数理解析研究所講究録
第534巻 1984年 247-261

Dynamic Orthogonal Segment Intersection Search and Its Applications

Hiroshi IMAI and Takao ASANO

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

Abstract

This paper develops data structures for maintaining a set of orthogonal segments, where a segment in the plane is called *orthogonal* if it is horizontal or vertical. By combining the data structures with graph algorithms for depth-first search, matchings, etc., or with algorithms in computational geometry, we obtain algorithms, with better time complexities, for problems on orthogonal segments, which arise in various fields such as VLSI design, computer graphics and database system.

1. Introduction

The *orthogonal segment intersection search problem* is stated as follows: Given a set S of n orthogonal segments in the plane, report all the segments in S that intersect a given orthogonal query segment. Here a segment in the plane is called *orthogonal* if it is horizontal or vertical. The problem is called *static* if the set S remains unchanged, and *dynamic* if the set S is updated by insertions or deletions. Until now, efficient data structures for this problem have been developed [1,2,3,16,18,23]. For the static problem, Chazelle's $O(\log n+k)$-time and $O(n)$-space algorithm is the best known solution [2], where k is the number of reported intersections. For the dynamic problem in which the size of the underlying set in the course of the algorithm is known to be $O(n)$ in advance, McCreight [18] showed an algorithm with $O((\log n)^2+k)$ query time, $O((\log n)^2)$ update time and $O(n)$ space, and Lipski and Papadimitriou [16] presented an algorithm with $O(\log n \log\log n+k)$ query time, $O(\log n \log\log n)$ update time and $O(n \log n)$ space. For the general dynamic problem, Edelsbrunner [3] gave an algorithm with $O((\log n)^2+k)$ query time, $O((\log n)^2)$ update time and $O(n \log n)$ space.

In this paper, we consider two restricted types 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 online in $O(n \log n+K)$ time and $O(n \log n)$ space, where K is the total number of reported intersections. That is, each of queries and updates in both problems can be executed in $O(\log n(+k))$ time in an *amortized* sense. Our data structure is a combination of the layered segment tree developed by Vaishnavi and Wood [23], which was

originally designed for the static problem, and linear-time set-union and set-splitting algorithms due to Gabow and Tarjan [5]. Here, in order to obtain the result for the insertion problem described above, we must generalize Gabow and Tarjan's set-splitting algorithm in such a way that the ground set can be incremented, which is the main problem discussed in section 2. In section 3, we discuss about dynamic layered segment trees, and show the main theorems for the insertion and deletion problems.

By combining the data structures with graph algorithms or with algorithms in computational geometry, we obtain various new results for problems on orthogonal segments, which arise in many fields. A basic idea combining the data structure for the deletion problem with graph algorithms such as depth-first search and breadth-first search was found in Lipski [13]. Imai and Asano [7] independently obtained a similar idea, and developed the idea further in [9]. Here, from the more general point of view, as applications of both insertion and deletion problems, we discuss, in section 4, seven problems the time complexities of which are improved by means of the data structures developed in this paper.

2. Set-Union and Set-Splitting Problems

Let $V=\{1,...,n\}$, and S and U be such that $S\subseteq U\subseteq V$. Suppose that $S=\{s_1,...,s_p\}$ and $0\equiv s_0<s_1<\cdots<s_p<s_{p+1}\equiv n+1$, where s_0 and s_{p+1} are dummy elements. Define $V(s_i)$ $(i=1,...,p+1)$ to be $\{u \mid u\in U, s_{i-1}<u\leqslant s_i\}$. Then $\{V(s_i) \mid i=1,...,p+1\}$ is a partition of U, and, for any $u_i\in V(s_i)$ and $u_j\in V(s_j)$ such that $1\leqslant i<j\leqslant p+1$, we have $u_i<u_j$. We call $(V(s_1),...,V(s_{p+1}))$ 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 $P(U,S)$, we consider the following four operations, *find*, *link*, *split* and *add* (Fig.2.1):

find: For $u\in U$, *find*(u) returns $s_i\in S$ such that $u\in V(s_i)$.

link: For $s_i\in S$, *link*(s_i) adds all the elements of $V(s_i)$ to $V(s_{i+1})$, and modifies the ordered partition to $P(U,S-\{s_i\})$. By *link*(s_i), S is updated to $S-\{s_i\}$.

split: For $u\in U-S$, *split*(u) first executes $s_i:=$*find*(u) and then divides the set $V(s_i)$ into two sets, one containing all $v\in V(s_i)$ with $v\leqslant u$, the other all $v\in V(s_i)$ with $v>u$. The ordered partition is modified to $P(U,S\cup\{u\})$. By *split*(u), S is updated to $S\cup\{u\}$.

add: For $v\in V-U$, let $v^*\in U$ be defined by $v^*=\min\{u \mid u\in U\cup\{n+1\}, u>v\}$. For v and v^*, *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 *add*(v,v^*), U is updated to $U\cup\{v\}$. (Note that, in *add*(v,v^*), v^* satisfying the above condition is given in advance with v.)

Concerning the general problem in which *find*, *link*, *split* and *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 *find* and *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 *link* operation can unite adjacent two sets in ordered partition only. For

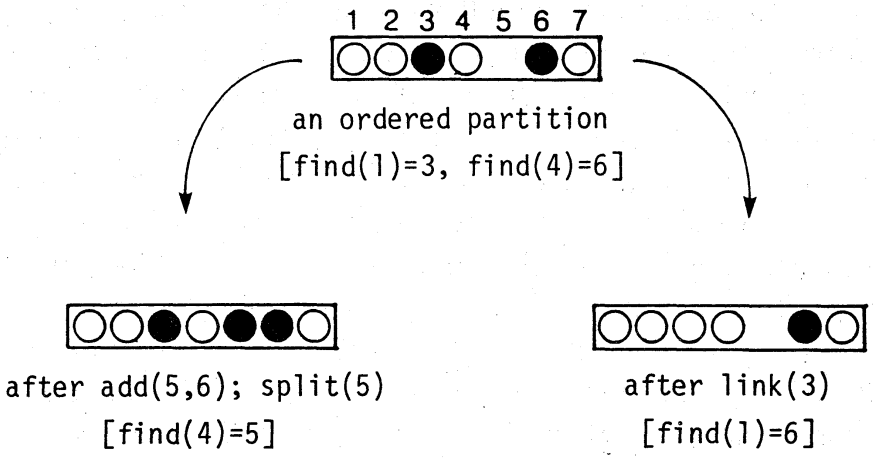

Fig.2.1. Ordered partitions

(○ implies i ∈ U; ● implies i ∈ S)

this problem, Gabow and Tarjan [5] showed the following.

Theorem 2.1. [5] Suppose that a subset U of V is given in sorted order. Let $n'=|U|$. Then, starting with an ordered partition $P(U,U)$, an intermixed sequence of m *find* and *link* operations can be executed in $O(m+n')$ time and $O(n')$ space. □

The problem in which *find* and *split* operations are executed is the *set-splitting problem* first considered by Hopcroft and Ullman [6]. They showed that, starting with an ordered partition $P(V,\varnothing)$, an intermixed sequence of m *find* and *split* operations can be executed in $O((m+n)\log^* n)$ time and $O(n)$ space. Gabow and Tarjan [5] showed that the sequence can be executed in $O(m+n)$ time and $O(n)$ space.

We call the problem, in which, starting with $P(\varnothing,\varnothing)$, an intermixed sequence of m *find* and *split* operations and l *add* operations is executed, the *incremental set-splitting problem*. Note that $l \leqslant n$. 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 first show that Hopcroft and Ullman's [6] set-splitting algorithm can be extended to the incremental set-splitting problem. Specifically, we show the following.

Theorem 2.2. Starting with an empty ordered partition $P(\varnothing,\varnothing)$, a sequence of m *find* and *split* operations and l *add* operations can be executed in $O((l+m)\log^* l)$ time and $O(l)$ space. □

Define a function $F(i)$ $(i \geqslant 0)$ by $F(0)=1$ and $F(i)=F(i-1)2^{F(i-1)}$ $(i \geqslant 1)$. Also, define a function $G(l)$ $(l \geqslant 1)$ by $G(l)=\min\{i \mid F(i) \geqslant l\}$. Note that $G(l)$ is $\Theta(\log^* l)$ and increases very slowly. Hopcroft and Ullman's set-splitting algorithm [6] 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 $i(\geqslant 1)$ has between one and $2^{F(i-1)}$ sons, hence it has at most $F(i)$ leaves. A node of level i is called *complete* if it has $F(i)$ leaves among its descendants, and is called *incomplete*, otherwise.

We keep the ordered partition by the following data structures. The ground set U is kept by a 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. A set of leaves of all trees correspond to some subset W of U, and not necessarily to the whole ground set U, where each element u such that *split*(u) has been executed is in the set W. For each $u \in U$, we record $p(u) \equiv \min\{u' \mid u' \in W, u' \geqq u\}$.

In order to execute *find*(u), we find the root of a tree that contains a leaf corresponding to $p(u)$, and return the name associated with that root. Concerning *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 *split*(u). If $u \in W$, we divide the tree containing u along a path from a leaf to the root of that tree as in Hopcroft and Ullman's static set-splitting algorithm. If $u \notin W$, 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 as above (if $p(u)$ is the leftmost leaf, we do not partition any tree). 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. At this stage, if the number of sons of a node of level i comes to be 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.

Let us evaluate the time complexity of the above algorithm. Each *add* operation takes a constant time, and each *find* operation takes a time linear to the height of a tree. In the above algorithm, each non-leaf node can have at most two incomplete sons, so that the height of a tree is at most $G(l)+1$. Hence, each *find* takes $O(G(l))$ time.

Now we consider the complexity for executing *split*. Concerning incomplete nodes, the number of incomplete nodes moved in one *split* operation is at most the height of a tree, i.e., at most $G(l)+1$, hence the total number of times to move an incomplete node is $O(lG(l))$. The remaining problem is to count the number of times to move a complete node. For this purpose, we use the following technique often used in analyzing the amortized time complexity of algorithms (see Tarjan [22]). In adding a new leaf, we put real-valued *chips* on non-leaf nodes, and, in moving a complete node, we consume one chip of the father of the node. Then, the number of times to move a complete node is bounded by the number of chips put in the course of the algorithm if the number of chips on each non-leaf node does not become negative at any stage of the algorithm.

A non-leaf node having k complete sons is called *admissible* if either (i) it has at least $k+(k \log k)/2$ chips and its rightmost son is complete, or (ii) it has at least $k+(k \log k)/2+\max\{0, (\log(k+1))/2+2-d\}$ chips where its rightmost son is incomplete and requires d leaves in order to become complete. If a non-leaf node is admissible, the number of chips on it is nonnegative.

We now show the following, from which, with the above discussions, Theorem 2.2 is obtained.

Lemma 2.1. The total number of times to move a complete node in all the *split* operations is $O(lG(l))$.

Proof: We put chips as follows: In adding a new leaf to a tree, we put two chips on each non-leaf node on the path from the leaf to the root. Then, the total number of chips needed is $O(lG(l))$, hence, we have only to show that all the non-leaf nodes are admissible in the computation when we put chips as above.

The number of chips changes when a leaf is added to a tree, or a tree is partitioned into two. We first consider the case when a leaf is added to a tree. New nodes created at this stage are apparently admissible. Among old nodes whose chips are modified by this operation, a node, whose rightmost son is not changed and is still incomplete, remains admissible, too. Among those old nodes, consider a node whose rightmost son becomes complete. This node currently has $[k+(k\log k)/2]+[2+(\log(k+1))/2]$ chips, whose number is easily seen to be greater than $(k+1)+((k+1)\log(k+1))/2$. Hence, this node is admissible. Among old nodes, consider a node whose rightmost son is a node newly created at this stage. Let i be the level of this node. In the case of $i=1$, this node is easily seen to be admissible. In the case of $i>1$, since $k\leqslant 2^{F(i-1)}-1$, we have

$$k+\frac{1}{2}k\log k+2+\frac{1}{2}\log(k+1)-(F(i-1)-1)\leqslant k+\frac{1}{2}k\log k+2,$$

and so this node is admissible.

Next, consider the case when a tree is divided into two. Consider a father node whose k complete sons are divided into two, where two cases occur. In the first case, a complete son among those k sons is divided into two incomplete nodes to both left and right trees. Suppose that the other $k-1$ complete nodes are distributed into k_1 and k_2 nodes in the left and right trees, respectively. Note that $k_1+k_2+1=k$. By simple calculation, we have

$$\min\{k_1,k_2\}+[(k_1+\frac{1}{2}k_1\log k_1)+(2+\frac{1}{2}\log(k_1+1))-1]+(k_2+\frac{1}{2}k_2\log k_2)$$

$$\leqslant k+\frac{1}{2}k\log k.$$

After consuming $\min\{k_1,k_2\}$ chips in moving complete nodes, chips left can be distributed so that two nodes corresponding to the original father node are admissible.

In the second case, $k=k_1+k_2$ complete sons are divided into k_1 and k_2 complete nodes in the left and right trees, respectively. Then we have

$$\min\{k_1,k_2\}+(k_1+\frac{1}{2}k_1\log k_1)+(k_2+\frac{1}{2}k_2\log k_2)\leqslant k+k\log k.$$

Hence, after consuming $\min\{k_1,k_2\}$ chips, chips at hand can be distributed so that two nodes corresponding to the original father node are admissible. □

Next, extending Gabow and Tarjan's algorithm, we show the following.

Theorem 2.3. An intermixed sequence of m *find* and *split* operations and l *add* operations can be executed in $O(l+m)$ time and $O(l)$ space with $O(n)$ preprocessing and extra space of size $O(n)$ for the answer table.

Proof: Since almost all of Gabow and Tarjan's algorithm for the set-splitting problem applies to the incremental problem, we note only those parts of their algorithm which should be modified or remarked in the incremental problem.

The first problem is how to represent the ground set U in order to obtain $O(l+m)$ bound. We use three levels of set: microsets, mezzosets and macrosets. (Since we employ the parent encoding scheme as described below, two level of sets seems to be insufficient for our purpose.) The set U is partitioned into consecutive elements, called mezzosets, of size at most $\lfloor \log n \rfloor$ in such a way that there is $O(|U|/\log n)$ mezzosets. Each mezzoset $\tilde{U}$ is partitioned into consecutive elements, called microsets, of size at most $\lfloor \log \log n \rfloor$ in such a way that the number of microsets in $\tilde{U}$ is $O(|\tilde{U}|/\log \log n)$.

The second problem is how to execute the table-lookup method on microsets so that *add* operations can also be executed efficiently. In the static problem, each microset is represented as a path, and is encoded into a single computer word through its parent table. In the incremental case, we represent each microset by a forest whose preorder gives an ordering of elements in the set in decreasing order, and encode it through the parent table.

This enables us to execute *add* operations quickly as follows. In $add(v,v^*)$, we add v to the forest of the microset containing v^* by making v^* the parent of v so that the preorder of the augmented forest satisfies the above condition, which is possible owing to the definition of v^* for v. If there comes to be a microset of size greater than $\lfloor \log \log n \rfloor$ by *add*, we first compute the preorder of the forest representing it, and then partition it into two halves according to the preorder, which can be done in $O(\log \log n)$ time. If there comes to be a mezzoset of size greater than $\lfloor \log n \rfloor$, we simply divide it into two halves.

Concerning the table lookup on microsets, we construct the answer table for parent tables and mark tables, which is the preprocessing. Here we must redefine the answer table for the incremental problem. Suppose that f is the forest, and a is a mark table of it, and j is a node in the forest. In Gabow and Tarjan [5], $answer(f,a,j)$ is defined to be the nearest marked ancestor of node j in the forest f. For our problem, we redefine $answer(f,a,j)$ to be the nearest marked predecessor of node j with respect to the preorder of f. Concerning the redefined answer table, given a forest f and a mark table a, we can compute the values of $answer(f,a,j)$ for all nodes j in the forest in time linear to the number of nodes, which can be done by traversing the forest in preorder. Thus, the preprocessing, to construct the answer table, can be done in $O(n)$ time and space.

The last problem to be mentioned is on the "relabel-the-smaller-half" method for the incremental set-splitting problem, since we use it for the unsplit microsets and also the unsplit mezzosets. It is noted that a sequence of $\tilde{m}$ *find* and *split* operations and $\tilde{l}$ *add* operations can be executed in $O(\tilde{m}+\tilde{l}\log \tilde{l})$ time by this method, which can be shown easily. □

Theorem 2.4. Consider sequences of m_i *find* and *split* operations and l_i *add* operations maintaining ordered partitions $P(U_i,S_i)$ whose ground set U_i are subsets, initially empty sets, of V. We assume that, given an element u of U_i, we can find the microset, $micro_i(u)$, containing u and the number, $number_i(u)$, of u within its microset in a constant time, totally using $O(\sum_i l_i)$ space (concerning $micro()$ and $number()$, see Gabow and Tarjan [5]). Then, we can execute these sequences

simultaneously in $O(n+\sum_i (l_i+m_i))$ time and $O(n+\sum_i l_i)$ space.

Proof: In each sequence, we use the same answer table. That is, in the preprocessing step, we construct only one answer table, which takes $O(n)$ time and space. Due to the assumption that, for $u \in U_i$, $micro_i(u)$ and $number_i(u)$ can be found in a constant time, we can execute each sequence in $O(l_i+m_i)$ time and $O(l_i)$ space by the algorithm in Theorem 2.3. Then, the total time and space complexities are $O(n+\sum_i (l_i+m_i))$ and $O(n+\sum_i l_i)$, respectively. □

3. Dynamic Layered Segment Tree

3.1. Layered segment tree

In this section, we recall the layered segment tree, developed by Vaishnavi and Wood [23], 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 [18] 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 $V \subseteq \{v_1,...,v_n\}$ 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 by a list and make them represent 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 $L(v_i)$ the interval of v_i with respect to ordinate. We suppose that $L(v_i)$ 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 $N(\leqslant 2n)$ 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 $T(a,\lfloor(a+b)/2\rfloor)$ and a right subtree $T(\lfloor(a+b)/2\rfloor,b)$; in the case of $b-a=1$, the left and right subtrees are empty (Fig.3.1 and Fig.3.2). 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 $L(v) \not\supset I(w)$, or

(ii) the parent of w is a t-node of v, and $\varnothing \neq L(v) \cap I(w) \subset I(w)$.

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

(i) w is the root of $T(1,N)$ and $L(v)=I(w)$, or

(ii) the parent of w is a t-node of v, and $I(w) \subseteq L(v)$.

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 $P(U(w),S(w))$. Also, for each non-root node w, we record an ordered partition $P(U(w_\pi),U(w))$ where w_π is the parent of w (note that $U(w) \subseteq U(w_\pi)$) (Fig.3.3). Initially, we construct these

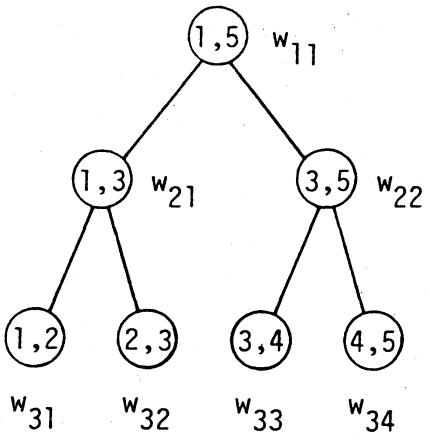

Fig.3.1. The segment tree T(1,5)

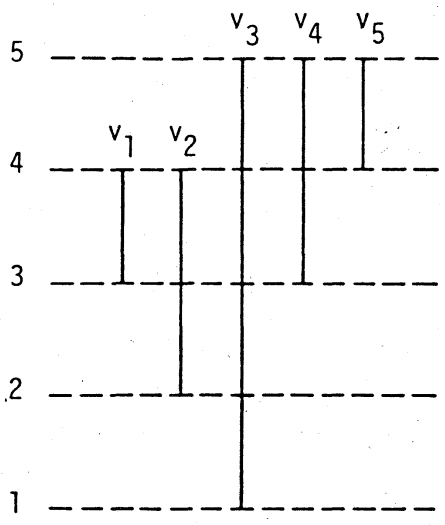

Fig.3.2. Vertical segments

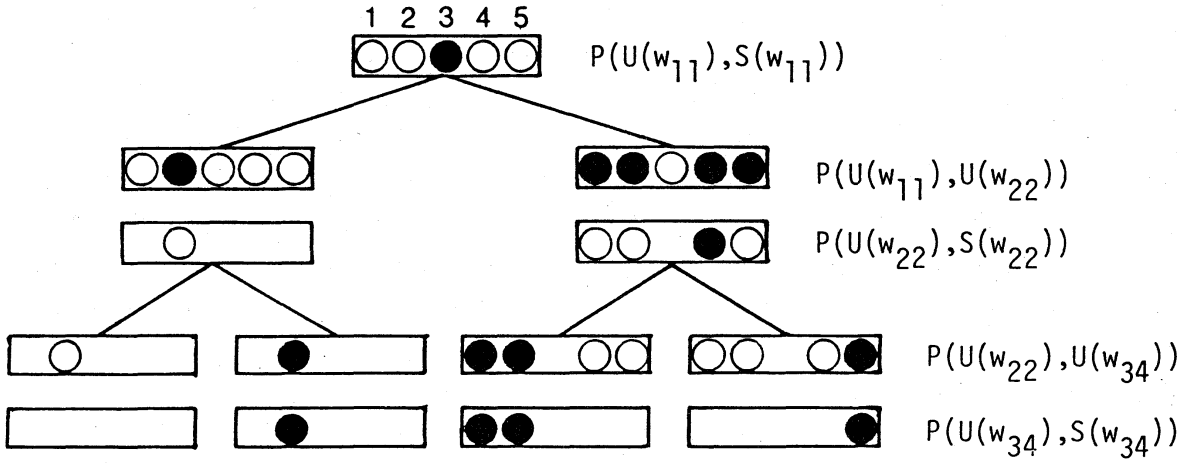

Fig.3.3. The layered segment tree for vertical segments in Fig.3.2

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 $[h_l,h_r]$ and whose ordinate is h_o $(1<h_o\leqslant N)$. Then the query for h can be executed as follows.

QUERY:

(i) find $v\in V$ such that $v=\min\{v'|\ v'\in V\cup\{n+1\}, h_l\leqslant v'\}$ by binary search;
if $v=n+1$, then stop; otherwise, let w be the root of $T(1,N)$;

(ii) $s_i:=find(v)$ with respect to the ordered partition $P(U(w),S(w))$;
report all the vertical segments $s\in S(w)$ such that $s_i\leqslant s\leqslant h_r$ by traversing the list representing $S(w)$;

(iii) if w is a leaf, then stop; otherwise, let w_σ be a son of w with $h_0 \in I(w_\sigma)$;
v:=*find*(v) with respect to the ordered partition $P(U(w), U(w_\sigma))$;
return to (ii);

In the algorithm QUERY, each *find* can be executed in a constant time, hence Vaishnavi and Wood's theorem [23] is obtained.

Theorem 3.1. [23] 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

In this problem, the set of vertical segments is updated by insertions. 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 $v^*=\min\{v' \mid v' \in V \cup \{n+1\}, v \leqslant v'\}$ by binary search;
let w be the root of $T(1,N)$;

(ii) with respect to the ordered partition $P(U(w),S(w))$,
add(v,v^*); if w is an s-node of v, then *split*(v);
if w is a leaf, then return; otherwise, let w_λ and w_ρ be the sons of w;
with respect to the ordered partition $P(U(w),U(w_\lambda))$,
v_λ:=*find*(v^*); *add*(v,v^*); if w_λ is an s- or t-node of v, then *split*(v);
with respect to the ordered partition $P(U(w),U(w_\rho))$,
v_ρ:=*find*(v^*); *add*(v,v^*); if w_ρ is an s- or t-node of v, then *split*(v);

(iii) if w is a t-node of v, then
if $L(v) \cap I(w_\lambda) \neq \emptyset$ then $v^*:=v_\lambda$; $w:=w_\lambda$ and return to (ii);
if $L(v) \cap I(w_\rho) \neq \emptyset$ then $v^*:=v_\rho$; $w:=w_\rho$ and return to (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 3.2. 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 $m_0(w)$ times, an the *add* operation is executed $l_0(w)$ times. Also, suppose that, for the ordered partition $P(U(w_\pi),U(w))$ of each non-root node w of $T(1,N)$, where w_π is the parent of w,

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 *micro*(v) and *number*(v) in the sequence in a constant time. We then see that, from Theorem 2.4, these sequences can be executed in $O(n+\sum_w\sum_i(m_i(w)+l_i(w)))$ time.

Thus, the whole sequence can be executed with the time complexity

$$O((q\log n+K)+(r\log n)+n+\sum_w\sum_i(m_i(w)+l_i(w))),$$

which is $O(n\log n+K)$. □

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 $v=\min\{v'|\ v'=n+1$ or $[v'\in V, i\leqslant v', j\in L(v')]\}$. 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 query. In this case, we do not need traversing the list of $S(w)$, so that the following theorem is obtained.

Theorem 3.3. An intermixed sequence of $O(n)$ *right* operations and insertions can be executed in $O(n\log n)$ 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 more be reported in queries. Specifically, on each s-node w of v, we remove v out of $S(w)$. This makes the ordered partition $P(U(w),S(w))$ update. 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 the ordered partition $P(U(w),S(w))$;

(iii) if w is a t-node of v, then
let w_λ and w_ρ be the sons of w;
if $L(v)\cap I(w_\lambda)\neq\varnothing$ then $w:=w_\lambda$ and return to (ii);
if $L(v)\cap I(w_\rho)\neq\varnothing$ then $w:=w_\rho$ and return to (ii);

The following theorem on the complexity can be obtained in a way similar to Theorem 3.2 (we use Theorem 2.1 in place of Theorem 2.4), so that we omit the proof.

Theorem 3.4. 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. □

4. Applications

(1) *Finding the connected and biconnected components of an intersection graph of n orthogonal segments* [9].

An *intersection graph* of segments in the plane is obtained by representing each segment by a vertex and connecting two vertices by an edge iff their corresponding segments intersect. In [9], it is shown that, if a sequence of $O(n)$ deletions and *right* operations can be executed in $T_D(n)$ time and $S_D(n)$ space, the connected and biconnected components of the intersection graph can be found in $O(T_D(n))$ time and $O(S_D(n))$ space. Hence, from Theorem 3.3, we see that the connected and biconnected components can be found in $O(n \log n)$ time and $O(n \log n)$ space. Note that, only for finding the connected components, $O(n \log n)$-time and $O(n)$-space optimal algorithms, that use the plane-sweep technique, are known [4], [8].

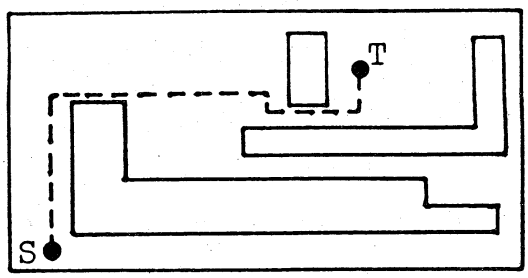

Fig.4.1. Rectilinear region and a shortest path between two points S and T in respect to L_1 metric

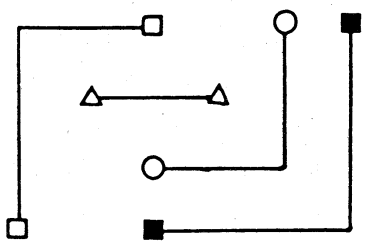

Fig.4.2. Manhattan wiring

(2) *Finding a shortest path, in respect to L_1 metric, between two given points in a rectilinear region with n edges* [12] (see Fig.4.1).

A *rectilinear region* is a polygonal region whose edges are orthogonal segments, and may contain "holes". In [12], Kojima, Sato and Ohtsuki showed that the problem can be solved in $O(n \log n+T_D(n))$ time and $O(S_D(n))$ space. Hence, it is seen from Theorem 3.3 that the problem (2) can be solved in $O(n \log n)$ time and $O(n \log n)$ space.

(3) *Discerning the existence of a Manhattan wiring of n nets on a single layer* (Fig.4.2).

For n pairs of points, called *nets*, on a grid, a wiring connecting all the pairs of points by wires along grid lines is called a *Manhattan wiring* if the wires do not intersect one another and no wire has more than one bend. This problem was first considered by Raghavan, Cohoon and Sahni [20], who gave $O(n^3)$-time algorithm. Masuda, Kimura, Kashiwabara and Fujisawa [17] gave an $O(n^2)$-time algorithm by considering an intersection graph of possible wires and employing depth-first search technique on that graph. Imai and Asano [9] gave an efficient implementation of their algorithm. Using the implementation with the data structure considered in this paper, we obtain an $O(n \log n)$-time algorithm for this problem (3).

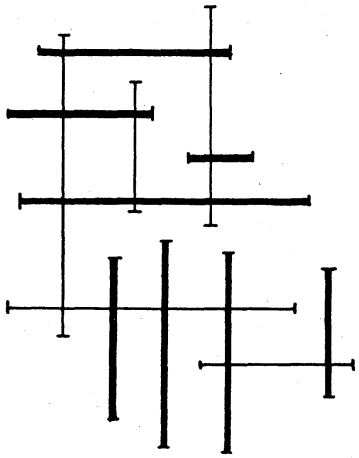

Fig.4.3. Maximum number of non-intersecting segments

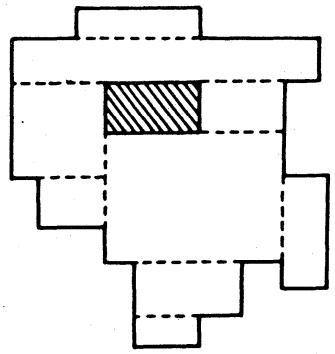

Fig.4.4. Minimum number of partition into disjoint rectangles

(4) *Finding, among n orthogonal segments such that no two of horizontal ones and no two of vertical ones intersect, a maximum number of non-intersecting segments* (Fig.4.3).

This problem is equivalent to finding a maximum independent set of an intersection graph of those segments, and, in this case, the intersection graph is bipartite, so that a maximum independent set can be found quickly. Combining the techniques developed in [9] with the data structure for the deletion problem, we can show that this problem can be solved in $O(n^{3/2}\log n)$ time and $O(n\log n)$ space.

(5) *Partitioning a given rectilinear region with n edges into a minimum number of disjoint rectangles* (Fig.4.4).

Algorithms for this problem were given in [15] and [19]. The main step of those algorithms is the problem (4), and so we can solve this problem in $O(n^{3/2}\log n)$ time and $O(n\log n)$ space.

The above problems are applications of the algorithm for the deletion problem. Here, it should be noted that, concerning the problems (1~5), a simpler data structure which gives the same results as above is given in Imai and Asano [9] by using proper structures of those problems. Also, note that a data structure similar to that in [9] is independently developed in Lipski [14].

We below present two applications of the algorithm for the insertion problem.

(6) *Finding the closure of a set of n rectangles with orthogonal sides* [16].

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 [16], Lipski and Papadimitriou considered the problem of finding the closure of a set of n given rectangles with orthogonal sides (Fig.4.5).

The algorithm given in [16] 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 rectangles, an $O(n\log n)$-time and $O(n)$-space optimal algorithm, which is different from that in [16] and uses the plane-sweep technique, is known [21].

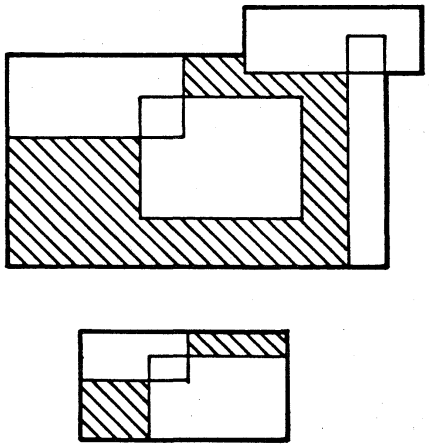

Fig.4.5. The closure of a set of rectangles

(7) *Partitioning a rectilinear region into disjoint rectangles by adding O(n) chords one by one in a given order.*

This problem can be solved in $O(n \log n)$ time and $O(n \log n)$ space. This problem can be used in several other problems. Here, we take up the problem of partitioning 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 [11], and several heuristics have been considered. For example, consider the following simple heuristic (see Fig.4.6).

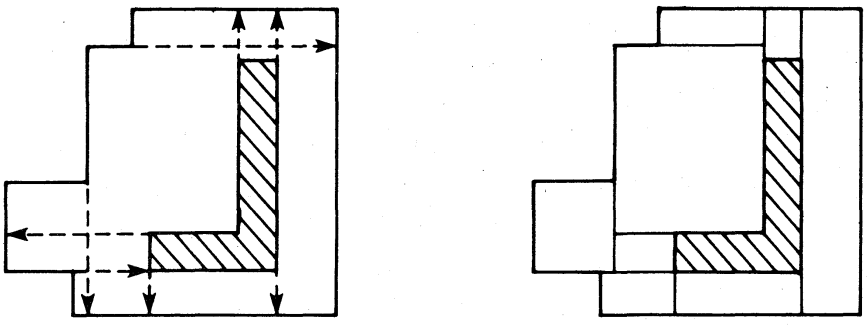

Fig.4.6. Minimum perimeter partition into disjoint rectangles
(a) Concave points P with directed segments l_p
(b) Approximate solution

(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 main step is just the problem (7), and so this heuristic can be executed in $O(n \log n)$ time and $O(n \log n)$ space.

Acknowledgment

The authors thank Professor Y. Kambayashi of Kyoto University for informing them of the paper [21].

References

[1] J. L. Bentley: Solutions to Klee's Rectangle Problems, unpublished note, Department of Computer Science, Carnegie-Mellon University, 1977.

[2] B. Chazelle: Filtering Search: A New Approach to Query-Answering. *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, Tucson, 1983, pp.122-132.

[3] H. Edelsbrunner: Dynamic Data Structures for Orthogonal Intersection Queries. *Report 59*, Institut für Informationsverarbeitung, Technische Universität Graz, 1980.

[4] H. Edelsbrunner, J. van Leeuwen, T. Ottmann and D. Wood: Connected Components of Orthogonal Geometric Objects. *Report 72*, Institut für Informationsverarbeitung, Technische Universität Graz, 1981.

[5] H. N. Gabow and R. E. Tarjan: A Linear-Time Algorithm for a Special Case of Disjoint Set Union, July 1982, submitted to *Journal of Computer and System Sciences* (also cf. Proceedings of the 15th Annual ACM Symposium on Theory of Computing, Boston, 1983, pp.246-251).

[6] J. E. Hopcroft and J. D. Ullman: Set Merging Algorithms. *SIAM Journal on Computing*, Vol.2 (1973), pp.294-303.

[7] H. Imai and T. Asano: An Efficient Algorithm for Finding a Maximum Matching of an Intersection Graph of Horizontal and Vertical Line Segments. *Papers of the Technical Group on Circuits and Systems*, CAS 83-143, Institute of Electronics and Communication Engineers of Japan, 1983.

[8] H. Imai and T. Asano: Finding the Connected Components and a Maximum Clique of an Intersection Graph of Rectangles in the Plane. *Journal of Algorithms*, Vol.4 (1983), pp.310-323.

[9] H. Imai and T. Asano: Efficient Algorithms for Geometric Graph Search Problems. *Research Memorandum RMI 83-05*, Department of Mathematical Engineering and Instrumentation Physics, University of Tokyo, 1983.

[10] H. Imai and T. Asano: Dynamic Orthogonal Segment Intersection Search. *Research Memorandum RMI 84-02*, Department of Mathematical Engineering and Instrumentation Physics, University of Tokyo, 1984.

[11] D. S. Johnson: The NP-Completeness Column: An Ongoing Guide. *Journal of Algorithms*, Vol.3 (1982), pp.182-195.

[12] I. Kojima, M. Sato and T. Ohtsuki: A Shortest Path Algorithm on a Planar Rectilinear Region (in Japanese). *Papers of the Technical Group on Circuits and*

Systems, CAS 83-205, Institute of Electronics and Communication Engineers of Japan, 1984.

[13] W. Lipski, Jr.: Finding a Manhattan Path and Related Problems. *Networks*, Vol.13 (1983), pp.399-409.

[14] W. Lipski, Jr.: An $O(n \log n)$ Manhattan Path Algorithm. *Research Report 141*, Laboratoire de Recherche en Informatique, Universite de Paris-Sud, 1983.

[15] W. Lipski, Jr., E. Lodr, F. Luccio, C. Mugnai and L. Pagli: On Two Dimensional Data Organization II. *Fundamenta Informaticae*, Vol.2 (1979), pp.245-260.

[16] W. Lipski, Jr., and C. H. Papadimitriou: A Fast Algorithm for Testing for Safety and Detecting Deadlocks in Locked Transaction Systems. *Journal of Algorithms*, Vol.2 (1981), pp.211-226.

[17] S. Masuda, S. Kimura, T. Kashiwabara and T. Fujisawa: On Manhattan Wiring Problem (in Japanese). *Papers of the Technical Group on Circuits and Systems*, CAS 83-20, Institute of Electronics and Communication Engineers of Japan, 1983.

[18] E. M. McCreight: Priority Search Trees. *Technical Report CSL-81-5*, Xerox Palo Alto Research Centers, 1982.

[19] T. Ohtsuki, M. Sato, M. Tachibana and S. Torii: Minimum Partitioning of Rectilinear Regions (in Japanese). *Transactions of Information Processing Society of Japan*, Vol.24, No.5 (1983), pp.647-653.

[20] R. Raghavan, J. Cohoon and S. Sahni: Manhattan and Rectilinear Wiring. *Technical Report 81-5*, Computer Science Department, University of Minnesota, 1981.

[21] E. Soisalon-Soininen and D. Wood: An Optimal Algorithm for Testing for Safety and Detecting Deadlocks in Locked Transaction Systems. *Proceedings of the 1st ACM Conference on Principle of Database System*, 1982, pp.108-116.

[22] R. E. Tarjan: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[23] V. K. Vaishnavi and D. Wood: Rectilinear Line Segment Intersection, Layered Segment Trees, and Dynamization. *Journal of Algorithms*, Vol.3 (1982), pp.160-176.