3

Intervals and Small Rectangles

The problem of how to represent the space spanned by collections of small rectangles arises in many applications. The space spanned by the rectangles can also be viewed as the d-dimensional Cartesian product of d one-dimensional intervals (i.e., spheres), and thus the problem is often cast as one of representing one-dimensional intervals. The most common application is one where rectangles are used to approximate other shapes for which they serve as the minimum enclosing objects (recall our discussion of the R-tree and other object-based hierarchical interior-based representations in Section 2.1.5.2 of Chapter 2). For example, rectangles can be used in cartographic applications to approximate objects such as lakes, forests, and hills [1258]. Of course, the exact boundaries of the object are also stored, but usually they are only accessed if a need for greater precision exists. Rectangles are also used for design rule checking in very large-scale integration (VLSI) design applications as a model of chip components for the analysis of their proper placement. Again, the rectangles serve as minimum enclosing objects. This process includes such tasks as determining whether components intersect and insuring the satisfaction of constraints involving factors such as minimum separation and width. These tasks have a practical significance in that they can be used to avoid design flaws.

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 design 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 chapter, we focus primarily on how to represent a large collection of rectangles common in VLSI design applications. However, our techniques 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. Our presentation makes use of representations for one-dimensional intervals rooted in computational geometry as well as combines representations for multidimensional point data (see Chapter 1) and objects (see Chapter 2).

The principal tasks to be performed are similar to those described in Chapter 1. They range from 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 [1899]. The most common of these queries involves proximity relations and is classified into two classes by Hinrichs [839]. The first is an intersection query that seeks to determine if two sets intersect. 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.

In this chapter, we focus on three types of proximity queries. The first, and most common, is the *point query*, which finds all the objects that contain a given point. It is important to distinguish this query from the point query discussed in Chapter 1 that seeks to determine if a given point p is actually in the dataset (more accurately described as an exact match query). The second type is a *point set query*, which, given a relation $\oplus$ and a set of points Q (typically a region), finds the set of objects S such that $S \oplus Q$ holds. An example is a query, more commonly known as a *window operation*, that finds all the rectangles that intersect a given region. In this example, the relation $\oplus$ is defined by $S \oplus Q$ if $S \cap Q \neq \emptyset$, and Q is the query window. The third type of query is a *geometric join query* (also known as a *spatial join query* [1415]), which, for a relation $\oplus$ and two classes of objects O_1 and O_2 with corresponding subsets S_1 and S_2, finds all pairs (P_1, P_2) with $P_1 \in S_1$, $P_2 \in S_2$, and $P_1 \oplus P_2$. An example is a query that determines all pairs of overlapping rectangles. In such a case, both O_1 and O_2 correspond to the set of rectangles, and $\oplus$ is the intersection relation. In our examples, S_1 and S_2 are usually the same subsets, although the more general problem can also be handled.

Initially, we present representations that are designed for use with the plane-sweep solution paradigm [119, 1511, 1740]. For many tasks, use of this paradigm yields worst-case optimal solutions in time and space. We examine its use in solving two problems:

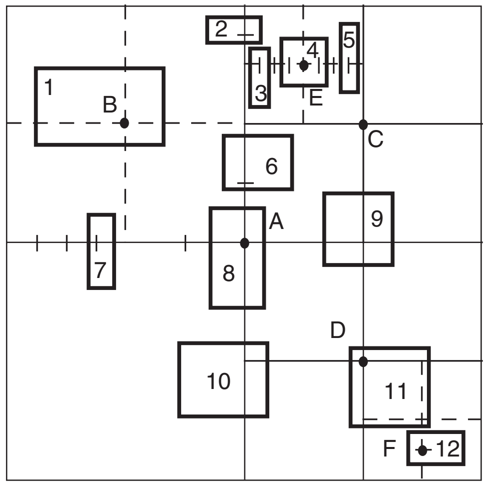
1. Reporting all intersections between rectangles (the rectangle intersection problem—Section 3.1)
2. Computing the area of a collection of d-dimensional rectangles (the measure problem—Section 3.2)

We focus on the segment, interval, and priority search trees. They represent a rectangle by the intervals that form its boundaries. However, these representations are usually for formulations of the tasks in a static environment. This means that the identities of all the rectangles must be known 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.

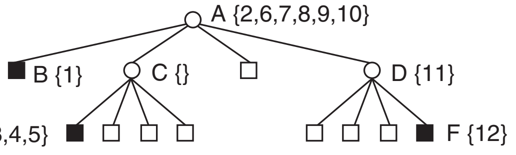
The remaining representations are for a dynamic environment. They are differentiated by the way in which each rectangle is represented. The first type of representation reduces each rectangle to a point in a usually higher-dimensional space and then treats the problem as if it involves a collection of points (Section 3.3). 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 (Section 3.4). Many of these representations are variants of the quadtree. We show that these quadtree variants 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 that the process of traversing it is analogous to a plane sweep in multiple dimensions.

3.1 Plane-Sweep Methods and the Rectangle Intersection Problem

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



(a)



(b)

Figure 3.23
(a) A collection of rectangles and the block decomposition induced by its MX-CIF quadtree and (b) the tree representation of (a).

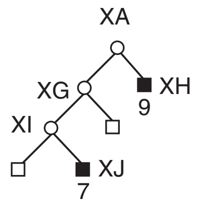
than a predetermined threshold size. This threshold is often chosen to be equal to the expected size of the rectangle [1009]. In this section, we will assume that rectangles do not overlap (but may touch), although our techniques can be modified to handle this situation. Figure 3.23 contains a set of rectangles and its corresponding MX-CIF quadtree. Once a rectangle is associated with a quadtree node, say P, it is not considered to be a member of any of the children of P. For example, in Figure 3.23, rectangle 11 overlaps the space spanned by both nodes D and F but is associated only with node D.

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. Abel and Smith [8] do not apply any ordering. This is equivalent to maintaining a linked list of the rectangles. Another approach, devised by Kedem [1009], is described below.

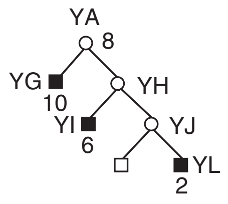
Let P be a quadtree node, and let S be the set of rectangles that are associated with P. Members of S are organized into two sets according to their intersection (or the collinearity of their sides) with the lines passing through the centroid of P's block. We shall use the terms *axes* or *axis lines* to refer to these lines. For example, consider node P, whose block is of size $2 \cdot L_x \times 2 \cdot L_y$ and centered at (C_x, C_y). All members of S that intersect the line $x = C_x$ form one set, and all members of S that intersect the line $y = C_y$ form the other set. Equivalently, these sets correspond to the rectangles intersecting the y and x axes, respectively, passing through (C_x, C_y). 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 3.24 illustrates the binary tree associated with the x and y axes passing through A, the root of the MX-CIF quadtree of Figure 3.23. The subdivision points of the axis lines are shown by the tick marks in Figure 3.23.

Note that a rectangle is associated with the shallowest node in the binary tree that contains it. For example, consider Figure 3.25, which contains the binary trees associated with the x and y axes passing through E in the MX-CIF quadtree of Figure 3.23. In particular, we see that no rectangle is stored in the left (right) subtree of node XN (XM) in Figure 3.25(a) even though rectangle 4 contains it. In this example, rectangle 4 is associated with the y axis that passes through node E (i.e., node YE in Figure 3.25(b)).

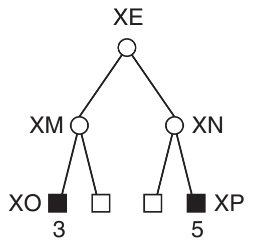


(a)

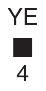


(b)

Figure 3.24
Binary trees for the (a) x axis and (b) (b) y axis passing through node A in Figure 3.23.



(a)



(b)

Figure 3.25
Binary trees for the (a) x axis and (b) y axis passing through node E in Figure 3.23.