i = (x_i, y_i, id_i)$ for node p_i. Choose the largest entry $e_j = (x_j, y_j, id_j)$ following e_i whose $x_j \leq t$. If $t <$ smallest x_j, choose the smallest entry.
(4) Repeat (2) and (3) until reaching a leaf node.

(5) At node entry $e_j = (x_j, y_j, id_j)$, if A is left of line id_j, check to see if line id_j intersects AD; if so, report line id_j.
(6) Use the *next* pointer at this x-entry to find the next adjacent polyline entry x_i. If $x_i > t$, $x_i \leftarrow x_{i-1}$. If $x_i \leq t$, $x_i \leftarrow x_{i+1}$.
(7) If line id_i intersects AD, report line id_i, and repeat step (6).
(8) We arrive at an entry $e_i = (x_i, y_i, id_i)$ in polyline p_i with a line id_i below D. Find the entry e_i in p_i with the largest y-entry value $\leq r$. Report id_i if it intersects DC.
(9) Use the *next* pointer at this y-entry to find the next adjacent polyline entry y_i. If $y_i > r$, $y_i \leftarrow y_{i-1}$. If $y_i \leq r$, $y_i \leftarrow y_{i+1}$.
(10) If line id_i intersects DC, report line id_i, and repeat step (9).
(11) We arrive at an entry $e_i = (x_i, y_i, id_i)$ with a line id_i right of C, so no possible lines remain that can intersect R.

Theorem 3 *Using an ordered polyline tree indexing n bounded lines in the plane, an algorithm exists to report the k lines intersecting an axis aligned query rectangle R in worst case time $O(\log_2(n) + k)$, where k is the number of lines in range.*

Proof. Without loss of generality, we assume that all n lines are oriented with slope $m \in (0, \infty]$. Assume w is the number of entries at the root node. Considering the 11 steps of the search algorithm above, we see that step (1) requires $O(\log_2 w)$ time. Steps (2), (3) and (4) take a combined $O(\log n)$ time to reach a leaf node. At step (3), when finding the largest entry $e_j = (x_j, y_j, id_j)$ following e_i whose $x_j \leq t$, we perform a binary search on the x-entries at node p_i. The worst case for step (3) arises when the root polyline p_i separates 2 sets of $n/2$ lines. Assuming A is on the right side of p_i, and that p_i is composed of two entries, this worst case requires up to $O(\log_2 n)$ time to find e_j. This can occur only once on the path to a leaf when the arrangement of lines to the right of p_i induces $O(n/2)$ segments in the right polyline of p_i. The remaining steps to a leaf require $O(1)$ time. Steps (5) through (10) require $O(1)$ time, and report the k lines intersecting R. The total required time for searching is $log_2 w$ + $O(\log_2 n)$ + k= $O(log_2 n + k)$ since $w \leq n$. □

5 Dynamic Update

We consider a limited form of dynamic updates. Line insertions are done on the right hand side and line deletions on the left hand side of the plane. This dynamic data structure would be useful, for example, when representing a set of moving objects on a graph's edge. For x representing time, and y representing positions along an edge, the (time $\times$ position) space admits new moving objects on the right (for the L_1 subset). Similarly, we delete the oldest moving objects from the left side of the (time $\times$ position) space.

5.1 Dynamic Data Structure

An ordered polyline tree with n nodes is a balanced binary tree where the depth of all leaves differs by at most one, and the depth of the tree is $\log_2 n$. As insertion of a new line happens at the rightmost node, the left child tree of an internal node is always a complete tree.

When all leaves of the left subtree T_L at the root node of an ordered polyline tree T are one level shallower than all leaves of the right subtree T_R of T (Figure 5a), the number of nodes of T_R with its depth $\log_2 n - 1$ is $(2^0 + .. + 2^{\log_2 n-2})$,

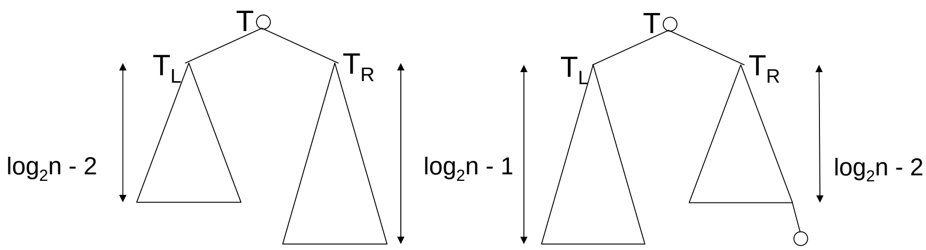

Fig. 5: (a) Nodes distributed in an ordered polyline tree T_i. (a) All leaves of T_L are 1 level shallower than those of T_R. (b) All leaves of T_L are 1 level deeper than those of T_R (except the rightmost leaf).

and the number of nodes of T_L with its depth $\log_2 n - 2$ is $(2^0 + .. + 2^{\log_2 n-3})$. There are $(2^0 + .. + 2^{\log_2 n-2}) - (2^0 + .. + 2^{\log_2 n-3}) = 2^{\log_2 n-1} = \frac{n}{4}$ more nodes in T_R than in T_L. Therefore, the left tree T_L contains $\lfloor \frac{3n}{8} \rfloor$ nodes, and the right tree T_R contains $\lfloor \frac{5n}{8} \rfloor$. Similarly, when all leaves of T_L is 1 level deeper than those of T_R (except the rightmost leaf) (Figure 5b), T_L contains $\lfloor \frac{5n}{8} \rfloor$ nodes and T_R contains $\lfloor \frac{3n}{8} \rfloor$. We obtain the following Lemma:

Lemma 1. *For an ordered polyline T containing n nodes constructed using the insertion at right-hand-side algorithm, the number of nodes in the left subtree T_L or the right subtree T_R of T is between $\lfloor \frac{3n}{8} \rfloor$ and $\lfloor \frac{5n}{8} \rfloor$, and $|T_L| + |T_R| + 1 = n$. The height of T is $\lfloor \log_2 n \rfloor$.*

Inserting a rightmost node to, or deleting a leftmost node from an ordered polyline tree can make the tree unbalanced. The $O(\log_2 n)$ nodes in the path from the involved leaf to the tree root have their height information updated. A rebalancing of the tree happens when the inserted rightmost node makes T unbalanced (e.g., an internal node P belonging to the rightmost path from the inserted rightmost node to the root of T has its right subtree's levels two levels deeper than its left subtree (see Figure 6)). We need a left rotation at P to make the tree balanced. Similarly, deleting a leftmost node from T can make T unbalanced (e.g., an internal node P belonging to the leftmost path from the

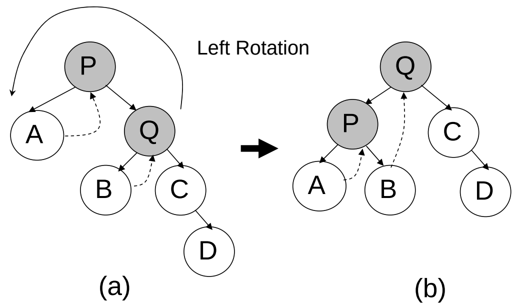

Fig. 6: (a) An example of an unbalanced tree at node P. (b) After rebalancing, P changes its right pointer, and Q changes its left pointer. Other nodes remain the same.

deleted node to the root of T has its right subtree's levels two levels deeper than its left subtree). A left rotation at P makes tree T rebalanced (see Figure 6).

Lemma 2. *Rebalancing an ordered polyline tree T after inserting a rightmost node or deleting a leftmost node involves at most three nodes of the tree.*

Proof. The tree is always unbalanced at a node P on the rightmost path or the leftmost path of the tree. If the tree is unbalanced at node P, a left rotation at P is applied to make the tree balanced (Figure 6). Let Q be the right child of P. As the result of rebalancing, the two involved nodes are P and Q. P changes its entries' right pointers, and Q changes its entries' left pointers. If P is not a root node, let U be P's parent node. After rebalancing the tree, U changes its entries' right (left) pointers if P is the right (left) child of U. All pointers of other nodes remain the same. □

Lemma 3. *At most four nodes have their entries' pointers changed as a result of inserting a rightmost line to, or deleting a leftmost line from the ordered polyline tree T.*

Proof. Let $node_R$ and $node_L$ be the existing rightmost and leftmost ordered polylines of T, respectively. Inserting a rightmost ordered polyline to T results in all $node_R$ entries' right pointers point to entries of the inserted node. Deleting $node_L$ from T results in all entries of $node_L$'s parent node changing their left pointers to *null* or to the next ordered polyline adjacent to $node_L$. Together with Lemma 3, there are at most four nodes having their entries's pointers changing after an insertion or a deletion. □

5.2 Insertion

If a new line ℓ is inserted on the right-hand-side, and there are μ intersection points between ℓ and ordered polylines $p_{n-(\mu-1)}$, .., p_{n-1}, p_n (see Figure 7), the required time to insert ℓ into the ordered polyline tree T is $O(\log n+\mu)$. There is one intersection between ℓ and each of the μ ordered polylines. Assume $u_1, .., u_\mu$ is the top-down y-sorted list of μ intersection points of ℓ and lines $\ell_1, .., \ell_\mu$ among

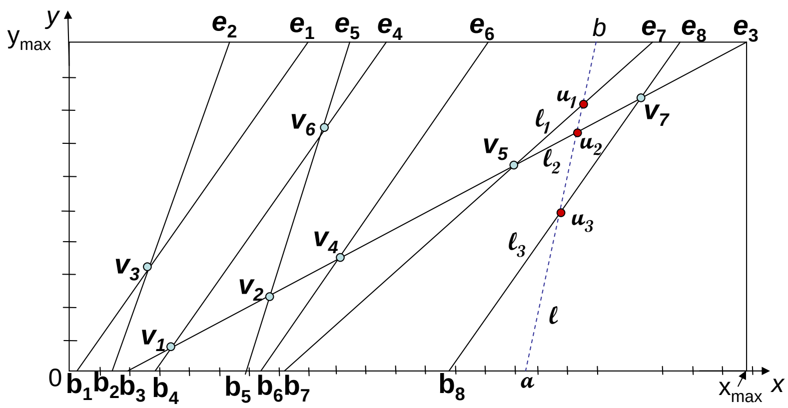

Fig. 7: Example of inserting the rightmost bounded line ℓ having endpoints a and b into the ordered polyline tree containing a set of eight lines (Figure 2). u_1, u_2, and u_3 are three intersection points between ℓ and lines o_6, o_3, and o_8, respectively.

μ ordered polylines. In this case, ℓ_μ belongs to the rightmost ordered polyline p_n in T.

Finding μ intersections requires $O(\log n + \mu)$ time by first finding the ordered polyline $p_{n-(\mu-1)}$intersecting ℓ, then finding the intersecting line ℓ_1 and computing the intersection point u_1. We use the next pointer at the current entry containing ℓ_1 to compute u_2, where u_2 is the intersection between ℓ and ℓ_2. This process is repeated until we reach ℓ_μ on p_n and obtain u_μ.

Updating μ ordered polylines requires $O(\mu)$ time. An ordered polyline containing points $e_1, ..., e_w$ is separated into two parts at the intersection point u_i of ℓ and ℓ_i $(1 \leq i \leq \mu)$. The first part contains entries $e_1, .., e_i, u_i$, and the second part is $(u_i, e_i, .., e_w)$. An updated ordered polyline is obtained by concatenating its first part to the $y = y_{max}$ end point of ℓ or to the second part of the previous ordered polyline. The first updated ordered polyline will concatenate the $y = y_{max}$ end point of ℓ. A new ordered polyline node p_{n+1} is created by concatenating the $y = 0$ end point of ℓ and the second part of p_n. Inserting an entry to each ordered polyline requires $O(1)$ time to find u_i and concatenation. It takes $O(1)$ time to travel from one inserted entry of an ordered polyline to the next inserted entry of the next ordered polyline. Therefore, the required time to insert μ entries to μ ordered polylines is $O(\mu)$. This leads to the following lemma:

Lemma 4. *The time to find the location of a new line ℓ, and to insert μ intersections from ℓ into each of μ existing ordered polylines is $O(\log n + \mu)$.*

Figure 7 shows an example of inserting a rightmost line to eight existing bounded lines stored in an ordered polyline tree shown in Figure 4. Line ℓ intersects three lines o_6, o_3, and o_8 at u_1, u_2, and u_3 respectively. Before being intersected by ℓ, the three last ordered polylines are $p_6 = \{b_6, v_4, v_5, e_7\}$,

$p_7 = \{b_7, v_5, v_7, e_8\}$, and $p_8 = \{b_8, v_7, e_3\}$. Let p_i^1 and p_i^2 be the first and second parts of p_i, respectively, after p_i is divided by an intersection point u. We have

$p_6^1 = \{b_6, v_4, v_5, u_1\}$, $p_6^2 = \{u_1, e_7\}$,
$p_7^1 = \{b_7, v_5, u_2\}$, $p_7^2 = \{u_2, v_7, e_8\}$,
$p_8^1 = \{b_8, u_3\}$, and $p_8^2 = \{u_3, v_7, e_3\}$.

Let $p_9^1 = \{a\}$ and $p_5^2 = \{b\}$. We obtain the three updated rightmost ordered polylines as follows:

$p_6 = p_6^1 + p_5^2 = \{b_6, v_4, v_5, u_1, b\}$,
$p_7 = p_7^1 + p_6^2 = \{b_7, v_5, u_2, u_1, e_7\}$,
$p_8 = p_8^1 + p_7^2 = \{b_8, u_3, u_2, v_7, e_8\}$.

The added ordered polyline $p_9 = p_9^1 + p_8^2 = \{a, u_3, v_7, e_3\}$. There are $\mu = 3$ ordered polylines with new entries to be inserted. When inserting node p_9 to the ordered polyline tree (Figure 4), we reorder p_7 and p_8 such that the right child of p_6 points to p_8. As a result, p_8's left child is p_7, and p_8's right child is p_9 (Figure 8). Three nodes p_7, p_8, and p_6 (the parent node of p_7 before rebalancing) are involved in the rebalancing.

Algorithm 1 shows the algorithm for inserting a rightmost line ℓ to the ordered polyline tree T. Function $entryIndex(p_i, \ell_j)$ finds the index of the entry in p_i containing line ℓ_j. Function $nextEntryIndex(e_k.xnext, \ell_j)$ locates the index of the entry in p_{i+1} containing line ℓ_j using the next pointer on x-value of e_k in p_i. Function $setEntriesPointers(p_n)$ sets p_n's next and right pointers to entries in the inserted ordered polyline p_{n+1}. Function $setPointers(u_j)$ sets the left, right, and next pointers for entry containing u_j. Statements 1.5, 1.6, 1.10, and 1.12 imply the left, right, next pointers are set appropriately.

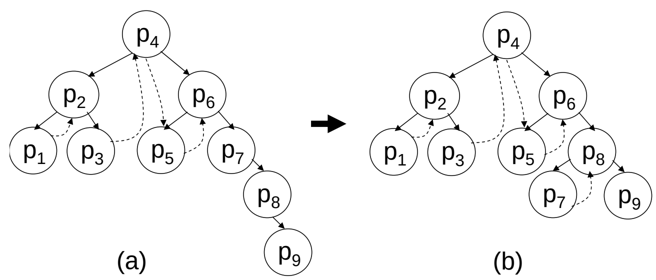

Fig. 8: (a) Unbalanced ordered polyline tree after inserting node p_9. (b) The tree after re-balancing.

Constructing a balanced ordered polyline tree by insertion of rightmost lines always results in a complete binary right sub-tree at any node of the tree. Inserting node p_{n+1} to the ordered polyline tree can make the tree unbalanced. The $\log_2 n$ nodes in the path from the rightmost leaf to the tree root have their height information updated, and at most 4 nodes are involved in tree re-balancing (Lemma 3). Each node contains at most n entries which need to reassign their left or right pointers. It requires $O(n)$ time to change left and right pointers in the nodes being re-balanced in this worst case. Assigning four pointers (i.e., left,

Algorithm 1: **RightmostInsert**$(T, \ell, \mu, (u_1, u_2, .., u_\mu), (\ell_1, \ell_2, .., \ell_\mu))$

The algorithm for inserting a rightmost line ℓ to the ordered polyline tree T indexing n lines.

input : Tree node T, rightmost line ℓ with two endpoints a and b on $y = 0$ and $y = y_{max}$, respectively, μ intersections $(u_1, u_2, .., u_\mu)$ between ℓ and lines $(\ell_1, \ell_2, .., \ell_\mu)$. In the algorithm, ordered polyline $p_i = \{e_1, .., e_{|p_i|}\}$. Each entry e_i contains a point (x, y), line id, three (left, right, next) pointers on x, and one next pointer on y.

output: Ordered polyline tree T with ℓ inserted.

begin
    if $\mu > 0$ **then**
        $i \leftarrow n - \mu + 1$//index of ordered polylines
        $j \leftarrow 1$//index of intersected points or lines
        $p^1_{n+1} \leftarrow \{a\}$
        $p^2_{i-1} \leftarrow \{b\}$
        Use b to travel down T to p_i
        $k \leftarrow entryIndex(p_i, \ell_j)$//the index of entry in p_i containing line ℓ_j
        while $i \leq n$ **do**
            $p_i \leftarrow \{e_1, .., e_k\} \cup u_j \cup p^2_{i-1}$
            $setPointers(u_j)$//set the left, right, and next pointers for entry u_j
            $p^2_i \leftarrow u_j \cup \{e_k, .., e_{|p_i|}\}$
            $j \leftarrow j + 1$
            $i \leftarrow i + 1$
            //from e_k find the index of the entry in p_{i+1} containing line ℓ_j
            $k \leftarrow nextEntryIndex(e_k.xnext, \ell_j)$
        $p_{n+1} \leftarrow p^1_{n+1} \cup p^2_n$
    else
        $p_{n+1} \leftarrow \{a, b\}$
    $setEntriesPointers(p_n)$//set p_n's next and right pointers
    if $unBalanced(T)$ **then**
        $Rebalance(T)$
    return T;
end

right, and next x-pointer, and its next y-pointer) for each new inserted entry takes $O(1)$ time by using the pointers of the previous entry in the same ordered polyline node. Therefore, the total required time is $O(\log n + \mu + n)$, or $O(n)$. We have the following Theorem:

Theorem 4 *The time to insert a new rightmost line ℓ into an ordered polyline tree indexing n lines is $O(n)$.*

Definition 1. *A sparse arrangement of n bounded lines in a plane has λ=$O(n)$.*

Theorem 5 *The time to insert a rightmost line ℓ into the ordered polyline tree of a sparse arrangement of n bounded lines in the plane is $O(\sqrt{n})$.*

Proof. The number of entries of an ordered polyline tree is $2(n + \lambda)$. From Definition 1, the number of intersection points λ is $O(n)$. The number of entries in the tree is $2(n+O(n))$, or $O(n)$. The maximum number of lines intersecting each other to form a sparse line arrangement is $O(\sqrt{n})$, which leads to $O(n)$ intersections among lines. With $O(n)$ intersections among $O(\sqrt{n})$ lines, the number of points in one ordered polyline is $O(\sqrt{n})$ (see [8]). When the ordered polyline tree with n nodes needs to be re-balanced, the height information of $\log n$ nodes is updated, and at most four nodes are involved in re-balancing (Lemma 3). Pointers of all entries of the involved nodes need to be updated. Thus, the required time is $O(\log n + \sqrt{n}) = O(\sqrt{n})$. With Lemma 4, the required time to insert μ intersection points to existing ordered polylines is $O(\log n + \mu)$. The total required time is thus $O(\log n + \mu + \sqrt{n})$, or $O(\sqrt{n})$.

Corollary 1. *The expected time to insert the rightmost line ℓ into the ordered polyline tree of a sparse arrangement is $O(\log n + \mu)$.*

Proof. With the sparse arrangement of n lines, the number of entries of the tree is $2(n+O(n))$, or $O(n)$. The average number of points belonging to one ordered polyline is $O(n)/n$, or $O(1)$. When the ordered polyline tree with n nodes needs to be re-balanced, the required time is $O(1)$, due to the fact that each node has at most $O(1)$ entries, and at most four nodes are involved in the update (Lemma 3). With Lemma 4, the required time to insert μ intersection points to existing ordered polylines is $O(\log n + \mu)$. We arrive at the proof.

5.3 Deletion

Deleting a leftmost line ℓ, having μ intersections with μ existing lines, from the ordered polyline tree requires $O(\log n+\mu)$ time. We need to delete μ intersection points from μ ordered polylines. Let $u_1, .., u_\mu$ be μ y-sorted intersection points between ℓ and lines $\ell_2, .., \ell_{\mu+1}$, where ℓ_2 is on the leftmost ordered polyline. Note that if an ordered polyline p_i contains ℓ, there exists a line segment (u_j, u_{j+1}) of ℓ belonging to p_i. This line segment needs to be removed from p_i. An ordered polyline p_i containing points $e_1, .., e_{j-1}, u_j, u_{j+1}, e_{j+2}, .., e_w$ is separated into three parts. The first part $e_1, .., e_{j-1}$ is kept in p_i. The middle part u_j, u_{j+1}

is removed from p_i. The third part $e_{j+2},..,e_w$ is concatenated to the first part of p_{i+1} to form the updated p_{i+1}. The updated ordered polyline p_i contains its first part concatenated with the third part of p_{i-1}. It takes $O(1)$ time to update an ordered polyline p_i by deleting the middle part u_j, u_{j+1} and concatenating the first part of p_i and the third part of p_{i+1}.

Figure 9 shows an example of deleting the leftmost line $\ell = o_3$ from a set of six lines. Before deleting ℓ, the six ordered polylines are as follows:

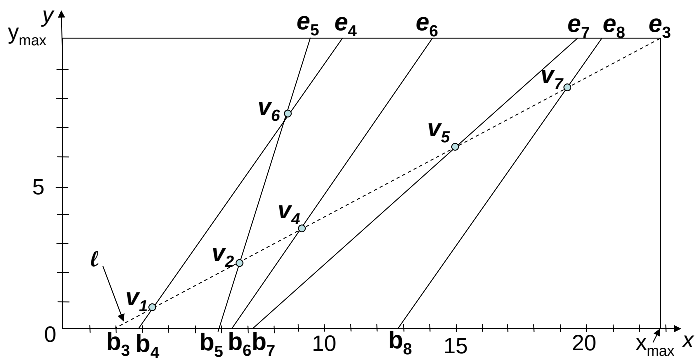

Fig. 9: Example of deleting the leftmost line ℓ from an ordered polyline tree for 6 lines from Fig. 2.

$p_3 = \{b_3, v_1, v_6, e_5\}$, $p_4 = \{b_4, v_1, v_2, v_6, e_4\}$,
$p_5 = \{b_5, v_2, v_4, e_6\}$, $p_6 = \{b_6, v_4, v_5, e_7\}$,
$p_7 = \{b_7, v_5, v_7, e_8\}$, and $p_8 = \{b_8, v_7, e_3\}$.

Let p_i^1, p_i^2, and p_i^3 be the first, second, and third parts of p_i, respectively. Based on the deleted line ℓ containing points $b_3, v_1, v_2, v_4, v_5, v_7$, and e_3, we have
$p_3^1 = \{\}$, $p_3^2 = \{b_3, v_1\}$, $p_3^3 = \{v_6, e_5\}$,
$p_4^1 = \{b_4\}$, $p_4^2 = \{v_1, v_2\}$, $p_4^3 = \{v_6, e_4\}$,
$p_5^1 = \{b_5\}$, $p_5^2 = \{v_2, v_4\}$, $p_5^3 = \{e_6\}$,
$p_6^1 = \{b_6\}$, $p_6^2 = \{v_4, v_5\}$, $p_6^3 = \{e_7\}$,
$p_7^1 = \{b_7\}$, $p_7^2 = \{v_5, v_7\}$, $p_7^3 = \{e_8\}$,
$p_8^1 = \{b_8\}$, $p_8^2 = \{v_7, e_3\}$, $p_8^3 = \{\}$.

At each updated ordered polyline p_i, we delete its second part p_i^2, then concatenate its first part p_i^1 and the third part p_{i-1}^3 of p_{i-1}. The updated ordered polylines are shown as follows:
$p_4 = p_4^1 + p_3^3 = \{b_4, v_6, e_5\}$,
$p_5 = p_5^1 + p_4^3 = \{b_5, v_6, e_4\}$,
$p_6 = p_6^1 + p_5^3 = \{b_6, e_6\}$,
$p_7 = p_7^1 + p_6^3 = \{b_7, e_7\}$,
$p_8 = p_8^1 + p_7^3 = \{b_8, e_8\}$.

Ordered polyline p_3 is removed resulting in p_3's parent having a right child but no left child. No node is involved in rebalancing the ordered polyline tree in this case (Figure 10).

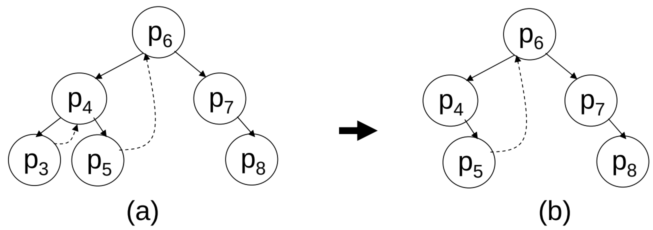

Fig. 10: The ordered polyline tree (a) before and (b) after deleting node p_3.

We then use the next pointer at the entry containing u_{j+1} of p_i to locate the entry on p_{i+1} containing u_{j+1}. This step requires $O(1)$ time. Now we have a new p_i with its middle part u_{j+1}, u_{j_2}, so we repeat the deletion and update operations until all μ intersections are visited. Updating μ ordered polylines thus requires $O(\mu)$ time.

Deleting node p_1 from n existing nodes of the ordered polyline tree can make the tree unbalanced. Similar to insertion, it requires $O(n)$ time to reorder all nodes of the tree in the worst case. Therefore, the total required time for deleting leftmost line ℓ is $O(\mu + n)$, or $O(n)$. We have the following Theorem:

Theorem 6 *The time to delete a leftmost line ℓ from an ordered polyline tree indexing n lines is $O(n)$.*

Theorem 7 *The time to delete a leftmost line ℓ from an ordered polyline tree of a sparse arrangement of n bounded lines in the plane is $O(\sqrt{n})$.*

Algorithm 2 shows the algorithm for deleting the leftmost line ℓ from the ordered polyline tree T. For the example of Figure 9, $\mu = 5$, and intersections $(u_1, u_2, u_3, u_4, u_5)$=$(v_1, v_2, v_4, v_5, v_7)$. Functions $index\_entryBefore(p_i, u_j)$ and $index\_entryAfter(p_i, u_j)$ find the index of the entry before and after u_j in p_i, respectively. Function $nextEntryIndex(u_{j+1}.xnext)$ finds the index of the entry in p_{i+1} containing point u_{j+1}, using the next pointer on x-value at u_{j+1} in p_i. Statement 2.14 implies the left, right, next pointers of related entries are set appropriately.

6 Conclusion

We present a new dynamic data structure for efficient axis aligned range search of a set of n lines on a bounded plane. To the best of our knowledge, this is the first dynamic data structure to solve this problem in $O(\log n + k)$ search time in the worst case to find all lines intersecting an axis aligned query rectangle R, for k the number of lines in range, and $O(n + \lambda)$ space.

Can the approach used here support general insertion or deletion of any bounded line? An open problem is how to build an I/O-efficient data structure to achieve logarithmic search time on a set of n bounded lines and linear storage space. The unpredictable number of intersections among lines makes the optimal branching factor hard to determine.

Algorithm 2: **LeftmostDelete**$(T, \ell, \mu, (u_1, u_2, .., u_\mu))$

The algorithm for deleting the leftmost line ℓ from the ordered polyline tree T indexing n lines.

input : Tree node T, leftmost line ℓ with two endpoints a and b on $y = 0$ and $y = y_{max}$, respectively, μ intersections $(u_1, u_2, .., u_\mu)$ between ℓ and lines $(\ell_2, \ell_3, .., \ell_{\mu+1})$. In the algorithm, ordered polyline $p_i = \{e_1, .., e_{|p_i|}\}$. Each entry e_i contains a point (x, y), line id, three (left, right, next) pointers on x, and one next pointer on y.

output: Ordered polyline tree T with the leftmost line ℓ deleted.

```
begin
    Travel down leftmost path of T to p_1
    if μ > 0 then
        k ← index_entryAfter(p_1, u_1)//index of entry after u_1 in p_1
        p_1^3 ← {e_k, .., e_|p_1|}
        i ← 2//index of ordered polylines
        j ← 1//index of intersection points
        while j ≤ μ do
            k ← index_entryBefore(p_i, u_j)//index of entry before u_j in p_i
            p_i^1 ← {e_1, .., e_k}
            k' ← index_entryAfter(p_i, u_{j+1})//index of entry after u_{j+1} in p_i
            p_i^3 ← {e_k', .., e_|p_i|}
            Delete(p_i, u_j, u_{j+1})//delete two entries containing u_j and u_{j+1} in p_i
            p_i ← p_i^1 ∪ p_{i-1}^3//the updated p_i
            i ← i + 1
            j ← j + 1
            //from u_{j+1} on p_i find the index of the entry in p_{i+1} containing point u_{j+1}
            k ← nextEntryIndex(u_{j+1}.xnext)
    Delete(p_1)//remove p_1 from T
    if unBalanced(T) then
        Rebalance(T)
    return T;
end
```

7 Acknowledgements

This research is supported, in part, by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the UNB Faculty of Computer Science, and the government of Vietnam.

References

1. P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61:194–216, 2000.
2. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1999.
3. I. J. Balaban. An optimal algorithm for finding segments intersections. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 211–219, New York, NY, USA, 1995. ACM.
4. B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *J. ACM*, 39(1):1–54, 1992.
5. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, 2000.
6. H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM J. Comput.*, 15(2):341–363, 1986.
7. T. T. T. Le and B. G. Nickerson. Ordered polyline trees for efficient search of objects moving on a graph. In *ICCSA 2010: The 2010 International Conference on Computational Science and Its Applications*, pages 401–413, Fukuoka, Japan, March 23-26 2010.
8. T. T. T. Le and B. G. Nickerson. A Dynamic Data Structure for Efficient Bounded Line Range Search. Technical report, TR10-200, Faculty of Computer Science, UNB, Fredericton, Canada, May, 2010, 12 pages.
9. J. Matoušek. Geometric range searching. *ACM Comput. Surv.*, 26(4):422–461, 1994.
10. C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 618–627, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.
11. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986.