Extracting Region Boundaries from Maps Stored as Linear Quadtrees

Michael B. Dillencourt
PAR Govt. Systems Corp.
1840 Michael Faraday Dr.
Reston, VA 22090 USA

and

Hanan Samet
Computer Science Dept.
University of Maryland
College Park, MD 20742

1. Introduction

Region representation plays a key role in Geographic Information Systems (GIS) and automated cartography. Many methods of region representation have been proposed. Because each method has its own advantages and disadvantages, it is desirable to be able to efficiently convert from one representation to another.

In this paper, we are concerned with boundary extraction from a map or image represented as a linear quadtree; that is, with determining the boundaries of the regions in a map from a quadtree-based representation of the map. This process is a form of raster to vector conversion. It can serve as the first step of a number of operations that a GIS may want to perform, such as computing a buffer zone of a given width about a region boundary, drawing a map on a vector device (such as a plotter), or displaying a map using a polygon-fill type algorithm.

We assume that each pixel has a value associated with it, which might be a country, primary crop, etc., depending on the type of map. This value is called the color of the pixel. A region consists of a set of contiguous pixels, each of which is associated with the same value. For example, the map shown in Figure 1 consists of five regions, labeled A, B, C, D, and E.

The boundary of a region can be expressed in several different ways. In this paper, the boundary is expressed as a sequence of vertices. For example, the boundary of region A in Figure 1 consists of the sequence

$$\{(0,16),(0,0),(16,0),16,4),(8,4),(8,7),(9,7),(9,8),(4,8),(4,16),(0,16)\}.$$

The algorithms of this paper can easily be adapted to produce other representations of the boundary, such as chain codes (Freeman, 1974).

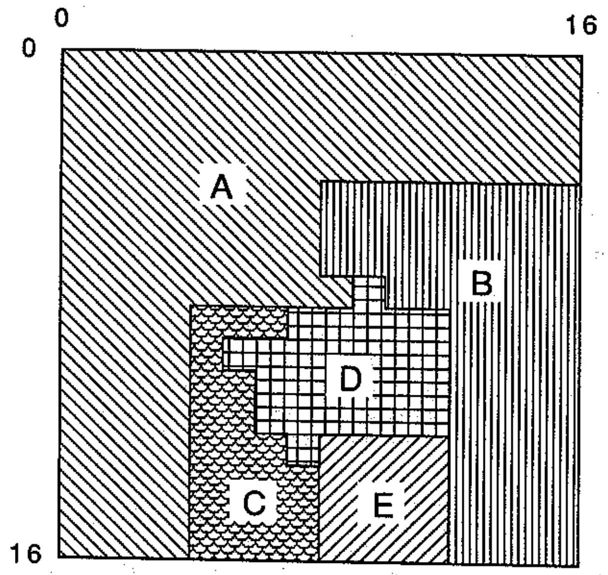


Figure 1. A sample map, consisting of five regions

2. The Linear Quadtree

One common method of storing region data is the region quadtree (Samet, 1984). In this representation, the map is decomposed into quadrants. Each quadrant that is not homogeneous (i.e., whose pixels do not all have the same associated values) is further decomposed. The result is a hierarchical decomposition of the map into disjoint homogeneous squares of different sizes. The decomposition is often stored as a tree, in which each internal (non-leaf) node has four children.

The region quadtree corresponding to Figure 1 is shown in Figures 2 and 3. Figure 2 shows the homogeneous blocks into which the region is decomposed, and Figure 3 illustrates the actual tree.

A useful alternative to the explicit (pointer-based) representation of the region quadtree is the linear quadtree (Gargantini, 1982; Abel and Smith, 1983). In the linear quadtree, only the leaf nodes of the region quadtree are explicitly stored. Each node is assigned a location code describing its position in the quadtree. The location codes are assigned in such a way that visiting the nodes in increasing order of the location code is equivalent to a preorder traversal of the leaf nodes of the region quadtree from which the linear quadtree was derived. The node numbering shown in Figure 2 reflects a quadrant ordering of Northwest, Northeast, Southwest, Southeast. The leaf nodes can then be stored in a B-tree (Comer, 1979), with the location code serving as the key.

The linear quadtree has two major advantages over the pointer-based region quadtree. It saves the space that would otherwise be occupied by the internal nodes and the pointers. Moreover, the linear quadtree method is more appropriate for data that resides on secondary storage, since it can use a data structure (e.g., the B-tree) that is designed specifically for random-access secondary storage devices.

A method for extracting the boundary of a binary image from a region quadtree appears in (Dyer, Rosenfeld, and Samet, 1980). It works by using a neighbor-finding algorithm to follow the boundary of the image through the quadtree. This method could be successively applied to each region in a map to produce all the boundaries. However, this approach can be quite inefficient in a linear quadtree: each call to the neighbor-finding routine may involve swapping pages in the underlying B-tree.

The algorithm for boundary extraction that we describe here computes the boundary of all map regions using a single traversal of the linear quadtree. It visits the nodes in order of ascending location-code, so each page in the B-tree needs to be read only once.

3. The data structures

Our algorithm maintains two basic data structures: an active border (Samet and Tamminen, 1985), and a list of active regions. The active border represents the border

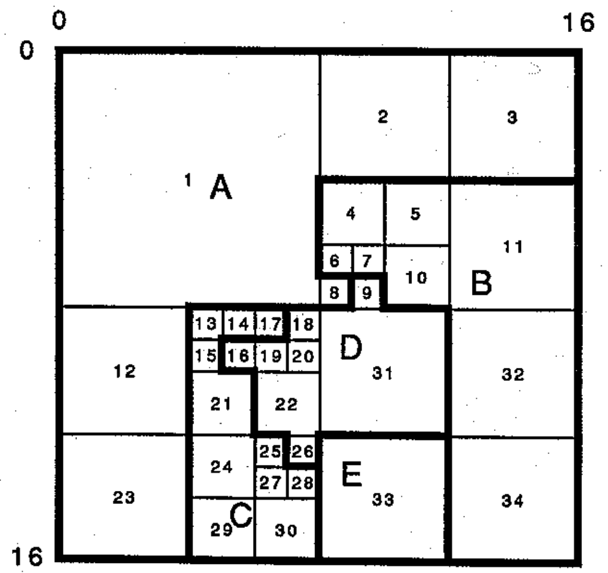


Figure 2. The decomposition of the map of Figure 1 into homogeneous blocks.

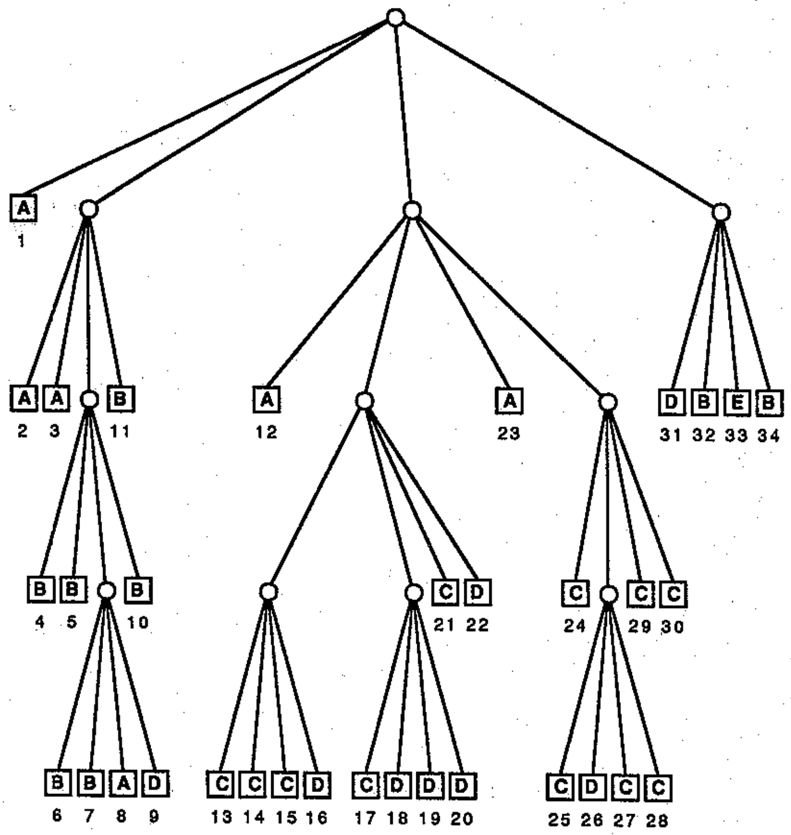


Figure 3. The quadtree for the image of Figure 2.

between those quadtree nodes that have been processed and those that have not. It consists of quadtree nodes, edges, and vertices.

The active border edges consist of all edges with the property that one of the two nodes adjacent to the edge has been processed while the other node has not. The active border edges form a "staircase," moving from southwest to northeast, as shown in Figure 4. The set of active border edges is implemented as a doubly linked list of records of type *edge*, ordered from southwest to northeast. Each *edge* record has two link fields named NEXT and PREV. In addition, active border edge records contains a PREVNODE field, which points to the quadtree node, adjacent to the edge, that has been processed. Moreover, the vertical and horizontal edges are also stored in separate balanced binary trees, for reasons to be given in the next section.

The active quadtree nodes, or simply active nodes, are those quadtree nodes that have been processed but are adjacent to nodes that have not been. Equivalently, a quadtree node is active if some portion of its eastern or southern boundary is an active edge. Each active node has associated with it a field REG, which points to the region descriptor of the active region to which it belongs.

The active regions are those regions that contain an active node. Each active region has associated with it a region descriptor. The region descriptor for each active region has associated with it a *partial boundary* for the region. The partial boundary is a linked list of chains. Each chain consists of a linked list of points, which represent a contiguous portion of the boundary of the region. Each region descriptor also contains a hash table (Knuth, 1973) which stores the first and last point of each of its chains.

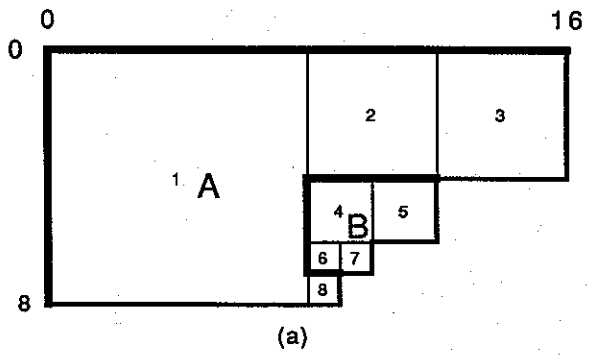
The active vertices are those vertices that serve as the endpoint of an active edge. Associated with each active vertex is an array, CHAINLOC, of (no more than three) pointers to the positions where the vertex appears in the partial boundaries active regions.

Figure 4(a) illustrates the active border corresponding to Figure 3, after nodes 1 through 8 have been processed. The heavy lines represent the partial boundaries that have already been encountered. There are nine active border edges, which are listed in Figure 4(b). Five of these edges are horizontal, and four are vertical. Nodes 1, 3, 5, 7, and 8 are active. There are two active regions, A and B. The partial boundary of A consists of two chains, $\{(0,8),(0,0),(0,16)\}$ and $\{(12,4),(8,4),(8,7),(9,7)\}$, because the nodes on both sides of elements on these chains have already been processed. The partial boundary of B consists of the single chain (9,7), (8,7), (8,4), (12,4) for the same reason.

Initially, the active border comprises the north and west borders of the map. When the algorithm terminates, the active border consists of the south and east borders.

4. The algorithm for boundary extraction in a linear quadtree

The boundary extraction algorithm processes each node exactly once. The general



Active Edges	PREVNODE
[(0,8),(8,8)]	1
[(8,8),(9,8)]	8
[(9,8),(9,7)]	8
[(9,7),(10,7)]	7
[(10,7),(10,6)]	7
[(10,6),(12,6)]	5
[(12,6),(12,4)]	5
[(12,4),(16,4)]	3
[(16,4),(16,0)]	3

(b)

Figure 4. (a) The active border of the map of Figure 2 after node 8 has been processed. (b) The list of active edges that make up the active border step in the algorithm uses the procedure PROCESS_1_NODE, defined below. A detailed description of the various steps follows the procedure definition. The basic steps are as follows. A new region descriptor is allocated, for the region to which the current node belongs. The set of active border edges that are part of the boundary of the current node is determined, and each active border edge in this set is examined. All active regions that share a boundary edge with the current node, and that have the same color as the current node, are merged into a single region. Each active border edge that separates the current node from a region with a different color is added to the boundary of the current region, and to the other region as well. Finally, the active border is updated, to reflect the fact that the current node has been processed.

Each node has a field called COLOR, which contains the color of the node or region. The function REVERSE, when applied to a segment, returns the segment with the end-points reversed. For example, REVERSE([(0,2),(0,8)]) returns the segment [(0,8),(0,2)]. The functions XW, XE, YS, and YN, when applied to a node, return the westernmost x-coordinate, the easternmost x-coordinate, the southernmost y-coordinate, and the northernmost y-coordinate, respectively.

Procedure PROCESS_1_NODE is applied to each node in the linear quadtree. After all the nodes are processed, the active border edges are the south and east borders of the map. These are most easily processed by applying the procedure PROCESS_1_NODE to two (fictitious) leaf nodes that are the same size as the entire map, to its south and east.

```
procedure PROCESS_1_NODE(NODE);
value pointer quadtree_node NODE;
pointer region_desc CURREG;
pointer segment SEG,STARTSEG,ENDSEG;
begin
    CURREG := ALLOCATE_NEW_REGION();
    NORMALIZE_ACTIVE_BORDER(XW(NODE),YS(NODE),XE(NODE),
          YN(NODE),STARTSEG,ENDSEG);
    SEG := STARTSEG
    loop
        if COLOR(PREVNODE(SEG) = COLOR(NODE) then
            CURREG = MERGE(REG(PREVNODE(SEG)),CURREG)
        else
            begin
                ADD_TO_BOUNDARY(SEG,CURREG);
                ADD_TO_BOUNDARY(REVERSE(SEG),REG(PREVNODE(SEG)));
            end;
        if SEG = ENDSEG then
            exit loop
        else
            SEG := NEXT(SEG);
```

```
    end loop;
    UPDATE_ACTIVE_BORDER(XW(NODE),YS(NODE),XE(NODE),YN(NODE);
end;
```

The variable CURREG represents a pointer to the descriptor of the region to which the current node belongs. It is initially allocated by the call to the function ALLOCATE_NEW_REGION().

Procedure NORMALIZE_ACTIVE_BORDER has two roles. First, it ensures that there is a set of active border edges that exactly covers the west and north boundaries of the current node. This may require splitting active border edges. For example, in Figure 2, before node 1 is processed, the active border edges are $[(0,16),(0,0)]$ and $[(0,0),(16,0)]$. In this case, NORMALIZE_ACTIVE_BORDER must split each of these two segments, thereby transforming the active border edges into the set of four edges $[(0,16),(0,8)]$, $[(0,8),(0,0)]$, $[(0,0),(8,0)]$, and $[(8,0),(16,0)]$. Second, it returns pointers to the first (i.e., most southwest) and last (i.e., most northeast active border edges that are adjacent to the current node. The process of locating the active border edge containing the southwestern and northeastern corners of the current node is made more efficient by the organization of the active border edges as two balanced binary trees, which was mentioned in the previous section. These trees are based on the leftmost x-coordinate, and the topmost y-coordinate values of the horizontal and vertical segments, respectively, of the active border.

The loop processes each active border edge that forms part of the boundary of the current node. There are two cases, depending on whether the edge separates two nodes of the same color or of different colors.

If the edge separates two nodes of the same color, then the two nodes must belong to the same region. This fact is recorded by invoking the procedure MERGE. MERGE takes as its arguments two pointers to region descriptors. If the two regions are identical, MERGE does nothing. Otherwise, it merges the partial boundaries of the two regions. One of the region descriptors (the survivor) has the data for the other region merged into it. The other region descriptor (the old descriptor) is overwritten with a pointer to the survivor. Subsequent references to the old descriptor will follow this pointer to the survivor. Chains of pointers, resulting from several merges, are shortened by using path-compression techniques (Sedgewick, 1983). Region descriptors that are not in use are detected using reference counters, so they can be deallocated and recycled (Samet and Tamminen, 1986). MERGE returns a pointer to the surviving region descriptor.

If the edge separates two nodes of different colors, then the boundary structure must be updated. This is done by two calls to the procedure ADD_TO_BOUNDARY. In order to produce a clockwise traversal of the boundary of a region, a segment must be added to the boundary of the region in such a way that the region is to the right of the segment. Thus the segment is added to the boundary of the current region, and the reverse of the segment is added to the boundary of the previous region. For example, when node 4 is processed in Figure 2, the segment $[(8,4),(8,6)]$ is added to the boundary of region B, and the segment $[(8,6),(8,4)]$ is added to the boundary of region A.

When ADD_TO_BOUNDARY adds a segment to a boundary, it looks for a chain to which the segment can be concatenated, using the hash table of chain endpoints described in the previous section. Depending on the result of this search, it may add the segment to a chain, combine two chains, start a new chain, or complete a cycle. (A cycle is a chain in which the first and last points are the same). The processing for MERGE is similar, except that instead of adding a single segment, all chains in the old region are added to the survivor region. In either case, the hash table of chain endpoints must be modified after each modification to the partial boundary structure.

When all the chains of the partial boundary of a region are cycles, the boundary is said to be *complete*, and it may be written to the output file. When this happens, the region is no longer active, and all storage used for the region descriptor may be reclaimed. If a region is simply connected (i.e., if it has no "holes"), then its completed boundary consists of a single component. Regions with holes are discussed in the following section.

The routine that writes the boundary to the output file is an appropriate place to perform conversions between appropriate representations. For instance, the representation of the boundary as a sequence of horizontal and vertical segments could be converted to a chain code.

The final step in PROCESS_1_NODE is a call to UPDATE_ACTIVE_BORDER. This procedure removes the west and north boundaries of the current node from the set of active border edges, and replaces them with the east and south edges. It also updates the active vertex and active quadtree node structures.

5. Regions with holes

Special considerations may apply when the map contains regions that are not simply connected. If the boundary of a region does not intersect itself, then the algorithm of the previous section works without modification. The last chain of the boundary of a region to be completed (i.e., to become a cycle) is the outer boundary, while the other cycles represent the boundaries of holes.

As a simple example, consider Figure 5(a). After node 15 is processed, the boundary of region B is complete and may be written to the output file. After the south and west boundaries of the map are processed, the boundary of region A is complete. The boundary of region B is $\{(1,1),(3,1),(3,3),(1,3),(1,1)\}$. The boundary of A consists of two components: the outer component of the boundary is $\{(0,0),(4,0),(4,4),(0,4),(0,0)\}$ and the inner component $\{(1,1),(1,3),(3,3),(3,1),(1,1)\}$.

Notice that each component of the boundary of A is oriented so that the region A is to its right. Notice also that the boundary of the inner region (B) is written before the boundary of the outer region (A). Both of these properties are characteristic of the

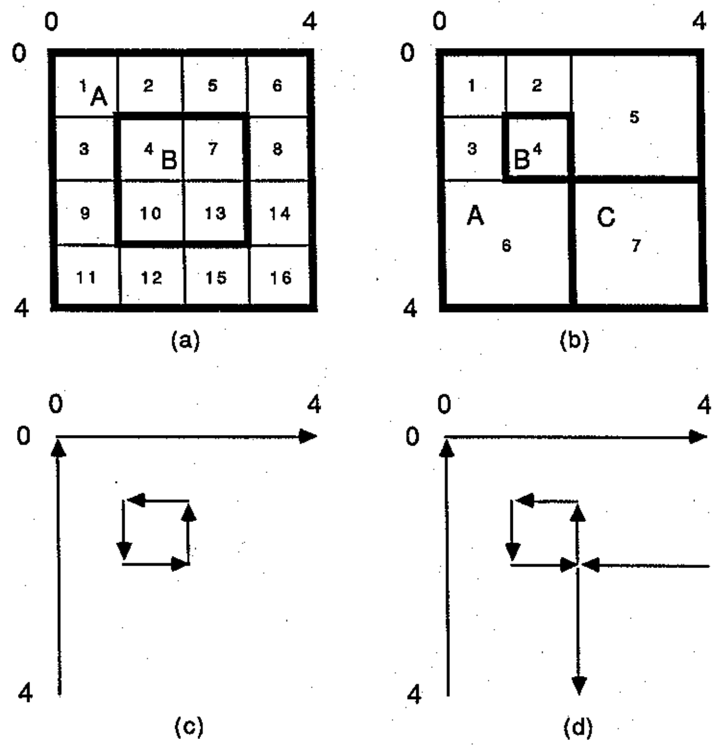


Figure 5. Regions with holes. (a) A region with a hole. (b) A region in which the boundary of region A intersects itself. (c) The partial boundary of region A in (b) before node 7 is processed. (d) The partial boundary of region A in (b) after node 7 is processed.

algorithm. These properties may reduce the amount of work that needs to be done by subsequent processing steps. For example, suppose the output of the boundary extraction algorithm is to be used to color the map using a polygon-fill algorithm on a device with destructive overwrite. Then correct results will be obtained if (1) only the outer component of each boundary is processed, and (2) the boundaries are processed in the opposite order from the order in which they were written to the output file. For example, in Figure 5(a), first the area inside the outer boundary of A (i.e., the whole map) will be colored with the appropriate color for A, and then area B will be overwritten with its color.

If the boundary of a region intersects itself, certain subtleties may come into play. For example, consider the map of Figure 5(b), which has three regions. The boundary of region A intersects itself at the point (2,2). If the algorithm described above is applied without modification, the boundary of region A will be determined to have two components: $\{(2,2),(2,1),(1,1),(1,2),(2,2)\}$ and $\{(0,0),(4,0),(4,2),(2,2),(2,4),(0,4),(0,0)\}$. In many cases, this may be an appropriate result. However, there are situations in which it is better to express the boundary of A as a single component, $\{(0,0),(4,0),(4,2),(2,2),(2,1),(1,1),(1,2),(2,2),(2,4),(0,4),(0,0)\}$. For example, it might be known that B and C were really the same region, and that their apparent separation was an artifact of the digitization process.

The fact that it is possible to connect two components of the boundary of A can be detected, and acted upon, as follows. After node 6 is processed, the partial boundary of region A consists of the two components $\{(0,4),(0,0),(4,0)\}$ and $\{(1,1),(1,2),(2,2),(2,1),(1,1)\}$, as shown in Figure 5(c). Processing node 7 causes the segments $[(2,2),(2,4)]$ and $[(4,2),(2,2)]$ to be added to the partial boundary of A, as in Figure 5(d). The fact that the point (2,2) is already part of the boundary indicates that the boundary intersects itself. It is easy to see that when the two pieces of the boundary are joined in such a way that region A is always to the right of the boundary, it must be true that the two turns taken by the boundary at (2,2) must both be 90 degree angles (measured counterclockwise). Thus the segment $[(4,2),(2,2)]$, directed to the west, must be followed by the segment $[(2,2),(2,1)]$, which is directed north. Similarly, $[(1,2),(2,2)]$ must be followed by $[(2,2),(2,4)]$. So after node 7 is processed, the partial boundary of A consists of the two components $\{(0,4),(0,0),(4,0)\}$ and $\{(4,2),(2,2),(2,1),(1,1)(1,2),(2,2),(2,4)\}$. After the south and east borders of the map are processed, a boundary consisting of a single component will result. It can be shown that the single rule applied above, namely that whenever a boundary intersects itself, it must take two 90 degree counterclockwise turns, can be applied to resolve all situations of this type.

Notice that efficient implementation of the steps just describe for handling a self-intersecting boundary requires being able to quickly detect that an active vertex (in this case, (2,2)) is already part of a boundary chain. This is the reason for associating the array CHAINLOC with each active vertex. This array contains pointers to the appearances of the vertex in the boundary chains to which it belongs, as described in Section 3.

6. Summary

An algorithm has been described for deriving the boundaries of all regions in a map that is stored as a quadtree. This algorithm has been implemented on a Sun 3 workstation. In contrast with the algorithm of (Dyer et al, 1980), which traces a single boundary, the algorithm described here makes a single pass over the quadtree and maintains partial descriptions of all boundaries. Thus our algorithm is particularly suited for use on a linear quadtree in secondary storage. With minimal adjustments, the algorithm handles regions with holes.

Acknowledgment

The support of the National Science Foundation under grant DCR-86-05557 is gratefully acknowledged.

References

D. J. Abel and J. L. Smith, A Data Structure and Algorithm Based on a Linear Key for a Rectangle Retrieval Problem, *Computer Vision, Graphics, and Image Processing 24*, 1 (October, 1983), 1–13.

D. Comer, The Ubiquitous B-tree, *ACM Computing Surveys 11*, 2 (June, 1979), 121–137.

C. R. Dyer, A. Rosenfeld, and H. Samet, Region Representation: Boundary Codes from Quadtrees, *Communications of the ACM 23*, 3 (March, 1980), 171–179.

H. Freeman, Computer Processing of Line-drawing Images, *ACM Computing Surveys 6*, 1 (March, 1974), 57–97.

I. Gargantini, An Effective Way to Represent Quadtrees, *Communications of the ACM 25*, 12 (December, 1982), 905–910.

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

H. Samet, The Quadtree and Related Hierarchical Structures, *ACM Computing Surveys 16*, 2 (June, 1984), 187–260.

H. Samet and M. Tamminen, Computing Geometric Properties of Images Represented by Linear Quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 3 (March, 1985), 229–240.

H. Samet and M. Tamminen, An Improved Approach to Connected Component Labeling of Images, *Proceedings of Computer Vision and Pattern Recognition 86*, Miami Beach, FL, June, 1986, 312–318.

R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1983.