Dynamic Segment Intersection Search with Applications

Hiroshi IMAI and Takao ASANO

Department of Mathematical Engineering and Instrumentation Physics,
Faculty of Engineering, University of Tokyo, Tokyo 113, Japan

Abstract

In this paper, we consider two restricted types of dynamic orthogonal segment intersection search problems. One is the problem in which the underlying set is updated only by insertions. The other, the set is updated only by deletions. We show that an intermixed sequence of \(O(n)\) queries and updates in both problems can be executed on-line in \(O(n \log n + K)\) time and \(O(n \log n)\) space, where \(K\) is the total number of reported intersections. Our algorithms utilize set-union and set-splitting algorithms. Especially, we present total number of reported intersections. Our algorithms utilize and updates in both problems can be executed on-line in insertions. In the other, the set is updated only by deletions. We show that an intermixed sequence of \(O(n)\) queries and updates in both problems can be executed on-line in \(O(n \log n + K)\) time and \(O(n \log n)\) space, where \(K\) is the total number of reported intersections. That is, in both problems, each of queries and updates can be performed in \(O(\log n)\) time in an amortized sense for two geometric problems, each of queries and updates can be performed in \(O(\log n^2 + k)\) query time, \(O(\log n \log \log n + k)\) query time, \(O(\log n \log \log n)\) update time and \(O(n \log n)\) space. For the general dynamic problem, Edelsbrunner [4] presents an algorithm with \(O(\log n^2 + k)\) query time, \(O(\log n^2)\) update time and \(O(n \log n)\) space.

We shall consider two restricted versions of dynamic orthogonal segment intersection search problems. One is called an insertion problem in which the underlying set of orthogonal segments is updated only by insertions. The other is called a deletion problem in which the set is updated only by deletions. We show that, in both problems, an intermixed sequence of \(O(n)\) queries and updates can be executed on-line in \(O(n \log n + K)\) time and \(O(n \log n)\) space, where \(K\) is the total number of reported intersections. That is, in both problems, each of queries and updates can be performed in \(O(\log n + k)\) time in an amortized sense. We utilize the layered segment tree presented in Vaishnavi and Wood [22] which is constructed by giving layered structures to the segment tree of Bentley [1]. We present techniques for dynamizing the layered structures by employing linear-time set-union and set-splitting algorithms due to Gabow and Tarjan [6]. Here, in order to obtain the result for the insertion problem, we introduce an incremental set-splitting problem, and, extending Gabow and Tarjan’s algorithm, develop a linear-time algorithm for it, which is of interest by itself. The techniques of dynamizing the layered structures apply to the range tree, and similar results are obtained for the orthogonal range search problem.

The data structures for both problems give new solutions with better time bounds to various geometric problems on orthogonal objects. We first show a paradigm of combining graph algorithms and the data structures for the deletion problem to solve geometric problems efficiently. A geometric problem on \(n\) objects can be solved as a graph problem on an intersection graph of the \(n\) objects by existing graph algorithms, if there is the graph problem corresponding to the original geometric problem. The paradigm shows how to execute graph algorithms on the intersection graph without recording its edges explicitly by using geometric searching data structures, which leads us to quite efficient solutions. Based on the paradigm, we describe five applications of the data structures for the deletion problem. Also, two applications of the insertion problem are presented.
2. Set-Union and Set-Splitting Problems

Algorithms for manipulating disjoint sets have been well studied. In this section, we first describe recent results of Gabow and Tarjan [6] on the set-union problem, and then develop a linear-time algorithm for the incremental set-splitting problem introduced here. For this purpose, we make some definitions on disjoint sets.

Let \( V = \{1, \ldots, n\} \), and \( S \) and \( U \) be such that \( S \subseteq U \subseteq V \). Suppose that \( S = \{s_1, \ldots, s_p\} \) and \( 0 = s_0 < s_1 < \cdots < s_p < s_{p+1} = n+1 \), where \( s_0 \) and \( s_{p+1} \) are dummy elements. Define \( V(s_i) = \{1, \ldots, s_{i+1}\} \) to be \( \{u \in U : s_i < u < s_{i+1}\} \). Then, \( \{V(s_i) : i = 1, \ldots, p+1\} \) is a partition of \( U \), and, for any \( u \in V(s_i) \) and \( v \in V(s_j) \) such that \( 1 \leq i < j \leq p+1 \), we have \( u < v \). We call \( \{V(s_1), \ldots, V(s_{p+1})\} \) an ordered partition of \( U \) with respect to \( S \), and denote it by \( P(U,S) \). The set \( U \) is called the ground set of the ordered partition \( P(U,S) \).

For the ordered partition \( P(U,S) \), the following three operations, \textit{find}, \textit{link}, and \textit{split} are fundamental:

- **find**: For \( u \in U \), \textit{find}(u) returns \( s_i \in S \cup \{n+1\} \) such that \( u \in V(s_i) \).
- **link**: For \( s_i \in S \), \textit{link}(s_i) adds all the elements of \( V(s_i) \) to \( V(s_{i+1}) \), and modifies the ordered partition to \( P(U,S \cup \{s_i\}) \). By \textit{link}(s_i), \( S \) is updated to \( S \cup \{s_i\} \).
- **split**: For \( u \in U \), \textit{split}(u) first executes \( \text{find}(u) \) and then divides the set \( V(s_i) \) into two sets, one containing all \( v \in V(s_i) \) with \( v < u \), the other all \( v \in V(s_i) \) with \( v > u \). The ordered partition is modified to \( P(U \setminus \{u\}, S \cup \{u\}) \). By \textit{split}(u), \( S \) is updated to \( S \cup \{u\} \).

In this paper, we introduce, besides these, the following \textit{add} operation.

- **add**: For \( v \in V \), let \( v^* \in U \cup \{n+1\} \) be defined by \( v^* = \min\{u \in U \cup \{n+1\} : u > v\} \). For \( v \) and \( v^* \), \textit{add}(v,v^*) inserts \( v \) into \( V(s_i) \) such that \( v^* \in V(s_i) \). The ordered partition becomes \( P(U \cup \{v\}, S) \). By \textit{add}(v,v^*), \( U \) is updated to \( U \cup \{v\} \).

Note that, in \textit{add}(v,v^*), \( v^* \) satisfying the above condition is given in advance with \( v \).

Concerning the general problem in which \textit{find}, \textit{link}, \textit{split}, and \textit{add} operations are executed, we can execute each of them in \( O(\log n) \) time by means of balanced trees. However, special cases of the problem can be solved more efficiently in an amortized sense as follows. The problem in which \textit{find} and \textit{link} operations are only executed is a kind of the set-union problem. In fact, it is a special case of the set-union problem, because the \textit{link} operation can unite adjacent two sets in ordered partition only. For this problem, the following is shown in [61], where it is assumed that, for each \( u \in U \), the rank of \( u \) in a sorted order can be found in \( O(1) \) time, totally using \( O(|U|) \) space.

**Theorem 2.1.** (Gabow and Tarjan [6]) For \( UC V \) (\( n = |U| \)), starting with an ordered partition \( P(U,S) \), an intermixed sequence of \( m \textit{find} \) and \textit{link} operations can be executed in \( O(m+n) \) time and \( O(n^2) \) space.

The problem in which \textit{find} and \textit{split} operations are executed is the set-splitting problem first considered by Hopcroft and Ullman [8]. They show that, starting with an ordered partition \( P(V,\emptyset) \), an intermixed sequence of \( m \textit{find} \) and \textit{split} operations can be executed in \( O((m+n) \log^* n) \) time and \( O(n^2) \) space, where \( \log^* n \) is the "iterated logarithm," the number of times the logarithm must be taken to obtain a number less than one. Gabow and Tarjan [6] show that the sequence can be executed in \( O(m+n) \) time and \( O(n^2) \) space.

We call the problem, in which, starting with \( P(\emptyset,\emptyset) \), an intermixed sequence of \( m \textit{find} \) and \textit{split} operations and \( l \textit{add} \) operations are executed, the incremental set-splitting problem. Note that \( 0 \leq l \leq n \). It seems that this problem has not yet been studied explicitly although it is rather straightforward to extend an algorithm for the set-splitting problem to that for the incremental problem. We show below, for completeness, two algorithms for the incremental set-splitting problem by extending the above-mentioned algorithms.

(1) An \( O((l+m) \log^* l) \)-time incremental set-splitting algorithm

We will show how to extend Hopcroft and Ullman's algorithm [8] so as to run in \( O((l+m) \log^* l) \) time and \( O(l) \) space for the incremental set-splitting problem. Define a function \( F(i) \) (\( i \geq 0 \)) by \( F(0) = 1 \) and \( F(1) = F(0) F(1) = F(0,1) \). The algorithm in [8] uses a tree defined as follows. A node of level 0 in the tree is a leaf, and does not have any son. A node of level \( i \geq 1 \) has between one and \( 2^{F(i-1)} \) sons, hence it has at most \( F(i) \) leaves. A node of level \( i \) is called \textit{complete} if it has \( F(i) \) leaves among its descendants, and is called \textit{incomplete} otherwise.

We maintain the ordered partition \( P(U,S) \) in the following way. \( U \) is kept by a doubly linked list in the increasing order from left to right. Each set in the ordered partition is made to correspond to a tree the root of which contains the name of the set. Some and not necessarily all of elements of each set are stored at the leaves of the tree, so that the set of all leaves of trees corresponds to a set \( W \) such that \( S \subseteq W \subseteq U \). \( W \) may be a proper subset of \( U \) (in fact, when executing \( \text{add}(v,v^*) \), we do not make a leaf for \( v \), but create it in some \textit{split} operation), and each \( u \) such that \textit{split}(u) has been executed (i.e., \( u \in S \)) necessarily has the corresponding leaf. For each \( u \in U \), we record \( p(u) = \min\{u' \in W : u' \geq u\} \). The execution of \textit{find}(u) is to find the root of a tree that contains a leaf corresponding to \( p(u) \). Concerning \textit{add}(v,v^*), we insert \( v \) in the left of \( v^* \) in the list representing \( U \), and let \( p(v) = p(v^*) \). Consider the operation \textit{split}(u). If \( u \notin W \), we divide the tree containing \( u \) along the path from the leaf \( u \) to the root of that tree as in Hopcroft and Ullman's static set-splitting algorithm. If \( u \in W \), we first find a leaf \( w \) that lies in the left of leaf \( p(u) \) among all the leaves, and then partition the tree containing \( w \) along the path from \( w \) to the root. Then, to the tree containing \( w \), we add, from the right side, elements from the next element of \( w \) to \( u \) in the list of \( U \) in this order one by one (added elements are now in the set \( W \)). At this stage, if the number of sons of a node of level \( i \) becomes greater than \( 2^{F(i-1)} \) by making a new node of level \( i-1 \), we make a new node of level \( i \) and let the new node of level \( i-1 \) be a son of the new node of level \( i \). In this algorithm, the height of any tree is \( O(\log^* l) \).

The analysis of the complexity of the above algorithm is omitted here. It should be noted that this algorithm works on pointer machines, whereas the following linear-time algorithm does not, and it makes use of power of random-access machines. We pose a conjecture that any pointer machine requires \( \Omega((l+m) \log^* l) \) time to solve the incremental set-splitting problem.
(2) An $O((1+m)-time incremental set-splitting algorithm.

Next extending Gabow and Tarjan's algorithm [6], we present, for the incremental set-splitting problem, an $O((1+m)$-time and $O(l)$-space algorithm on a random access machine with unit cost measure and $O(\log n)$ word length, where arguments described below are almost similar to those in Gabow and Tarjan's static set-splitting algorithm.

We partition the ground set $U$ into consecutive elements, called mezzosets $U_i$, of size at most $\log \log n$ in such a way that there are $O(|U|/\log \log n)$ mezzosets. Each mezzoset $U_i$ is partitioned into consecutive elements, called microsets $U_{i,j}$, of size at most $\log \log n$ in such a way that the number of microsets in $U_i$ is $O(|U_i|/\log \log n)$. We represent the ordered partition $P(U,S)$ by the three levels of ordered partitions. In order to do so, we need some definitions. For a subset $U'$ of $U$, define $\bar{U}'$ to be $\max \{u \in U' \}$, and $s(U')$ to be $\min \{s \in U' \}$ if $U' \cap S \neq \emptyset$ and null, otherwise. For each mezzoset $U_i$, let $\bar{U}_i = \{\{U_i(j) \} \}_{j \in \mathbb{U}}$, and $S_i = \{s(U_i(j)) \}_{j \in \mathbb{U}}$. Also, let $\overline{U}_i = \{\overline{U}(U_i(j)) \}_{j \in \mathbb{U}}$, and $S_i = \{s(U_i(j)) \}_{j \in \mathbb{U}}$.

We maintain the following three levels of ordered partitions: (i) $P^0 = P(\bar{U}, \overline{U})$; (ii) $P_i^1 = P(U_i, \overline{U})$ for each mezzoset $U_i$; (iii) $P_{i,j}^2 = P(U_{i,j}, S_i)$ for each microset $U_{i,j}$, where $S_i = \{U_i(j) \}_{j \in \mathbb{U}}$.

Then, we can execute $\text{find}(u)$ (with respect to $P(U,S)$) as follows: First, for the microset $U_{i,j}$ containing $u$, execute s := $\text{find}(u)$ with respect to $P_{i,j}^2$, if $s \in S_i$, return $s$ and halt; otherwise, execute $s := \text{find}(u)$ with respect to $P_i^1$, if $s \in S_i$, return $s$ and halt; otherwise, execute $s := \text{find}(u)$ with respect to $P_i^1$ and return $s$. We can perform analogously $\text{split}$ operations for the three levels of ordered partitions. To execute $\text{add}(v,*)$, we execute $\text{add}(v,*)$ with respect to $P_i^1$ for the microset $U_{i,j}$ containing $v^*$. This may cause the size of the microset $U_{i,j}$ to become greater than $\log \log n$, and, in such a case, we divide this microset into halves. This division may further make the size of a mezzoset greater than $\log \log n$, and, in such a case, the mezzoset is divided into halves. Note that the scheme of dividing sets in above way maintains the properties that each mezzoset $U_i$ contains $O(|U_i|/\log \log n)$ microsets and the number of mezzosets is $O(|U|/\log \log n)$.

Concerning the implementations of $\text{find}$, $\text{split}$, and $\text{add}$ for the top and middle levels where ordered partitions are $P_i^1$ and $P_i^2$, we employ the "relabel-the-smaller-half" method as in [6]. It is seen that this method executes a sequence of $\text{find}$ and $\text{split}$ and $\text{add}$ operations in $O(m+1/\log n)$ time. Note that, in this method, the mezzoset whose size comes to be greater than $\log \log n$ can be simply divided into halves in $O(\log \log n)$ time.

In order to implement $\text{find}$, $\text{split}$, and $\text{add}$ for the bottom level where ordered partitions are $P_{i,j}^2$, we use the table-lookup method as in [6]. We represent each microset, which is numbered from 1, by a tree whose preorder gives an ordering of elements in the set in decreasing order, and encode it by the parent table, and execute it by the parent table as follows. For each element (or, node) $v$ in the microset, we store $\text{micro}(v)$, the number of this microset, and $\text{number}(v)$, the number, which is from 1 to $b = \log \log n$, of $u$ within this microset. By node $(i,j)$, we denote the node in microset $i$ with number $j$. For the tree representing the microset, let $\text{parent}(i,j) = k$ if the parent of node $(i,j)$ is node $(i,k)$. The value of $\text{parent}(i,j)$ can be represented in (at most) $\log \log n$ bits, and the parent table $\text{parent}(i,*)$ for the tree can be recorded in $\log \log n$ bits. Since $b = \log \log n$, we can fit the parent table $\text{parent}(i,*)$ into a single computer word (hence we can treat each parent table as an integer, and perform arithmetic on this integer in a constant time). We also store a mark table for the microset $i$ where $\text{mark}(i,j) = 1$ if $\text{split}(\text{node}(i,j))$ has been performed, and $\text{mark}(i,j) = 0$ otherwise. The mark table $\text{mark}(i,*)$ can be represented in $b$ bits in a direct manner, and it is fit into a single computer word.

Suppose $q$ is a parent table representing a tree, and $a$ is a mark table, and $j$ is a node number. Then, $\text{answer}(q,a,j)$ is defined to be the nearest marked predecessor of node $j$ with respect to the preorder of $q$. Concerning this answer table, given a tree $q$ and a mark table $a$, we can compute the values of $\text{answer}(q,a,j)$ for all nodes $j$ in the tree in time linear to the number of nodes, which can be done by traversing the tree in preorder. The value of $\text{answer}(q,a,j)$ can be recorded in $\log \log n$ bits, and can be fitted into a computer word. In the answer table, $q$, as an integer, ranges from 0 to $2^\log \log n-1$, and $j$ ranges from 1 to $b$. Hence, the entire answer table can be constructed, in preprocessing step, in $O(b \log \log n) = O(n)$ time and space.

To execute $\text{find}(u)$ at the bottom level, we only look up the $\text{answer}(q,a,\text{number}(u))$ for the parent table $q$ and the mark table $a$ of microset $\text{micro}(u)$. To execute $\text{split}(u)$, we simply mark $u$ in the mark table. Each of $\text{find}$ and $\text{split}$ can be done in $O(1)$ time on the microset. The scheme of recording microsets as above enables us to execute $\text{add}$ operations quickly as follows. In $\text{add}(v,*)$, we add $v$ to the tree of the microset containing $v^*$ by making $v$ a son of $v^*$ so that the preorder of the augmented tree satisfies the above condition, which is possible owing to the definition of $v^*$ for $v$. This can be done in $O(1)$ time. If there comes to be a microset of size greater than $\log \log n$ by $\text{add}$, we first compute the preorder of the tree representing it, and then partition it into halves according to the preorder, which can be done in $O(\log \log n)$ time.

We now analyze the complexity of the above algorithm. Each $\text{find}$ only takes $O(1)$ time. Each $\text{split}$ at the bottom level (on the microset) can be performed in $O(1)$ time. At the top level, $\text{split}$ operations, which are performed by the "relabel-the-smaller-half" method, can be totally executed in $O(m+1/\log n \log \log n) = O((1+m)\log n)$ time. Also, a detailed analysis shows that $\text{split}$ operations at the middle level can be totally executed in $O(m+1/\log n \log \log n) = O((1+m)\log n)$ time.

Concerning $\text{add}$ operations, we divide a microset into halves $\log \log n$ times at the bottom level and a mezzoset $\log \log n$ times at the middle level. Dividing a microset and a mezzoset takes $O(\log n)$ and $O(\log \log n)$ time, respectively. Executing $\text{add}$ on the microset takes $O(1)$ time. Hence, $\text{add}$ operations can be done in $O(1)$ time in total.

In extending Gabow and Tarjan's algorithm to the above algorithm, the main step is to represent a microset by a tree the preorder of which gives the ordering of elements and to employ the above-mentioned answer table. In the above algorithm, we assume that the value $n$ such that $l \leq n$ is known in advance. However, the technique, presented in [6], of initializing the answer table each time $l$ doubles
enables us to remove such a restriction. We thus obtain the following.

**Theorem 2.2.** An intermixed sequence of $m$ find and split operations and $l$ add operations can be executed in $O(l+n+m)$ time and $O(l+n)$ space. □

**Theorem 2.3.** Suppose we execute, by the above algorithm, sequences of $m$ find and split operations and $l$ add operations maintaining ordered partitions $P(U_v,S_v)$ (each $U_v \subseteq V$ is initially an empty set). We assume that, given an element $u$ of $U_v$, we can find $\min(u)$ and $\max(u)$ in the corresponding sequence in a constant time, totally using $O(\Sigma l_i)$ space. Then, we can execute these sequences simultaneously in $O(\Sigma (l_i+m_i))$ time and $O(\Sigma l_i)$ space. □

3. Dynamization of Layered Structures

3.1. Layered segment tree

In this section, we recall the layered segment tree, developed by Vaishnavi and Wood [22], for the static orthogonal segment intersection search. Concerning intersections of parallel segments, the problem is essentially the one-dimensional interval intersection problem, for which efficient dynamic data structures such as priority search tree [17] are known. So, in the following, we consider the case in which a set of vertical segments is given, and a query is a horizontal segment. The case in which a query is a vertical segment is analogous.

Let $V \subseteq \{v_1, \ldots, v_N\}$ be a set of vertical segments. For simplicity, we assume that abscissae of vertical segments are different from one another. (It is easy to modify our algorithm so that it can treat vertical segments with the same abscissa; for example, when several vertical segments with the same abscissa must be handled simultaneously, we record them in a list and make them represented by the value of the abscissa.) We can then suppose that $v_i$ itself denotes the abscissa of vertical segment $v_i \in V$. Denote by $L(v)$ the interval of $v$ with respect to ordinate. We suppose that $L(v)$ is an open-closed interval, i.e., $(a,b]$, and that all the endpoints of segments in $V$ have integer values from 1 to $n$ with respect to abscissa and from 1 to $N$ (i.e., $2n$) with respect to ordinate.

We now recall the segment tree due to Bentley [1]. For an integer interval $(a,b]$, a segment tree $T(a,b)$ consists of a root $w$ associated with interval $I(w)=[a,b]$, and in the case of $b-a>1$, of a left subtree $T(a,(a+b)/2]$ and a right subtree $T((a+b)/2,b]$; in the case of $b-a=1$, the left and right subtrees are empty. For $v \in V$, a node $w$ of $T(1,N)$ is called an $t$-node of $v$ if

(i) $w$ is the root of $T(1,N)$ and $L(v) \cap I(w)$, or
(ii) the parent of $w$ is a t-node of $v$, and $L(v) \cap I(w) \subseteq I(w)$.

A node $w$ is called an $s$-node of $v$ if

(i) $w$ is the root of $T(1,N)$ and $L(v) \subseteq I(w)$, or
(ii) the parent of $w$ is a t-node of $v$, and $L(w) \subseteq L(v)$.

For each node $w$ of $T(1,N)$, let $S(w)$ be the set of $v \in V$ such that $w$ is an $s$-node of $v$, and $U(w)$ be the set of $v \in V$ such that $w$ is an $s$- or $t$-node of $v$. Note that $S(w) \subseteq U(w)$. We keep the set $S(w)$ by a doubly linked list in increasing order.

In the layered segment tree, we manipulate two kinds of ordered partitions. For each node $w$ of $T(1,N)$, we record an ordered partition $P(U(w),S(w))$. For each non-root node $w$, we record an ordered partition $P(U(w),S(w))$ where $w$ is the parent of $w$ (note that $U(w) \subseteq U(w')$). Initially, we construct these structures, which can be done in $O(n \log n)$ time and space.

Let $h$ be a horizontal segment, whose interval with respect to abscissa is $[h_1,h_2]$ and whose ordinate is $h_3$ ($1 < h_3 \leq N$). Then the query for $h$ can be executed as follows.

**QUERY:**

(i) find $v \in V$ such that $v-min(v') \in V \cup \{n+1\}$, $v \leq v'$;

(ii) if $v > n$, then stop; otherwise, $w$ := the root of $T(1,N)$;

(iii) $w := \text{find}(v)$ with respect to $P(U(w),S(w))$;

(iv) report all the vertical segments $v \in S(w)$ such that $s_1 \leq v \leq h_2$ by traversing the list representing $S(w)$;

(v) if $w$ is a leaf, then stop;

(vi) let $w_1$ be a son of $w$ with $h_3 \in I(l(w_1))$;

(vii) let $w_2$ be a son of $w$ with $h_3 \in I(l(w_2))$;

(viii) $v := \text{find}(v)$ with respect to $P(U(w_1),U(w_2))$;

(ix) return to (ii);

In the algorithm QUERY, each find can be executed in a constant time, hence Vaishnavi and Wood's result [22] is obtained: The static orthogonal segment intersection search problem can be solved in $O(\log n+k)$ query time and $O(n \log n)$ space where $k$ is the number of reported intersections.

3.2. The insertion problem

We shall mainly consider the restricted insertion problem in which segments to be inserted are in $\{v_1, \ldots, v_N\}$ and hence the ordinates of endpoints of those segments are known in advance to be integers from 1 to $N$. In applications of the insertion problem in section 4.2, this restricted problem suffices. After presenting an algorithm for this problem, we note the strategy to solve the general insertion problem in the same complexity with the restricted problem.

Concerning queries, the algorithm QUERY given above works if we maintain respective ordered partitions correctly.

We now give an algorithm INSERT which inserts a vertical segment $v$ into the set with handling those ordered partitions correctly.

**INSERT:**

(i) find $v^* \in V$ such that $v^* := \min(v') \in V \cup \{n+1\}$, $v \leq v'$;

(ii) let $w$ be the root of $T(1,N)$;

(iii) with respect to $P(U(w),S(w))$,

$\text{add}(v,v^*)$;

if $w$ is an $s$-node of $v$, then $\text{split}(v)$;

if $w$ is a leaf, then return;

let $w_s$ and $w_p$ be the sons of $w$;

with respect to $P(U(w_s),U(w_p))$,

$v_s := \text{find}(v)$;

$\text{split}(v)$;

$\text{add}(v,v^*)$;

if $w_s$ is an $s$- or $t$-node of $v$, then $\text{split}(v)$;

with respect to $P(U(w_s),U(w_p))$,

$v_p := \text{find}(v)$;

$\text{split}(v)$;

if $w_p$ is an $s$- or $t$-node of $v$, then $\text{split}(v)$;

if $w$ is an $s$-node of $v$, then return;

if $w$ is a $t$-node of $v$, then

if $L(v) \cap I(w_s) \neq \emptyset$ then

$v^* := v_s$;

$w^* := w_s$;

and recursively execute (ii);

if $L(v) \cap I(w_p) \neq \emptyset$ then

$v^* := v_p$;

$w^* := w_p$;

and recursively execute (ii);
It is seen that, in the step (ii) of INSERT, \( v^* \) for \( v \) satisfies the condition required for executing \add. It is then easy to show that the algorithm INSERT correctly maintains all the ordered partitions. We now consider the complexity of the above algorithm.

**Theorem 3.1.** An intermixed sequence of \( O(n) \) queries and insertions can be executed in \( O(n \log n + K) \) time and \( O(n \log n) \) space, where \( K \) is the total number of reported intersections.

**Proof:** The above algorithm obviously takes \( O(n \log n) \) space, so that we consider the time complexity. Suppose that, in the sequence, there are \( q \) queries and \( r \) insertions, where \( q+r=O(n) \). In the query step, \( q \) queries except those parts for \find operations can be executed in \( O(q \log n + K) \) time. In the insertion step, \( r \) insertions except those parts for \find, \split and \add operations can be executed in \( O(r \log n) \) time.

In all the steps, suppose that, for the ordered partition \( P(U(w), S(w)) \) of each node \( w \) of \( T(1,N) \), the \find and \split operations are executed \( m_1(w) \) times, and the \add operation is executed \( l_1(w) \) times. We consider that, for the root \( w \), \( m_1(w)=l_1(w)=0 \). Then, we have \( \sum_w (m_0(w)+m_1(w))=O((q+r) \log n) \) and \( \sum_w (l_0(w)+l_1(w))=O(r \log n) \), where the summations are taken over all the nodes \( w \) of \( T(1,N) \).

Let us consider the complexity to execute sequences of \( m_i(w) \) \find and \split operations and \( l_i(w) \) \add operations \( (i=0,1; \) all nodes \( w \) of \( T(1,N) \)). Owing to the structure of the segment tree, given \( v \in V \) in some sequence, we can devise a procedure to find \( \text{micro}(v) \) and \( \text{number}(v) \) in the sequence in a constant time. We then see that, from Theorem 2.3, these sequences can be executed in \( O(\sum_w (m_i(w)+l_i(w)))=O((q+r) \log n) \) time. Thus, the whole sequence can be executed in \( O(n \log n + K) \) time.

We next introduce a new operation, called \right. For a set \( V \) of vertical segments, the operation \right(i,j) finds \( v \in V \) such that \( v=\min\{v'\mid v'=n+1 \text{ or } \{v'\in V, i\leq v', j\in L(v')\} \}. That is, \right(i,j) finds the first segment in \( V \) that meets a line stretching from a point \( (i,j) \) in increasing abscissa. We can execute this \right operation by an algorithm similar to that for the \find. In this case, we do not need traversing the list of \( S(w) \), so that the following theorem is obtained.
Theorem 3.2. An intermixed sequence of $O(n)$ right operations and insertions can be executed in $O(n \log n)$ time and $O(n \log \log n)$ space. □

Finally, we consider the general insertion problem such that the coordinates of $n$ segments to be inserted are unknown and so we cannot provide the segment tree $T(1,N)$ in advance as in the above algorithm. In [4], Edelsbrunner shows that the general insertion problem (in fact, deletions are allowed in [4]) can be solved in $O(n \log n)^{1+K}$ time and $O(n \log n)$ space by dynamicizing segment trees without layered structures by means of bounded balanced trees due to Nievergelt and Reingold [18], where such a dynamization technique is first given by Lueker [15] and Willard [23] for the dynamic orthogonal range search. Combining this dynamization technique with the above algorithm on the dynamic layered segment tree, we can solve the general insertion problem in $O(n \log n + K)$ time and $O(n \log n)$ space.

3.3. The deletion problem

Using the operations link and find, we can efficiently solve the deletion problem. Given a vertical segment $v$ to be deleted, we do not remove all the elements corresponding to $v$ from the data structure, but modify them so that $v$ will no longer be reported in queries. Specifically, on each s-node $w$ of $v$, we remove $v$ out of $S(w)$. This makes the ordered partition $P(U(w),S(w))$ updated. To obtain the updated ordered partition, we have only to execute link($v$). Combining this procedure with the algorithm QUERY, we can solve the deletion problem correctly. For completeness, we describe below a procedure for deleting a vertical segment $v$.

DELETE:
(i) let $w$ be the root of $T(1,N)$;
(ii) if $w$ is an s-node of $v$, then
   link($v$) with respect to $P(U(w),S(w))$, and return;
   if $w$ is a t-node of $v$, then
   let $w_{l}$ and $w_{r}$ be the sons of $w$;
   if $L(v) \cap I(w) \neq \emptyset$ then
      $w:=w_{l}$, and recursively execute (ii);
   if $L(v) \cap I(w) \neq \emptyset$ then
      $w:=w_{r}$, and recursively execute (ii);

The above theorem on the complexity can be obtained in a way similar to Theorem 3.1 (we use Theorem 2.1 in place of Theorem 2.3), so that we omit the proof.

Theorem 3.3. An intermixed sequence of $O(n)$ queries and deletions can be executed in $O(n \log n + K)$ time and $O(n \log n)$ space, where $K$ is the total number of reported intersections. □

3.4. Special cases in which candidates for queries are known in advance

When we use the data structures for the dynamic segment intersection search in order to solve other geometric problems as in section 4, it is often the case that candidates for queries are known in advance. Here, it should be noted that we do not consider the off-line or batched versions of those problems in which a sequence of queries and updates is completely given in advance. In our problems, we know $O(n)$ candidates for queries and updates, but we do not know any sequence of those operations and we must execute each of them on-line. We will show that such problems can be solved in a simpler fashion.

First, we consider the problem of executing a sequence of $O(n)$ deletions and right operations, where candidate points for right are known in advance. Let $V$ be a set of vertical segments, and $Q$ be a set of those candidate points. We suppose that the endpoints of segments in $V$ and the points in $Q$ have integer values, different from one another, from 1 to $n$ with respect to abscissa and integer values from 1 to $N$ with respect to ordinate. In section 3.1, we consider that $v \in V$ and $q \in Q$ themselves denote their respective abscissa. For each node $w$ of the segment tree $T(1,N)$, let $S(w)$ be the set of $v \in V$ such that $w$ is an s-node of $v$, and $Q(w)$ be the set of $q \in Q$ such that the ordinate of $q$ is contained in $I(w)$.

We can then solve the problem in $O(n \log n)$ time and space only by maintaining ordered partitions $P(U(w),S(w))$ for all nodes $w$ of $T(1,N)$ as follows. First, those ordered partitions are constructed. In order to delete $v \in V$, we have only to execute link($v$) with respect to $P(U(w),S(w))$ for each s-node $w$ of $v$ in $T(1,N)$. To execute right for $q \in Q$, we let $v:=\text{find}(q)$ with respect to $P(U(w),S(w))$ for nodes $w$ of $T(1,N)$ such that the ordinate of $q$ is in $I(w)$ (note that the number of such nodes $w$ is $O(\log n)$), and find the segment nearest to $q$ among those $w$'s. It is easily seen that this procedure takes $O(n \log n)$ time and space. Although we here consider deletions and right operations only, it is straightforward to allow queries in the above algorithm.

For the problem of executing a sequence of $O(n)$ insertions and right operations, where candidate segments for insertions and candidate points for right are known in advance, similar techniques can be used. Thus, when candidates for queries are known in advance, we can solve the problem in a simpler manner than in sections 3.2 and 3.3.

3.5. The orthogonal range search problem

The orthogonal range search problem is stated as follows: Given a set $Q$ of $n$ points $q_{1}, q_{2}, \ldots, q_{n}$ in the plane, report all the points in $Q$ that are contained in a given orthogonal query rectangle. (A rectangle is called orthogonal if its sides are orthogonal.) Concerning this problem, Chazelle [3] gives an algorithm, to the static problem, with $O(\log n+k)$ query time and $O(n \log n \log \log n)$ space. Overmars and van Leeuwen [19] show the dynamic problem can be solved in $O((\log n)^{2}+k)$ query time, $O((\log n)^{3})$ insertion time, $O(n \log n)$ deletion time and $O(n \log n)$ space.

We shall show that the techniques in sections 3.2 and 3.3 can also be applied to this problem. We first outline the range tree (Bentley [1]). Suppose that the abscissae of points in $Q$ are integers, different from one another, ranging from 1 to $n$, and that the ordinate of points in $Q$ are integers from 1 to $N \leq n$. We consider that $q \in Q$ itself denotes its abscissa. For each node $w$ of the segment tree $T(1,N)$, let $Q(w)$ be the set of $q \in Q$ such that the ordinate of $q$ is in $I(w)$. The range tree is $T(1,N)$ such that, at each node $w$, the set $Q(w)$ is stored in a sorted order. Then, the static orthogonal range search problem can be solved by this range tree in $O((\log n)^{2}+k)$ query time and $O(n \log n)$ space.

It is known that the range tree is given layered structures to solve the static problem in $O((\log n)^{2}+k)$ query time and
that, at each node \( w \), the ordered partition \( P(Q(w,)),Q(w)) \) is stored where \( w_r = w \) if \( w \) is the root, and \( w_p \) is the parent of \( w \), otherwise. To this layered range tree, the techniques of dynamicizing layered structures can be directly applied, and we obtain the following.

**Theorem 3.4.** Concerning the orthogonal range search problem, an intermixed sequence of \( O(n) \) queries and updates can be executed on-line in \( O(n \log n + K) \) time and \( O(n \log n) \) space if, as updates, only insertions are allowed or only deletions are allowed, where \( K \) is the total number of reported points. □

4. Applications to Geometric Problems on Orthogonal Objects

This section presents applications of the data structures developed in section 3 to various geometric problems related to orthogonal segments. Here we first describe a paradigm which enables us to apply the data structures for the deletion problem to problems for a geometric intersection graph of orthogonal segments. Such a paradigm is given in Lipski [12], Imai and Asano [10]. We then show applications of both deletion and insertion problems.

4.1. A paradigm of solving geometric problems by combining graph algorithms and geometric data structures

An intersection graph of \( n \) orthogonal segments is obtained by identifying each segment with a vertex and connecting two vertices by an edge if their corresponding segments intersect. Various geometric problems on orthogonal segments can be solved by applying graph algorithms to their intersection graph. In such cases, if we first construct the intersection graph completely by computing all the intersections among them, represent the graph by adjacency lists, and execute graph algorithms in an ordinary manner, it costs \( O(n^2) \) time and space. However, there is another way of employing such approaches without explicitly enumerating all the edges of the intersection graph, as we will show, by executing graph algorithms in a different manner with using the data structures developed in section 3.

First, consider depth-first search, which finds a depth-first spanning forest, on a directed graph \( G=(V,A) \) with vertex set \( V \) and arc set \( A \). Let \( n=|V| \) and \( m=|A| \). An undirected graph \( G=(V,E) \) with vertex set \( V \) and edge set \( E \) is regarded as a directed graph obtained from \( G \) by replacing each edge by two reversely-oriented arcs connecting the same vertices. In Fig.2, we give a procedure SEARCH for finding a depth-first spanning forest, which is represented by \( p() \), and computing \( df\text{number}(v) \).

In an ordinary implementation of the procedure, a vertex removed from \( W \) is labeled, and, in DFS(\( u \)), we iterate to scan arcs going out of \( u \) until an arc \((u,v)\) such that \( v \) is not labeled is found. If this technique is employed and the graph is represented by adjacency lists, the procedure can be executed in \( O(m) \) time and space.

We can implement the procedure in a different way, for which two procedures are introduced. At any stage of the procedure, we update the graph (specifically, delete all arcs coming into vertices in \( V-W \)) and maintain the data structures representing the graph at that time. Then, for \( v \in W \), DEL(\( v \)) is a procedure which makes \( W=W-\{v\} \) and updates the data structures as mentioned just above. For \( u \in V \), LIST1(\( u \)) is a function which returns a vertex \( v \) such that \((u,v)\) is an arc in the current graph. By means of DEL and LIST1, the procedure can be executed as follows. When removing \( w \) out of \( W \) in the main part, and removing \( v \) out of \( W \) in DFS(\( u \)), we execute DEL(\( v \)) and DEL(\( (u,v) \)), respectively. In order to find an arc \((u,v)\in A \) with \( v \in W \) in DFS(\( u \)), we execute \( v=\text{LIST1}(u) \). The validity of this implementation is obvious. Concerning the complexity, since DEL and LIST1 are executed \( O(n) \) times in this implementation, we have the following.

**Theorem 4.1.** Suppose that a sequence of \( O(n) \) DEL's and LIST1's can be executed in \( g_T(n,m)=\Omega(n) \) time and \( g_S(n,m)=\Omega(n) \) space. Then, the depth-first search can be executed in \( O(g_T(n,m)) \) time and \( O(g_S(n,m)) \) space. □

Of course,德尔 and LIST1 can be implemented by representing \( G \) with adjacency lists, in this case, we trivially have \( g_T(n,m),g_S(n,m)=O(m) \), which reduces to the ordinary implementation.

Now, consider the depth-first search on an intersection graph \( G \) of \( n \) orthogonal segments. \( G \) can be represented by the data structure for the deletion problem in section 3, where the edges of \( G \) are not surely recorded explicitly, but they can be immediately obtained by means of queries. DEL(\( v \)) for \( G \) precisely corresponds to deleting the segment \( v \) from our data structure, and LIST1(\( u \)) is similar to the right operation for \( u \) in fact, LIST1 is easier than right. Then, from the discussions in section 3, it is seen that, by these data structures, \( g_T(n,m),g_S(n,m)=O(n \log n) \), and we obtain the following.

**Theorem 4.2.** The depth-first search on an intersection graph of \( n \) orthogonal segments can be executed in \( O(n \log n) \) time and space. □

This result is quite preferable since \( m \), the number of edges of the intersection graph, can be \( \Theta(n^2) \), and, in such a case, the naive algorithm takes \( \Theta(n^2) \) time and space. Our algorithm may be a bit inefficient with respect to the space complexity in case \( m=O(n \log n) \), but, in such a case, we can
modify our algorithm so as to require $O(\min(m, cn \log n))$ space, where $c$ is a given constant, by $O(n \log n)$ preprocessing as follows. First, we compute $m$, the number of intersections among $n$ segments, which can be done in $O(n \log n)$ time and $O(n)$ space by the plane-sweep algorithm in Bentley and Ottmann [2]; if $m > cn \log n$, we solve the problem as described above; otherwise, we actually construct the intersection graph by enumerating all the intersections, which can be done in $O(n \log n + m)$ time and $O(m + n)$ space by the algorithm in [2], and solve the problem by representing the graph by adjacency lists.

This kind of approach can be applied to other graph problems, besides depth-first search, concerning breadth-first search, the biconnected components, the max-flow problem of unit capacities, and so on; we can then solve problems on intersection graphs on orthogonal segments by using our data structures. Here it should be noted that graph notions on the intersection graph such as a maximum independent vertex set and the connectedness have respective geometric meanings, which enables us to solve practically useful geometric problems based on this paradigm.

4.2. Applications

(1) Finding the biconnected components of an intersection graph of $n$ orthogonal segments.

Based on the paradigm presented in section 4.1, we can find the biconnected components of the graph $G$ in $O(g_f(n,m))$ time and $O(g_b(n,m))$ space as follows. First, construct a depth-first search spanning tree as above. Then, the main step is to compute, for each $u \in V$, $df\min(u) \equiv \min\{df\text{number}(v)\}$ where the minimum is taken over $v \in V$ such that $v = u$ or $(u, v)$ is an edge in $G$ with $v \neq p(u)$ (see Tarjan [21]). Again, based on the paradigm, it is straightforward to show that $df\min(u)$ for all $u \in V$ can be computed in $O(g_f(n,m))$ time and $O(g_b(n,m))$ space. For the intersection graph of $n$ orthogonal segments, we have $g_f(n,m), g_b(n,m) = O(n \log n)$ by employing the data structure for the deletion problem, so that this problem (1) can be solved in $O(n \log n)$ time and space. Related to the problem (1) is the problem of finding the connected components of the intersection graph. For this problem, techniques developed here yields an algorithm with $O(n \log n)$ time and space, while there is an $O(n \log n)$-time and $O(n)$-space algorithm [9], which uses the plane-sweep technique.

(2) Finding, among $n$ orthogonal segments such that no two of horizontal segments and no two of vertical ones intersect, a maximum number of non-intersecting segments. (Fig.3)

This problem is equivalent to finding a maximum independent set of the intersection graph of those segments. Since no two of parallel segments intersect, the intersection graph is bipartite, so that a maximum independent set can be found by using a bipartite matching algorithm. Since the main steps of the efficient bipartite matching algorithm given by Hopcroft and Karp [7] are to execute depth-first searches and breadth-first searches, hence the paradigm also applies to their algorithm. However, those depth-first searches and breadth-first searches (especially, the former) must be performed on graphs obtained by partly modifying the original graph, so that only the data structures in section 3 are not sufficient to handle the modified graphs. In [10], by slightly extending those data structures, we present data structures that can handle the modified graphs, and show that the problem (2) can be solved in $O(n^{3/2} \log n)$ time and $O(n \log n)$ space.

This problem (2) is a major step in solving the following problem (3) (see Lipski et al. [13]).

(3) Partitioning a given rectilinear region with $n$ edges into a minimum number of disjoint rectangles (Fig.4).

Using the result for the problem (2), we see that the problem (3) can be solved in $O(n^{3/2} \log n)$ time and $O(n \log n)$ space.

(4) Among $n$ orthogonal segments, finding a minimum number of segments whose removal makes the remaining set disconnected.

A set of segments is connected if its intersection graph is connected. Then, this problem (4) is equivalent to finding the connectivity $k_0$ of the intersection graph of those $n$ orthogonal segments. The connectivity problem can be solved by algorithms for the max-flow problem with unit capacities (e.g., an algorithm by Even and Tarjan [5]) and again the paradigm applies. Only the data structures considered in this paper are insufficient to execute the algorithm in [5], but, as in the problem (2), by using the modified data structures presented in [10], we can solved this problem in $O(k_0 n^{3/2} \log n)$ time and $O(n \log n)$ space.

(5) Discerning whether the set of $n$ point pairs can be wired in Manhattan fashion on a single layer.

A wiring connecting $n$ pairs of points on the grid by wires along grid lines so that the wires do not intersect one another is called a Manhattan wiring if no wire has more than one
The problem (5) is to discern the existence of a Manhattan wiring for \( n \) pairs of points. There are at most two wires connecting a pair of points in Manhattan fashion, each of which is called a \( M \)-wire of the pair of points. Then, there is a Manhattan wiring iff there is an independent set of size \( n \) in an intersection graph of all the \( M \)-wires. In [16], Masuda et al. present an \( O(n^2) \) algorithm by employing a bit modified depth-first search on the intersection graph. Based on the paradigm in section 4.1, Imai and Asano [10] show how to execute this modified depth-first search efficiently. Combining such an implementation with the data structure for the deletion problem, the problem (5) can be solved in \( O(n \log n) \) time and space.

An application of the data structure for the deletion problem is discussed in [11] to the problem of finding a shortest path, in respect to \( L_1 \) metric, between two given points in a rectilinear region. The applications described above are those for the deletion problem. We below show two applications of the data structures for the insertion problems.

(6) Finding the closure of a set of \( n \) orthogonal rectangles (Fig.6).

A region \( R \) in the \((x,y)\)-plane is called closed if, for any pair of points \((x_1,y_1), (x_2,y_2)\) \( \in R \) with \((x_1-x_2)(y_1-y_2)<0 \) which are connected in \( R \), we have \((x_1,y_2), (x_2,y_1)\) \( \in R \). The unique smallest closed region containing \( R \) is called the closure of \( R \). In [14], Lipski and Papadimitriou consider the problem of finding the closure of a set of \( n \) given orthogonal rectangles. The algorithm given in [14] can be implemented so as to run in \( O(n \log n) \) time and \( O(n \log n) \) space by using the algorithm for the insertion problem. Here, it should be noted that, concerning the problem itself of finding the closure of \( n \) rectangles, an \( O(n \log n) \)-time and \( O(n) \)-space optimal algorithm is recently given [20], which is different from that in [14] and uses the plane-sweep technique.

(7) Heuristics for the minimum-perimeter partitioning of a rectilinear region.

The minimum-perimeter partitioning problem is to partition a rectilinear region into disjoint rectangles in such a way that the sum of the perimeters of those rectangles is as small as possible. This problem is known to be NP-complete, and several heuristics have been considered. We here take up the following simple heuristic, to which the data structures for the insertion problem can apply (Fig.7).

(i) For each concave point \( P \) in the given rectilinear region, stretch horizontal and vertical directed segments starting from \( P \) into the region until they meet an edge of the region; take the shorter of the two, and denote it by \( l_P \).

(ii) Arrange all the concave points \( P \) in the increasing order of the lengths of \( l_P \).

(iii) In that order, stretch and add a segment starting from \( P \) in the direction of \( l_P \) until it meets an edge of the region or another segment that has already been added to the region.

In this heuristic algorithm, the primary step is to stretch and add \( O(n) \) segments one by one in the given order, and so this heuristic can be executed in \( O(n \log n) \) time and \( O(n \log n) \) space.
5. Concluding Remarks

In this paper, we have shown that, in the orthogonal segment intersection search and the orthogonal range search problems, each of queries and updates can be performed in \(O(\log n)\) time if the update operation is restricted to either insertion or deletion and the running time is evaluated in an amortized sense. By means of the algorithms developed here for the orthogonal segment intersection search, we can solve various geometric problems relevant to \(n\) orthogonal segments with better running time (mostly, \(O(\log n)\)) than by existing algorithms that make use of data structures for the dynamic segment intersection search. Besides applications presented in this paper, our algorithms would have much more applications in many fields. Still, there have remained open the questions of how to handle insertions and deletions together and how to obtain the above time complexity even in the worst case.

We have also introduced the incremental set-splitting problem on disjoint sets, and have presented a linear-time algorithm for it. It seems that less attention has been paid for the set-splitting problem than for the set-union problem. As is seen from the discussions in section 3, the set-splitting problem is "dual" in some sense to the set-union problem, and so the (incremental) set-splitting algorithms would have further applications besides those presented in this paper.

Acknowledgments

The authors would like to express their sincere gratitude to Professor Masao Iri of the University of Tokyo for his encouragements and valuable discussions. They are also grateful to Dr. Robert E. Tarjan of AT&T Bell Laboratories for his comments on the set-splitting problem. This research was supported by the Grant in Aid for Scientific Research of the Ministry of Education, Science and Culture of Japan under Grant: 59550239 (1984).

References