An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles

JON LOUIS BENTLEY, MEMBER, IEEE, AND DERICK WOOD

***Abstract*—In this paper we investigate the problem of reporting all intersecting pairs in a set of n rectilinearly oriented rectangles in the plane. This problem arises in applications such as design rule checking of very large-scale integrated (VLSI) circuits and architectural databases. We describe an algorithm that solves this problem in worst case time proportional to $n \lg n + k$, where k is the number of interesecting pairs found. This algorithm is optimal to within a constant factor. As an intermediate step of this algorithm, we solve a problem related to the range searching problem that arises in database applications. Although the algorithms that we describe are primarily theoretical devices (being very difficult to code), they suggest other algorithms that are quite practical.**

***Index Terms*—Computational geometry, geometric intersection problems, optimal algorithms, range searching, VLSI design rule checking.**

I. Introduction

Computational geometry is the study of the computational complexity of finite geometric problems. The members of one interesting set of problems in computational geometry have come to be known as the *geometric intersection problems.* These problems deal with the computation of intersection properties among sets of planar objects such as line segments, circles, and half-spaces. Shamos and Hoey [13] and later Bentley and Ottmann [5] have described a number of algorithms for computing the union or intersection of sets of such objects and for counting and reporting all intersecting pairs in sets of such objects. In this paper we will extend that work by investigating intersection problems defined on sets of rectangles.

The primary problem that we will investigate is that of *rectangle intersection;* we are given a set of rectangles (with sides parallel to the coordinate axes) in the plane, and asked to *report*[1] all pairs of rectangles that intersect each other. As we will see later, this process is a crucial step in checking the design rules for very large-scale integrated (VLSI) circuitry and currently consumes large quantities of computer time in real applications. We will investigate an algorithm that solves this problem in optimal worst case time. A crucial step in our algorithm is a solution of the two-dimensional *batched range searching* problem. In geometric terms we are given a set of rectangles and a set of points and must report, for each point, all of the rectangles in which it lies. This problem also arises in certain database applications. We will describe an optimal worst case algorithm for solving this problem. Although both of the algorithms that we will describe are very complex to code and must therefore be considered primarily of theoretical interest, they do suggest practical algorithms.

We will investigate the above problems in a top-down fashion. In Section II we will examine the rectangle intersection problem and see how it can be reduced to the batched range searching problem. In Section III we will then investigate how the batched range searching problem can be solved by use of a "segment tree" data structure, which we will examine in Section IV. After having derived all the necessary components in these sections, we will see in Section V how they can be put together to form a single algorithm. Directions for further work and conclusions are then offered in Sections VI and VII.

II. The Rectangle Intersection Problem

In this section we will investigate the following problem.

The Rectangle Intersection Problem: Given n rectilinearly oriented rectangles in the plane, report all pairwise intersections.

Two rectangles are said to intersect either if their edges intersect or if one entirely encloses the other. For example, in Fig. 1 there are the five rectangles $A,B,C,D,$ and E. On the one hand, A and C have an *edge intersection as* do (A,B), (A,E), and (B,E), while on the other hand D is *rectangle-enclosed* in B and this is the only rectangle enclosure. There are therefore a total of five intersecting pairs of rectangles in the set.

The rectangle intersection problem arises in many applications. Eastman and Lividini [6] discuss how this problem arises in maintaining architectural databases. A crucial application of the problem is in VLSI *design rule checking.* After a chip has been laid out, it must be verified to ensure that it meets all design rules. These rules are of the form "objects that are not to be connected in the circuit must be separated by at least x_1 units" or "objects that are to be connected in the circuit must overlap by at least x_2 units." (Both of these rules stem

Manuscript received June 10, 1979; revised January 20, 1980. This research was supported in part by the U.S. Office of Naval Research under Contract N00014-76-C-0370 and in part by Natural Sciences and Engineering Research Council of Canada under Grant A-7700.

J. L. Bentley is with the Departments of Computer Science and Mathematics, Carnegie-Mellon University, Pittsburgh, PA 15213.

D. Wood is with the Unit for Computer Science, McMaster University, Hamilton, Ont., Canada.

[1] Throughout this paper we will use the word "report" to mean an enumeration of a set in which each element is included exactly once, and with no relative ordering implied.

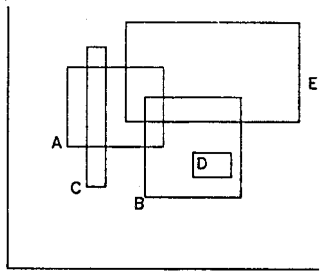

Fig. 1. A set of rectilinearly oriented rectangles.

from the fact that the potential for lossage of edge acuity in the fabrication process must be accounted for in the design.) Since it is very expensive to apply these design rule checks to all $\theta(n^2)$ pairs[2] of objects on an n-element chip, many systems use a preliminary test to discard most pairs from consideration. A common approach is based on "bounding boxes"—each object is represented by the smallest rectangle enclosing it, and then two objects are checked in detail if and only if their respective rectangles intersect. Note that this calls for solving precisely the rectangle intersection problem. Details of the design rule checking problem can be found in Baird [1], Lauther [8], or Mead and Conway [10]. To emphasize the practical importance of the problem, we note that Baird [1] discusses one system in which twenty-four CPU hours are required to check a chip of one hundred thousand elements.

Much research has been devoted to the rectangle intersection problem. Baird [1] surveys many pragmatic approaches to VLSI design rule checkings, and Lauther [8] describes an approach based on rectangle intersections among bounding boxes. Although no theoretical analysis of these algorithms is known, Baird [1] does report some timing results. At the empirical level, the authors know of no comparisons of the available systems. In a theoretical context, Bentley and Ottmann [5] have described an algorithm to solve a special case of the rectangle intersection problem that can be stated as follows: Given n rectilinearly oriented line segments, report all pairwise edge intersections. Their algorithm, which is optimal, requires $\theta(n \lg n + k)$ time, where k is the number of pairwise intersections reported.[3] Since a rectangle is composed of four line segments, their algorithm can be used to find all k pairwise edge intersections for n rectangles in $\theta(n \lg n + k)$ time. Their algorithm does not detect rectangle enclosures, however, so the thrust of this paper is to extend their results to solve the rectangle intersection problem.

Our algorithm for solving the rectangle intersection problem will consist of two stages. The first stage will find all pairs of rectangle with edge intersections using Bentley and Ottmann's [5] algorithm. Finding all enclosing pairs is the responsibility of the second stage (note that having found all edge intersections and all enclosing pairs, we have found all intersecting rectangles.) We will accomplish this by associating with each rectangle X a *representative point* x in its interior. There are now two important properties to note about rectangles and their representative points.

1) If rectangle A enclosed rectangle B, then point b (the representative of rectangle B) lies within rectangle A.

2) If point b lies within rectangle A, then rectangle B intersects rectangle A (either by being wholly enclosed within it or by having an edge intersection).

Our algorithm will exploit these two properties by solving the batched range searching problem of reporting, for each rectangle, all the points that lie inside it. Although the first property tells us that this will indeed report *all* rectangle enclosures, the second tells us that it may also report some of the edge intersections, thus seemingly a great deal of extra work is done. The crucial observation, however, is that we don't care, since we need to report all edge intersections anyway!

The approach that we have taken, therefore, is to reduce the original problem of finding rectangle enclosure to that of finding the point enclosures for n points and n rectangles. This is a special case of the batched range searching problem, which we discuss in the following section.

III. The Batched Range Searching Problem

In this section we will study the following problem.

The Batched Range Searching Problem: Given n points and m rectangles in the plane, report for each rectangle all of the points that lie inside it.

Fig. 2 shows a set of 4 points and 4 rectangles; our algorithm should report for this set that rectangle A contains points a and b, etc. We will investigate this problem by first examining solutions based on repeated searching, and then study an approach based on "scanning" (which is the method used in our final algorithm). As we have seen, a solution to this problem (which arises in certain database applications; see Bentley and Maurer [4]) will enable us to solve the rectangle intersection problem.

One method of solving the batched range searching problem is to cast it as a *searching problem:* We organize the set of points into a data structure and then for each rectangle we ask for a list of all the points contained in it. Queries of this form are called *range queries,* because the rectangle is defined by a range of values in both the x and the y dimensions. Many algorithms for range searching have been investigated recently; a summary of the results can be found in Bentley and Maurer [4]. The algorithms described in that paper can be used to give algorithms for batched range searching that have running time of $\theta(n \lg n + k)$ (where k is the total number of points reported to lie within the rectangles) for the case that there are many more points than rectangles. More formally, this holds if m grows more slowly than $n^{1-\epsilon}$ for some positive constant ϵ (for

[2] We say that $f(n) = \theta(g(n))$ if there exist positive constants $c,d,$ and m such that for all $n \geq m$, $cg(n) \leq f(n) \leq dg(n)$. Similarly, we write $f(n) = O(g(n))$ if there exist positive constants c and m such that for all $n \geq m, f(n) \leq dg(n)$.

[3] To facilitate comparison with algorithms described later in this paper, we will now briefly sketch their algorithm. The underlying idea is to scan through the set of line segments from bottom to top. When the bottom endpoint of a vertical segment is encountered, its x-value is inserted into a total order R, when the top endpoint is encountered, the x-value is deleted from R. If the total order is implemented as a balanced binary tree, then each of these steps can be performed in logarithmic time. Whenever a horizontal segment is encountered, all x-values in R between the extreme x-values of the segment are (correctly) reported as intersecting the segment. This requires time proportional to the logarithm of the number of vertical segments plus the number of intersections.

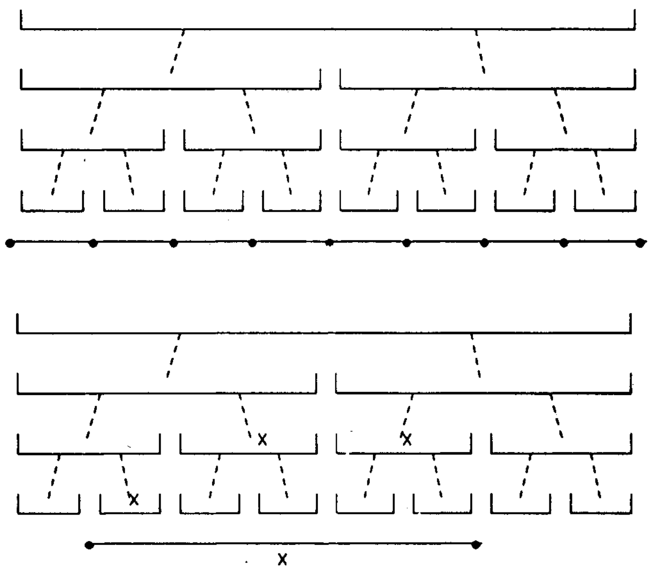

Fig. 2. The segment tree.

instance, if the number of rectangles is proportional to the square root of the number of points). An algorithm with time $\theta(m \lg m + k)$ can likewise be given for sets in which the number of points grows much more slowly than the number of rectangles. Applying the methods described by Bentley and Maurer to the case that $m = n$, however, leads to an algorithm with running time $\theta(n \lg^2 n + k)$. Since our main interest is the case $m = n$ and we would like a running time of $\theta(n \lg n + k)$, the repeated searching method must be discarded in favor of some other approach.

The new approach that we will take to solving the batched range searching problem is based on *scanning*. The description of the algorithm in this section will be rather intuitive; a more formal description will be presented in Section V. The primary step of our algorithm is to "sweep" a horizontal line through the points and rectangles. During this sweep we maintain a set S of line segments that are the projections of rectangles in the set onto the x axis. Initially S is empty. When we encounter a new rectangle R during the sweep we insert its corresponding segment into S; as we leave a rectangle we delete the corresponding segment from S. When we encounter a point p during the sweep we must report all of the rectangles that contain it. Notice that any containing rectangle must satisfy the following two properties: it is currently contained in S (otherwise its y range does not contain p's y value) and its projection onto the x axis must contain p's x value. Combining these observations, we can find all the rectangles that contain point p by merely reporting all segments in S that contain p's x value.

We can now describe our algorithm for batched range searching more precisely. The first step is to sort the rectangles and the points by y values (notice that each rectangle is represented twice—once for its bottom edge and once for its top). We then initialize S to the empty set, and scan through the sorted list of rectangles and points. As we enter a rectangle we insert the corresponding segment into S, as we leave a rectangle we delete the segment, and as we encounter a point we report all segments that overlap its x-projection as rectangles containing the point. The correctness of this algorithm can be established by formalizing the observations mentioned above.

We have now reduced the batched range searching problem to the problem of maintaining a set of line segments such that segments can be efficiently inserted and deleted, and a query asking for all segments that cover a given point can be answered quickely. (We say a segment covers a point if the point lies between the segment's endpoints.) In the next section we will investigate a data structure called the *segment tree* that permits all these operations to be performed in logarithmic time. We will now state precisely the properties required of the segment tree to ensure the efficiency of the batched range searching algorithm.

The Segment Tree: The segment tree data structure must be able to perform the following operations on m line segments within the described time bounds.

1) *Insert* a new line segment in time proportional to $\lg m$.
2) *Delete* a line segment in time proportional to $\lg m$.
3) *Report* all line segments that cover a given point p in time proportional to $\lg m + k_p$, where k_p is the number of line segments covering p.

Note that the $m + 1$ endpoints that define the line segments are known before execution of any of the above operations.

IV. The Segment Tree

In this section we will show how the *segment tree* data structure can be used to maintain a set of line segments in the plane. The segment tree is based on the idea of representing a set of intervals on the line by a perfectly balanced binary tree; a line segment is then represented by "covering" it with certain nodes of the tree. In this section we will first examine some abstract properties of line segments and binary trees, and then show how these properties can be used to implement our segment tree data structure.

The first property of segments that we will utilize is the fact that for the particular application of batched range searching, the values (locations) of the endpoints of all the segments are known before any processing at all. Without loss of generality, we can assume that these endpoints occupy the integer coordinates $0, 1, 2, \cdots, m$, where $m \leq 2n$ (that is, the number of endpoints cannot be greater than twice the number of rectangles). We can make this simplification to integers because the essential property of the endpoints is their relative ordering and not their absolute value; implementations, however, must be careful at this point. Having now made the simplification that the endpoints of the segments are integers, a segment can be defined by a pair of integers, (i,j), such that $1 \leq i < j \leq m$. We will also make the simplifying assumption that m is a power of 2; this is for pedagogic purposes only, and will not be detrimental to implementations.

We will now see how a set of (consecutive) intervals on the line can be represented by a binary "interval tree." Specifically, we wish to represent the m intervals $[0,1], [1,2], \cdots, [m - 1, m]$. We represent these segments by the tree depicted in Fig. 3(a) where each node on the bottom level represents a single interval of length one. On the next higher level each node represents an interval of length two (that starts at a power of 2). This continues, and on the jth level from the bottom there

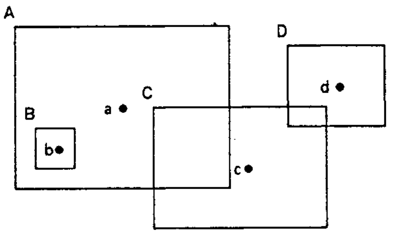

Fig. 3. A collection of points and rectangles.

are $2^{\lg m-j}$ intervals represented, each of length 2^j and beginning at a power of 2^j. It is now easy to represent a line segment X in such an interval tree: We "mark" with an X every node in the tree whose interval is contained in X, *on the condition that that node's father is not also contained in* X. We call this the *canonical covering* of X in the tree. A segment X and its canonical covering in an interval tree are shown in Fig. 3(b).

It is easy to describe a recursive algorithm to compute the canonical covering of a line segment X in an interval tree T in $\theta(\lg m)$ time. The algorithm is initially invoked at the root of the tree. As it visits any node it compares the interval represented by the node with the input segment X. If X wholly contains the node's interval, then the procedure marks the node accordingly and returns (notice that the sons do, in fact, remain correctly unmarked—their father is wholly contained in X). If the segment X does not wholly contain the node's interval, does intersect one of the node's sons, and does not intersect with the node's other son, then the node is not marked and only the intersecting son is recursively visited. The last possible case is that the node's interval is not wholly contained in segment X but X does intersect both of its sons; in this case both sons are visited recursively. To establish the $\theta(\lg m)$ running time of this algorithm it is sufficient to observe that the algorithm visits at most four nodes and marks at most two nodes on each of the $\lg m + 1$ levels of the tree when representing any segment.

With this background in interval trees and canonical coverings, it is easy to define a segment tree. The underlying structure of a segment tree is an interval tree, with the intervals at the leaves of the tree defined by the endpoints of the set of segments that we are to process (which we assume are the integers $0, 1, 2, \cdots, m$). We represent a segment X in this tree by marking with "X" the tree's nodes of the segment X's canonical covering. An illustration of a segment set and its corresponding segment tree is shown in Fig. 4(a). The binary tree structure is implemented in the typical manner (see Knuth [14]). The set of segments currently marking a given node in the tree is implemented by a double linked list. The concrete implementation of the abstract tree of Fig. 4(a) is depicted in Fig. 4(b).

We must now define how the three operations of *inserting* a segment, *querying* (asking for all segments that cover a given point) and *deleting* a segment are to be implemented. The operation of inserting a segment X is easy: we just use the algorithm that visits the canonical covering of X to find each of the nodes at which X must be represented. At each of these nodes we add X to the front of the doubly linked list representing the segments that currently contain the interval corresponding to the node. The time complexity of this operation is $\theta(\lg m)$.

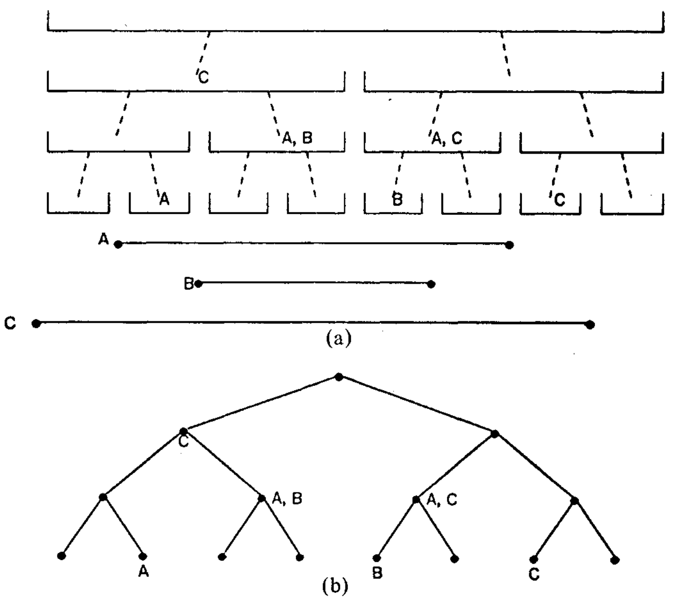

Fig. 4. (a) Abstract and (b) concrete segment trees.

To answer a query asking for all the segments that contain a given point p we "go down" the binary tree, finding the unit segment in which p lies. As the search visits each node, we know that every segment that contains a visited node must contain p. Thus as we visit each node we traverse the linked list and report each segment in the list as containing p. Since any segment containing p must contain one of the intervals on the path from the root to p, this algorithm correctly performs the query. The time required by this search will be logarithmic in m (for going down the tree), and linear in the number of segments found to intersect the point p, which we will call k_p. Thus the total cost of performing a query is $\theta(\lg m + k_p)$.

The problem of deleting a line segment from a segment tree is more subtle than either insertion or querying. The algorithm for finding the canonical covering would allow us to find all the nodes in which the segment is currently marked; the problem then becomes that of locating the particular node in the linked list to be deleted—a sequential scan of the list is much too slow. To accomplish deletion quickly we will instead keep an auxiliary table[4] that contains an entry for each segment currently stored in the segment tree. The auxiliary table contains at most m elements, so the entry for segment X can be obtained in $O(\lg m)$ time. The entry for segment X is a linked list containing points to each of the at at most $2 \lg m - 2$ markers in node lists representing segment X. These linked lists can be easily built as each segment X is inserted (into the segment tree). The pointer will tell us the location of the node to be deleted in each chain, and since the chains in each tree node are doubly-linked we can accomplish deletion in constant time at each node. The total cost of deleting an entire segment is therefore $\theta(\lg m)$.

[4] If this table is implemented as a balanced binary tree (see Knuth [14]), then each access will require time logarithmic in the size of the table. If each rectangle is identified by a unique integer in the range $1 \cdots n$, then implementing the table as an array will allow constant access time.

To summarize the segment tree that we have studied in this section, we will briefly review the costs of manipulating it when it represents an m-element set. The empty tree of m elements can be built in $\theta(\mathrm{m})$ time. Inserting or deleting a line segment requires $\theta(\lg \mathrm{m})$ time, and all segments that intersect a given point can be reported in time proportional to lg m plus the number of segments reported. The space required by the empty tree is $\theta(\mathrm{m})$ words, and since each segment can be represented up to $\theta(\lg m)$ times in the tree, the total worst case space required by a segment tree is $\theta(m \lg m)$ words.

V. The Entire Algorithm

In the previous sections of this paper we have investigated the rectangle intersection problem in a top-down fashion. **Algorithm RecInt** is a complete description of the resulting procedure.

Algorithm RecInt

Input

A list of n rectilinearly oriented rectangles. Each rectangle is defined by four reals giving its bottom, top, left, and right extreme points.

Output

A list of all intersecting pairs of rectangles. An element is added to this list by calling the "report" function in the following procedure. A pair of rectangles is said to intersect if either their edges intersect or one lies wholly within the other.

Procedure

1) Find all pairs of rectangles with intersecting edges. This step is accomplished by use of Bentley and Ottmann's [1979] algorithm for reporting all intersecting pairs among a set of horizontal and vertical line segments.

2) Find all pairs of rectangles in which one is entirely enclosed by the other. To accomplish this we will first generate a set of points T in which each rectangle is represented by one of its interior points (its centroid, for example). We will now find for all points a list of all the rectangles containing that point. We then process that list by comparing each rectangle on the list to the rectangle that the point is representing. If the rectangles have an edge intersection, nothing happens; otherwise the new rectangle entirely contains the rectangle represented by the point, and we report that pair of rectangles as intersecting. We now describe the procedure that will find for each point all the rectangles that contain it.

a) Sort the set of all rectangles and points by y-values. Each rectangle is represented twice, once for its bottom value and once for its top value, and each point is represented once. (This sorted list will be used in the bottom-to-top scan.)

b) Build an empty segment tree representing the (at most) $2n - 1$ x-intervals defined by the left and right endpoints of the n rectangles.

c) Scan through the sorted list created in step a). As each element is encountered, take the appropriate one of the following actions.

i) *Bottom of a Rectangle:* Insert the segment that is the projection of that rectangle onto the x-axis into the segment tree.

ii) *Top of a Rectangle:* Delete the corresponding segment.

iii) *A Point:* Search the segment tree to determine all segments that overlap the projection of this point onto the x-axis. The rectangles corresponding to these segments are precisely the rectangles containing the point, and they should be processed as described previously.

Time Analysis

We will show that the worst case running time of the preceding procedure is $\theta(n \lg n + k)$, where k is the number of intersecting rectangles reported. It is clear that our algorithm must take at least that much time (since it performs *a* sort of n elements and reports k intersecting pairs); we will now show that both steps 1) and 2) take no more than $O(n \lg n + k)$ time.

1) Bentley and Ottmann [5] show that the time required by this step is proportional to $n \lg n$ plus the number of edge intersections found. Since the number of such edge intersections is less than or equal to k (the total number of rectangle intersections), this step has running time of $O(n \lg n + k)$.

2) We analyze the components of this step as follows.

a) The sort takes time proportional to $n \lg n$.

b) The endpoints of the rectangles must be sorted by x-value in this step. After that, the empty tree can be constructed in linear time. The total time required by this step is therefore proportional to $n \lg n$.

c) During this scan each rectangle is examined twice and each point is examined once. The total number of objects examined is therefore $3n$. Each time a rectangle is examined we either insert or delete an element of the segment tree. Since the cost of these operations is logarithmic, the total time spent in insertions and deletions is proportional to $n \lg n$. The cost for each of the n segment searches performed in step iii) is proportional to lg n plus the number of line segments found. The total number of such containing segments found during the entire algorithm is bounded above by k, the number of intersecting rectangles. Hence, the total amount of time spent in this step is $O(n \lg n + k)$.

Space Analysis

The space required by step 1) is proportional to n, and the space required by step 2) is $\theta(n \lg n)$ [for representing each segment up to $\theta(\lg n)$ times in the segment tree]. The total space required by the algorithm is therefore $\theta(n \lg n)$ words.

Discussion

The $\theta(n \lg n + k)$ running time of this algorithm can be proved optimal by the methods described by Shamos and Hoey [13] (see also Fredman and Weide [7]). Bentley *et al.* [3] have implemented a program (based on **Algorithm RecInt**) in which the edge-intersection algorithm of Bentley and Ottmann [5] and the rectangle enclosure algorithm are combined in one scan through the data. Their program achieves linear expected time at the cost of having quadratic worst case time. Preliminary results indicate that their algorithm should be able to solve practical design rule problems very efficiently.

VI. Further Work

Although **Algorithm RecInt** answers an important question in the study of geometric intersection problems, a number of open problems remain. We will now briefly mention some of these directions open for research.

The $\theta(n \lg n + k)$ worst case running time of our algorithm is optimal. It is not known whether the $\theta(n \lg n)$ space it requires is also optimal. We conjecture that the space requirements can be reduced to $\theta(n)$.[5]

We have solved the *rectangle intersection problem* by combining two algorithms. The first algorithm (due to Bentley and Ottmann [5] determines the pairwise edge intersections, while the second, presented in the previous sections, determines all rectangle enclosures and some of the edge intersections. This leaves open the *rectangle enclosure problem* of determining all pairwise rectangle enclosures among a set of n rectangles. Can this be carried out in $\theta(n \lg n + k)$ time in the worst case, where k is the number of such enclosures?

A final open problem is to develop a general algorithm for reporting intersections in geometric sets. The algorithms of Shamos and Hoey [13] and Bentley and Ottmann [5] as well as those described in this paper, all crucially depend on exploiting a particular property of the objects among which they are finding intersections (such as circles, line segments, or rectangles). All of the algorithms do, however, share the same "scanning" strategy, and this suggests that there might exist a single $\theta(n \lg n + k)$ algorithm for reporting intersections in sets that contain many different kinds of geometric objects (such as circles, line segments, polygons, etc.). An obvious extension calls for working on objects in three (and higher) dimensional space; examples and applications of such algorithms are described by van Leeuwen and Wood [9] and Eastman and Lividini [6]. Preparata [11] and Shamos [12] discuss problems of searching in geometric sets; it would be nice to include the problems of searching in a general theory.

VII. Conclusions

In this paper we have examined two optimal worst case algorithms: one for the rectangle *intersection problem,* the other for the *batched range searching problem.* The rectangle intersection problem is particularly important in practice because of its relation to VLSI design rule checking. Two methods that we employed in deriving these algorithms might probe to be fundamental in computational geometry. The idea of scanning (solving a problem by passing through the point set in bottom-to-top order) has already found a number of applications; see Shamos [12]. A more novel technique is the idea of segment trees for representing line segments; an example of a very similar structure in another geometric problem can be found in Bentley [2].

[5] Dr. E. McCreight of Xerox Palo Alto Research Center has shown that this is indeed the case (in a private communication).

Acknowledgment

The helpful comments of M. Ast, D. Haken, and the anonymous referees are gratefully acknowledged.

References

[1] H. S. Baird, "Fast algorithms for LSI artwork analysis," *J. Des. Autom. Fault-Tolerant Comput.*, vol. 2, no. 2, pp. 179–209, 1978.

[2] J. L. Bentley, "Algorithms for Klee's rectangle problems," unpublished notes, Carnegie-Mellon Univ., Pittsburgh, PA, 1977 (described in van Leeuwen and Wood [9]).

[3] J. L. Bentley, D. Haken and R. Hon, "Fast geometric algorithms for VLSI tasks," IEEE CompCon Spring '80, 1980, pp. 88–92.

[4] J. L. Bentley, and H. A. Maurer, "Efficient worst-case data structures for range searching," *Acta Informatica,* vol. 13, no. 2, pp. 155–168, 1980.

[5] J. L. Bentley and T. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Trans. Comput.*, vol. C-28, pp. 643–647, Sept. 1979.

[6] C. M. Eastman and J. Lividini, "Spatial search," Institute of Physical Planning, Res. Rep. 55, Carnegie-Mellon University, Pittsburgh, PA, (16 pp.), May 1975.

[7] M. Fredman and B. W. Weide, "On the complexity of computing the measure of $u[a_i,b_i]$," *Commun. Assoc. Comput. Mach.*, vol. 21, no. 7, pp. 540–544, 1978.

[8] U. Lauther, "4-dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits," *J. Des. Autom. Fault-Tolerant Comput.*, vol. 2, no. 3, pp. 241–247, 1978.

[9] J. van Leeuwen and D. Wood, "The measure problem for rectangular ranges in *d*-space," Rijkuniversiteit Ultrecht Rep. RUU-CS-79-6, July 1979; also *J. Algorithms,* to be published.

[10] C. Mead and L. Conway, *Introduction to VLSI Systems.* Reading, MA: Addison-Wesley, 1980.

[11] F. P. Preparata, "A new approach to planar point location," Tech. Rep. ACT-11, Coordinated Sci. Lab., Univ. of Illinois, Urbana, IL, 1979.

[12] M. I. Shamos, "Problems in computational geometry," Ph.D. dissertation, Yale University, New Haven, CT.

[13] M. I. Shamos and D. J. Hoey, "Geometric intersection problems," in *Proc. 17th Annu. IEEE Symp. Foundations of Computer Science,* pp. 208–215, 1976.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching.* Reading, MA: Addison-Wesley, 1973.

Jon L. Bentley (M'79) was born in Long Beach, CA, on February 20, 1953. He received the B.S. degree in mathematical sciences from Stanford University, Stanford, CA, in 1974, and the M.S. and Ph.D. degrees

in computer science from the University of North Carolina at Chapel Hill in 1976.

He worked as a Research Intern at the Xerox Palo Alto Research Center from 1973 to 1974. During the summer of 1975 he was a Visiting Scholar at the Stanford Linear Accelerator Center. From 1975 to 1976 he was a National Science Foundation Graduate Fellow. He was awarded the Second Prize in the 1974 ACM Student Paper Competition. In 1977 he joined the faculty of Carnegie-Mellon University, Pittsburgh, PA, as an Assistant Professor of Computer Science and Mathematics. His primary research interests include the design and analysis of computer algorithms (especially for geometrical and statistical problems) and the mathematical foundations of computation. Other research areas in which he has worked include software engineering tools and novel computer architectures.

Dr. Bentley is a member of the Association for Computing Machinery and Sigma Xi.

Derick Wood received the Ph.D. degree in mathematics from the University of Leeds, Leeds, England in 1968.

From 1968 to 1970 he was associated with the Department of Computer Science, Courant Institute of Mathematical Sciences as an Assistant Research Scientist. In 1970 he joined the Department of Applied Mathematics at McMaster University, Hamilton, Ont., Canada, as an Assistant Professor. From 1978 he has been Professor of Applied Mathematics and currently is Chairman of the Unit for Computer Science. His research interests include formal language theory, search trees, data encoding, analysis of algorithms and computational geometry.

Dr. Wood is a member of the Association for Computing Machinery, the American Mathematical Society, the Canadian Information Processing Society and a member of the Council of the European Association for Theoretical Computer Science. He is also a member of the Editorial Board of the *International Journal of Computer Mathematics*.

A Dynamically Microprogrammable Computer with Low-Level Parallelism

HIROSHI HAGIWARA, SHINJI TOMITA, SHIGERU OYANAGI, AND KIYOSHI SHIBAYAMA

***Abstract*—A new microprogrammable computer with low-level parallelism was built and has been utilized as a research vehicle for solving different classes of research-oriented applications such as real-time processings on static/dynamic images, pictures and signals, and emulations of both existing and virtual machines including high (intermediate) level language machines. The design goal of the machine was to achieve a high degree of processing enhancement in research-oriented applications by means of a low-level parallel processing organization combined with dynamically microprogrammable control. The machine has the capability to process multiple data streams, performing parallel operations with four 16-bit ALU's. These ALU's are independently controlled by the different fields of a 160-bit horizontal-type microinstruction, and have simultaneous access to 15 working registers. This microprogrammed MIMD organization is expected to provide a greater degree of flexibility for low-level parallel processing. In addition, not only does the machine contain powerful ALU's and a large number of registers, but also it employs flexible control structures and a hierarchical organization of control storage. All of these combine to yield extensive microprogramming capability which the user can effectively tailor to a wide spectrum of applications.**

Throughout this paper, emphasis will be placed on the advantages of the low-level parallel processing system over conventional machines and on how it can be effectively tailored to a wide spectrum of applications.

***Index Terms*—Computer animation, emulation, firmware, microprogramming, parallel processing, real-time applications, virtual control storage.**

Manuscript received April 15, 1978; revised October 26, 1979. This work was supported in part by the Japanese Ministry of Education.

The authors are with the Department of Information Science, Kyoto University, Kyoto, Japan.

I. Introduction

VARIOUS requirements from the fields of hardware, software, and application have provided an impetus for the development of innovative techniques for the advancement of computer architectures. Microprogramming is one of the most typical techniques that attracts our attention by its flexibility in bridging the gap between hardware and software. It not only affords computer manufacturers systematic approaches to hardware design and maintenance, but also enables users to tailor the machine architectures to the applications at hand through the use of writable control storage.

We developed a research-oriented computer called QA-1. The design goal of the QA-1 was to achieve a high degree of processing enhancement in research-oriented applications by means of a low-level parallel organization combined with dy-