Hierarchical Representations of Collections of Small Rectangles

HANAN SAMET

Computer Science Department, Center for Automation Research, and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742

A tutorial survey is presented of hierarchical data structures for representing collections of small rectangles. Rectangles are often used as an approximation of shapes for which they serve as the minimum rectilinear enclosing object. They arise in applications in cartography as well as very large-scale integration (VLSI) design rule checking. The different data structures are discussed in terms of how they support the execution of queries involving proximity relations. The focus is on intersection and subset queries. Several types of representations are described. Some are designed for use with the plane-sweep paradigm, which works well for static collections of rectangles. Others are oriented toward dynamic collections. In this case, one representation reduces each rectangle to a point in a higher multidimensional space and treats the problem as one involving point data. The other representation is area based—that is, it depends on the physical extent of each rectangle.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids—*layout; placement and routing*; E.1 [**Data**]: Data Structures—*trees*; E.5 [**Data**]: Files—*organization/structure*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*geometrical problems and computations; routing and layout*; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*; I.2.10 [**Artificial Intelligence**]: Vision and Scene Understanding—*representations, data structures, and transforms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*curve, surface, solid, and object representations; geometric algorithms, languages, and systems*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided design (CAD)*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Cartography, geographic information systems, interval trees, hierarchical data structures, multidimensional data structures, plane-sweep methods, priority search trees, quadtrees, R-trees, rectangle intersection problem, rectangles, representative points, segment trees, VLSI design rule checking

INTRODUCTION

The problem of how to represent collections of small rectangles arises in many applications. The most common example is when a rectangle is used to approximate other shapes for which it serves as the minimum enclosing object. Of course, the exact boundaries of the object are also stored; but usually they are only accessed if a need for greater precision exists. For example, bounding rectangles can be used in cartographic applications to approximate objects such as lakes, forests, and

CONTENTS

hills [Matsuyama et al. 1984]. In such a case, the approximation gives a rough indication of the existence of an object. This is useful in processing spatial queries in a geographic information system. Such queries can involve the detection of overlapping areas, a determination of proximity, and so on. Another application is the detection of cartographic anomalies that require further resolution when a map is printed.

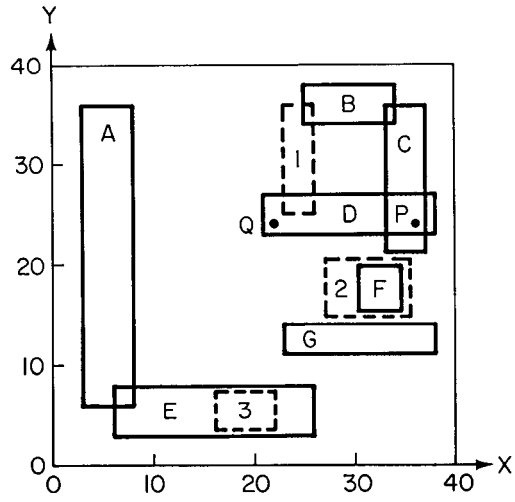
Rectangles are also used in the process of very large-scale integration (VLSI) design rule checking as a model of chip components for the analysis of their proper placement. Again, the rectangles serve as minimum enclosing objects. This process includes tasks such as determining whether components intersect and ensuring the satisfaction of constraints involving factors as minimum separation and widths. These tasks have a practical significance in that they can be used to avoid design flaws, and so on.

The size of the collection depends on the application; it can vary tremendously. For example, in cartographic applications the number of elements in the collection is usually small, and frequently the sizes of the rectangles are of the same order of magnitude as the space from which they are drawn. On the other hand, in VLSI applications, the size of the collection is quite large (e.g., millions of components), and the sizes of the rectangles are several orders of magnitude smaller than the space from which they are drawn.

In this tutorial we focus primarily on how to represent a large collection of rectangles as is common in VLSI applications. Our techniques, however, are equally applicable to other domains. We assume that all rectangles are positioned so that their sides are parallel to the x and y coordinate axes. We first give a general introduction to the problem domain and to the tasks whose solutions such representations are intended to facilitate. The representations and issues that we discuss are also common to multiattribute data. In order to compare the different representations, we will use the collection of rectangles given in Figure 1 and the collection of points corresponding to the locations of the cities given in Figure 2.

Initially, we present representations that are designed for use with the plane-sweep solution paradigm [Preparata and Shamos 1985; Shamos and Hoey 1976]. This solution consists of two passes. The first pass sorts the data along one dimension, and the second pass processes the result of the first pass. In this case, the rectangles are represented by the intervals that form their boundaries. Use of this paradigm requires maintaining a changing set of intervals whose endpoints are known in advance (a by-product of the initial sorting pass). One representation that we discuss is the segment tree. The segment tree represents the intervals as unions of atomic segments whose endpoints are members of the collection of rectangles. It turns out that the segment tree is suboptimal with respect to its space requirements, and this leads to the development of the interval tree. The interval tree does not decompose each interval into segments and hence reduces the space requirements without increasing the execution time. In essence, the interval tree is a balanced binary tree of suitably chosen points from the space spanned by the intervals so that interval I is associated with the point P that is closest to the root such that P is in I.

Our discussion of the plane-sweep solution paradigm uses the rectangle intersection problem as a motivating example. For



(a)

rectangle or point	X_L	X_R	Y_B	Y_T
A	3	8	6	36
B	25	34	34	38
C	33	37	21	36
D	21	38	23	27
E	6	26	3	8
F	31	35	15	19
G	23	38	11	14
1	23	26	25	36
2	27	35.5	14.5	20.5
3	16	22	3.5	7.5
P	36	—	24	—
Q	21	—	24	—

(b)

Figure 1. (a) A collection of rectangles. Members of the collection are designated by solid lines and labeled alphabetically (A–G). Query rectangles are designated by broken lines and labeled numerically (1–3). P and Q are query points. (b) The locations of the endpoints of the rectangles and the points in (a). For a rectangle, x_L and x_R correspond to its left and right boundaries, respectively, and y_B and y_T correspond to its bottom and top boundaries, respectively. For a point, x_L and y_B are its x and y coordinate values, respectively.

NAME	X	Y
CHICAGO	35	40
MOBILE	50	10
TORONTO	60	75
BUFFALO	80	65
DENVER	5	45
OMAHA	25	35
ATLANTA	85	15
MIAMI	90	5

Figure 2. A collection of points.

this task, as well as many other tasks, the plane-sweep paradigm leads to worst-case optimal solutions in time and space. Representations such as the segment and interval tree, however, are designed primarily for formulations of the tasks in a static environment. This means that the identity of all of the rectangles must be known a priori if the worst-case time and space bounds are to hold. Furthermore, for some tasks, the addition of a single object to the database may force the reexecution of the algorithm on the entire database.

The remaining representations are for a dynamic environment. They are differentiated by the way in which each rectangle is represented. The first type of representation reduces each rectangle to a point in a higher (usually) dimensional space and then treats the problem as if it involves a collection of points. The second type is region based in the sense that the subdivision of the space from which the rectangles are drawn depends on the physical extent of the rectangle—it does not just treat a rectangle as one point. A number of these region-based representations make use of variants of a data structure commonly referred to as a quadtree. Interestingly, these quadtree representations are very similar to the segment and interval trees that are used with the plane-sweep paradigm. Moreover, we observe that the quadtree serves as a multidimensional sort and the process of traversing it is analogous to a plane sweep in multiple dimensions. We conclude with a discussion of some representations that are more commonly used in a problem domain that involves a relatively small number of rectangles (e.g., as found in a cartographic application).

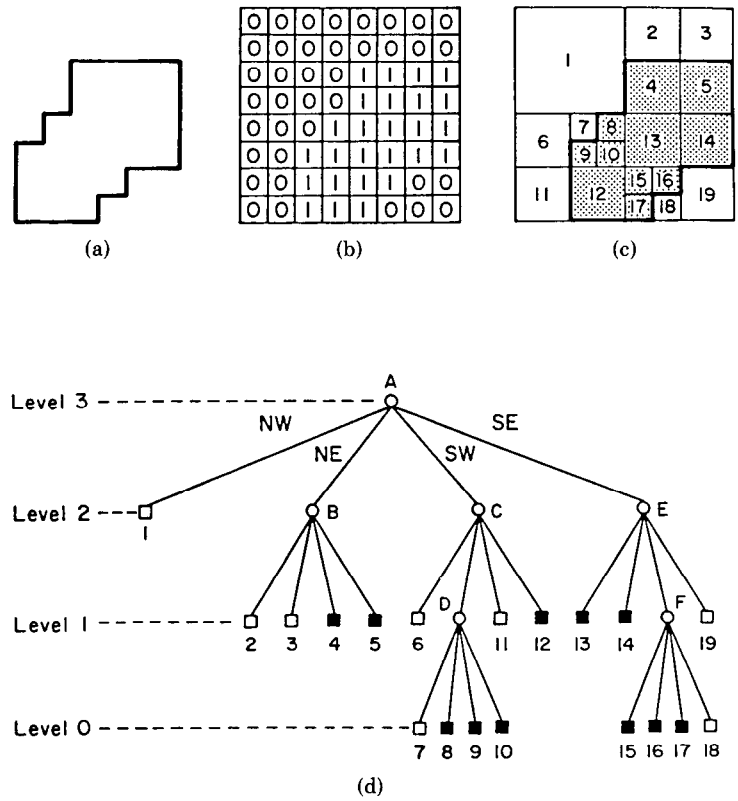


Figure 3. An example (a) region, (b) its binary array, (c) its maximal blocks (blocks in the region are shaded), and (d) the corresponding quadtree.

Since the focus of this tutorial is on hierarchical data structures, we will find ourselves making frequent comparisons with *quadtree* like data structures. Hence in the following we briefly review them. The term *quadtree* is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space (similar to divide and conquer methods [Aho et al. 1974]). They can be differentiated on (1) the type of data they are used to represent and (2) the principle guiding the decomposition process. The decomposition may be into equal parts on each level, termed a regular decomposition, or it may be governed by the input. The second distinction is very similar to that between a trie [Fredkin 1960] and a tree, respectively. The most common quadtree representations are the region quadtree [Hunter 1978; Klinger 1971; Samet 1984] (really a trie) and the point quadtree [Finkel and Bentley 1974].

The region quadtree is used to represent region data and is based on the successive subdivision of an image into four equal-size blocks until each block is of a uniform color or type. Figure 3 is an example of a binary image consisting of a region and its region quadtree representation. The region quadtree is based on a regular decomposition. On the other hand, the point quadtree is a representation of multiattribute data where the subdivision lines are determined by the input data. In essence, it is a multidimensional binary search tree. For example, Figure 4 is the point quadtree for the data in Figure 2 when the cities are inserted in the order in which they appear in Figure 2. See

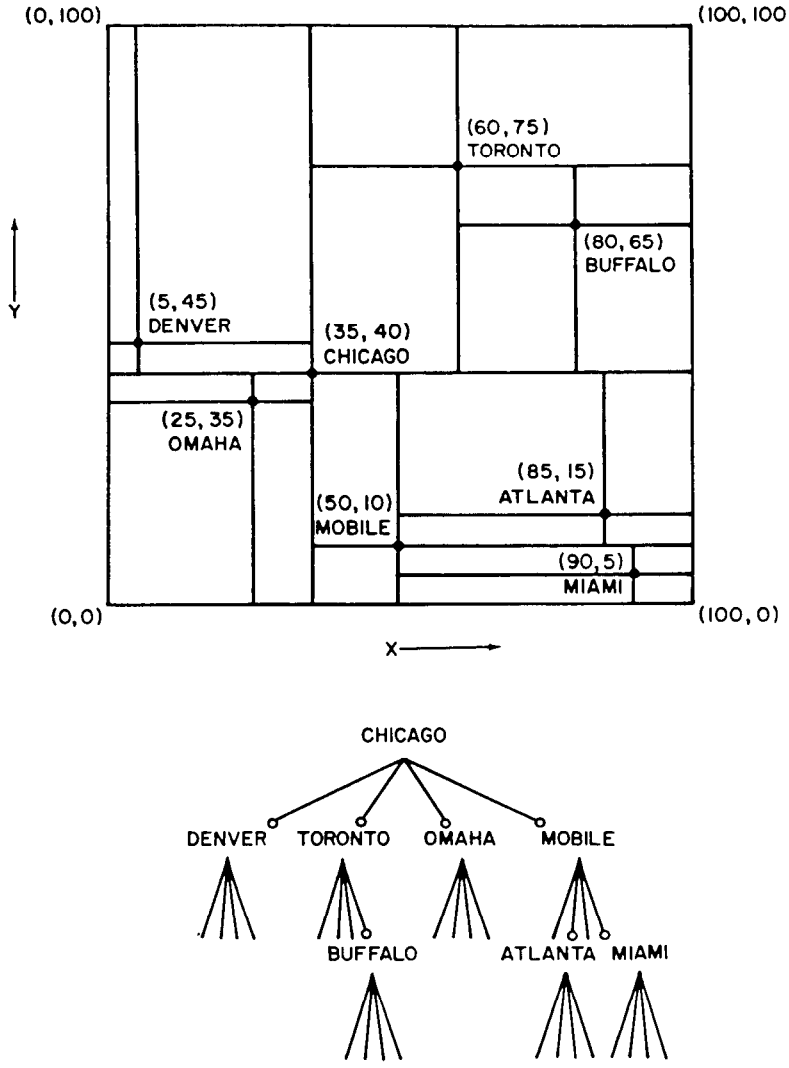


Figure 4. A point quadtree corresponding to the points in Figure 2.

Samet [1984, 1989a, 1989b] for more details on quadtrees and related hierarchical data structures.

1. CHOOSING A REPRESENTATION

In choosing a representation for a collection of objects[1] we are faced with two issues. First, we must choose a representation for the objects. Second, we must decide whether (and if so, how) to organize the objects that make up the collection. During our decision process we will be confronted with many of the same issues that arise in the representation of multiattribute data. For example, we must decide between a static and a dynamic representation, between making use of comparative search and address computation, and between retrieval on one key that is a combination of all the keys and just using a subset of the

[1] In this section we will speak of arbitrary objects although the subsequent discussion will be restricted to rectangles.

keys. Of course, there are many other options and factors, and thus a choice can only be made after a careful consideration of the type of operations (including queries) we wish to support. Not surprisingly, the operations are similar to those commonly considered for points. It should be mentioned that there are situations where the representation issue is not as crucial. For example, implementing an operation using plane-sweep methods (see Section 2) implies a sequential process in which only a subset of the objects is generally of interest. Thus, there is no need to be concerned about how to represent the entire collection of objects.

Hinrichs and Nievergelt [1983] and Hinrichs [1985a] suggest that the representations of the individual objects can be grouped into three principal categories, which are briefly described below. First, we can use a representative point (e.g., the centroid). Such an approach is not good for proximity queries if the object's extent (e.g., lengths of the sides for a rectangle) is not stored together with the coordinate values of the representative point.

Second, we can represent an object by its characteristic parts. There are many choices, some of which are outlined below:

(1) The representation can be based on the interior of the object. For example, we could decompose the object into smaller units (e.g., a decomposition of a rectangle into squares as would be done by a region quadtree). Each unit contains a pointer to the complete description of the object.

(2) The representation can be based on the boundary of the object. For example, polygons are often represented as an ordered collection of vertices or, equivalently, by the line segments comprising their boundaries.

(3) The representation can be procedural—that is, a combination of (1) and (2) according to a well-defined set of rules. The combination could also be based on a decomposition into units of a smaller dimension. For example, a rectangle is often represented as the Cartesian product of two one-dimensional spheres (i.e., intervals).

There are several difficulties with such approaches. One problem is that updating (e.g., insertion or deletion) will generally require processing several units. Another more serious drawback is that at times a query is posed in such a manner that none of the characteristic parts of an object, say O, that satisfies the query will match the query's description, yet O does satisfy the query. The problem is that not all properties are inherited by the parts. For example, suppose we are dealing with a collection of polygons stored so that each polygon is represented by the line segments that constitute its edges. We wish to determine whether a given polygon contains a given rectangle. Clearly, no edge of the polygon will contain the rectangle. Hence a solution to this problem requires that with each edge of a polygon (or rectangle) we store an identifier that indicates the polygons or objects associated with each of its sides.

Third, we can represent an object by partitioning the space from which it is drawn into cells that are adapted to the objects. Each cell is like a bucket that contains references to all objects that intersect it. The cells may be disjoint or may be permitted to overlap. In the latter case, if the partition is such that there always exists at least one cell that contains the object in its entirety, we can avoid the redundancy that is a natural consequence of the multiple references to the object. Of course, the fact that cells may overlap will increase the costs of certain query operations since several cells may cover a specific point.

Once a representation has been selected for the individual objects, we must choose how to represent the collection of the objects. There are numerous ways of doing so. In this tutorial we are primarily interested in hierarchical representations, so the bulk of our discussion concentrates on hierarchical methods and on operations for which they are useful. As we will see, the method that is used depends to a large degree on the manner in which the individual objects are represented. When objects are represented using representative points, data structures for multiattribute data are applicable (e.g., grid file [Nievergelt et al. 1984], *k-d* trees [Bentley 1975], point quadtrees [Finkel and Bentley 1974], PR

quadtrees [Orenstein 1982; Samet 1984], variations on B-trees [Comer 1979]). The choice depends on whether we wish to organize the data to be stored (i.e., methods based on comparative search) or the embedding space from which the data are drawn (i.e., methods based on address computation). Similar considerations apply when individual objects are represented by their characteristic parts, as is the case when using variants of the region quadtree.

The principal tasks that are to be performed are similar to those for multiattribute data. They range from the basic operations such as insertion and deletion to more complex queries that include exact match, partial match, range, partial range, finding all objects (e.g., rectangles) in a given region, finding nearest neighbors with respect to a given metric for the data domain, and even join queries [Ullman 1982]. The most common of these queries involves proximity relations and are classified into two classes by Hinrichs [1985a]. The first is an intersection query that seeks to determine if two sets intersect. This could be in the form of a window operation or query that finds all the rectangles that intersect (i.e., partially overlap) a given region. An alternative query is to determine all rectangles that intersect (i.e., partially overlap) other rectangles. The second is a subset relation and can be formulated in terms of enclosure (i.e., is A a subset of B) or of containment (i.e., does A contain B).

In describing queries involving these relations we must be careful to distinguish between a point and an object. A *point* is an element in the d-dimensional space from which the objects are drawn. It is not an element of the space into which the objects may be mapped by a particular representation. For example, in the case of a collection of rectangles in two dimensions, a point is an element of the Euclidean plane and not a rectangle even though we may choose to represent each rectangle by a point in some multidimensional space.

2. PLANE-SWEEP METHODS

The term *plane-sweep* is used to characterize a paradigm employed to solve geometric problems by sweeping a line (plane in three dimensions) across the plane (space in three dimensions) and halting at points where the line (plane) makes its first or last intersection with any of the objects being processed. At these points, the solution is partially computed so that at the end of the sweep a final solution is available. In this discussion we are dealing with two-dimensional data. Assume, without loss of generality, that the line is swept in the horizontal direction and from left to right. In order to use this solution technique, we need to organize two sets of data. The first set consists of the *halting points* of the line (i.e., the points of initial or final intersection). It is usually organized as a list of x coordinate values sorted in ascending order. The second set consists of a description of the status of the objects that are intersected by the current position of the sweep line. This status reflects the information relevant to the problem that is being solved, and it must be updated at each halting point. Thus, the data structure used to store the status must be *dynamic*. The characteristics of this data structure will determine, to a large extent, the efficiency of the solution.

The application of plane-sweep methods to rectangle problems is much studied. The solutions to many of these problems require the data to be ordered in a manner that makes use of some variant of multidimensional sorting. In such cases, the execution times of optimal algorithms are often constrained by how fast we can sort, which for N objects usually means a lower bound of $O(N \cdot \log_2 N)$. At times, an increase in speed can be obtained by making use of more storage. The text of Preparata and Shamos [1985] contains an excellent discussion of a number of problems to which such techniques are applicable.

We assume that each rectangle is specified by four values; the x coordinates of its two vertical sides and the y coordinates of its two horizontal sides (equivalently, these are the x and y coordinates of its lower-left and upper-right corners). We also assume that each rectangle is closed on its left and bottom sides and open on its top and right sides. Applying the same open–closed convention to the boundaries of a rectangle finds that its horizontal (vertical) bounda-

ries are closed on their left (bottom) ends and open on their right (top) ends. Alternatively, the boundaries can be described as being *semiclosed.*

In this section we focus on the efficient solution of the problem of reporting all intersections between rectangles and, to a lesser extent, on some related problems. We assumed that a static environment, that is, the identity of all rectangles is known a priori. Note that a naive way to report all intersections is to check each rectangle against every other rectangle, which requires $O(N^2)$ time for N rectangles. The plane-sweep solution of the problem consists of two passes. The first pass sorts the left and right boundaries (i.e., x coordinate values) of the rectangles in ascending order and forms a list. For example, consider the collection of rectangles given in Figure 1. Letting R_l and R_r denote the left and right boundaries of rectangle R, the result of the first pass is a list consisting of 3, 6, 8, 21, 23, 25, 26, 31, 33, 34, 35, 37, 38, 38 corresponding to A_l, E_l, A_r, D_l, G_l, B_l, E_r, F_l, C_l, B_r, F_r, C_r, D_r, G_r, respectively.

The second pass sweeps a vertical scan line through the sorted list from left to right halting at each one of these points. This pass requires solving a quasi-dynamic version of the one-dimensional intersection problem. At any instant, all rectangles that intersect the scan line are considered *active* (e.g., rectangles D, E, and G for a vertical scan line through $x = 24$ in Figure 1). We must report all intersections between a newly activated rectangle and currently active rectangles that lie on the active scan line. The sweep process halts every time a rectangle becomes active (causing it to be inserted in the set of active rectangles) or ceases to be active (causing it to be deleted from the set of active rectangles). The key to a good solution is to organize the active rectangles so that intersection detection, insertion, and deletion are executed efficiently.

The first pass involves sorting, and thus it requires $O(N \cdot \log_2 N)$ time. Insofar as the second pass is concerned, each rectangle is nothing more than a one-dimensional vertical line segment. There are several data structures that can be used to represent line segments. If we only care about reporting the intersections of boundaries (i.e., vertical boundaries with horizontal boundaries), then a balanced binary tree is adequate to represent the bottom and top boundaries (i.e., y coordinate values) of the active line segments [Bentley and Ottmann 1979]. Unfortunately, such a representation fails to account for intersections that result when one rectangle is totally contained within another rectangle.

In the rest of this section we focus on solutions that use the segment, interval, and priority search trees to represent the active line segments. We first explain the segment tree and then show how the order of the space requirements of the solution can be reduced by using either the interval or priority search trees while still retaining the same order of execution time. We conclude by briefly explaining how some related problems can be solved using the same techniques.

2.1 Segment Trees

The segment tree is a representation for a collection of line segments devised by Bentley [1977]. It is useful for detecting all the intervals that contain a given point. It is best understood by first examining a simplified version that we call a *unit-segment tree*, which is used to represent a single line segment. For the moment, assume that the endpoints of the line segments of our collection are drawn from the set of integers $\{i \mid 0 \leq i \leq 2^h\}$. Let S be a line segment with endpoints l and r ($l < r$). S consists of the set of consecutive unit intervals $[j: j + 1)$ ($l \leq j < r$). The unit-segment tree is a complete binary tree of depth h such that the root is at level h, and nodes at level 0 (i.e., the bottom) represent the sequence of consecutive intervals $[j: j + 1)$ $(0 \leq j < 2^h)$. A node at level i in the unit-segment tree represents the interval $[p: p + 2^i)$ (i.e., the sequence of 2^i consecutive unit intervals starting at p where $p \bmod 2^i$ is 0).

Representing line segment S in a unit-segment tree is easy. We start at the root of the tree and check if its corresponding interval is totally contained in S. If yes, then we *mark* the node with S. In such a

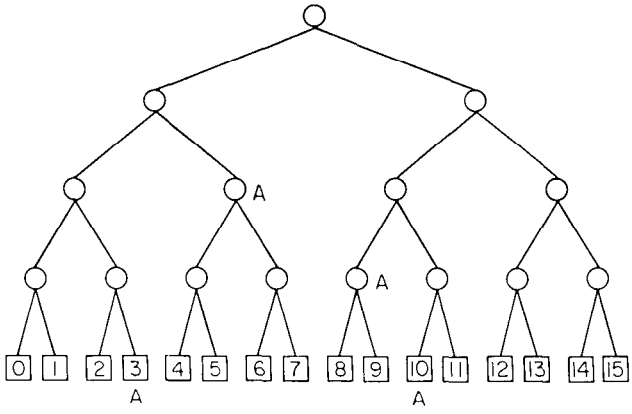


Figure 5. The unit-segment tree for the segment [3:11) labeled A in the range [0:16).

case, we say that S *covers* the node's interval. Otherwise, we repeat the process for the left and right sons of S. This process visits at most four nodes at each level while marking at most two of them. Thus, it is easy to see that inserting a line segment into a unit-segment tree in a top-down manner can be achieved in $O(h)$ time. An equivalent bottom-up description of this process is that a node is marked with S if all (i.e., both) the intervals corresponding to its sons are totally contained in S, in which case the sons are no longer marked with S.

As an example of the unit-segment tree, consider a collection of line segments with integer endpoints that are in the range [0:16). In this case, there are 16 possible intervals, each of unit length. The unit-segment tree for a line segment, named A, of length 8 whose endpoints are at 3 and 11 is given in Figure 5. Note that the interval $[i: i + 1)$ is represented by the node labeled i. From the figure it is easy to observe the close analogy between the unit-segment tree and a one-dimensional region quadtree [Rosenfeld and Kak 1982], where the unit intervals are the one-dimensional analog of pixels. The analogy is completed by letting BLACK (WHITE) correspond to the labeled (unlabeled) nodes and merging brother nodes of the same color.

The unit-segment tree is inadequate for two reasons: (1) it can only represent one line segment, and (2) it is only defined for line segments with integer endpoints. The segment tree is an adaptation of the unit-segment tree that enables the use of one data structure to represent a collection of line segments with arbitrary endpoints by removing the restriction that the intervals be of uniform length, and by replacing the mark at each node by a linked list of the names of the line segments that contain that node. This is achieved in the following manner. Given a set of N line segments, we first sort their endpoints and remove duplicates to obtain $y_0, y_1, \ldots, y_m$ $(m < 2N)$. Next, we form the segment tree in the same way as the unit-segment tree with the exception that interval $[j: j + 1)$ is replaced by the interval $[y_j: y_{j+1})$ $(0 \leq j < 2^h$ and $2^{h-1} \leq m < 2^h)$. Each line segment S with endpoints y_l and y_r consists of the sequence of consecutive intervals $[y_j: y_{j+1})$ $(l \leq j < r)$. A node at level i in the segment tree represents the interval $[y_p: y_{p+2^i})$ (i.e., the sequence of 2^i consecutive intervals starting at y_p, where p mod 2^i is 0). Each node is marked with the names of all the line segments that cover the node's corresponding interval and that do not cover the corresponding interval of the parent node. As in the case of the unit-segment tree, a node and its brother cannot be both marked with the same line segment. The set of line segments associated with each node is represented as a doubly linked list.

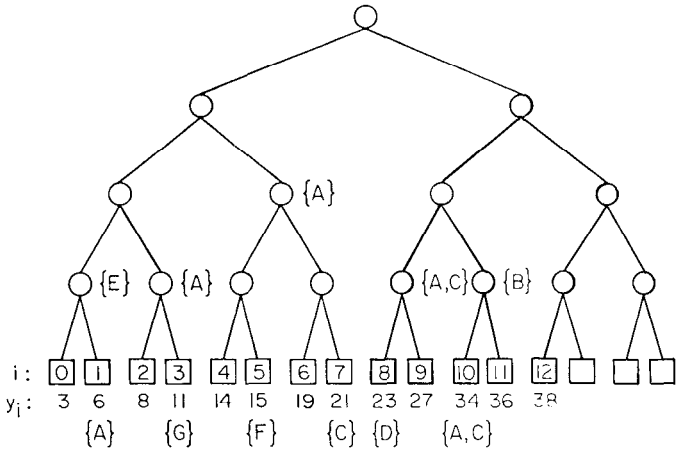


Figure 6. The segment tree for the set of line segments corresponding to the vertical boundaries of the rectangles in Figure 1. Terminal node i corresponds to the interval $[y_i : y_{i+1})$.

For example, Figure 6 is the segment tree for the set of line segments that correspond to the vertical boundaries of the rectangles in Figure 1. Although there are seven line segments, the segment tree contains 12 intervals since there are only 13 different endpoints. Since the segment tree is a complete binary tree, in this case it has four unused intervals. Each terminal node is labeled with its corresponding interval number and the leftmost endpoint of the interval—that is, node i corresponds to the interval $[y_i: y_{i+1})$. Nodes are also labeled with the sets of names of the line segments that cover their corresponding intervals. For example, the interval [23 : 34) is labeled with {A, C} since it is covered by these line segments.

Inserting a line segment in the segment tree is analogous to inserting it in the unit-segment tree. The only difference is that the line segment must also be placed in the list of the line segments that is associated with the node. It can be placed anywhere in the list and thus we usually attach it to the front of the list. In a domain of N line segments, insertion (into the tree and list) takes $O(\log_2 N)$ time per line segment.

Deleting a line segment from a segment tree is somewhat more complex. We must remove the line segment from the doubly linked list that is associated with each node that contains it. This could be expensive, since in the worst case it requires the traversal of $O(\log_2 N)$ linked lists, each containing $O(N)$ entries. This difficulty is avoided by maintaining an auxiliary table with one entry for each of the N line segments. Each table entry points to a list of pointers. Each pointer points to a list entry for the line segment in a node, say P, of the segment tree such that P's interval is covered by the line segment. This table is built as the line segments are entered into the segment tree. It contains at most N entries and can be accessed or updated in $O(\log_2 N)$ time when implemented as a balanced binary tree (or even in constant time if implemented as an array, in which case each line segment must be represented by a unique integer in the range $1 \cdots N$). We can use an array instead of a dictionary because we know the identities of the rectangles in advance (i.e., a static environment).

A segment tree for N line segments has a maximum of $2 \cdot N$ leaf nodes. Each line segment covers the intervals of at most $2 \cdot \lceil \log_2 N \rceil$ nodes of the segment tree. At each of these nodes, deletion of the line segment can be done in constant time, since the segment lists that are associated with these nodes are implemented as doubly linked lists. Thus, the total cost of deleting a line segment is $O(\log_2 N)$. The

segment tree has a total of $O(N)$ nodes, and since each line segment can appear in (i.e., cover) $O(\log_2 N)$ nodes, the total space required (including the auxiliary table) in the worst case is $O(N \cdot \log_2 N)$. Interestingly, Bucher and Edelsbrunner [1983] have shown that the average space requirement for a segment tree is also $O(N \cdot \log_2 N)$.

Given F rectangle intersections, using a segment tree to determine the set of rectangles that intersect each other is somewhat complex [Bentley and Wood 1980] if we want to do it in $O(N \cdot \log_2 N + F)$ time. In particular, it involves considerably more work than just inserting a line segment and reporting the rectangles associated with the line segments that were encountered during the insertion process. Conceptually, the problem is quite straightforward—for each line segment S, with starting and ending points l and r, respectively, we want the set of line segments A such that $A_i \cap S$ is nonempty for each $A_i \in A$. Recalling that the segment tree is good for detecting all intervals that contain a given point, we formulate the problem as an infinite set of point queries—that is, for each point p_i in line segment S find all line segments that contain it. This process requires $O(\log_2 N)$ time for each point that is queried. In order to avoid looking at every point in S (an infinite number!), we can restrict our search to the endpoints of the line segments that are overlapped by S. An obvious, but unacceptable solution is to explicitly store with each line segment the set of segments that intersect it, at a total storage cost of $O(N^2)$.

A more reasonable solution, which makes use of the above restriction on the search, is given by Six and Wood [1980, 1982], who decompose the search into two disjoint problems. They make use of the obvious fact that any line segment whose starting point is greater than r or whose ending point is less than l does not intersect the line segment S. The first problem consists of determining all line segments with starting points less than l whose intersection with S is nonempty. The second problem consists of determining all line segments with starting points that lie between l and r. Thus, there is really only a need to be concerned with an ordering that is based on the starting points.

The first problem is solved by performing a point query for point l on the segment tree representation of the line segments. In order to determine all the line segments that contain l, we simply locate the smallest interval that contains it. Since a segment tree for N line segments has a maximum of $2 \cdot N$ leaf nodes, this search visits at most $\lceil \log_2 N \rceil + 1$ nodes. For each node that is visited, we traverse its associated list of line segments and report them as containing l. This process requires $O(\log_2 N + F_l)$ time, where F_l is the number of line segments that contain l. Since a segment tree is used, it needs $O(N \cdot \log_2 N)$ space.

The second problem is solved by performing a range query for range $[l:r)$ on the set of starting points of the line segments. This query is one for which a range tree [Bentley 1979, Bentley and Maurer 1980] that stores the starting points of the line segments is ideal. A range tree is a balanced binary tree where the data points are stored in sorted order in the leaf nodes, which are linked in this order by use of a doubly linked list. Therefore, insertion and deletion are both $O(\log_2 N)$ processes. A range query consists of locating the node corresponding to the start of the range, say L, and the closest node to the end of the range, say R, and then reporting the line segments corresponding to the nodes that lie between them by traversing the linked list of nodes. This process requires $O(\log_2 N + F_{lr})$ time, where F_{lr} is the number of line segments with starting points in $[l:r)$. Since a balanced binary tree is used, it needs $O(N)$ space. The combination of the point and range query solution requires $O(N \cdot \log_2 N)$ space and $O(N \cdot \log_2 N + F)$ time, where F is the number of rectangle intersections.

2.2 Interval Trees

Suppose we try to determine the set of rectangles that intersect each other by just using a segment tree. The problem with this approach is that upon insertion of a line segment, say S, in a segment tree, we

cannot find all of the existing line segments in the tree that are totally contained in S, or partially overlap S, without examining every node in each subtree that contains S. For example, consider the segment tree of Figure 6 without line segment A (i.e., for segments B, C, D, E, F, and G). Upon inserting line segment A in the node corresponding to interval [14:23), the only way to determine the existence of line segment F (corresponding to the interval [15:19)) that is totally contained in A is to descend to the bottom of the subtree rooted at [14:23). A similar example can be constructed to show that this problem also arises in the case of partial overlap. Unfortunately, checking for total containment or partial overlap in this way takes $O(N)$ time for each line segment, or $O(N^2)$ for all the line segments.

The above problem can be overcome in part by making the following modifications to the segment tree. Link each marked node (i.e., a node whose corresponding interval overlaps at least one line segment), say P, to some of the nodes in P's subtrees that are marked. This could be implemented by an auxiliary binary tree whose elements are the marked nodes. Since each line segment can be associated with more than one node in the segment tree, the number of intersections that can be detected is bounded by $2 \cdot N^2 \cdot \log_2 N$, while the number of different intersections is bounded by N^2. Removing duplicates will require sorting, and even use of the bin method [Weid 1978], which is linear, still leaves us with an $O(N^2 \cdot \log_2 N)$ process. The duplicate entries, however, can be avoided by redefining the segment tree so that a line segment is only associated with one node—the nearest common ancestor[2] of all the intervals contained in the line segment (e.g., the node corresponding to the interval [3:38) for line segments A and C in Figure 6). The absence of duplicate entries also means that the space requirements can be reduced to $O(N)$.

[2] The principle of associating key information with the nearest common ancestor is similar to Chazelle and Guibas' [1986a, 1986b] *fractional cascading*. It is also used as the basis of an efficient solution of the point location problem by Edelsbrunner et al. [1986].

The above modifications serve as the foundation for the development of the *interval tree* of Edelsbrunner [1980a, 1983a, 1983b] and the tile tree of McCreight [1980]. The difference between them is that the tile tree is based on a regular decomposition, while the interval tree is not. In the rest of this section, we only discuss the interval tree.

The interval tree is designed specifically to detect all intervals that intersect a given interval. It is motivated by the dual goals of reducing the space requirement to $O(N)$ while maintaining an execution time of $O(N \cdot \log_2 N + F)$. The interval tree solution also makes use of the decomposition of the search into the two disjoint tasks of

(1) determining all line segments that overlap the starting point of the query line segment, and

(2) determining all line segments whose starting point lies within the query line segment.

Once again, assume that we are given a set of N line segments such that line segment S_i corresponds to the interval $[l_i : r_i)$—that is, l_i and r_i are its left and right endpoints, respectively. The endpoints of the N line segments are sorted (with duplicates removed) to obtain the sequence $y_0, y_1, \ldots, y_m$ ($m < 2N$ and $2^{h-1} \leq m < 2^h$). The interval tree is a three-level structure, where the first (and principal) level is termed the *primary* structure, the second level is termed the *secondary* structure, and the third level is termed the *tertiary* structure. We shall illustrate our discussion with Figure 7, the interval tree for the set of line segments corresponding to the vertical boundaries of the rectangles in Figure 1.

The primary structure is a complete binary tree with $m + 1$ external (i.e., terminal) nodes such that when the tree is flattened and the internal nodes are removed, external node i corresponds to y_i. In Figure 7, the primary structure is denoted by solid lines. In our example, although there are 7 line segments, the primary structure contains only 13 external nodes as there are only 13 different endpoints. Each terminal node is labeled

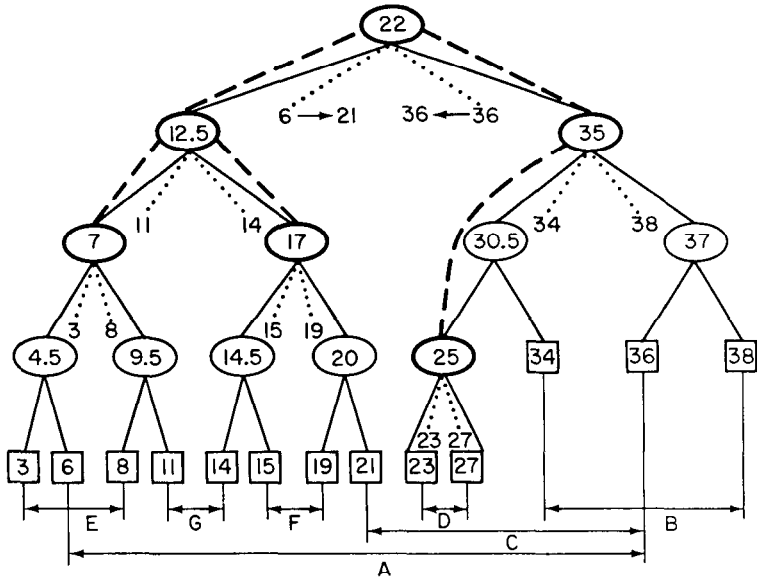


Figure 7. The interval tree for the set of line segments corresponding to the vertical boundaries of the rectangles in Figure 1. The primary structure is shown by solid lines. The secondary structure is shown by dotted lines. The tertiary structure is shown by broken lines, with the active nodes circled with thick lines. The interrelationships between the endpoints of the line segments are also shown.

with its corresponding endpoint [i.e., y_i for terminal node i $(0 \leq i < 2N)$]. Each internal node is assigned an arbitrary value, stored in the field VAL, that lies between the maximum value in its left subtree and the minimum value in its right subtree (usually their average). For example, the root node in Figure 7 is labeled with 22.

Given a line segment corresponding to the interval $[l{:}r)$, we say that its *nearest common ancestor* in the interval tree is the internal node that contains l and r in its left and right subtrees, respectively. For example, in Figure 7, node 22 is the nearest common ancestor of line segment A, which corresponds to the interval [6, 36).

Each internal node in the primary structure, say v, serves as the key to a pair of secondary structures LS and RS that represent the sets of left and right endpoints, respectively, of the line segments for which v is the nearest common ancestor (i.e., they contain v's value). Elements of the sets LS and RS are linked in ascending and descending order, respectively. In Figure 7, the secondary structure is denoted by dotted lines emanating from each internal node that has a nonempty secondary structure. The sets LS and RS are distinguished by dotted lines emanating from the internal node to its left and right sides, respectively. When LS and RS contain more than one entry, we show them linked in increasing and decreasing order, respectively (e.g., LS of the root node shows 6 pointing to 21 since the corresponding intervals are [6 : 36) and [21 : 36)). Each starting (ending) point appears in LS (RS) as many times as there are line segments that have it as a starting (ending) point. For example, 36 appears twice in RS of the root node in Figure 7 as it is the ending point of line segments A and C. In order to support rapid insertion and deletion, the sets LS and RS are implemented as balanced binary trees (as well as doubly linked lists).

Each internal node in the primary structure has eight pointers—two to its left and right subtrees in the primary structure (LP and RP), two to the roots of LS and RS in

the secondary structure, one to the minimum value in LS, one to the maximum value in RS, and two (LT and RT) to its left and right subtrees in the tertiary structure, which we discuss below.

An internal node in the primary structure is marked *active* if its corresponding secondary structure is nonempty or both of its sons have active descendants; otherwise, it is marked *inactive*. The active nodes of the primary structure form the tertiary structure, which is a binary tree. It is rooted at the root of the primary structure and is linked via the LT and RT fields of the internal nodes of the primary structure. If node v of the primary structure is inactive, then LT(v) and RT(v) are Ω [i.e., pointers to Nil]. If v is active, then LT(v) points to the closest active node in the left subtree of v [i.e., in LP(v)], and RT(v) points to the closest active node in the right subtree of v [i.e., in RP(v)]. If there are no closest active nodes in the left and right subtrees of v, then LT(v) and RT(v), respectively, are Ω (i.e., pointers to NLL). In Figure 7, the tertiary structure is denoted by broken lines linking all of the active nodes (e.g., nodes 22, 12.5, 7, 17, 35, and 25), which are also marked with thicker ellipses. The tertiary structure is useful in collecting the line segments that intersect a given line segment and enables us to avoid examining primary nodes whose corresponding line segments do not. It can be shown that more than half of the elements of the tertiary structure (i.e., active nodes) have nonempty secondary structures.

Inserting the line segment $[l:r)$ in the interval tree is very simple. We start at the root and locate the first node v such that $l < \text{VAL}(v) < r$. In this case, we insert l into LS(v) and r into RS(v). Both of these processes can be achieved in $O(\log_2 N)$ time. Updating the tertiary structure requires us to traverse it in parallel with the primary structure and takes $O(\log_2 N)$ time. Deletion of a line segment is performed in a similar manner and with the same complexity.

Reporting the rectangle intersections in an interval tree is straightforward, although there are a number of cases to consider. Again, this task is performed while inserting a vertical line segment, say S, corresponding to the interval $[l:r)$. During this process we search for and report the line segments that overlap S. Assume that the interval tree is rooted at T_1. The search has the following three stages:

(1) Start at T_1 and find the first node v such that $l < \text{VAL}(v) < r$.
(2) Start at v and locate l in the left subtree of v.
(3) Start at v and locate r in the right subtree of v.

This search involves the secondary structures of the nodes in the primary structure. The tertiary structure is used to limit the number of nodes in the primary structure, with empty secondary structures, that must be examined. Note that all of the overlapping line segments will be reported, and each will be reported only once since it is associated with the secondary structure of just one node in the primary structure.

In the following we present the main ideas of the three stages. Figure 8 aids the visualization of the symbols used in the explanation for $[l:r)$. All secondary and tertiary structures that are visited are marked with dotted and broken lines, respectively.

First, we explain stage (1). $\{T_i\}$ denotes the set of nodes encountered during the search for v. We use the insertion of line segment [6:20) into the interval tree of Figure 7 as our example. The secondary structures associated with each T_i must be checked for a possible overlap with S. This is quite simple. Either $l < r < \text{VAL}(T_i)$ or $\text{VAL}(T_i) < l < r$.

If $r < \text{VAL}(T_i)$, then we need only report the line segments in the secondary structure of T_i whose starting points are less than r (e.g., line segment A upon examining the internal node with value 22). To achieve this we visit LS(T_i) in ascending order until we encounter a line segment whose starting point exceeds r. We then search the left subtree of T_i [i.e., LP(T_i)].

Similarly, if $\text{VAL}(T_i) < l$, we need only report the elements of the secondary structure of T_i whose ending points are greater than l. To do this, we visit RS(T_i) in de-

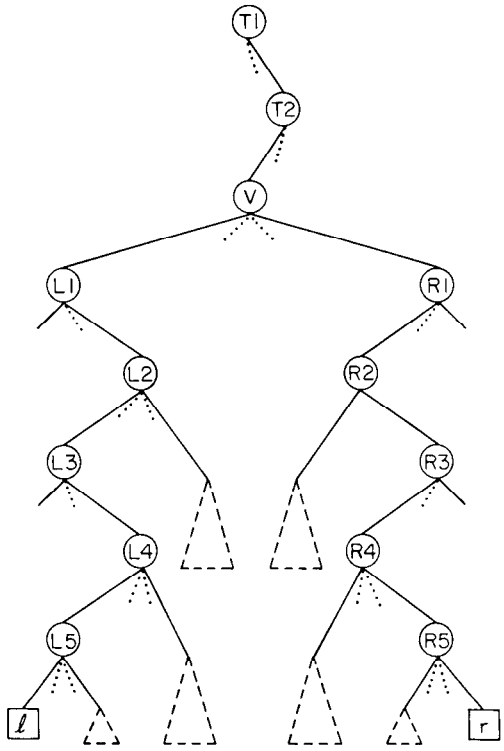


Figure 8. Example of an interval tree search for the interval $[l : r)$. All secondary structures that are visited are marked with dotted lines. All tertiary structures that are visited are marked with broken lines.

scending order until encountering a line segment whose ending point is less than l. The search is then continued in the right subtree of T_i [i.e., RP(T_i)].

Both of these cases are executed in time proportional to the number of intersections that are reported. Once node v has been located we report all elements of its secondary structure as intersecting S. In our example, we would report line segment G, since v is the internal node with value 12.5.

Now, we explain stages (2) and (3). They are very similar and thus we just discuss stage (2). We use the insertion of line segment [6:34) into the interval tree of Figure 7 as our example. In this case, v is the root of the tree (the internal node with value 22). Let $\{L_i\}$ denote the set of nodes encountered during the search for l in this stage. Recall that $l <$ VAL(v). Either $l <$ VAL(L_i) or VAL(L_i) $< l$.

If $l <$ VAL(L_i), then S intersects every line segment in the secondary structure of L_i as well as all the line segments in the secondary structures in the right subtree of L_i. The first set consists of just the line segments in RS(L_i) (e.g., line segment G upon examining the internal node with value 12.5). The second set is obtained by visiting all the active nodes in the right subtree of L_i, RP(L_i) (e.g., line segment F during the processing of the internal node with value 12.5 since F is associated with the active internal node with value 17). In order to avoid visiting irrelevant nodes we make use of the tertiary structure using the pointers LT(L_i) and RT(L_i). It can be shown that more than half of the elements of the tertiary structure have nonempty secondary structures, and thus the time necessary to execute this process is proportional to the number of intersections that are reported. The search is continued in the left subtree of L_i, LP(L_i).

If VAL(L_i) $< l$, then we report the line segments in the secondary structure of L_i whose ending points are greater than l. To do this, we visit RS(L_i) in descending order until encountering a line segment whose ending pont is less than l. This process is executed in time proportional to the number of intersections that are reported. The search is continued in the right subtree of L_i, RP(L_i).

Solving the rectangle intersection problem using an interval tree requires $O(N)$ space and $O(N \cdot \log_2 N + F)$ time, where F is the number of rectangle intersections.[3] The space requirements are obtained by observing that for N line segments we need at most $2 \cdot N$ terminal nodes in the primary structure and likewise for the secondary structures. The tertiary structure is constructed from nodes in the primary structure, and thus it requires no additional space except for the pointer fields. Making use of the fact that the in-

[3] For the tile tree, the execution time is $O(N \cdot \log_2(\max(N, K)) + F)$, where the horizontal boundaries of the rectangles are integers between 0 and $K - 1$. The execution time becomes $O(N \cdot \log_2 N + F)$ if the $2 \cdot N$ y coordinate values are first sorted and then mapped into the integers from 0 to $2 \cdot N - 1$.

terval tree is a complete binary tree, the number of internal nodes in the primary and secondary structures is bounded by $2 \cdot N - 1$.

The execution time requirements are obtained by noting that searching the primary structure for the starting and ending points of a line segment takes $O(\log_2 N)$ time. The number of nodes in the secondary structure that are visited is of the same order as the number of rectangle intersections that are found. Since at least one-half of the active nodes have nonempty secondary structures, the number of nodes in the tertiary structure that are visited is no more than twice the number of nodes visited in the secondary structure. Constructing the interval tree takes $O(N \cdot \log_2 N)$ time since the endpoints of the line segments that form the sides of the rectangles must be sorted.

2.3 Priority Search Trees

Using an interval tree, as described in Section 2.2, yields an optimal worst-case space and time solution to the rectangle intersection problem. The interval tree solution requires that we know in advance the endpoints of all of the vertical intervals since they must be sorted and stored in a complete binary tree. Thus, given N rectangles, the storage requirement is always $O(N)$. The solution can be slightly improved by adapting the priority search tree of McCreight [1985] to keep track of the active vertical intervals. In this case, the storage requirements for the sweep pass only depend on the maximum number of vertical intervals that can be active at any one time, say M. Moreover, there is no need to know their endpoints in advance, and thus there is no need to sort them. This also has an effect on the execution time of the algorithm since the data structure used to keep track of the endpoints of the vertical intervals is the determinative factor in the amount of time necessary to do a search. Thus, when using the priority search tree, the sweep pass of the solution to the rectangle intersection problem can be performed in $O(N \cdot \log_2 M + F)$ time, rather than $O(N \cdot \log_2 N + F)$ time. However, sorting the endpoints of the horizontal intervals, which is the first pass of the plane sweep, still requires $O(N \cdot \log_2 N)$ time.

A priority search tree keeps track of points in a two-dimensional space. It is built in the following manner. Assume that no two data points have the same x coordinate value. Sort all the points along the x coordinate and store them in the leaf nodes of a balanced binary search tree, say T. We proceed from the root node toward the leaf nodes. With each internal node of T, say I, associate the point in the subtree rooted at I with the maximum value for its y coordinate that has not already been stored at a shallower depth in the tree. If such a point does not exist, then leave the node empty. For example, treating the vertical boundaries $[y_B, y_T)$ of the rectangles in Figure 1 as points (x, y), Figure 9 is their corresponding priority search tree. For N points, the priority search tree requires $O(N)$ storage.

The priority search tree is designed for solving queries involving semi-infinite ranges in two-dimensional space. Performing a semi-infinite range query $([L_x : R_x], [L_y : \infty])$ using a priority search tree is done as follows. Descend the tree looking for the nearest common ancestor of L_x and R_x, say Q. Now, recursively, apply the following search procedure to the subtree rooted at Q. Let T denote the root of the subtree that is currently being processed, and let P be the point associated with it. If no such P exists, then we are finished with the entire subtree rooted at T since all points in its subtrees have already been examined and/or reported. Examine the y coordinate value of P, say P_y. If $P_y < L_y$, then we are finished with the entire subtree rooted at T since P is the point with the maximum y coordinate value in the subtree. Otherwise, perform the following steps:

(1) Check if the x coordinate value of P is in the range $[L_x : R_x]$; if yes, then output P as satisfying the query.

(2) Determine where the search is to be continued. If both T and its right son are on the path from Q to L_x, then continue in the right son of T; else if both T and its left son are on the path from Q to R_x, then continue in the left

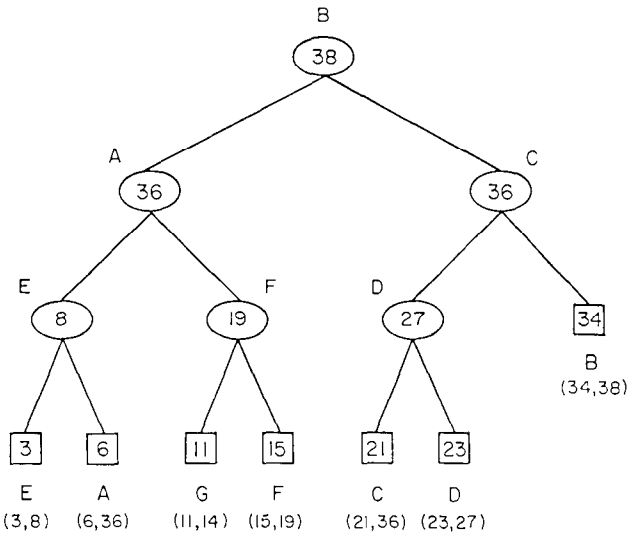


Figure 9. The priority search tree for the set of line segments corresponding to the vertical boundaries $[y_B, y_T)$ of the rectangles in Figure 1. Each vertical boundary $[y_B, y_T)$ is treated as a point (x, y) in a two-dimensional space. The leaf nodes contain the y_B values and the internal nodes contain the maximum y_T values.

son of T; else continue in the two sons of T.

For N points, this process requires $O(\log_2 N + F)$ time, where F is the number of points found.

There are two keys to understanding the use of the priority search tree in solving the rectangle intersection problem. Assume, again, that all intervals are semi-closed (i.e., they are closed on their left ends and open on their right ends). First, each one-dimensional interval, say $[a:b)$, is represented by the point (a, b) in two-dimensional space. This two-dimensional space is represented by a priority search tree. Second, we observe that the one-dimensional interval $[a:b)$ intersects the interval $[c:d)$ if and only if $a < d$ and $c < b$. An equivalent observation is that the point (c, d) lies in the range $([-\infty:b), (a:\infty])$. This equivalence means that in order to find all one-dimensional intervals that intersect the interval $[a:b)$, we need only perform the semi-infinite range query $([-\infty:b), (a:\infty])$ in the priority search tree. If the priority search tree contains M one-dimensional intervals, then this operation requires $O(\log_2 M + F)$ time, where F is the number of intersecting intervals found.

In order for the space and time bounds to be comparable with the interval tree, we must also show that a one-dimensional interval can be inserted and deleted from a priority search tree in $O(\log_2 M)$ time. This is achieved by implementing the priority search tree as a "red–black" balanced binary tree [Guibas and Sedgewick 1978]. The red–black balanced binary tree has the property that, for M items, insertions and deletions take $O(\log_2 M)$ time with $O(1)$ rotations [Tarjan 1983]. McCreight [1985] shows that the priority search tree adaptation of the red–black balanced binary tree can be maintained at a cost of $O(\log_2 M)$ per rotation. The use of a red–black balanced binary tree does not affect the $O(M)$ storage requirements of the priority search tree. Hence the desired time and space bounds are achieved.

When the priority search tree is implemented as a red–black balanced binary tree, its node structure differs from the way it was defined earlier in this section. In particular, the internal nodes now also contain intervals. The interval associated with an

internal node, say I, is the one whose left endpoint, say L, is the median value of the left endpoints of the intervals in I's subtrees. All intervals in I's left subtree have left endpoints that are less than L, whereas the intervals in I's right subtree have left endpoints that are greater than L.

Comparing the interval and priority search tree solutions to the rectangle intersection problem, we find that the priority search tree is considerably simpler from a conceptual standpoint than the interval tree. The execution time requirements of the priority search tree are lower when the sort pass is ignored. Also, the priority search tree enables a more dynamic solution than the interval tree because for the priority search tree only the endpoints of the horizontal intervals need to be known in advance. On the other hand, for the interval tree the endpoints of both the horizontal and vertical intervals must be known in advance.

2.4 Applications

Data structures such as the segment tree can be used within the plane-sweep paradigm to solve a number of problems other than rectangle intersection. In fact, the segment tree was originally developed by Bentley [1977] as part of a plane-sweep solution to compute the area (also termed a *measure* problem [Klee 1977]) of a collection of rectangles where overlapping regions are only counted once. It can also be used to compute the perimeter.

The central idea behind the use of the segment tree to compute the area is to keep track of the total length of the parts of the vertical scan line, say L_i at halting point X_i, that overlap the vertical boundaries of rectangles that are active just to the left of X_i. This quantity is adjusted at each halting point. The total area is obtained by accumulating the product of this quantity with the difference between the current halting point and the next halting point—that is, $L_i \cdot (X_i - X_{i-1})$. In order to facilitate this computation, each node of the segment tree contains the length of the overlap of the marked[4] components of its corresponding interval with the vertical scan line.

As an example of the computation of the area, consider the collection of rectangles in Figure 1. When the scan line passes over $x = 7$, the lengths of its overlaps with the nodes corresponding to intervals [6:8), [3:8), [8:14), [3:14), [14:23), [3:23), [23:34), [34:36), [34:38), [23:38), [23:∞), and [3:∞) are 2, 5, 6, 11, 9, 20, 11, 2, 2, 13, 13, and 33, respectively. In addition, for each marked node of the segment tree, we record the number of times, if any, it is marked. In our example, when $x = 7$, the nodes corresponding to the intervals [6:8), [3:8), [8:14), [14:23), [23:34), and [34:36) are marked once; however, at no time in this example will any node be marked more than once. These values are adjusted whenever a vertical boundary of a rectangle is inserted into, or deleted from, the segment tree—that is, at each halting point. For N rectangles, this adjustment process requires $O(\log_2 N)$ steps per halting point or $O(N \cdot \log_2 N)$ for the area of the entire collection. The total space requirements is $O(N)$.

As stated earlier, the unit-segment tree is analogous to a region quadtree in one dimension. This analogy is exploited by van Leeuwen and Wood [1981] in solving measure problems in higher dimensional spaces (i.e., $d > 2$). Lee [1983] also uses the same technique to develop an algorithm that finds the maximum number of rectangles whose intersection is not empty (also termed a maximum clique [Harary 1969]). As an example of this method consider the problem of computing the volume that is occupied by the union of a collection of three-dimensional rectangular parallelepipeds. van Leeuwen and Wood [1981] use a plane-sweep solution that sorts the boundaries of the parallelepipeds along one direction (say x) and then sweeps a plane (instead of a line) parallel to the y–z plane across it. At any instant of time, the plane consists of a collection of cross sections

[4] By *marked* we mean that the node's interval is completely contained in a vertical boundary of one of the rectangles.

(i.e., two-dimensional rectangles). This collection is represented as a region quadtree in a manner analogous to the segment tree.

The region quadtree is built as follows. Assume that there is a maximum of N boundaries in all directions. First, sort the y and z boundaries of the parallelepipeds (removing duplicates) obtaining $y_0, y_1, \ldots, y_p$ $(p < 2N)$, $z_0, z_1, \ldots, z_q$ $(q < 2N)$, and $2^{h-1} \leq \max(p, q) < 2^h$. Assume without loss of generality that the boundaries are distinct. If not, then there are fewer subdivisions. Also, add enough subdivision lines so that there are 2^h subdivisions in each of y and z. Next, form a grid with an origin at the lower left corner such that the two-dimensional interval with a lower left corner at (i, j) corresponds to the rectangular parallelepiped with (y_i, z_j) as its lower left corner. Each two-dimensional interval is marked with the name of the parallelepiped of which it is a part. At this point, only the terminal nodes of the quadtree are marked. The nonterminal nodes are marked as follows. Visit the intervals in an order such as that used in building a region quadtree from a binary array [Samet 1980]. Whenever four brother two-dimensional intervals, say I_i, are in the same parallelepiped, say P, then their father node, say F, gets the label of the parallelepiped and P is no longer associated with any of I_i. Whenever no parallelepipeds are associated with four brother intervals, then they are merged and their corresponding nodes are removed from the quadtree. Building the region quadtree is an $O(N^2)$ process.

As the scan line is swept, each halting point causes the insertion and/or deletion of two-dimensional intervals from the quadtree. Insertion and deletion of a two-dimensional interval is an $O(N)$ process since a rectangle can appear in at most $O(N)$ nodes. Thus, we see that a plane-sweep algorithm employing the region quadtree will execute in time $O(N \cdot \log_2 N + N^2)$ or $O(N^2)$ since there are $O(N)$ halting points. This is an improvement over a solution of Bentley [1977], which recursively performs a plane sweep across each of the planes (i.e., it reduces it to N one-dimensional subproblems) for an execution time of $O(N^2 \cdot \log_2 N)$. Generalizing these results to d dimensions reduces the time requirement of Bentley's solution from $O(N^{d-1} \cdot \log_2 N)$ to $O(N^{d-1})$; however, this increases the space requirement from $O(N)$ to $O(N^2)$. This is achieved by recursively reducing the d-dimensional problem to N problems in $(d - 1)$ dimensions until we obtain the three-dimensional case and then solving each three-dimensional problem as discussed above.

Another related problem is finding the containment set (also known as the inclusion or enclosure set) of a collection of rectangles. The *containment set* is the set of all pairs of rectangles A and B such that A contains B. Variants of the segment tree are used by Vaishnavi and Wood [1980] to solve this problem in $O(N \cdot \log_2^2 N + F)$ time and $O(N \cdot \log_2^2 N)$ space. The space requirement is reduced to $O(N)$ by Lee and Preparata [1982], who map each rectangle into a point in four-dimensional space and solve a point dominance problem.

As described in Section 1, the rectangle intersection problem is closely related to the following problems:

(1) Determining all rectangles that are contained in a given rectangle,
(2) Determining all rectangles that enclose a given rectangle,
(3) Determining all rectangles that partially overlap or are contained in a given rectangle (a window query).

The plane-sweep approach is not appropriate for these problems since, regardless of the data structures employed (e.g., segment, interval, or priority search trees), sorting is a prerequisite and thus any algorithm requires at least $O(N \cdot \log_2 N)$ time for N rectangles. In contrast, the naive solution of intersecting each rectangle with the query rectangle is an $O(N)$ process.

The problems described above can be solved by segment trees and interval trees without making use of a plane sweep [Overmars 1988]. The key is to adapt these representations to store two-dimensional intervals in a manner similar to that in which a two-dimensional range tree is de-

veloped from a one-dimensional range tree [Edelsbrunner 1982]. For example, a segment tree can be adapted to represent rectangles as follows. Project the rectangles on the x axis and store these intervals in a segment tree, say T. Let I be an internal node in T and let R_I denote the rectangles whose horizontal sides are associated with I. For each I build a segment tree for the projections of R_I on the y axis. We can also build an interval tree for the projections of R_I on the y axis.

Using such adaptations of the segment and interval trees, for N rectangles the execution time of the solution to the window query is $O(\log_2^2 N + F)$, where F is the number of rectangles that satisfy the query. The difference is in their storage requirements—the segment tree solution requires $O(N \cdot \log_2^2 N)$ space, whereas the interval tree solution requires $O(N \cdot \log_2 N)$ space. For both of these structures, judicious use of doubly linked lists ensures that rectangles can be inserted and deleted in $O(\log_2^2 N)$ time. Of course, these structures must still be built, which requires $O(N \cdot \log_2^2 N)$ time in both cases. In the case of the segment tree solution, the query time can be reduced further to $O(\log_2 N + F)$ with the same space and preprocessing costs by adding some pointers (termed a layered tree) [Vaishnavi and Wood 1982]. It is not clear how to adapt the priority search tree to store two-dimensional intervals.

The plane-sweep paradigm for solving the geometric problems discussed earlier in this section (e.g., the rectangle intersection problem) assumes that the set of rectangles is only processed once. It can be shown that for many of these problems, plane-sweep methods yield a theoretically optimal solution. A disadvantage is that such solutions assume a static environment.

In contrast, in a dynamic environment where update operations occur frequently (i.e., rectangles are added and deleted), the plane-sweep approach is less attractive. Plane-sweep methods require that the endpoints of all rectangles are known a priori and that the endpoints be sorted before the sweep pass. This is not a major problem, since it is easy to maintain a dynamically changing set of points in sorted order. A more serious problem is that in a dynamic environment, the sweep pass of a plane-sweep algorithm will usually have to be reexecuted, since there is no data structure corresponding to it.[5] In the following sections we discuss methods for a dynamic environment whose worst-case behavior is not as good as that of plane sweep. Interestingly, the data structures that were used in the plane-sweep approach are also applicable in the dynamic environment.

3. POINT-BASED METHODS

In this section we first discuss the representation of rectangles as points and then examine the representation of the collection of points. A common solution to the problem of representing a collection of objects is to approximate elements of the collection by simpler objects. One technique is to represent each object by using one of a number of primitive shapes that contain it. Up to now we have used rectangles, but other shapes such as triangles, circles, cubes, parallelepipeds, cylinders, and spheres are also possible. This approach is motivated, in part, by the fact that it is easier to test the containing objects for intersection than it is to perform the test using the actual objects. For example, it is easier to compute the intersection of two rectangles than of two polygons for which the rectangles serve as approximations. More complex approximations can be created by composing Boolean operations and geometric transformations on instances of the primitive types. In fact, this is the basis of the constructive solid geometry [Requicha 1980; Voelcker and Requicha 1977] (CSG) technique of representing three-dimensional objects.

The advantage of using such approximations is that each primitive can be described by a small set of parameters and can in turn represent a large class of objects. In particular, if primitive P is

[5] Application of newly introduced methods employing persistent search trees due to Sarnak and Tarjan [1986] to the rectangle intersection problem may be useful in avoiding the reexecution of the sweep pass.

described by k parameters, then each set of parameter values defines a point in a k-dimensional space assigned to the class of objects whose members are all the possible instances of primitive P. Such a point is termed a *representative point*. Note that a representative point, and the class to which it belongs, completely define all of the topological and geometric properties of the corresponding object.

Most primitives can be described by more than one set of parameters. For example, using Cartesian coordinates, a circle is described by a representative point in three-dimensional space consisting of the x and y coordinates of its center and the value of its radius. On the other hand, using polar coordinates, a circle can also be described by a representative point in three-dimensional space consisting of the ρ and θ coordinates of its center and the value of its radius. For other primitives, the choices are even more varied. For example, the class of objects formed by a rectangle in two dimensions whose sides are parallel to the x and y coordinate axes is described by a representative point in four-dimensional space. Some choices for the parameters are as follows:

(1) The x and y coordinates of two diagonally opposite corners of the rectangle (e.g., the lower left and upper right).
(2) The x and y coordinates of a corner of the rectangle, together with its horizontal and vertical extents.
(3) The x and y coordinates of the centroid of the rectangle, together with its horizontal and vertical extents (i.e., the horizontal and vertical distances from the centroid to the relevant sides).

The actual choice depends on the type of operations we intend to perform on the objects formed by them.

Different parameters have different effects on the queries, and thus making the right choice is important. Hinrichs and Nievergelt [1983] and Hinrichs [1985a] lump the parameters into two classes—location and extension. Location parameters specify the coordinates of a point such as a corner or a centroid, whereas extension parameters specify size, such as the radius of a circle. This distinction is always possible for objects that can be described as Cartesian products of spheres of varying dimension. Many common objects can be described in this way. For example, a rectangle is the Cartesian product of two one-dimensional spheres, whereas a cylinder is the Cartesian product of a one-dimensional sphere and a two-dimensional sphere. Whenever such a distinction between location and extension parameters can be drawn, the proximity queries that are described in Section 1 have cone-shaped search regions where the tip of the cone is usually in the subspace of the location parameters and has the shape of the query point or query object.

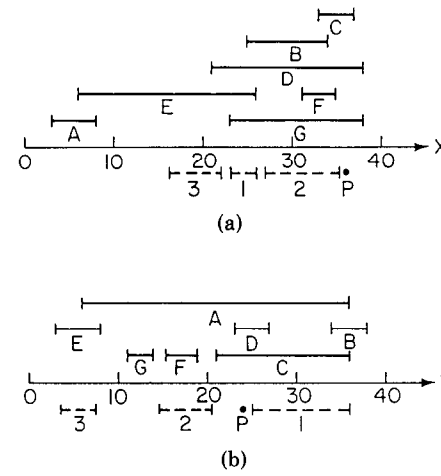


Figure 10. (a) Horizontal (i.e., x) and (b) vertical (i.e., y) intervals corresponding to the sides of the rectangles in Figure 1. Solid lines correspond to rectangles in the collection, and broken lines correspond to the query rectangles.

The importance of making the right choice can be seen by examining the class of one-dimensional intervals on a straight line. As an example, consider the collection of rectangles given in Figure 1. Each rectangle can be represented as the Cartesian product of two one-dimensional spheres corresponding to the sides that are given as horizontal and vertical intervals in Figure 10. These intervals can be represented using any of the three representations enumerated above. Representation (1)

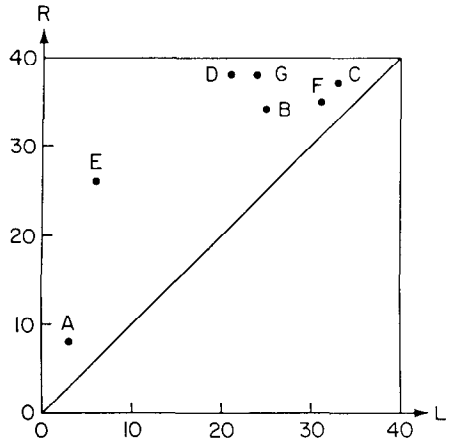


Figure 11. Representation of the horizontal intervals of Figure 1 as ordered pairs (L, R), where L and R are the left and right endpoints, respectively, of the interval.

yields an ordered pair (L, R) where L and R correspond to the left and right endpoints of the interval, respectively. Figure 11 shows how the horizontal intervals would be represented using this method.

In most applications the intervals are small. Therefore, for representation (1), L and R are very close in value. $L < R$ means that the representative points are clustered near and above the diagonal. Thus, the representative points are not well distributed and hence any data structure that is based on organizing the embedding space of the data (e.g., address computation), in contrast to one based on the actual representative points that are stored (e.g., comparative search), will have to pay a price for the empty half of the embedding space. On the other hand, Hinrichs and Nievergelt [1983] point out that separating the location parameters from the extension parameters results in a smaller embedding space, which is filled more uniformly. For example, representation (3) is used in Figure 12, where the horizontal intervals are represented as an ordered pair (CX, DX) such that CX is the centroid of the interval and DX is the distance from the centroid to the end of the interval.

Bearing the above considerations in mind, representation (3) seems to be the most appropriate. In such a case, a rectangle is represented by the 4-tuple (c_x, d_x, c_y, d_y), which is interpreted as the Cartesian product of a horizontal and a vertical one-dimensional interval—that is, (c_x, d_x) and (c_y, d_y), respectively.[6] This representation is used by Hinrichs and Nievergelt [1983] and Hinrichs [1985a], and the following examples of its utility are due to them.

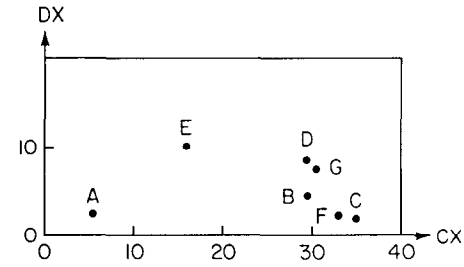


Figure 12. Representation of the horizontal intervals of Figure 1 as ordered pairs (CX, DX), where CX and DX are the centers and half-lengths, respectively, of the interval.

Proximity queries involving point and rectangular query objects are easy to implement. Their answers are conic-shaped regions in the four-dimensional space formed by the Cartesian product of the two interval query regions. This is equivalent to computing the intersection of the two query regions, but is much more efficient. It also enables us to visualize our examples since the horizontal and vertical intervals correspond to the projections of the query responses on the c_x–d_x and c_y–d_y planes, respectively.

We illustrate our discussion with the collection of rectangles given in Figure 1 along with query point P and query rectangles 1, 2, and 3. Note that when the query objects are not individual points or rectangles, the representation of a rectangle as the Cartesian product of two orthogonal intervals is not that useful (e.g., query regions in the form of an arbitrary line or circle).

For a point query, we wish to determine all intervals that contain a given point, say

[6] The notation (c_x, d_x) corresponds to a point in a two-dimensional space. It is *not* the open one-dimensional interval whose left and right endpoints are at c_x and d_x, respectively.

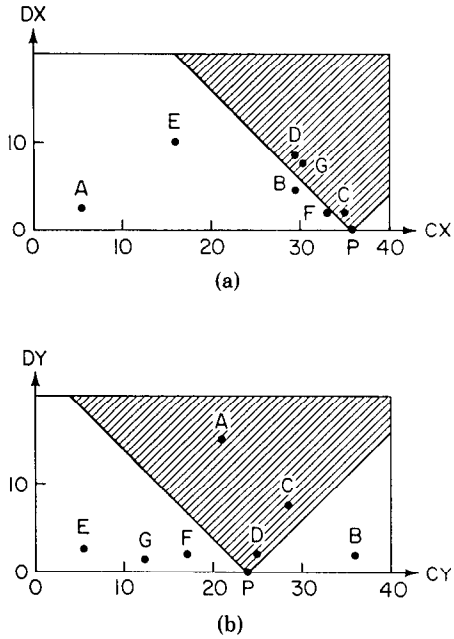


Figure 13. Search region for a point query on P for (a) the horizontal intervals and (b) the vertical intervals of Figure 1. All intervals containing P are in the shaded regions. Intervals appearing in the shaded regions of both (a) and (b) correspond to rectangles that contain P.

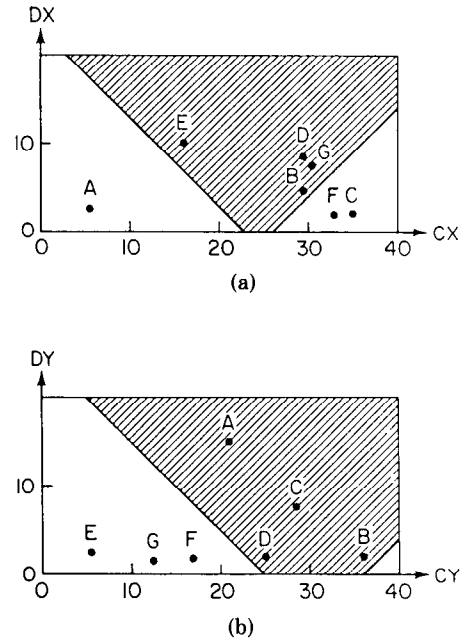


Figure 14. Search region for a window query on query rectangle 1 of Figure 1 for (a) the horizontal intervals and (b) the vertical intervals of that figure. All intervals that contain points of rectangle 1 are in the shaded regions. Intervals appearing in the shaded regions of both (a) and (b) correspond to rectangles that intersect rectangle 1.

p. These intervals form a cone-shaped region whose tip is an interval of length zero centered at p.[7] For example, the horizontal and vertical intervals containing P are shown shaded in Figures 13a and 13b, respectively. To find all the rectangles that contain a given point, we access a specific region in the four-dimensionl space defined by the Cartesian product of the horizontal and vertical point-in-interval query regions. For example, P is in the set of rectangles with representative points in the intersection of the shaded portions of Figures 13a and 13b—that is, {C, D} is the intersection of {C, D, G} and {A, C, D}.

A window query is a bit more complex. In this case, the one-dimensional analog of this query is to find all intervals that overlap a given interval, say I. Again, the set of overlapping intervals consists of a cone-shaped region whose tip is the interval I. For example, the horizontal and vertical intervals that overlap the horizontal and vertical sides of query rectangle 1 are shown shaded in Figures 14a and 14b, respectively. To find all the rectangles that overlap the query window, we access a specific region in the four-dimensional space defined by the Cartesian product of the horizontal and vertical interval-in-interval query regions. For example, query rectangle 1 overlaps the intersection of the shaded portions of Figures 14a and 14b—that is, {B, D} is the intersection of {B, D, E, G} and {A, B, C, D}.

For a containment query, the one-dimensional analog is to find all the intervals that are totally contained within a given interval, say I. The set of contained intervals consists of a cone-shaped region whose tip is at I and that opens in the direction of smaller extent values. This makes sense since all intervals within the cone are to-

[7] McCreight [1980] uses the same technique in conjunction with representation (1) to solve the problem.

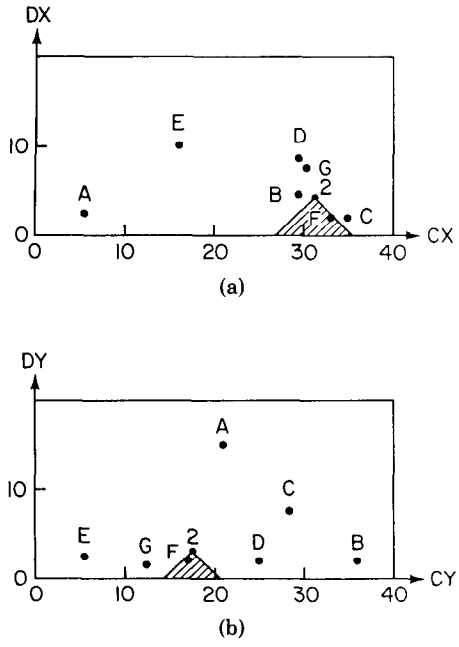


Figure 15. Search regions for a containment query on query rectangle 2 of Figure 1 for (a) the horizontal intervals and (b) the vertical intervals of that figure. All intervals that are contained in one of the intervals forming rectangle 2 are in the shaded regions. Intervals appearing in the shaded regions of both (a) and (b) correspond to rectangles that are contained in rectangle 2.

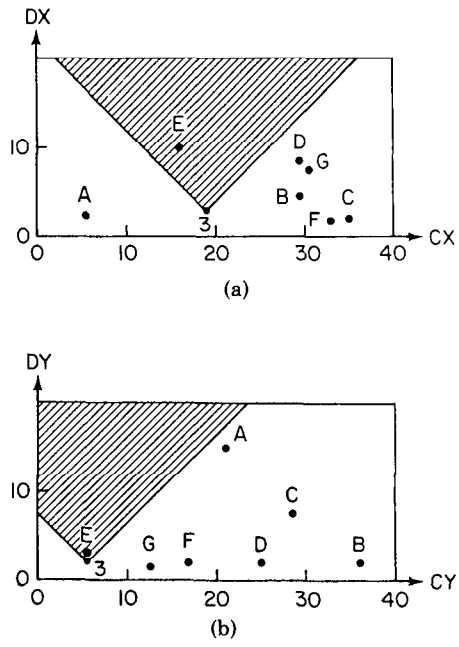


Figure 16. Search regions for an enclosure query on query rectangle 3 of Figure 1 for (a) the horizontal intervals and (b) the vertical intervals of that figure. All figures that enclose one of the intervals forming rectangle 3 are in the shaded regions. Intervals appearing in the shaded regions of both (a) and (b) correspond to the rectangles that enclose rectangle 3.

tally contained in the interval represented by the tip. For example, the horizontal and vertical intervals that are contained in the horizontal and vertical sides of query rectangle 2 are shown shaded in Figures 15a and 15b, respectively. To find all the rectangles that are contained in the query window, we access a specific region in the four-dimensional space defined by the Cartesian product of the horizontal and vertical contained-in-interval query regions. For example, query rectangle 2 contains the intersection of the shaded portions of Figures 15a and 15b—that is, {F}.

For an enclosure query, the one-dimensional analog is to find all the intervals that enclose the given interval, say I. The set of enclosing intervals consists of a cone-shaped region whose tip is at I and that opens in the direction of larger extent values. This is logical since the interval represented by the tip is contained (i.e., enclosed) by all intervals within the cone. For example, the horizontal and vertical intervals that enclose the horizontal and vertical sides of query rectangle 3 are shown shaded in Figures 16a and 16b, respectively. To find all the rectangles that enclose the query window, we access a specific region in the four-dimensional space defined by the Cartesian product of the horizontal and vertical enclose-interval query regions. For example, query rectangle 3 contains the intersection of the shaded portions of Figures 16a and 16b—that is, {E}.

In spite of the relative ease with which the above queries are implemented using the representative point method with representation (3), there are queries for which it is ill suited. For example, suppose we wish to solve the rectangle intersection problem. The fact is, no matter which of the three representations we use, in order to solve this problem we must intersect each rectangle with every other rectangle.

The problem is that none of these representations is area oriented—that is, they reduce a spatial object to a single representative point. Although the extent of the object is reflected in the representative point, the final mapping of the representative point in the four-dimensional space does not result in the preservation of nearness in the two-dimensional space from which the rectangles are drawn. In other words, two rectangles may be very close (and possibly overlap), yet the Euclidean distance between their representative points in four-dimensional space may be quite large, thereby masking the overlapping relationship between them. For example, even though rectangles B and D intersect query rectangle 1, we cannot easily tell if they intersect each other except by checking their sizes.

Our discussion has emphasized representation (3). Nevertheless, as we will see below, the other representations are also commonly used. Interestingly, although a rectangle whose sides are parallel to the x and y axes requires four values to be uniquely specified, it is also frequently modeled by a representative point in a two-dimensional space. The representative point corresponds to the centroid of the rectangle or to one of its corners (e.g., lower left). If rectangles are not permitted to overlap, then such a representation is sufficient to ensure that no two rectangles have the same representative point. Of course, since two values do not uniquely specify the rectangle, the remaining values are retained in the record corresponding to the rectangle that is associated with the representative point. If rectangles are permitted to overlap, then such a representation means that there may be more than one record associated with a specific representative point.

Once a specific representative point method is chosen for the rectangle, we can use any one of a number of techniques for representing multiattribute data to organize the collection of representative points. Again, the choice of representation depends to a large extent on the type of operations that we will be performing. As an example, Lauther [1978] and Rosenberg [1985] make use of a balanced k-d tree to organize the rectangles whose representative point uses representation (1). The k-d tree [Bentley 1975] is a binary search tree where at each level of the tree a different coordinate is tested when determining the direction in which a branch is to be made. Therefore, in the two-dimensional case (i.e., a 2-d tree!), we compare x coordinates at the root and at even levels (assuming the root is at level 0) and y coordinates at odd levels. For example, Figure 17 is the k-d tree corresponding to the data of Figure 2.

The balanced k-d tree is a k-d tree where at each level the number of nodes in the two subtrees is either equal or differ by 1. Lauther [1978] discusses the solution of the rectangle intersection problem using the balanced k-d tree. The solution is an adaptation of the $O(N^2)$ algorithm. It first builds the tree (equivalent to a sorting process) and then traverses the tree in inorder and intersects each rectangle, say P, with the remaining unprocessed rectangles (i.e., the inorder successors of P[8]). Two rectangles with sides parallel to the axes intersect if their projections on the x axis intersect and their projections on the y axis intersect. The one-dimensional analog of this condition has been used in the segment and interval tree solutions to the rectangle intersection problem (see Sections 2.1 and 2.2). More formally, we say that in order for rectangle Q to intersect rectangle P, all four of the following conditions must be satisfied:

(1) $x_{\min}(Q) \leq x_{\max}(P)$
(2) $y_{\min}(Q) \leq y_{\max}(P)$
(3) $x_{\min}(P) \leq x_{\max}(Q)$
(4) $y_{\min}(P) \leq y_{\max}(Q)$

Armed with this formulation of the problem, we see that there is no need to visit all of the inorder successors of P since whenever one of these conditions fails to hold at a node Q, the appropriate subtree of Q need not be searched. These conditions can be restated in the following manner, which is

[8] In this discussion we use P to refer to both a rectangle and its corresponding node in the tree. The correct meaning should be clear from the context.

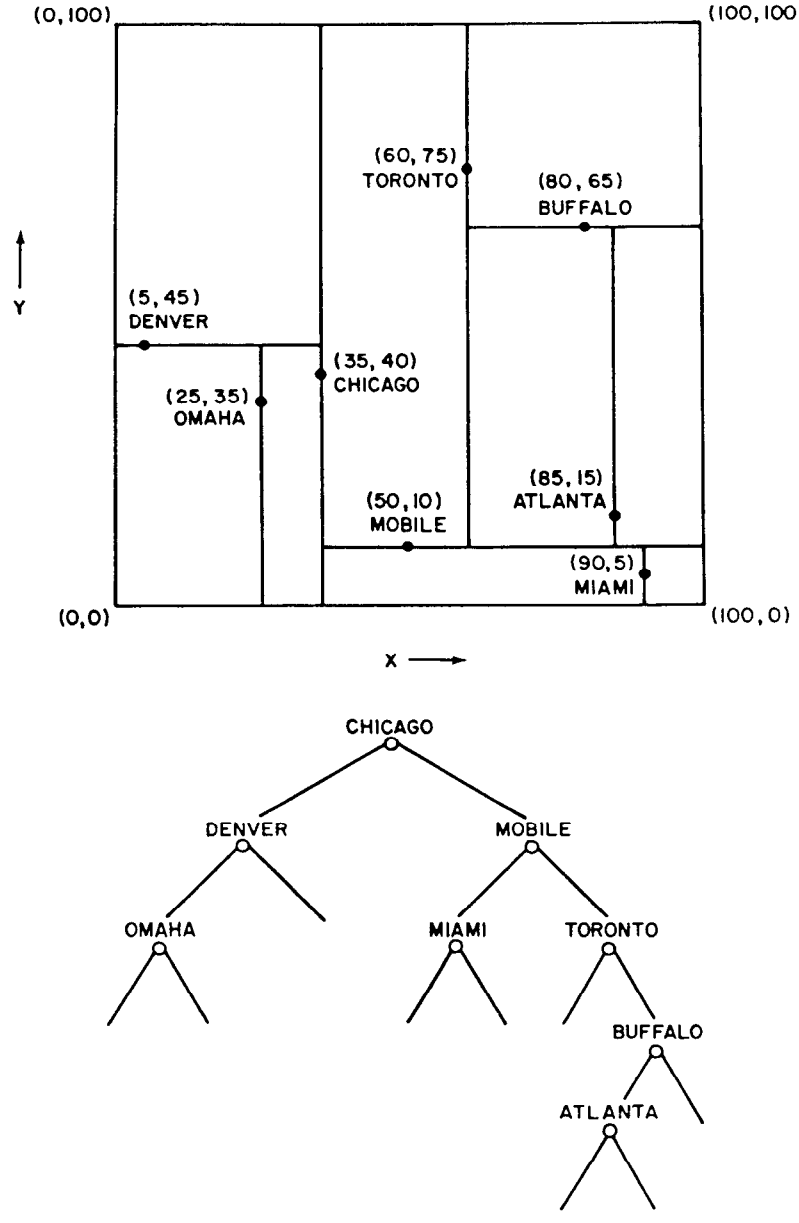


Figure 17. A *k-d* tree corresponding to the points in Figure 2.

more compatible with the way in which the balanced *k-d* tree is traversed:

(5) $x_{\min}(Q) \leq x_{\max}(P)$
(6) $y_{\min}(Q) \leq y_{\max}(P)$
(7) $-x_{\max}(Q) \leq -x_{\min}(P)$
(8) $-y_{\max}(Q) \leq -y_{\min}(P)$

Now, we build a balanced *k-d* tree with discriminators K_0, K_1, K_2, K_3 corresponding to $x_{\min}$, $y_{\min}$, $-x_{\max}$, and $-y_{\max}$, respectively. Whenever we encounter node Q discriminating on K_d such that $K_d(Q) > K_{(d+2) \bmod 4}(P)$, then all the nodes in the right subtree of Q need not be examined further.

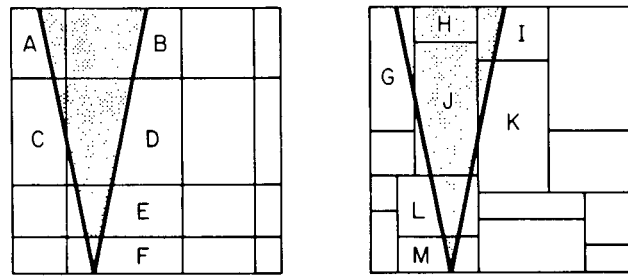


Figure 18. Blocks examined when searching for points within a conic region for a collection of intervals represented by (a) a grid file and (b) a *k-d* tree.

Solving the rectangle intersection problem as described above has an upper bound of $O(N^2)$ and a lower bound of $O(N \cdot \log_2 N)$. The lower bound is achieved when pruning is assumed to occur at every right subtree. When the rectangle intersection problem is posed in terms of conditions (5)–(8), the relation $\leq$ between Q and P is said to be a *dominance relation* [Preparata and Shamos 1985]. In such a case, the intersection problem is called the *dominance merge* problem by Preparata and Shamos [1985]. Given F rectangle intersections, the algorithm of Preparata and Shamos solves the rectangle intersection problem in $O(N \cdot \log_2^2 N + F)$ time instead of the optimal $O(N \cdot \log_2 N + F)$ time.

Building a balanced *k-d* tree takes more time than an ordinary *k-d* tree since medians must be computed in order to assure balance. The balanced *k-d* tree makes point searches and region searches quite efficient. Rosenberg [1985] compares the performance of the *k-d* tree, a point method in his formulation, with linked lists[9] and the area-based quadtree approaches discussed in Section 4.1 below and concludes that the point methods are superior. However, he only takes into account point and window queries. Comparisons using queries such as finding all intersecting rectangles may lead to a different conclusion.

Hinrichs and Nievergelt [1983] and Hinrichs [1985a, 1985b] make use of the grid file to organize the rectangles whose representative point uses representation (3). A grid file yields a partition of space into variable-sized blocks having a finite capacity that are accessed with the aid of a directory. For example, Figure 18a is an example space partition induced by a grid file. The result is that proximity queries are answered by examining all grid blocks that intersect the cone-shaped search regions. They prefer this method to one based on a tree (e.g., the *k-d* tree) because the relevant grid blocks are in contiguous regions, whereas in a tree, contiguous blocks may appear in different subtrees. Hinrichs and Nievergelt are quite concerned with reducing the number of disk access operations necessary to process such queries. For example, Figure 18 shows a conic-shaped search region when the rectangles are organized with a grid file (Figure 18a) and a *k-d* tree (Figure 18b). In the case of a grid file, blocks A, B, C, D, E, and F would be examined, whereas for a *k-d* tree blocks G, H, I, J, K, L, and M would be examined. Note that blocks I and K are in a different subtree of the *k-d* tree than blocks G, H, J, L, and M. In the worst case, solving the rectangle intersection problem when using a grid file takes $O(N^2)$ time. This is achieved by using the naive method of Section 2. The expected cost, however, will be lower since it is assumed that the points corresponding to the rectangles are well distributed among the grid blocks. For an analysis of grid file methods on randomly distributed point data, see Regnier [1985].

[9] The rectangles are stored in a doubly linked list.

The techniques discussed above for organizing the collection of representative points assume that the representative point lies in four-dimensional space. Another possibility is to use a representative point in two-dimensional space. For example, suppose that in our application we must perform a point query (i.e., determine the rectangles that contain a given point). In this case, when the representative point is the centroid, some of the tree representations require that we search the entire data structure. For example, this is the case for the point quadtree (see Figure 4) because the rectangle that is centered at a given point can lie in all of the quadrants. Of course, pruning can occur at deeper levels in the tree. In contrast, using the lower left corner as the representative point may permit the pruning of up to three quadrants in the search. For instance, when the query point lies in the SW quadrant, then no rectangle whose representative point lies in the NW, NE, or SE quadrants can contain the query point.

4. AREA-BASED METHODS

The problem with using trees in conjunction with representative point methods such as those discussed in Section 3 is that the placement of the node in the tree (i.e., its depth) does not reflect the size (i.e., the spatial extent) of the rectangle. It primarily depends on the location of the representative point. In this section we focus on alternative representations provided by area-based methods that associate each rectangle with blocks that contain it or blocks that it contains. The sizes and positions of these blocks may be predetermined as is the case in an approach based on the region quadtree. This need not be the case, however, nor must the blocks be disjoint.

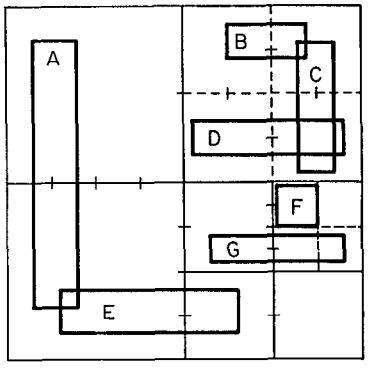
As an example of a representation based on the region quadtree, suppose that we represent each rectangle by its minimum enclosing quadtree block (i.e., a square). The rectangle is associated with the center of the quadtree block. Of course, more than one rectangle can be associated with a given enclosing square and a technique must be used to differentiate among them. Observe that in this case, we do not explicitly store a representative point. Instead, there is a predefined set of representative points with which rectangles can be stored. In some sense this is analogous to hashing [Knuth 1973], where the representative points correspond to buckets. These techniques, which we term *CIF quadtrees*,[10] have been developed independently by Kedem [1982] (called a *quad-CIF tree*) and by Abel and Smith [1983].

In this section we expand further on the area-based approaches. We first present a detailed implementation of one variant of the CIF quadtree, which is related to the MX quadtree representation of point data [Samet 1984]. Next, we describe some quadtree-based alternatives that permit a rectangle to be associated with more than one quadtree block. We conclude with a discussion of the R-tree and some of its variants. It is a hierarchy of rectangular regions that contain the data rectangles. The hierarchy is constructed by rules similar to those used to define a B-tree. The regions need not be disjoint. Analyzing the space requirements of these representations as well as the execution time of algorithms that use them is quite difficult, since it depends heavily on the distribution of the data. In most cases, a limited part of the tree must be traversed and thus the execution time depends, in part, on the depth and the shape of the tree.

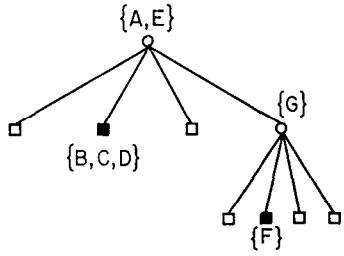
4.1 MX-CIF Quadtrees

The *MX-CIF quadtree* associates each rectangle, say R, with the quadtree node corresponding to the smallest block that contains R in its entirety. Rectangles can be associated with both terminal and nonterminal nodes. Subdivision ceases whenever a node's block contains no rectangles. Alternatively, subdivision can also cease once a quadtree block is smaller than a predetermined threshold size. This threshold is often chosen to be equal to the expected size of the rectangle [Kedem 1982]. Figure 19 is the MX-CIF quadtree for the set of rectangles in Figure 1. Once a rectangle is associated with a quadtree node, say P, it is not considered to be a member of any

[10] CIF denotes Caltech Intermediate Form.



(a)



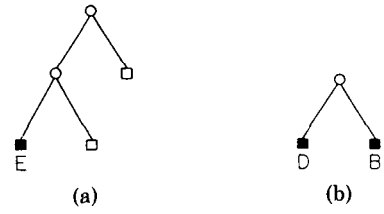
(b)

Figure 19. The MX-CIF quadtree (b) and the block decomposition induced by it (a) for the rectangles in Figure 1.

of the sons of P. For example, in Figure 19, rectangle G overlaps the space spanned by both the SE son of the root and the NE son of the SE son of the root; yet G is only associated with the SE son of the root.

It should be clear that more than one rectangle can be associated with a given enclosing block (i.e., node). There are several ways of organizing these rectangles. The simplest solution is to maintain a linked list of these rectangles. Another approach, due to Kedem [1982], is described below.

Let P be a quadtree node and let S be the set of rectangles that are associated with P. Members of S are organized into two sets according to their intersection (or colinearity of their sides) with the lines passing through the centroid of P's block. We shall use the term *axes* or *axis lines* to refer to these lines. For example, consider node P centered at (CX, CY). All members of S that intersect the line $x = CX$ form



(a) (b)

Figure 20. Binary trees for the y axes passing through (a) the root of the MX-CIF quadtree in Figure 19 and (b) the NE son of the root of the MX-CIF quadtree in Figure 19.

one set, and all members of S that intersect the line $y = CY$ form the other set. Equivalently, these sets correspond to the rectangles intersecting the y and x axes, respectively, passing through (CX, CY). If a rectangle intersects both axes (i.e., it contains the centroid of P's block), then we adopt the convention that it is stored with the set associated with the y axis. These subsets are implemented as binary trees (really tries), which in actuality are one-dimensional analogs of the MX-CIF quadtree. For example, Figure 20 illustrates the binary tree associated with the y axes passing through the root and the NE son of the root of the MX-CIF quadtree of Figure 19. The subdivision points of the axis lines are shown by tick works in Figure 19.

At this point, the following observations can be made. The MX-CIF quadtree is related to the region quadtree in the same way as the interval tree is related to the segment tree. The MX-CIF quadtree is the two-dimensional analog of the tile tree[11] (without the tertiary structure). The tile tree and the one-dimensional MX-CIF quadtree are identical. In particular, when rectangles are not permitted to overlap, the secondary structures of the tile tree consist of at most one rectangle. When the tile tree is used in this context, it is not a complete binary tree. Alternatively, it is not necessarily balanced since the subdivision points are fixed by virtue of regular decomposition rather than being determined by the endpoints of the domain of rectangles as in the definition of the interval tree.

[11] The analogy is with the tile tree instead of the interval tree because the MX-CIF quadtree is based on a regular decomposition.

A rectangle is inserted into an MX–CIF quadtree by a top-down search for the position that it is to occupy. This position is determined by a two-step process. First, the first subdivision point must be located such that at least one of its axis lines (i.e., the quadrant lines emanating from the subdivision point) intersects the input rectangle. Second, having found such a point and an axis, say point P and axis V, the subdivision process is repeated for the V axis until the first subdivision point that is contained within the rectangle is located. During the process of locating the destination position for the input rectangle, the space spanned by the MX–CIF quadtree may have to be repeatedly subdivided (termed *splitting*) creating new nodes in the process. In the worst case, each rectangle is at the maximum depth of the tree, say n.[12] Thus, the worst-case cost of building an MX–CIF quadtree for N rectangles is $O(n \cdot N)$ in space and time. Of course, the expected behavior should be better. It should be clear that the shape of the resulting MX–CIF quadtree is independent of the order in which the rectangles are inserted into it. Deletion of nodes is more complex and may require collapsing of nodes—that is, the direct counterpart of the node splitting process outlined above.

The most common operations are determining whether a given rectangle overlaps (i.e., intersects) any of the existing rectangles or performing a window query. Another popular query seeks to determine whether one collection of rectangles can be overlaid on another collection without any of the component rectangles intersecting one another. These two operations can be implemented by using variants of algorithms developed for handling set operations (i.e., union and intersection) in region-based quadtrees [Hunter and Steiglitz 1979; Shneier 1981]. The window query is answered by intersecting the query window with the MX–CIF quadtree. The overlay query is answered by a two-step process. The two MX–CIF quadtrees are first intersected. If the result is empty, then they can be safely overlaid, and all that is needed is to perform a union of the two MX–CIF quadtrees. Boolean queries can also be easily handled.

When the rectangles are allowed to intersect, reporting the pairs of rectangles that intersect each other is achieved by traversing the MX–CIF quadtree and for each node examining all neighboring nodes that can contain rectangles that intersect it. In the worst case, for some rectangles we may have to examine the entire MX–CIF quadtree. If this is the case, however, the remaining rectangles will not require this much work. Nevertheless, the worst-case execution time of this task is $O(n \cdot N^2)$ for a tree of maximum depth n with N rectangles. The expected behavior should be better.

Abel and Smith [1983] also represent a collection of rectangles by an MX–CIF quadtree. The difference between their approach and that of Kedem [1982] is that they do not use binary trees, or any other data structure, to organize the rectangles that are associated with each quadtree node separately. They represent each rectangle by its locational code. The locational code represents a sequence of 2-bit directional codes that locate the quadtree node along a path from the root of the quadtree. The code also reflects the level at which the node is found. These codes are subsequently organized in a B^+-tree [Comer 1979]. Note that many rectangles have identical locational codes, and thus the rectangle dimensions must also be stored along with the locational codes in the B^+-tree. For example, for the set of rectangles in Figure 19, rectangles B, C, and D have the same locational code, and so do rectangles A and E.

4.2 Multiple Quadtree Block Representations

One of the problems with the MX–CIF quadtree and other representations that associate each rectangle with the smallest enclosing quadtree block is that determining how many rectangles intersect a window (e.g., in the form of a rectangle) may be quite costly. The problem is that the quadtree nodes that intersect the query rectangle may contain many rectangles that

[12] n is the sum of the maximum depths of the MX–CIF quadtree and of the binary tree.

do not intersect the query rectangle, yet each one of them must be individually compared with the query rectangle to determine the existence of a possible intersection.

For example, consider the MX-CIF quadtree in Figure 19, which corresponds to the collection of rectangles given in Figure 1. Although query rectangle 1 (see Figure 1) is in the NE quadrant of the root of the MX-CIF quadtree, we still have to check some of the rectangles that are stored at the root of the entire quadtree since these rectangles could conceivably overlap the query rectangle. This work could be avoided by using a more compact (in an area sense) representation of each rectangle. Such a representation would use more, and smaller, quadtree blocks to represent each rectangle, but the total area of the blocks would be considerably less than that of the smallest enclosing quadtree block. The result is that more rectangles would be eliminated from consideration due to the pruning that occurs during the search of the quadtree. A number of alternatives are available to achieve this effect. They are examined briefly below.

One possibility is to use a region quadtree representation for each rectangle. Such a representation would lead to many nodes since its underlying decomposition rule requires that the block corresponding to each node be homogeneous (i.e., that it be totally contained within one of the rectangles or not be in any of the rectangles). Permitting rectangles to overlap forces a modification of the decomposition rule. In particular, it implies that decomposition ceases when a block is totally contained within one or more rectangles. If a block is contained in more than one rectangle, however, it must be totally contained in all of them.

Abel and Smith [1985] present a less radical alternative. They propose that instead of using the minimum enclosing quadtree block, each rectangle is represented by a collection of enclosing quadtree blocks. They suggest that the collection contain a maximum of four blocks, although other amounts are also possible. The four blocks are obtained by determining the minimum enclosing quadtree block, say B, for each rectangle, say R, and then splitting B once to obtain quadtree blocks

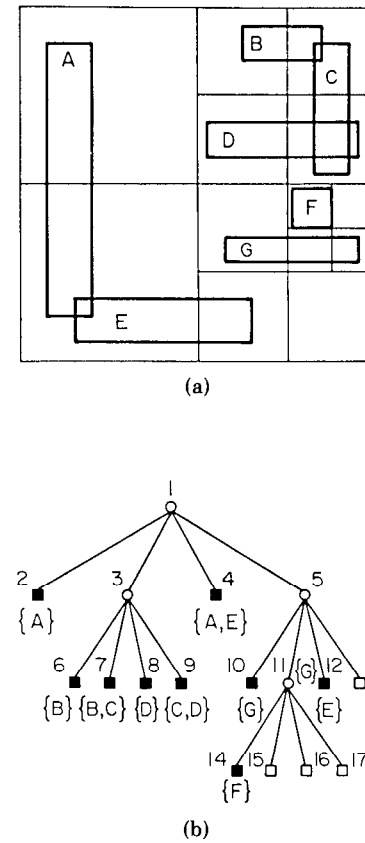


Figure 21. The expanded MX-CIF quadtree (b) and the block decomposition induced by it (a) for the rectangles in Figure 1.

B_i ($i \in \{NW, NE, SW, SE\}$) such that R_i is the portion of R, if any, that is contained in B_i. Next, for each B_i we find the minimum enclosing quadtree block, say D_i, that contains R_i. Now, each rectangle is represented by the set of blocks consisting of D_i. We term such a representation an *expanded MX-CIF quadtree.*

As an example of the expanded MX-CIF quadtree, consider Figure 21, which corresponds to the collection of rectangles of Figure 1. Several items are worthy of note. First, each node appears at least one level lower in the expanded MX-CIF quadtree than it did in the MX-CIF quadtree. Second, some of the D_i may be empty (e.g., rectangle A in Figure 21 is covered by blocks 2 and 4; rectangle F in Figure 21 is covered by block 14). Third, the covering

blocks are not necessarily of equal size (e.g., rectangle E in Figure 21 is covered by blocks 4 and 12). It should be clear that the area covered by the collection of blocks D_i is not greater than that of B.

The worst-case execution time for building the expanded MX-CIF quadtree and the space requirements are the same as for the MX-CIF quadtree—that is, $O(n \cdot N)$ for a tree of maximum depth n with N rectangles. The worst-case execution time of the rectangle intersection problem is also the same as that for the MX-CIF quadtree—that is, $O(n \cdot N^2)$. Abel and Smith suggest that the search process can be made more efficient by applying the splitting process again to the blocks D_i. Of course, the more times that we split, the closer we get to the region quadtree representation of the rectangles. Also, this increases the space requirement and the insertion and deletion costs.

Shaffer [1986] presents a pair of data structures termed an *RR quadtree* that is somewhat related to the expanded MX-CIF quadtree. Two variants are given. The first, called an *RR_1 quadtree* makes use of a decomposition rule that splits until each node contains either just one rectangle or all of the rectangles in the node intersect each other. Thus, all rectangles are associated with terminal nodes. When rectangles are not permitted to overlap, this decomposition rule means that no block can contain a part of more than one rectangle. For example, consider Figure 22, which is the RR_1 quadtree corresponding to the collection of rectangles of Figure 1. Note that node 3 had to be decomposed further since rectangles B, C, and D do not mutually intersect each other.

The storage requirements of the RR_1 quadtree are much higher than those of the MX-CIF and expanded MX-CIF quadtrees. This is due to the need to decompose the collection when many rectangles are near each other without mutually intersecting each other—for example, a chain formed by intersecting rectangles. This problem is partially resolved by loosening the decomposition criterion of the RR_1 quadtree to permit a node, say N, to contain a pair of rectangles if they intersect or are

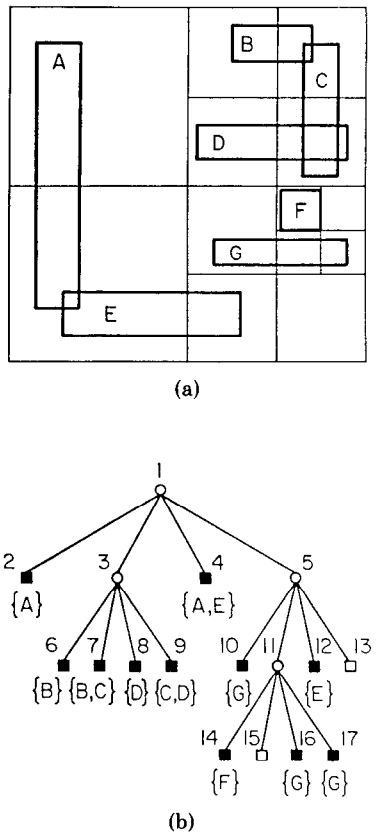


Figure 22. The RR_1 quadtree (b) and the block decomposition induced by it (a) for the rectangles in Figure 1.

a part of a chain of connected rectangles so that each rectangle in the chain is also in the node. The resulting structure is called an *RR_2 quadtree*. For example, consider Figure 23, which is the RR_2 quadtree corresponding to the collection of rectangles of Figure 1. Note that now, unlike the RR_1 quadtree, node 3 need not be further decomposed to deal with rectangles B, C, and D. These three rectangles form a chain of intersecting rectangles but they do not mutually intersect each other.

The RR_2 quadtree still requires considerably more storage than the MX-CIF and expanded MX-CIF quadtrees. The advantage of the RR quadtree family, however, is that if two rectangles intersect, then they must be stored in the same node. This makes window queries quite efficient since

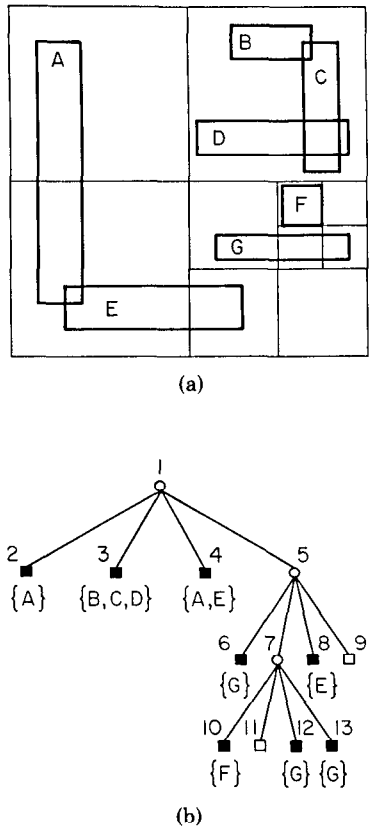


Figure 23. The RR_2 quadtree (b) and the block decomposition induced by it (a) for the rectangles in Figure 1.

fewer rectangles must be examined for intersection. In particular, the number of rectangle comparisons required by a window query in an RR_2 quadtree is equal to the number of comparisons that would be made were the query rectangle being inserted into the tree.

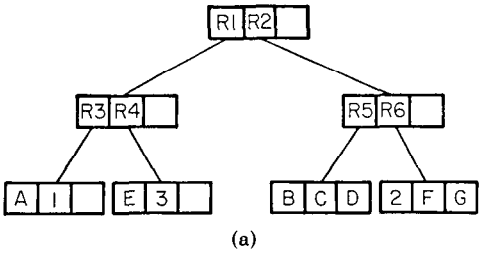
The space requirements of the region quadtree and the RR quadtree family are dependent on the amount of space that is required to store an individual rectangle. In all three cases, for a tree of maximum depth n, a rectangle requires $O(2^n)$ space. For N rectangles, the time required to build the region quadtree is $O(N \cdot 2^n)$, whereas for the RR quadtree family it may be as high as $O(N^2 \cdot 2^n)$ since each rectangle must be checked against the rectangles in each node in which it is contained—there are $O(2^n)$ such nodes and each can contain N rectangles. Solving the rectangle intersection problem is quite easy since it is done by traversing the tree and reporting all nodes that contain more than one rectangle. The time required is the same as the space requirement. It can be shown that each intersection is only reported once.

4.3 R-Trees

The R-tree of Guttman [1984] is a hierarchical data structure that is derived from the B-tree. Each node in the tree corresponds to the smallest d-dimensional rectangle that encloses its son nodes. The leaf nodes contain pointers to the actual geometric objects in the database, instead of sons. The objects are represented by the smallest aligned rectangle in which they are contained. Often the nodes correspond to disk pages, and thus the parameters defining the tree are chosen so that a small number of nodes is visited during a spatial query. Note that rectangles corresponding to different nodes may overlap. Also, a rectangle may be spatially contained in several nodes, yet it can only be associated with one node. This means that a spatial query may often require several nodes to be visited before ascertaining the presence or absence of a particular rectangle. Our discussion is limited to the representation of rectangles in two dimensions.

The basic rules for the formation of an R-tree are very similar to those for a B-tree. All leaf nodes appear at the same level. Each entry in a leaf node is a 2-tuple of the form (R, O) such that R is the smallest rectangle that spatially contains data object O. Each entry in a nonleaf node is a 2-tuple of the form (R, P) such that R is the smallest rectangle that spatially contains the rectangles in the child node pointed at by P. An R-tree of order (m, M) means that node in the tree, with the exception of the root, contains between $m \leq \lceil M/2 \rceil$ and M entries. The root node has at least two entries unless it is a leaf node.

For example, consider the collection of rectangles given in Figure 1, and treat the query rectangles (i.e., 1, 2, and 3) as elements of the collection so that there is a



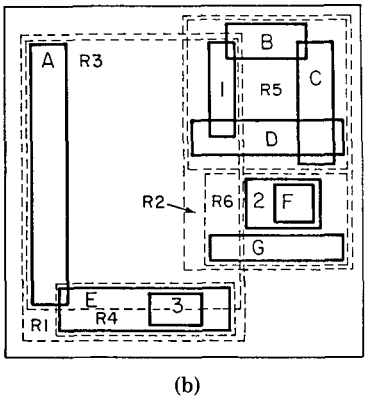


Figure 24. (a) R-tree for the collection of rectangles in Figure 1 and (b) the spatial extents of the enclosing rectangles.

total of 10 rectangles. Let $M = 3$ and $m = 2$. One possible R-tree for this collection is given in Figure 24a. Figure 24b shows the spatial extent of the rectangles of the nodes in Figure 24a, with broken lines denoting the rectangles corresponding to the subtrees rooted at the nonterminal nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual rectangles were inserted into (and possibly deleted from) the tree.

The algorithm for inserting an object (i.e., a record corresponding to its enclosing rectangle) in an R-tree is analogous to that used for B-trees. New rectangles are added to leaf nodes. The appropriate leaf node is determined by traversing the R-tree starting at its root and at each step choosing the subtree whose corresponding rectangle would have to be enlarged the least. Once the leaf node has been determined, then check if insertion of the rectangle will cause the node to overflow. If yes, then it must be split and the $M + 1$ records must be

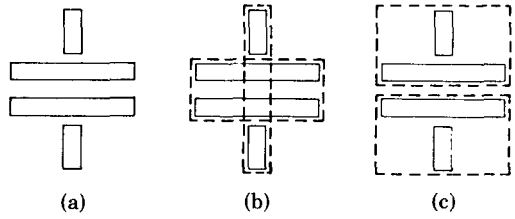


Figure 25. (a) Four rectangles and the splits that would be induced (b) by minimizing the total area of the covering rectangles and (c) by minimizing the area common to the covering rectangles of both nodes.

distributed in the two nodes. Splits are propagated up the tree.

There are many possible ways to split a node. One possible goal is to distribute the records among the nodes so that the likelihood that the nodes will be visited in subsequent searches will be reduced. This is accomplished by minimizing the total areas of the covering rectangles for the nodes (i.e., coverage). An alternative goal is to reduce the likelihood that both nodes are examined in subsequent searches. This is accomplished by minimizing the area common to both nodes (i.e., overlap). Of course, at times these goals may be contradictory. For example, consider the four rectangles in Figure 25a. The first goal is satisfied by the split in Figure 25b, whereas the second goal is satisfied by the split in Figure 25c.

Deletion of a rectangle, say R, from an R-tree proceeds by locating the leaf node, say L, containing R and removing R from L. Next, adjust the covering rectangles on the path from L to the root of the tree while removing all nodes in which underflow occurs and adding them to the set U. Once the root node is reached, if it has just one son, then the son becomes the new root. The nodes in which underflow occurred (i.e., members of U) are reinserted at the root. Elements of U that correspond to leaf nodes result in the placement of their constituent rectangles in the leaf nodes, whereas other nodes are placed at a level so that their leaf nodes are at the same level as those of the whole tree.

The deletion algorithm for an R-tree differs from that for a B-tree in that in the case of underflow, nodes are reinserted in-

stead of being merged with adjacent nodes. The difficulty is that there is no concept of adjacency in an R-tree, although we could merge with the sibling whose area will be increased the least or even just distribute the elements of the underlowing node among its siblings. Nevertheless, reinsertion is used by Guttman [1984] because of a feeling that it enables the tree to reflect the changing spatial structure of the data dynamically rather than the gradual degradation that could arise when a rectangle maintains the same parent throughout its lifetime!

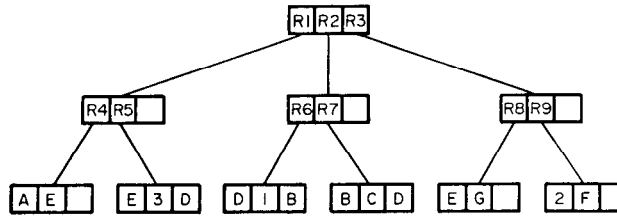
Searching for points or regions in an R-tree is straightforward. The only problem is that a large number of nodes may have to be examined since a rectangle may be contained in the covering rectangles of many nodes while its corresponding record is only contained in one of the leaf nodes (e.g., in Figure 24, rectangle 1 is contained in its entirety in R1, R2, R3, and R5). For example, suppose we wish to determine the identity of the rectangle element in the collection of rectangles given in Figure 1, which contains point Q at coordinates (21, 24). Since Q can be in either of R1 and R2, we must search both of their subtrees. Searching R1 first, we find that Q could only be contained in R3. Searching R3 does not lead to the rectangle that contains Q even though Q is in a portion of rectangle D that is in R3. Thus, we must search R2, and we find that Q can only be contained in R5. Searching R5 results in locating D, which is the desired rectangle.

The insertion algorithm, and the accompanying node splitting techniques, described above are based on a dynamic database. If the database can be expected to be static and all of the objects are known a priori, then a different technique can be used to build an R-tree. Roussopoulos and Leifker [1985] propose the use of a *packed R-tree*, which is an R-tree that is built by successively applying a nearest-neighbor relation to group objects in a node after the set of objects has been sorted according to a spatial criterion. Once an entire level of the tree is built, the algorithm is reapplied to add nodes at the next higher level, terminating when a level contains just one node. This is a static method that results in each node being filled to capacity.

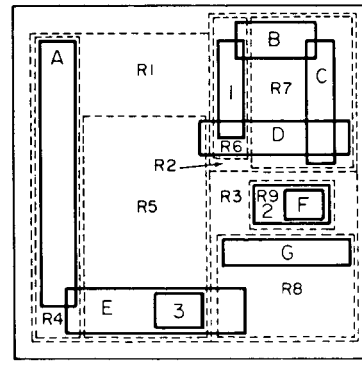
Although the packed R-tree does not necessarily result in a minimum coverage and/or overlap, empirical tests [Roussopoulos and Leifker 1985] of its performance on point searches in a database of two-dimensional points show its use to lead to significant improvements vis à vis an R-tree that was built using the conventional R-tree insertion routine. In these tests each node was constructed by selecting an object from a spatially sorted list and then adding its $M-1$ nearest neighbors. Once all the nodes at a given level have been constructed, the process is applied recursively, forming nodes at successively higher levels in the tree until just one node remains. A better approach, although far costlier from a combinatorial standpoint, is to choose the M objects simultaneously so that the area of the resulting covering rectangle is minimized.

Another alternative to the R-tree is the R^+-tree [Sellis et al. 1987; Stonebraker et al. 1986], which is an extension of the *k-d*-B-tree [Robinson 1981] to deal with rectangles. The motivation for the R^+-tree is to avoid overlap among the bounding rectangles. In particular, all bounding rectangles (i.e., at levels other than the leaf) are nonoverlapping. Thus, each rectangle is associated with all the bounding rectangles that it intersects. The result is that there may be several paths starting at the root to the same rectangle. This will lead to an increase in the height of the tree. Retrieval time, however, is sped up. The *cell tree* of Gunther [1987] is similar to the R^+-tree, with the principal difference being that the nonleaf nodes of the cell tree are convex polyhedra instead of bounding rectangles.

Figure 26 is an example of one possible R^+-tree for the collection of rectangles in Figure 1. Once again, the query rectangles (i.e., 1, 2, and 3) are treated as elements of the collection so that there is a total of 10 rectangles. This particular tree is of order (2, 3), although in general it is not possible to always guarantee that all nodes will have a minimum of two entries. Notice that rectangles D and E appear in three different nodes, whereas rectangle B appears in two



(a)



(b)

Figure 26. (a) R^+-tree for the collection of rectangles in Figure 1 and (b) the spatial extents of the enclosing rectangles.

different nodes. Of course, other variants are possible since the R^+-tree is not unique.

It is interesting to note that the decomposition into blocks induced by the R^+-tree is similar to the way a region quadtree would be used to represent a collection of rectangles (see Section 4.2). Since the R^+-tree is an extension of the *k-d*-B-tree, it has a drawback that B-tree performance guarantees are no longer valid. For example, pages are not guaranteed to be 50% full without very complicated record insertion and deletion procedures. Nevertheless, empirical tests by Faloutsos et al. [1987] reveal reasonable behavior in comparison to the conventional R-tree. These tests were coupled with a limited analysis of the behavior of the two data structures when used to represent one-dimensional intervals of equal length by transforming them to points in two dimensions using representation (2) of Section 3. Sellis et al. [1987] suggest that performance of the R^+-tree can be improved by a judicious choice of partition lines, as well as by a careful initial grouping of the rectangles at the leaf level. The latter can be achieved by applying heuristics similar to those used to build a packed R-tree.

Assume that the R-tree and the R^+-tree are constructed in a batch manner—that is, all the rectangles are known before the construction and hence are arbitrarily grouped together. For N rectangles, the construction time and space requirements of these two data structures are both $O(N)$ and $O(N^2)$, respectively. The reason for the higher costs for the R^+-tree is that a rectangle may appear in N nodes because of its intersection with N other rectangles. This analysis assumes that no optimization is performed when a node overflows. In both cases, the worst-case execution time of the rectangle intersection problem is $O(N^2)$.

5. CONCLUDING REMARKS

In this tutorial we gave an overview of a number of different hierarchical represen-

tations for collections of rectangles. The choice of a representation is not clear-cut. It depends on the problem domain and the tasks to be performed. Although our coverage has been limited, we have seen that the various methods are quite similar. In particular, we have shown a close relationship between quadtree like area-based representations and the representations used to support solutions based on the plane-sweep paradigm.

In retrospect, this relationship is not surprising. It is based on the observation that algorithms that make use of traversals of a quadtree are in actuality performing a two-dimensional plane sweep. Furthermore, a quadtree decomposition is really a multi-dimensional sort. In contrast, solutions based on representations such as the interval tree perform a sort in one dimension and then a sweep in the other direction. Of course, a similar solution could also be devised in a multidimensional environment. In such a case, the concept of an active border [Samet and Tamminen 1985, 1988] used in the multidimensional plane sweep embodied by a quadtree traversal is the analog of the scan line in the one-dimensional plane-sweep. At any instant in a traversal of the quadtree, the *active border* represents the boundary between the nodes that have been processed and those that have not. The similarity is completed by devising data structures to represent the active border that have properties analogous to those of the segment and interval trees. It would be interesting to explore this relationship further.

ACKNOWLEDGMENTS

I have benefitted greatly from many discussions with Mike Dillencourt as this paper was written. I would also like to think Timos Sellis for his comments on R-trees, and Azriel Rosenfeld and an anonymous referee for a thorough review of the paper. The support of the National Science Foundation under Grant IRI-88-02457 is gratefully acknowledged.

REFERENCES

ABEL, D. J. 1985. Some elemental operations on linear quadtrees for geographic information systems. *Comput. J. 28*, 1 (February), 73–77.

ABEL, D. J., AND SMITH, J. L. 1983. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Comput. Vision Graph. Image Process. 24*, 1 (Oct.), 1–13.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept.), 509–517.

BENTLEY, J. L. 1977. Algorithms for Klee's rectangle problems. Computer Science Department, Carnegie-Mellon University, Pittsburgh.

BENTLEY, J. L. 1979. Decomposable searching problems. *Inf. Process. Lett. 8*, 5 (June), 244–251.

BENTLEY, J. L., AND MAURER, H. A. 1980. Efficient worst-case data structures for range searching. *Acta Inf. 13*, 2, 155–168.

BENTLEY, J. L., AND OTTMANN, T. A. 1979. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput. 28*, 9 (Sept.), 643–647.

BENTLEY, J. L., AND WOOD, D. 1980. An optimal worst-case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput. 29*, 7 (July), 571–577.

BUCHER, W., AND EDELSBRUNNER, H. 1983. On expected and worst-case segment trees. In *Advances in Computing Research*, vol. 1, *Computational Geometry*, F. P. Preparata, Ed. JAI Press, Greenwich, Conn., pp. 109–125.

CHAZELLE, B., AND GUIBAS, L. J. 1986a. Fractional cascading: I. A data structuring technique. *Algorithmica 1*, 2, 133–162.

CHAZELLE, B., AND GUIBAS, L. J. 1986b. Fractional cascading: II. Applications. *Algorithmica 1*, 2, 163–191.

COMER, D. 1979. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June), 121–137.

EDELSBRUNNER, H. 1980a. Dynamic rectangle intersection searching. Institute for Information Processing Rept. 47, Technical University of Graz, Graz, Austria.

EDELSBRUNNER, H. 1980b. Dynamic data structures for orthogonal intersection queries. Institute for Information Processing. Rept. 59, Technical University of Graz, Graz, Austria.

EDELSBRUNNER, H. 1982. Intersection problems in computational geometry. Institute for Information Processing Rept. 93, Technical Univ. of Graz, Graz, Austria.

EDELSBRUNNER, H. 1983a. A new approach to rectangle intersections: Part I. *Int. J. Comp. Math. 13*, 3–4, 209–219.

EDELSBRUNNER, H. 1983b. A new approach to rectangle intersections: Part II. *Int. J. Comput. Math. 13*, 3–4, 221–229.

EDELSBRUNNER, H., GUIBAS, L. J., AND STOLFI, J. 1986. Optimal point location in a monotone subdivision. *SIAM J. Comput. 15*, 2 (May), 317–340.

FALOUTSOS, C., SELLIS, T., AND ROUSSOPOULOS, N. 1987. Analysis of object oriented spatial access

methods. In *Proceedings of the SIGMOD Conference* (San Francisco, May). ACM, New York, pp. 426–439.

FINKEL, R. A., AND BENTLEY, J. L. 1974. Quad trees: A data structure for retrieval on composite keys. *Acta Inf. 4*, 1, 1–9.

FREDKIN, E. 1960. Trie memory. *Commun. ACM 3*, 9 (Sept.) 490–499.

GUIBAS, L. J., AND SEDGEWICK, R. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science* (Ann Arbor, Mich., Oct.). IEEE, New York, pp. 8–21.

GUNTHER, O. 1987. Efficient structures for geometric data management. Ph.D. dissertation, UCB/ERL M87/77, Electronics Research Laboratory, College of Engineering, Univ. of California at Berkeley, Berkeley, Calif.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference* (Boston, June). ACM, New York, pp. 47–57.

HARARY, F. 1969. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.

HINRICHS, K. 1985a. The grid file system: Implementation and case studies of applications. Ph.D. dissertation, Institut fur Informatik, ETH, Zurich, Switzerland.

HINRICHS, K. 1985b. Implementation of the grid file: Design concepts and experience. *BIT 25*, 4, 569–592.

HINRICHS, K., AND NIEVERGELT, J. 1983. The grid file: A data structure designed to support proximity queries on spatial objects. In *Proceedings of the WG'83 (International Workshop on Graph-theoretic Concepts in Computer Science)*, M. Nagl and J. Perl, Eds. (Trauner Verlag, Linz, Austria), pp. 100–113.

HUNTER, G. M. 1978. Efficient computation and data structures for graphics. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, N.J.

HUNTER, G. M., AND STEIGLITZ, K. 1979. Operations on images using quad trees. *IEEE Trans. Pattern Anal. Mach. Intell. 1*, 2 (Apr.), 145–153.

KEDEM, G. 1982. The quad-CIF tree: A data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference* (Las Vegas, June), pp. 352–357.

KLEE, V. 1977. Can the measure of $\cup [a_i, b_i]$ be computed in less than $O(n \log n)$ steps? *Am. Math. Monthly 84*, 4 (Apr.), 284–285.

KLINGER, A. 1971. Patterns and search statistics. In *Optimizing Methods in Statistics*, J. S. Rustagi, Ed. Academic Press, Orlando, Fla., pp. 303–337.

KNUTH, D. E. 1973. *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*. Addison-Wesley, Reading, Mass.

LAUTHER, U. 1978. Four-dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits. *J. Design Automat. Fault-Tolerant Comput. 2*, 3 (July), 241–247.

LEE, D. T. 1983. Maximum clique problem of rectangle graphs. In *Advances in Computing Research*. Vol. 1, *Computational Geometry*. F. P. Preparata, Ed. JAI Press, Greenwich, Conn., pp. 91–107.

LEE, D. T., AND PREPARATA, F. P. 1982. An improved algorithm for the rectangle enclosure problem. *J. Algorithms 3*, 3 (Sept.), 218–224.

MATSUYAMA, T., HAO, L. V., AND NAGAO, M. 1984. A file organization for geographic information systems based on spatial proximity. *Comput. Vision Graph. Image Process. 26*, 3 (June), 303–318.

MCCREIGHT, E. M. 1980. Efficient algorithms for enumerating intersecting intervals and rectangles. Report CSL-80-9, Xerox Palo Alto Research Center, Palo Alto, Calif. (June).

MCCREIGHT, E. M. 1985. Priority search trees. *SIAM J. Comput. 14*, 2 (May), 257–276.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. 1984. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst. 9*, 1 (Mar.), 38–71.

ORENSTEIN, J. A. 1982. Multidimensional tries used for associative searching. *Inf. Process. Lett. 14*, 4 (June), 150–157.

OVERMARS, M. H. 1988. Geometric data structures for computer graphics: An overview. In *Theoretical Foundations of Computer Graphics and CAD*, R. A. Earnshaw, Ed. Springer-Verlag, Berlin, pp. 21–49.

PREPARATA, F. P., AND SHAMOS, M. I. 1985. *Computational Geometry: An Introduction*. Springer-Verlag, New York.

REGNIER, M. 1985. Analysis of grid file algorithms. *BIT 25*, 2, 335–357.

REQUICHA, A. A. G. 1980. Representations of rigid solids: Theory, methods, and systems. *ACM Comput. Surv. 12*, 4 (Dec.), 437–464.

ROBINSON, J. T. 1981. The *k-d*-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the SIGMOD Conference* (Ann Arbor, Mich., Apr.). ACM, New York, pp. 10–18.

ROSENBERG, J. B. 1985. Geographical data structures compared: A study of data structures supporting region queries. *IEEE Trans. Comput. Aided Design 4*, 1 (Jan.), 53–67.

ROSENFELD, A., AND KAK, A. C. 1982. *Digital Picture Processing*, 2nd ed. Academic Press, New York.

ROUSSOPOULOS, N., AND LEIFKER, D. 1985. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the SIGMOD Conference* (Austin, Tx., May). ACM, New York, pp. 17–31.

SAMET, H. 1980. Region representation: Quadtrees from binary arrays. *Comput. Graph. Image Process. 13*, 1 (May), 88–93.

SAMET, H. 1984. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2 (June), 187–260.

SAMET, H. 1989a. *Fundamentals of Spatial Data Structures*. Addison-Wesley, Reading, Mass.

SAMET, H. 1989b. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, Mass.

SAMET, H., AND TAMMINEN, M. 1985. Computing geometric properties of images represented by linear quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 7*, 2 (Mar.), 229–240.

SAMET, H., AND TAMMINEN, M. 1988. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Trans. Pattern Anal. Mach. Intell. 10*, 4 (July), pp. 579–586.

SARNAK, N., AND TARJAN, R. E. 1986. Planar point location using persistent search trees. *Commun. ACM 29*, 7 (July), 669–679.

SELLIS, T., ROUSSOPOULOS, N., AND FALOUTSOS, C. 1987. The R^+-tree: A dynamic index for multidimensional objects. Computer Science TR-1795, Univ. of Maryland, College Park, Md.

SHAFFER, C. A. 1986. Application of alternative quadtree representations. Ph.D. dissertation, TR-1672, Computer Science Dept., Univ. of Maryland, College Park, Md.

SHAMOS, M. I., AND HOEY, D. 1976. Geometric intersection problems. In *Proceedings of the 17th Annual IEEE Symposium on the Foundations of Computer Science* (Houston, October). IEEE, New York, pp. 208–215.

SHNEIER, M. 1981. Calculations of geometric properties using quadtrees, *Comp. Graph. Image Process. 16*, 3 (July), 296–302.

SIX, H. W., AND WOOD, D. 1980. The rectangle intersection problem revisited. *BIT 20*, 4, 426–433.

SIX, H. W., AND WOOD, D. 1982. Counting and reporting intersections of d-ranges. *IEEE Trans. Comput. 31*, 3 (March), 181–187.

STONEBRAKER, M., SELLIS, T., AND HANSON, E. 1986. An analysis of rule indexing implementations in data base systems. In *Proceedings of the 1st International Conference on Expert Database Systems* (Charleston, S.C., Apr.), pp. 353–364.

TARJAN, R. E. 1983. Updating a balanced search tree in $O(1)$ rotations. *Inf. Process. Lett. 16*, 5 (June), 253–257.

ULLMAN, J. D. 1982. *Principles of Database Systems*, 2nd ed. Computer Science Press, Rockville, Md.

VAISHNAVI, V., AND WOOD, D. 1980. Data structures for the rectangle containment and enclosure problems. *Comp. Graph. Image Process. 13*, 4 (Aug.), pp. 372–384.

VAISHNAVI, V., AND WOOD, D. 1982. Rectilinear line segment intersection, layered segment trees and dynamization. *J. Algorithms 3*, 2 (June), pp. 160–176.

VAN LEEUWEN, J. AND WOOD, D. 1981. The measure problem for rectangular ranges in d-space. *J. Algorithms 2*, 3 (Sept.), 282–300.

VOELCKER, H. B., AND REQUICHA, A. A. G. 1977. Geometric modeling of mechanical parts and processes. *IEEE Comp. 10*, 12 (Dec.), 48–57.

WEIDE, B. W. 1978. Statistical methods in algorithm design and analysis. Res. Rep. CMU-CS-78-142, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa.

Received January 1988; final revision accepted June 1988.