International Journal of Computer and Information Sciences, Vol. 15, No. 1, 1984

An Optimal Time and Minimal Space Algorithm for Rectangle Intersection Problems[1]

D. T. Lee[2]

Received June 1983; revised January 1984

Given a set of n iso-oriented rectangles in the plane whose sides are parallel to the coordinate axes, we consider the rectangle intersection problem, i.e., finding all s intersecting pairs. The problem is well solved in the past and its solution relies heavily on unconventional data structures such as range trees, segment trees or rectangle trees. In this paper we demonstrate that classical divide-and-conquer technique and conventional data structures such as linked lists are sufficient to achieve a time bound of $O(n \log n) + s$, and a space bound of $\Theta(n)$, both of which are optimal.

KEY WORDS: Computational geometry; analysis of algorithms; rectangle intersection; divide-and-conquer; disjoint set union-find.

1. INTRODUCTION

The problem of determining whether any two of n intersect, when given plane figures, was studied and solved in time $\Theta(n \log n)$ by Shamos and Hoey.[1] Since then many results have been obtained on finding all intersecting pairs among n given objects. In particular, Bentley and Ottmann[2] presented an algorithm for finding all intersecting pairs of n line segments in the plane in time $O(n \log n + s \log n)$ and space $O(n + s)$, where s is the number of intersecting pairs reported. The space required was later improved to $O(n)$ by Brown[3] If the line segments are rectilinear, i.e., horizontal or

[1] Supported in part by the National Science Foundation under Grants MCS 8342682 and ECS 8340031.

[2] Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201.

0091-7036/84/0200-0023$03.50/0

vertical, Bentley and Ottmann's algorithm runs in optimal time $\Theta(n \log n + s)$. Bentley and Wood[4] made use of Bentley and Ottmann's result, and obtained an algorithm for finding all intersecting pairs of n iso-oriented rectangles in optimal time $\Theta(n \log n + s)$ but in space $\Theta(n \log n)$ using segment tree structures.[5] (Iso-oriented rectangles mean those whose sides are parallel to the coordinate axes.) However, Bentley and Wood's algorithm may find some intersecting pairs more than once and cannot solve in optimal time the rectangle containment problem, i.e., finding all pairs of rectangles such that one is contained in the other. Six and Wood[6] presented an algorithm for reporting the intersecting pairs of rectangles exactly once in $\Theta(n \log n + s)$ time and $\Theta(n \log n)$ space using segments trees. As for the rectangle containment problem, which is of interest as well, Vaishnavi and Wood[7] presented an algorithm that runs in $0(n \log^2 n + t)$ time and $\Theta(n \log^2 n)$ space, where t is the number of containment pairs. Lee and Wong,[8] independently of Edelsbrunner,[9] studied the rectangle intersection problem in k-dimensional space and provided a unified approach for solving both rectangle intersection and containment problems by transforming them into appropriate $2k$-dimensional range searching problems.[10,11] Lee and Preparata[12] later provided a divide-and-conquer algorithm to solve the rectangle containment problem in $0(n \log^2 n + t)$ time and $\Theta(n)$ space. Edelsbrunner[13] developed a very elegant data structure, called rectangle trees, with which he provided a $\Theta(n \log n + s)$ time and $\Theta(n)$ space algorithm for reporting s intersecting pairs of rectangles. McCreight,[14] independently of Edelsbrunner also obtained an optimal time and space algorithm for the same problem using so-called tile tree structures. The rectangle intersection problem is therefore considered "solved" since Edelsbrunner's and McCreight's algorithms are both time and space optimal. Nevertheless, we shall demonstrate that the classical divide-and-conquer technique and conventional data structures such as linked lists are sufficient to solve the intersection problem in optimal time and minimal space. Besides, the subproblem that results from the "merge" step of the standard divide-and-conquer method, as we shall see, is interesting in its own right.

2. MAIN RESULT

Let $S = \{r_1, r_2, ..., r_n\}$ be a set of n iso-oriented rectangles in the plane, where $r_i = (x_1^i, x_2^i) \times (y_1^i, y_2^i)$ with $x_1^i < x_2^i$ and $y_1^i < y_2^i$. As an initial step, we sort the $2n$ left and right boundary values and $2n$ bottom and top boundary values of these n rectangles respectively. We then divide the set S of rectangles into two sets S_1 and S_2 such that $|S_1| = |S_2| = n/2$, (without loss of generality we assume that n is a power of 2) and for all $r_i \in S_1$ and $r_j \in S_2$, we have $x_2^i < x_2^j$, namely, we divide S into two sets S_1 and S_2 of

equal size according to the right boundary values of the rectangles in S. Recursively apply the algorithm and solve the rectangle intersection problems for S_1 and S_2. Note that because of our division step, some rectangles in S_2 may extend to the left crossing the dividing line. Let us denote the set of rectangles in S_2 that cross the dividing line by S_2' and the rest of S_2 by S_2''. Since no rectangles in S_1 may intersect rectangles in S_2'', we need only consider the sets S_2' and S_1 and find the intersecting pairs of rectangles (r_i, r_j), where $r_i \in S_1$ and $r_j \in S_2'$. For example, in Fig. 1 $S_1 = \{r_1, r_2, r_3, r_4\}$, $S_2' = \{r_5, r_6, r_7, r_8\}$ and (r_1, r_5), (r_1, r_6), (r_3, r_5), (r_3, r_8), (r_4, r_5) are the intersecting pairs that we are looking for in the merge step. Note that rectangles r_i and r_j intersect if and only if the following intersection conditions hold, i.e., intervals (x_1^i, x_2^i) and (x_1^j, x_2^j) have nonempty intersection and intervals (y_1^i, y_2^i) and (y_1^j, y_2^j) have nonempty intersection. Note also that intervals (t, u) and (v, w) have nonempty intersection if and only if (i) $t < v < u$ or (ii) $v < t < w$, namely, left (or similarly right) endpoint value of one interval must lie inside another interval. For simplicity we assume here that no two points have the same x or y coordinates. The case where points may have the same coordinate values can be handled without too much difficulty.

Now consider rectangles $r_i \in S_1$ and $r_j \in S_2'$. Suppose that the x-coordinate of the right boundary of r_i, namely x_2^i, lies in the interval (x_1^j, x_2^j). Since $x_2^i < x_2^j$ is true because of the division, r_i and r_j intersect if and only if $(y_1^i, y_2^i) \cap (y_1^j, y_2^j) \neq \emptyset$. Since there are two possibilities, $y_1^j < y_1^i < y_2^j$ and $y_1^i < y_1^j < y_2^i$, we shall detect these in two passes. The first

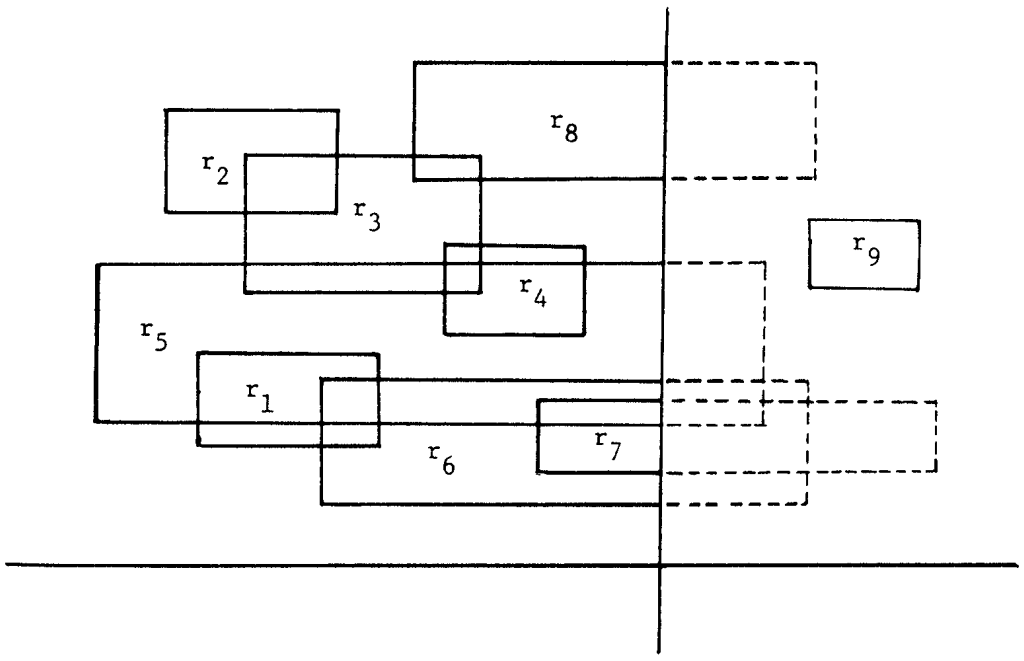

Fig. 1. Some rectangles of S_2 are cut by the dividing line.

pass identifies the possibility $y_1^j < y_1^i < y_2^j$ and the second pass identifies $y_1^i < y_1^j < y_2^i$. Specifically in the first pass we shall scan the set $S_1 \cup S_2'$ of rectangles from right to left starting from the dividing line and stopping at the right boundaries of the rectangles in S_1 and at the left boundaries of the rectangles in S_2'. When the right boundary of $r_i \in S_1$ is seen, we insert y_1^i into a linked list that maintains all the y_1's encountered so far in ascending order. When the left boundary of $r_j \in S_2'$ is seen, we perform a *retrieval* operation by retrieving from the linked list all those y_1's that lie in the interval (y_1^j, y_2^j). The sweeping operation ensures that when the left boundary of $r_j \in S_2'$ is seen, all the rectangles r_i whose y_1^i's are in the linked list satisfy $x_1^j < x_2^i < x_2^j$. The second pass is carried out in an analogous manner. We scan the rectangles in $S_1 \cup S_2'$ from left to right until the dividing line is encountered. During the scan we stop at the left boundaries of the rectangles in S_2' and at the right boundaries of the rectangles in S_1. When the left boundary of $r_j \in S_2'$ is seen, its y_1 value, y_1^j, is inserted into an ordered linked list and when the right boundary of $r_i \in S_1$ is seen, we retrieve from the list those y_1's that lie in the interval (y_1^i, y_2^i). Since these two passes are similar, we shall examine only the first pass in detail.

It is obvious that the first pass can be achieved in $O(m \log m + s')$ time, where $m = |S_1| + |S_2'|$ and s' is the number of intersecting pairs reported, using any balanced tree scheme[15] to implement the ordered linked list. Since the merge step of the divide-and-conquer algorithm takes $O(n \log n)$ time, the overall time of $O(n \log^2 n + s)$ and space of $\Theta(n)$ would result, if this straightforward approach was taken. However, we shall make full use of the fact that all the rectangles are given *a priori* and solve the problem in $\Theta(n \log n + s)$ time and $\Theta(n)$ space. The key issue is how we can maintain the ordered linked list and perform retrieval operations in $O(m)$ time instead of $O(m \log m)$. The problem essentially is the following. We are given a set of m numbers (the y_1's of the rectangles) whose ranks[3] are known via an initial sorting step, and we are to insert $k = |S_1|$ numbers into an ordered linked list, and to perform $q = |S_2'|$ retrievals by retrieving from the list the numbers that lie in ranges of the form (y_1^j, y_2^j). This problem is referred to as the *off-line insertion-retrieval problem* (OIRP for short). In the next section we give an algorithm for solving this problem without giving its timing analysis.

2.1. Off-line Insertion-Retrieval Problem

Consider two sets $A = \{a_1, a_2, ..., a_k\}$ and $B = \{b_1, b_2, ..., b_q\}$ of numbers, where $k + q = m$ and the ranks of a's and b's in $A \cup B$ are known. Suppose

[3] The rank of a number in a set is the position in which the number is if the numbers in the set are sorted.

that we have a list $C = (c_1, c_2, ..., c_m)$, where either $c_i \in A$ or $c_i \in B$. If $c_i \in A$, then c_i is to be inserted into an ordered list L at an appropriate position. If $c_i \in B$, then we need to perform a retrieval operation to retrieve those numbers in L that are greater than c_i.[4] For example, let $C = (4, 1, 5, 7, \overline{3}, 8, \overline{6}, 9, 2)$, where the numbers denote the ranks and those with overbars belong to B and the rest belong to A. When $\overline{3}$ is seen, the list contains 1, 4, 5 and 7 in that order; and 4, 5 and 7 are retrieved. When $\overline{6}$ is seen, the list contains 1, 4, 5, 7, 8; and 7 and 8 are retrieved. The key to the OIRP is that for each element c_i we need to find the element c_j in L that is the immediate predecessor of c_i if c_i is inserted into L. In other words, if PRED(c_j) and SUCC(c_j) denote, respectively, the predecessor and successor of c_j in L, then for each element c_i we need to find c_j such that $c_j < c_i < \text{SUCC}(c_j)$. Let LINK($c_i$) denote the element c_j. Thus, if we can compute LINK(c_i) for all c_i in C, then we can solve OIRP as follows. If $c_i \in A$, insert c_i into L immediately after LINK(c_i); otherwise ($c_i \in B$), retrieve elements in L from SUCC[LINK(c_i)] and onwards following SUCC pointers. To compute LINK(c_i) we first assume that $B = \varnothing$, i.e., no retrieval is to be made, and consider the case $B \neq \varnothing$ later. In the following LINK(c_i) = 0 means that c_i is to be inserted in front of the list L. Let $T = \{p_1, p_2, ..., p_m\}$ be a set of points, each corresponding to an element in C

[4] We have modified the problem a bit by leaving out the upper bounds when retrievals are performed. It is straightforward, however, to incorporate them later on.

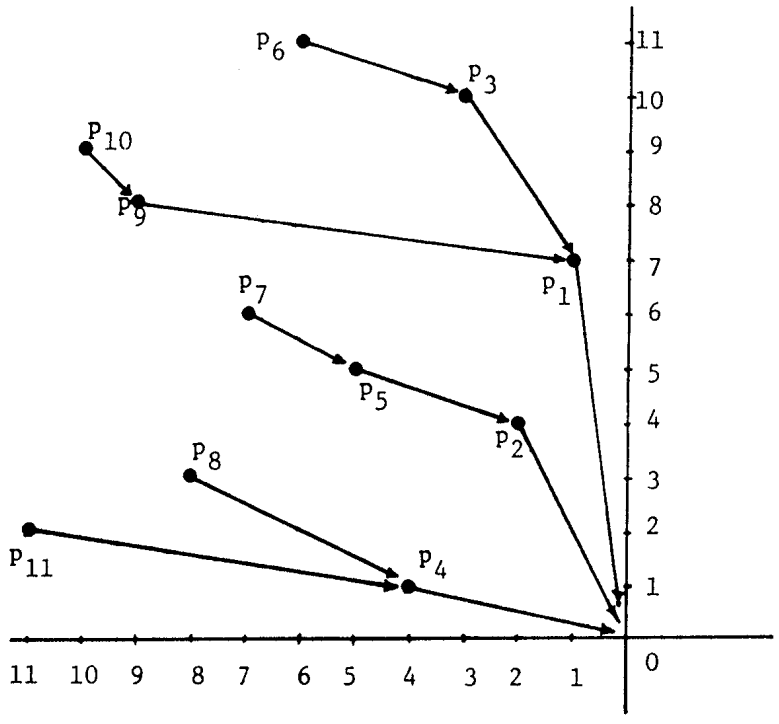

Fig. 2. Point representation of $C = (7, 4, 10, 1, 5, 11, 6, 3, 8, 9, 2)$

such that the x- and y-coordinates of p_i are the position i of c_i in C and its value respectively. Figure 2 is the point representation of $C = (7, 4, 10, 1, 5, 11, 6, 3, 8, 9, 2)$, and the arrowhead directing from p_i to p_j means that $\text{LINK}(c_i)$ is c_j. Since the ranks of elements in C are known, we may assume that there is a doubly-linked list P of the points in T arranged in ascending y-coordinate such that $\text{PREV}(p_i)$ and $\text{NEXT}(p_i)$ denote, respectively, the previous and next points of p_i in P. For example, in Fig. 2 $\text{PREV}(p_1) = p_7$ and $\text{NEXT}(p_1) = p_9$. We shall use p_i and c_i interchangeably in the sequel. To compute $\text{LINK}(c_i)$ for all c_i in C we shall scan the list C *backward*. For the last element c_m, $\text{LINK}(c_m)$ must be c_j for $p_j = \text{PREV}(p_m)$. The key observation is that when c_m is scanned while processing C (*forward*), all elements in C except c_m have been inserted into L. Therefore $\text{LINK}(c_m)$ is correctly computed. Thus, if we discard c_m from C and obtain $C' = (c_1, \ldots, c_{m-1})$ by deleting p_m from P, we have exactly the same configuration as before and the computation of LINK's continues. This method was used by Lee and Preparata[12] to obtain an improved algorithm for rectangle containment problem. The detailed algorithm is given in the following section.

2.2. Algorithm Link

Input: A list $C = (c_1, c_2, \ldots, c_m)$ and a doubly-linked list P for elements in C such that $\text{PREV}(c_i)$ and $\text{NEXT}(c_i)$ denote, respectively, the immediate predecessor c_j and immediate successor c_k of element c_i in the list P, i.e., $c_j < c_i < c_k$ and no other element lies in between.

Output: $\text{LINK}(c_j)$ for $j = 1, 2, \ldots, m$

Method:

```
For j = m downto 1 do
    begin LINK(c_j) = PREV(c_j)
        Delete c_j from P
    end
```

The arrowheads in Fig. 2 are the output of Algorithm LINK when it is applied to $C = (7, 4, 10, 1, 5, 11, 6, 3, 8, 9, 2)$. It is easy to see that Algorithm LINK runs in $O(m)$ time.

Now let us address ourselves to the case where $B \neq \varnothing$. Recall that the elements in B are not to be inserted into L and hence Algorithm LINK cannot be applied directly. We shall first scan the list P by "marking" off those elements that belong to B. This marking-off operation results in a doubly-linked list P' consisting only of the elements in A and in the meantime creates a "grouping" of elements in B as follows. Let $a_i, b_1, b_2, \ldots, b_j, a_k$ be elements of C that appear in P in that order, i.e.,

$a_i < b_1 < \cdots < b_j < a_k$ and $a_i, a_k \in A$ and $b_1,..., b_j \in B$. The set $\{b_1, b_2,..., b_j\}$ is said to belong to *group*(a_i) and PREV(b_1),..., PREV(b_j) are set to a_i initially. We then apply Algorithm LINK to the elements in P', i.e., to compute LINK(c_i), for all $c_i \in A$. To compute the LINK's for elements in B we shall work with the list P' of initially "unmarked" elements and the elements in B at the same time by scanning the list C *backward*. When an element $c_i \in A$ is encountered, it is "marked" in P', meaning that c_i is not present from this point on or c_i is not in the list until now if C is processed. When an element $c_i \in B$ is seen, we follow the pointers PREV to get to the first "unmarked" element c_j in P' and set LINK(c_i) to c_j; in the meantime we perform a path compression[17] by letting all pointers PREV(c_k) point to c_j for those elements c_k that are visited during the traversal. (Note that those elements c_k must all be marked.) A detailed description of this algorithm is given by the following.

2.3. Algorithm Link*

Input: A list $C = (c_1, c_2,..., c_m)$, where c_i is either in A or in B, and a doubly-linked list P of elements in C such that PREV(c_i) and NEXT(c_i) denote the immediate predecessor and immediate successor of c_i, respectively.

Output: LINK(c_j) for $j = 1, 2,..., m$.

Method:

1. Visit the elements in P in order and create a new list P' of elements of A. Set PREV$(c_j) = c_i$ for all $c_j \in B$, $c_i \in A$ such that c_j belongs to group(c_i).
2. Invoke Algorithm LINK to compute LINK's for elements in A by providing the list C', which is C with elements in B removed, and P'.
3. For $j = m$ downto 1 do
   if $c_j \in A$ then "mark" c_j in P'
   else (Comment: $c_j \in B$)
   begin $c =$ PREV(c_j)
   while c is marked do begin Append c to a queue Q
   $c =$ PREV(c)
   end
   LINK$(c_j) = c$
   (Comment: Perform path compression.)
   For all d in Q do PREV$(d) = c$
   end

The correctness of the algorithm LINK* can be verified easily. Since step 3 is a special case of disjoint set union-find problem and can be shown

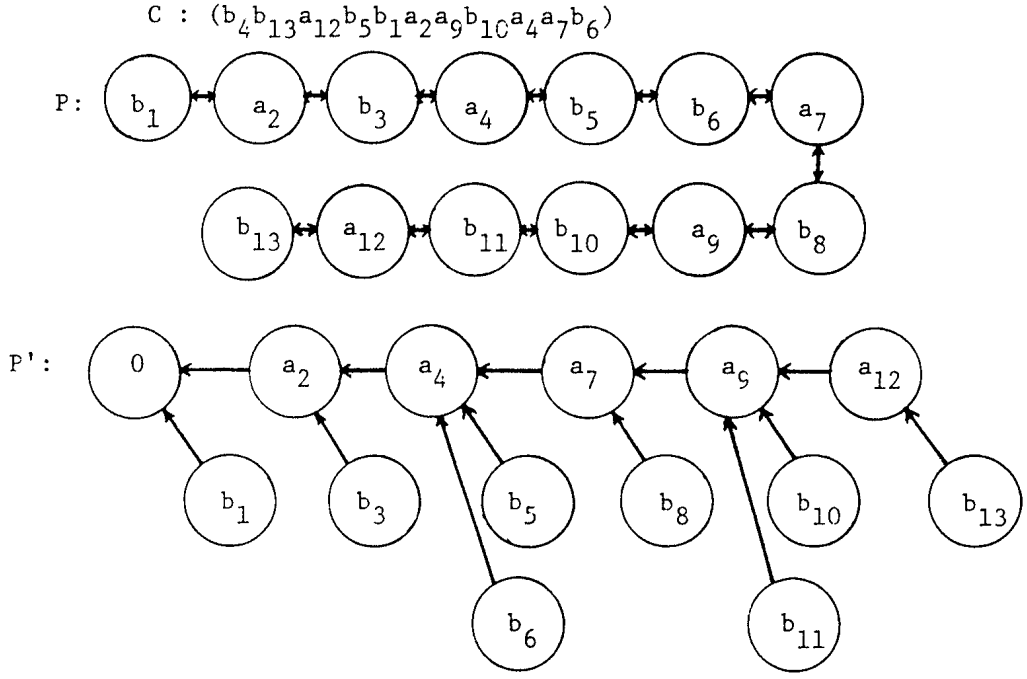

Fig. 3. Linked list P and P' after the grouping.

to take $O(m)$ time (see Ref. 16 for a proof), the total time involved is $O(m)$. (We remark here that in the context of Ref. 16, P' is the static set union tree and whenever an element of P' is "marked," a *union* operation is performed.) Figure 3 illustrates the diagram of a list P, where the a's are in A and b's are in B. After step 1 we obtain a new list P' of elements in A to which various numbers of b's are attached. Figure 4 illustrates the path compressions performed at various stages of step 3. Now the entire algorithm for OIRP is presented below. It is obvious that the running time of algorithm OIRP is $O(m+s)$, where s is the output size.

2.4. Algorithm OIRP

Input: Same as in Algorithm LINK*

Output: The set $C_i = \{c_j | j < i \text{ and } c_j > c_i\}$ for all $c_i \in B$.

Method:
1. Call LINK* (Comment: Set up LINK pointers for all c in C.)
2. For $i = 1$ to m do
   if $c_i \in A$ then insert c_i into L after element LINK(c_i)
   else output $C_i = \{c_j | c_j$ follows LINK(c_i) in $L\}$.

As an example, let us refer to Fig. 1 again, where $S_1 = \{r_1, r_2, r_3, r_4\}$ and $S_2' = \{r_5, r_6, r_7, r_8\}$. The list C and P are $C = (y_1^4, y_1^7, y_1^3, y_1^1, y_1^8, y_1^2, y_1^6, y_1^5)$ and $P = (y_1^6, y_1^7, y_1^1, y_1^5, y_1^4, y_1^3, y_1^2, y_1^8)$. When y_1^4 of C is scanned, since it

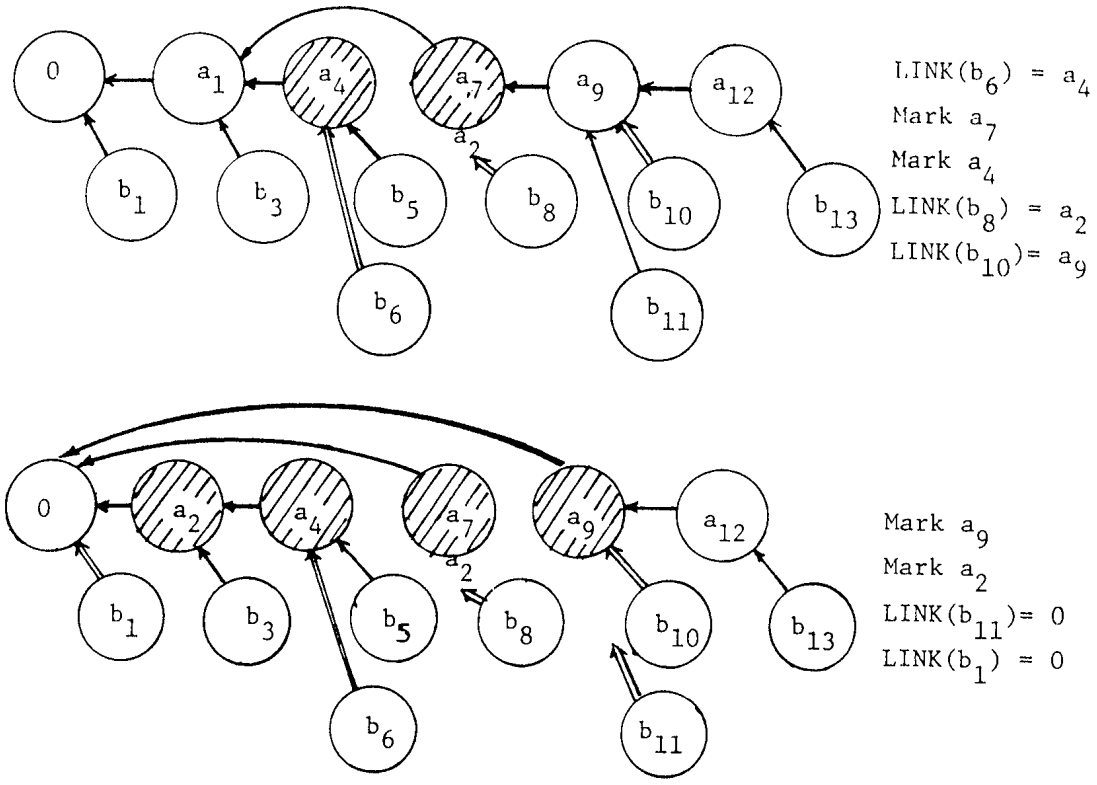

Fig. 4. Diagram of P' after elements of C have been processed backward.

is the right boundary of r_4 in S_1, we insert it into the list L. When y_1^7 is seen, we perform a retrieval operation by retrieving the y_1's in L that are greater than y_1^7 (and less than y_2^7) while following the pointer LINK(y_1^7) and the pointer SUCC in L. The next two values y_1^3 and y_1^1 are inserted into L. When y_1^8 is seen, we perform a retrieval operation as before. Thus, when the list C is exhausted, we will have reported the intersecting pairs (r_6, r_1), (r_5, r_4), and (r_5, r_3) in that order. The remaining intersecting pairs (r_1, r_5) and (r_3, r_8) are reported in the second pass by scanning the list C backward and interchanging the roles of sets S_1 and S_2', i.e., when elements in S_1 are seen, retrieval operations are performed and when elements in S_2' are seen, insertions are executed.

Now if $T(n)$ denotes the time for the rectangle intersection problem, we have the following recurrence relation.

$$T(1) = O(1)$$

$$T(n) \leqslant 2T(n/2) + M(n/2, n/2) + O(n)$$

where $M(n/2, n/2)$ denotes the time for the "merge" step—the OIRP problem, and $O(n)$ is for the "divide" step and other operations necessary to start the merge. Since $M(n/2, n/2)$ is $O(n)$, $T(n)$ is $O(n \log n)$, excluding the

time for the output. The space requirement of the algorithm is obviously $\Theta(n)$.

Theorem. The rectangle intersection problem of finding all s intersecting pairs among n iso-oriented rectangles in the plane can be solved in $O(n \log n) + s$ time and $\Theta(n)$ space, which is both time and space optimal.

Proof. The time and space complexities follow from the previous discussion. Optimality of its time complexity follows from the result[1] that determining if any two rectangles intersect requires $\Omega(n \log n)$ comparisons under the linear decision tree model.[17]

REFERENCES

1. M. I. Shamos, and D. Hoey, Geometric intersection problems, *Proc. 17th IEEE Symp. Foundations of Computer Science*, pp. 208–215 (October 1976).
2. J. L. Bentley and T. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.*, **C28**: 643–647 (September 1979).
3. K. Q. Brown, Comments on 'Algorithms for reporting and counting geometric intersections', *IEEE Trans. Comput.*, **C30**: 147–148 (February 1981).
4. J. L. Bentley and D. Wood, An optimal worst-case algorithm for reporting intersections of rectangles, *IEEE Trans. Comput.*, **C29**: 571–576 (July 1980).
5. J. L. Bentley, Algorithms for Klee's rectangle problem, (unpublished manuscript) Carnegie-Mellon University, 1977.
6. H. W. Six and D. Wood, The rectangle intersection problem revisited, *BIT*, **20**: 426–433, 1980.
7. V. Vaishnavi and D. Wood, Data structures for the rectangle containment and enclosure problems, *Computer Graphics and Image Processing*, 13: 372–384, 1980.
8. D. T. Lee and C. K. Wong, Finding intersection of rectangles by range search, *J. Algorithms*, **2**: 337–347, 1981.
9. H. Edelsbrunner, Dynamic rectangle intersection searching, Inst. for Informationsverarbeitung, TU Graz, Bericht 47 (February 1980).
10. J. L. Bentley, Decomposable searching problems, *Info. Proc. Lett.*, **8**: 244–251, 1979.
11. G. S. Lueker and D. E. Willard, A data structure for dynamic range queries, *Info. Proc. Lett*, 15(5): 209–213 (December 1982).
12. D. T. Lee and F. P. Preparata, An improved algorithm for the rectangle enclosure problem, *J. Algorithms*, **3**(3): 218–224, 1982.
13. H. Edelsbrunner, A time- and space-optimal solution for the planar all intersecting rectangles problem, Inst. for Informationsverarbeitung, TU Graz, Bericht 50, (April 1980).
14. E. M. McGreight, Efficient algorithms for enumerating intersecting intervals and rectangles, Res. Rep. CSL-80-9, Xerox PARC, Palo Alto, California, (June 1980).
15. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1973.
16. H. N. Gabow and R. E. Tarjan, A linear time algorithm for a special case of disjoint set union problem, *Proc. 15th ACM Symp. Theory of Computing*, pp. 246–251, (April 1983).
17. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Designand Analysis of Efficient Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.