Algorithmica (1996) 15: 82–102

Algorithmica

Using Topological Sweep to Extract the Boundaries of Regions in Maps Represented by Region Quadtrees[1]

M. B. Dillencourt[2] and H. Samet[3]

Abstract. A variant of the plane-sweep paradigm known as topological sweep is adapted to solve geometric problems involving two-dimensional regions when the underlying representation is a region quadtree. The utility of this technique is illustrated by showing how it can be used to extract the boundaries of a map in $O(M)$ space and $O(M\alpha(M))$ time, where M is the number of quadtree blocks in the map, and $\alpha(\cdot)$ is the (extremely slowly growing) inverse of Ackerman's function. The algorithm works for maps that contain multiple regions as well as holes. The algorithm makes use of active objects (in the form of regions) and an active border. It keeps track of the current position in the active border so that at each step no search is necessary. The algorithm represents a considerable improvement over a previous approach whose worst-case execution time is proportional to the product of the number of blocks in the map and the resolution of the quadtree (i.e., the maximum level of decomposition). The algorithm works for many different quadtree representations including those where the quadtree is stored in external storage.

Key Words. Computational geometry, Topological sweep, Plane sweep, Region representation, Boundary extraction, Active borders, Region quadtrees, Computer graphics, Image Processing, Geographic information systems.

1. Introduction.

Efficient processing of geometric data is an important issue in computational geometry, computer graphics, image processing, geographic information systems, VLSI design, etc. The algorithm and problems frequently depend on the nature of the data and, most importantly, on its representation. One of the most popular problem-solving paradigms is that of plane sweep [3], [10], [17], [18], [20]. It attacks the problem in two stages. The first stage organizes the data using techniques such as sorting (e.g., rectangle problems [9]), arrangements [11], quadtrees [19], etc. The second stage sweeps a line or a topological equivalent through the result of the first stage, and performs the operation in a more restricted setting (e.g., on a subset of the data).

By organizing the data in the first stage, we are often able to reduce the necessary computation by a dramatic amount since the irrelevant data can be pruned from the search space. In applications involving geometric data, the search space can be large on account of its size (i.e., the area spanned by it). In such a case the physical organization of the data may also play an important role in evaluating the efficiency of the solution.

[1] M. B. Dillencourt was supported in part by a UCI Faculty Research Grant, and H. Samet was supported in part by the National Science Foundation under Grant IRI-90-17393.

[2] Department of Information and Computer Science, University of California, Irvine, CA 92717, USA. dillenco@ics.uci.edu.

[3] Center for Automation Research and Computer Science Department, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.

Received April 15, 1991; revised December 7, 1992. Communicated by D. P. Dobkin.

For example, if the data is stored in external storage (e.g., on disk), then we want to pursue a solution strategy that minimizes page faults. This means that the organization of the data imposed by stage one should be tightly coupled to the algorithm used in stage two. Thus, in the case of two-dimensional regions, an algorithm that computes a geometric property by following the boundary or connectivity of the region (which may be arbitrary) may be less attractive than one that computes it by exploiting properties of the space in which the region lies.

In this paper we focus on two-dimensional region data. Our domain is a collection of regions whose borders are rectilinear. The regions may also contain holes. Such data arises frequently in automated cartography (e.g., a map of counties or states). We assume that the regions are represented by a region quadtree (e.g., [20]). This is a flexible representation which is based on sorting geometric data according to their relationship with the space that they occupy. We show how to adapt a variant of the plane-sweep paradigm known as a topological sweep to solve problems in this domain.

We illustrate the utility of the topological sweep technique by using it to extract the boundaries of a map in $O(M\alpha(M))$ time where M is the number of quadtree blocks in the map, and $\alpha(\cdot)$ is the (extremely slowly growing) inverse of Ackerman's function. The algorithm presented here makes a single pass over the image. It represents an improvement over a previous approach proposed in [8], which extracts the boundary of each region by using boundary-following techniques. The worst-case cost of applying the boundary-following algorithm of [8] to the entire map region is proportional to the product of the number of blocks in the map and the resolution of the quadtree (i.e., the maximum level of decomposition). The boundary-following algorithm requires the entire map to be stored in memory for the entire run of the algorithm. This is not a requirement for our algorithm. This is an important practical consideration: it means, for example, that our algorithm can process a very large map on a computer with limited memory (such as a personal computer). In Section 8 below, we show how to reduce the memory requirement of our algorithm, at the cost of an increase in processing time.

The techniques of this paper can also be adapted to maps whose boundaries need not be rectilinear, provided the map is represented appropriately. For example, they can be applied to polygonal data stored in a PM quadtree [23].

The rest of this paper is organized as follows. Section 2 contains a discussion of the plane-sweep paradigm and shows how its relative, the topological sweep, can be adapted to quadtrees. Section 3 discusses the problem of boundary extraction in quadtrees and reviews prior approaches, as well as gives an overview of our algorithm. Sections 4 and 5 describe the data structures and the new algorithm, while Sections 6 and 7 discuss the correctness of the algorithm and give an analysis of its space and execution time requirements. Section 8 shows how memory can be conserved, and Section 9 contains concluding remarks as well as directions for future research. The Appendix contains pseudocode for the algorithm.

2. Topological Sweep and Region Quadtrees.

Plane sweep is one of the basic paradigms of computational geometry [3], [10], [17], [18], [20]. It consists of sweeping a straight line (called the *sweep line*) across the plane through a collection of objects. Plane-sweep algorithms consist of two phases: a *sort phase*, which sorts the points at which the objects make their first and last encounter with the sweep line (termed *halting points*), and a *sweep phase*, in which the actual plane sweep is performed and partial solutions are computed. Plane sweep requires maintaining two sets of data. The first set consists of the set of halting points, while the second set consists of the objects intersected by the current position of the sweep line (termed *active objects*).

Topological sweep [11] is a variant of plane sweep that makes use of a sweep line consisting of a simple (i.e., non-self-intersecting) curve which need not be a straight line. It can be applied when an appropriate combinatorial structure among the input objects is available. It uses the combinatorial structure to guide the sweep and thereby eliminates the sort phase.

Some of the first problems to be systematically attacked by plane-sweep methods involved the computation of certain properties of collections of rectangles with sides parallel to the axes. Among the properties that can be computed using this approach are the total area [5], intersections [6], [9], [16], and maximum clique in the intersection graph [15]. In the most general formulation of these problems, rectangles are allowed to overlap. Under these circumstances, the sort phase cannot be avoided. Indeed, it can be shown using the methods of [4] that these problems have $\Omega(n \log n)$ lower bounds in the algebraic decision-tree model (also see [18]).

The situation changes when further restrictions are placed on the rectangles. In particular, in many image processing and geographic information systems (GIS) applications, the image space (i.e., map) is partitioned into a collection of nonoverlapping rectangles that span the image space. Algorithms for computing properties of the image may then proceed by performing a topological sweep of the image. The sweep line is a connected sequence of horizontal and vertical segments, and the adjacency relations among rectangles are used to guide the sweep.

The importance of topological-sweep methods is especially apparent when the image is stored using a hierarchical representation based on a rectangular decomposition, such as the bintree and the region quadtree (e.g., [20]). The region quadtree decomposes a map 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, or *blocks*, of different sizes. The decomposition is often stored as a tree, in which each internal (nonleaf) node has four children.

The block decomposition induced by the region quadtree of Figure 1 is illustrated in Figure 2. The blocks are numbered according to the order in which they would be visited when the quadtree is scanned using a northwest, northeast, southwest, southeast scanning order.

If the children of each node are ordered consistently, then a listing of the leaf nodes in a preorder traversal (or equivalently postorder since the nonleaf nodes are ignored) corresponds to a valid topological sweep of the image space. For example, if the nodes are in northwest, northeast, southwest, southeast order, then at any instant during the sweep, the sweep line is a "staircase" moving from southeast to northwest. The staircase is termed the *active border*. The simple curve that forms the active border is in fact the topological sweep frontier. The set of active objects consists of the active quadtree blocks which are the blocks that are adjacent to the staircase in the sense that their boundaries coincide with the staircase or are adjacent to blocks whose boundaries have

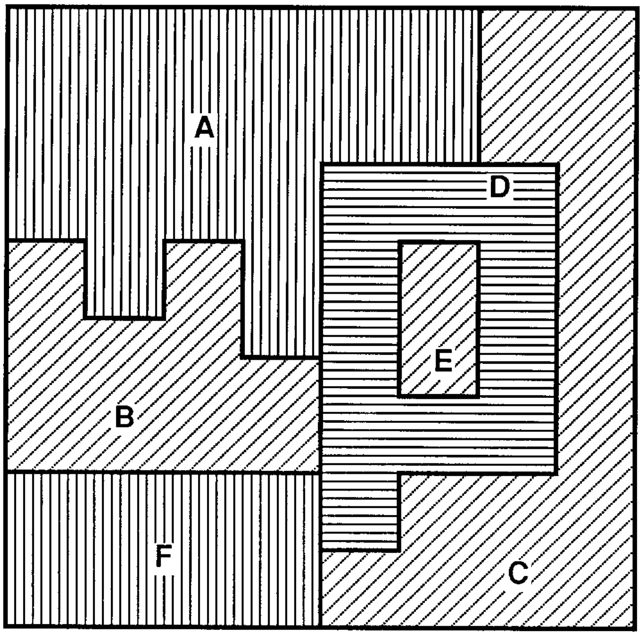

Fig. 1. A sample map, consisting of six regions.

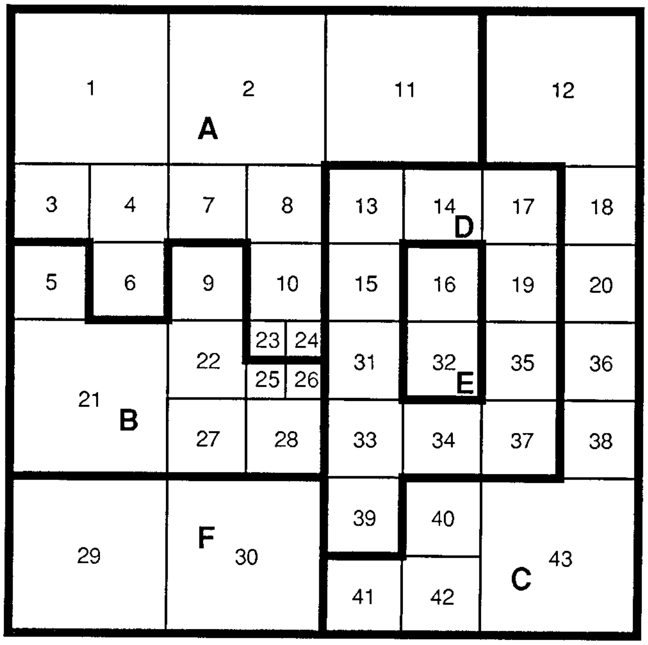

Fig. 2. The decomposition of the map of Figure 1 into homogeneous blocks.

been scanned in their entirety. Alternatively, the active objects could also be viewed as the regions whose quadtree blocks are active. The correct interpretation depends on the application. These analogies underlie algorithms based on “active border methods” [19], [21], [22].

3. Boundary Extraction in Region Quadtrees. Boundary extraction 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 fitting a spline to the boundary of a region.

To see this problem at its simplest, consider an image or map in the form of an array of image elements (termed pixels). 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 six regions, labeled A, B, C, D, E, and F.

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, starting at the upper left corner and proceeding clockwise around the boundary (i.e., with the image to the right), consists of the sequence

$$\{(0,0),(12,0),(12,4),(8,4),(8,9),(6,9),(6,6),(4,6),(4,8),(2,8),(2,6),(0,6)\}.$$

Of course, other representations of the boundary (e.g., chain codes [12]) are also possible, and the algorithms of this paper can be readily adapted to produce them.

We are interested in boundary extraction in an image where the regions are represented using a region quadtree. An algorithm for extracting the boundary of a single region in a (pointer-based) region quadtree appears in [8]. The algorithm of [8] works by following the boundary of the region through the quadtree. Its worst-case time is proportional to the product of the number of blocks in the image and the resolution. This algorithm could be adapted to deal with multiple regions by applying it to each region. Holes may also pose a problem in the sense that the extension of the algorithm does not yield a correspondence between holes and the containing regions without additional processing. An additional drawback of this algorithm is that the image elements that comprise the border may not necessarily be stored next to each other. This will result in page faults if the image is not entirely in the main memory.

In contrast, the solution that we present processes the adjacencies in the image in a predetermined order. In particular, it makes a single pass over the image by using a topological sweep in the form of a traversal of the blocks comprising the quadtree in the order northwest, northeast, southwest, southeast. The key data structure is a set of active regions, which represent the regions that meet the sweep line. Associated with each region is a partial boundary, consisting of one or more simple closed curves (termed *cycles*). The partial boundary represents the algorithm’s best current guess at the boundary of the region, based on the information it has seen up to now.

One cycle, always present, represents the outer boundary of the region. Some portions of this cycle, called *chains*, represent portions of the boundary that are known to belong to the boundary (because the block on the other side has been visited and is known to be of a different color). Other portions, called *bridges*, represent portions of the boundary that may or may not belong to the boundary (their status is unclear because the block on the other side has not yet been visited.) The remaining cycles, if present, represent “holes” in the region.

When a new quadtree block is visited, its adjacencies in the western and northern directions are examined. Each adjacency between two blocks of different colors causes bridges to be replaced by portions of chains. In addition, it may be that such an adjacency causes the boundary of one of the regions to become closed (and hence complete). In this case the boundary of this region is written to the output file and the region is removed from the active region data structure (i.e., the region becomes *inactive*). An adjacency between two blocks of the same color may result in two regions being merged, a hole being detected, or neither of these two possibilities occurring. A simple test, described more fully in Section 5.3, distinguishes among these three cases.

One key to the success of this algorithm is that the nature of the sweep and the associated data structures ensure that when a block is visited, we know its containing region. This means that when an adjacency causes the boundary of a hole to be closed, we know the identity of the region surrounding it, so assigning holes to the right region is easy. In addition, there is no computational overhead in locating the appropriate position in the active border when processing a block. Consequently, we can perform the sweep phase of the algorithm in almost-linear time. The periodic merging of disjoint regions results in a worst-case overhead factor of $\alpha(M)$ which, while undesirable from a theoretical point of view, is undetectable in practice.

The new algorithm requires additional storage for the active border which is on the order of the perimeter of the image, as well as the partial boundaries of the regions. This storage is less than the storage required to hold the entire image. Thus if enough storage is available to permit the algorithm of [8] to run efficiently (i.e., enough to hold the entire image), the new algorithm will also perform well. If smaller amounts of storage are available (so that the entire image cannot fit in memory at once), then the algorithm of [8] (because of its nonsequential access to portions of the image) will cause many more page faults while reading the image. The new algorithm can be altered to run in less memory, at some cost in processing speed; such an approach is outlined in Section 8.

Another very important property of the new algorithm is that it works with both pointer-based and pointerless quadtree representations. Pointerless representations are of interest because the space required for storing pointers from a node to its sons may be significant. Of greater importance is the fact that when the quadtree is represented in external storage, processing pointer chains can be time consuming due to the presence of page faults.

Two approaches to pointerless quadtree representations have been proposed. In the first approach the image is treated as an ordered collection of leaf nodes. Each leaf is represented by a locational code corresponding to a sequence of directional codes that locate the leaf along a path from the root of the tree [1], [13]. In the second approach the image is represented as a preorder traversal of the nodes of its quadtree, known as a *DF-expression* [14], [24]. In this representation the symbol “G” represents a gray node (i.e., a block that is subdivided further), and any other symbol represents a leaf node corresponding to a homogeneous block of the indicated color. The algorithm that we present works for all of these representations, although our pseudocode and description assume the DF-expression representation.

4. The Data Structures.

Our algorithm visits the blocks of the quadtree in NW, NE, SW, SE scanning order. It maintains two basic data structures: an active border [21] and a set of active regions. The active border is represented by a list of records of type **activeborderelement**, and each active region is represented by a collection of records of type **region**, as described below. We first describe what the data structures represent and how they interact, and then we describe their implementations (i.e., the record structure).

4.1. *The Active Border.*

The *active border* represents the border between those quadtree blocks that have been processed and those that have not. The elements of the active border form a "staircase" of vertical and horizontal edges, moving from southwest to northeast, as shown in Figure 3(a). Initially, the active border consists of the north and west borders of the image. When the algorithm terminates, the active border consists of the south and east borders of the entire image. The set of active border elements is implemented as a

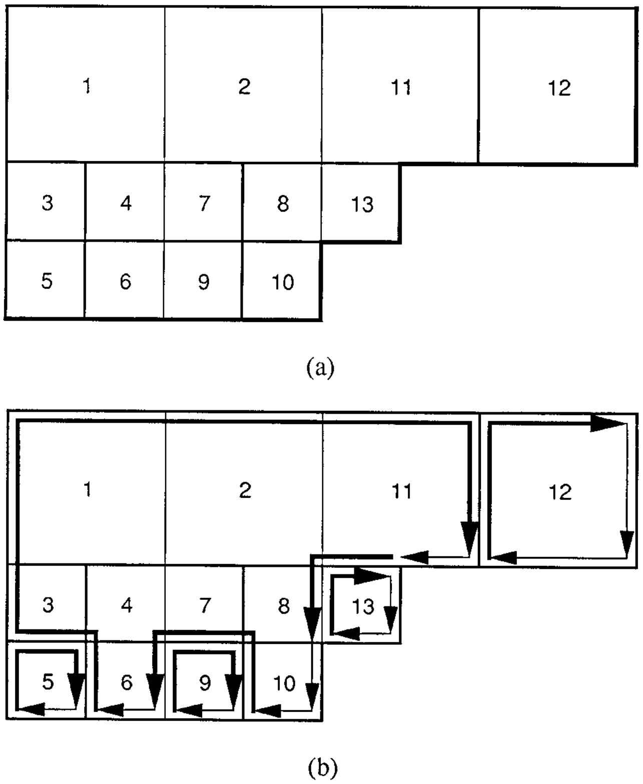

Fig. 3. (a) The active border of the map of Figure 2 after node 13 has been processed. (b) The list of active edges that constitute the active border.

doubly linked list of records of type **activeborderelement**, ordered from southwest to northeast.

4.2. *The Active Regions*. Because of the order in which the blocks of the quadtree are processed, a quadtree block is *active* if some portion of its eastern or southern boundary is an active border edge. An *active region* is a region (i.e., a continguous set of blocks of the same color) that contains an active quadtree block. For example, consider Figure 3(a), which illustrates the active border of Figure 2 after block 13 has been processed. There are seven active blocks: 5, 6, 9, 10, 11, 12, and 13. There are five active regions. One of the active regions contains the quadtree blocks 1, 2, 3, 4, 6, 7, 8, 10, and 11. The remaining four regions consist of a single quadtree block: 5, 9, 13, and 12. Two of the active regions—the region consisting of block 5 and the region consisting of block 9—will subsequently be merged into a single region.

The *partial boundary* of a region describes the boundary of the region as known at the current point in the scan. It consists of a list of *cycles*. One cycle, called the *principal cycle*, corresponds to the outer boundary of the region. The remaining cycles, which correspond to the boundaries of holes, are called *auxiliary cycles*. Each cycle consists of a collection of *boundary elements*. Each boundary element may be either a *bridge* or a *chain*. A *chain* is a contiguous set of edges that are known to form part of the boundary of the region. A *bridge* is an edge that coincides with a portion of the active border and forms part of the boundary of that portion of the region that has already been scanned. Notice that the principal cycle may consist of both bridges and chains, but the auxiliary cycles consist only of chains.

As an example of these concepts, consider Figure 3(b), which illustrates the partial boundaries of the regions that are active at the time depicted in Figure 3(a) (i.e., after block 13 of Figure 2 has been processed). The heavy lines represent the chains, and the thin lines represent the bridges. The partial boundary of region A at this point in the scan consists of a single cycle, which in turn consists of the following boundary elemements:

(1) the chain {(2, 8), (2, 6), (0, 6), (0, 0), (12, 0), (12, 4)},
(2) the bridge from (12, 4) to (10, 4),
(3) the chain {(10, 4), (8, 4), (8, 6)},
(4) the bridge from {(8, 6) to (8, 8),
(5) the bridge from (8, 8) to (6, 8),
(6) the chain {(6, 8), (6, 6), (4, 6), (4, 8)}, and
(7) the bridge from (4, 8) to (2, 8).

The partial boundaries of the remaining four regions, which have much simpler structure, are also illustrated in Figure 3(b).

Auxiliary cycles are necessary because of regions that have holes. For example, in Figure 2, after block 37 has been processed the partial boundary of region D consists of two cycles, the principal cycle and a single auxiliary cycle. The auxiliary cycle represents the boundary of the hole resulting from the existence of region E. It consists of the chain {(12, 10), (12, 6), (10, 6), (10, 10), (12, 10)}. Notice that the chain is ordered so that the region D is to its right.

The active regions are represented by a collection of records of type **region**. The records of this type are partitioned into equivalence classes, and each active region is represented by an equivalence class. When two regions are merged into a single region (for example, when block 22 is processed in Figure 2), the corresponding equivalence classes are merged (i.e., one consisting of block 9 and one consisting of blocks 5 and 21). The equivalence classes are maintained using the well-known UNION-FIND algorithm for disjoint set union [2]. This algorithm implements the equivalence classes as trees in which nodes are linked to their fathers (but not to their sons). Thus each active region is represented by a tree of records of type **region**. The partial boundary of a region is associated with the root of the tree of records of type **region** that represents the region. This record is called the *primary record* associated with the region.

4.3. *The Record Structures*. We now describe the record formats in more detail. Regions are represented by records of type **region**. Such a record has seven fields: REGCOLOR, FATHER, COUNT, BORDERCOUNT, PRINCIPALCYCLE, AUXCYCLEFRONT, and AUXCYCLEREAR. These seven fields can be decomposed into three groups: color, data to support the union-find process, and partial boundary information.

Let r be a record of type **region**. REGCOLOR contains the color associated with the region. FATHER, COUNT, and BORDERCOUNT support the union-find process. FATHER(r) points to r's father, COUNT(r) is the number of proper descendants of r, and BORDERCOUNT(r) is the number of elements of the active border that reference r. The COUNT field supports weight balancing. A record r of type **region** may be safely reused if COUNT$(r) = 0$ and BORDERCOUNT$(r) = 0$.

A region consists of a principal cycle and a set of auxiliary cycles. The elements of the partial boundary of a region can be parts of any of these cycles. The partial boundary of a region is represented by the three fields: PRINCIPALCYCLE, AUXCYCLEFRONT, and AUXCYCLEREAR. PRINCIPALCYCLE points to a boundary element in the circular list of boundary elements that comprise the principal cycle. The auxiliary cycles are maintained as a linked list with AUXCYCLEFRONT and AUXCYCLEREAR pointing to its front and rear, respectively.

A list of auxiliary cycles is represented by a record of type **cyclelist** with two fields, FIRSTBOUNDARYELEMENT and NEXTCYCLE. FIRSTBOUNDARYELEMENT points to a boundary element in the circular list of records of type **boundaryelement** that comprise the cycle. NEXTCYCLE points to the next cycle in the list.

Each boundary element e is represented by a record of type **boundaryelement**. The fields FLINK and PLINK are always present and point at the successor and predecessor of e in the circular list of boundary elements comprising the cycle to which e belongs. It also has either two additional fields FRONT and REAR if it corresponds to a chain, or one additional field REG if it corresponds to a bridge. If e corresponds to a bridge, then it contains an additional field, REG, which points to a record of type **region** in the tree of records representing the region to whose boundary e belongs. If e corresponds to a chain, then it contains two pointers, FRONT and REAR, to the front and rear of a singly linked list of records containing the vertices that comprise the chain. Both front and rear pointers are used because new vertices may be added to either the front or the rear of the chain. The list is singly linked because vertices are never deleted from the middle of a chain.

The chain of vertices comprising a boundary element is represented by a record of type **chain** with two fields DATA and NEXT with the obvious meaning. Each vertex is represented by a record of type **vertex** with two fields X and Y corresponding to the x and y coordinate values of the vertex, respectively.

Element e of the active border is represented by a record of type **activeborderelement**. NEXT and PREV are link fields that support the doubly linked list of elements, ordered from southwest to northeast. LEN(e) contains the length of e. HORIZONTAL(e) is a Boolean value that is true if e is horizontal, and false if e is vertical. Being an element of the active border, e is a bridge in the partial boundary of the region immediately to the left of e (if e is vertical) or immediately above e (if e is horizontal), and DATA(e) points to the **boundaryelement** record for this bridge.

5. Algorithm.

The boundary extraction algorithm assumes a DF-expression representation of a quadtree. The quadtree is stored on disk and read sequentially, thus minimizing the number of page faults. Each quadtree block is processed exactly once. The main routine is the recursive procedure TRAVERSE. The parameters to TRAVERSE enable it to locate the appropriate element in the active border and to keep track of the size and location of the current quadtree block. When TRAVERSE encounters a leaf block P, it calls PROCESSLEAFBLOCK to process P. Procedure PROCESSLEAFBLOCK, in turn, calls PROCESSBORDERELEMENT to process each active border element that is adjacent to P.

In the following discussion it is important to remember that the entire quadtree is not stored in memory. Instead, only the active border and current region structures are stored in memory. The quadtree is processed one block at a time. Each time TRAVERSE is invoked, it gets the next element from the DF-expression by calling the function GET. If the corresponding quadtree block, say P, is gray, TRAVERSE calls itself recursively to process each of the four sublocks of P. Otherwise, P is a leaf block, in which case PROCESSLEAFBLOCK is called. Variables XLEFT, YTOP, and SIZE keep track of the position of the upper-left corner and the size of block P.

We describe the algorithm in several stages. First, we describe how procedures TRAVERSE and PROCESSLEAFBLOCK are able to find the appropriate entries in the active border list (Section 5.1). We then describe how these two routines process each quadtree block (Section 5.2). Next, we describe how PROCESSBORDERELEMENT processes each active border element (Section 5.3). In Section 5.4 we describe one of the primitive operations of the boundary extraction algorithm, namely adding an edge to a chain. A complete specification of the algorithm, in the form of pseudocode for the high-level routines and informal descriptions of the lower level routines, is presented in the Appendix.

5.1. *Keeping Track of Position in the Active Border List.*

Whenever a leaf block is processed, the portion of the active border that is adjacent to the block must be located. This is accomplished as follows. When TRAVERSE is called, it is passed a pointer to the uppermost active border element along the left border of the block about to be processed (UPPERLEFT). When TRAVERSE calls PROCESSLEAFBLOCK, it passes this pointer. By following PREV and NEXT links, PROCESSLEAFBLOCK can find all active border elements adjacent to the block in $O(1)$ time per element.

Whenever TRAVERSE and PROCESSLEAFBLOCK are called, they return two pointers, UPPERRIGHT and PREVLOWERLEFT. UPPERRIGHT is the uppermost active border element along the right border of the processed block, *after* it has been processed and the active border has been updated. PREVLOWERLEFT is the predecessor, in the active border element list, of the first element that is adjacent to the processed block. For example, when PROCESSLEAFBLOCK is called to process block 31 in Figure 2, UPPERLEFT points to the active border element separating block 31 from block 24. After PROCESSLEAFBLOCK completes, UPPERRIGHT and PREVLOWERLEFT point, respectively, to the active border elements separating block 31 from block 32 and block 28 from block 33. As another example, let P be the gray block whose sons are the leaf blocks 39, 40, 41, and 42. After TRAVERSE is finished processing block P, PREVLOWERLEFT points to the bottom border of block 30, and UPPERRIGHT points to the active border elements separating block 40 from block 43.

With these definitions, it is easily seen that the following four relations hold for any gray block P.

1. UPPERLEFT(NW(P)) = UPPERLEFT(P).
2. UPPERLEFT(NE(P)) = UPPERRIGHT(NW(P)).
3. UPPERLEFT(SW(P)) = PREVLOWERLEFT(NW(P)).
4. UPPERLEFT(SE(P)) = UPPERRIGHT(SW(P)).

The fact that the topological sweep processes the blocks of the quadtree in the order northwest, northeast, southwest, southeast ensures that the correct value of UPPERLEFT is always passed to each invocation of TRAVERSE and PROCESSLEAFBLOCK.

5.2. *Processing Blocks*. Procedures TRAVERSE and PROCESSLEAFBLOCK combine to process each block as follows. First, TRAVERSE checks whether either the left or top border of the block is properly contained in an active border element. This situation can arise if the block is smaller than its western or northern neighbor. For example, in Figure 2 when block 3 is processed, its top border is properly contained in an active border element, namely the entire bottom border of block 1. In this case, the left and/or top border element is split to achieve the desired size. Note that the split operation could be applied several times. In particular, if the side length of the neighboring block was four times that of the current block, then two splits would be required. If the block is a gray block, then TRAVERSE is invoked recursively for each of the four sons. If the block is a leaf block, then PROCESSLEAFBLOCK is called.

PROCESSLEAFBLOCK processes a leaf block P by first walking down the left side of P until it finds LOWERLEFT, the lowermost active border element adjacent to the left border of P. Next, it calls ALLOCATENEWREGION to allocate a new region descriptor CURREG corresponding to the region to which P belongs. The region descriptor returned by ALLOCATENEWREGION has a border consisting of three bridges: the east border of P, the south border of P, and a third bridge, CURBRIDGE. CURBRIDGE represents the portion of the west and north border of P that has not yet been processed.

Next, PROCESSLEAFBLOCK processes the active border elements along the west and north border of P, following NEXT links (i.e., moving upward along the west border, then eastward along the north border). This is done with two loops, one for the west border and one for the north border. PROCESSLEAFBLOCK calls PROCESSBORDERELEMENT to process each active border element.

After the last call to PROCESSBORDERELEMENT, CURBRIDGE corresponds to a null boundary element, as the entire northern and western boundaries of P have been processed. Hence CURBRIDGE can be deleted from the principal cycle of the region containing

P. Finally, UPDATEACTIVEBORDER is called to delete from the active border those active border elements that have been processed in this invocation of PROCESSBLOCK (i.e., those along the western and north borders of P) and to replace them with two new active border elements, representing the southern and eastern borders of P.

5.3. *Processing Border Elements*. PROCESSBORDERELEMENT processes one border element. Throughout this section assume PROCESSBORDERELEMENT has been called to process the border element e, that PROCESSLEAFBLOCK is processing block P, and that Q is the block on the opposite side of e from P. There are four cases, illustrated in Figure 4. The left half of each figure shows the situation before the border element e is processed, and the right half shows the situation after e is processed. To simplify the following explanation, we introduce some terminology, corresponding to variables in the code. The "current region" (CURREG) is the region containing P. The "neighbor region" (NEIGHBORREG) is the region containing Q. The current bridge (CURBRIDGE) is labeled **c**, while the "neighbor bridge" (NEIGHBORBRIDGE), labeled **n**, is the bridge that forms the part of the neighbor region's partial boundary corresponding to the active border element e. Notice that in the code, some processing common to cases (b)–(d) is performed *after* the processing for the individual cases.

Case (a). P and Q are not the same color. In this case the boundary of the neighbor region must be updated to reflect the fact that the border between P and Q is part of its boundary (i.e., part of a chain). A similar update must be made to the boundary of the current region. This is done by two calls to NEWCHAINEDGE, discussed in Section 5.4, below. After this is done, the routine CHECKFOROUTPUT is called to determine whether the neighbor region's boundary is complete. (This can be determined by checking whether the two fields COUNT(NEIGHBORREG) and BORDERCOUNT(NEIGHBORREG) are both zero.) If so, the boundary of the neighbor region can be written to the output file, and all the storage used by the neighbor region and its cycles can be reclaimed.

Case (b). P and Q are the same color, but they do not belong to the same region. In this case the current region and the neighbor region must be merged. This merging is done in two phases. First, the "union" portion of the standard union-find algorithm is executed. Next, the partial boundaries of the two regions are merged. This requires appending the auxiliary cycle list of the "losing" region (the one that did not become the root) to the auxiliary cycle list of the "winning" region. It also requires readjusting pointers to combine the two principal cycles, and deleting the neighbor bridge from the border of the (newly combined) region.

Case (c). P and Q already belong to the same region (i.e., the neighbor region and the current region are the same region) and a new auxiliary cycle has not been detected. This case is characterized by CURBRIDGE being immediately followed by NEIGHBORBRIDGE in a traversal of the partial boundary of CURREG. In this case all that is necessary is to delete NEIGHBORBRIDGE from the partial boundary of CURREG.

Case (d). P and Q already belong to the same region (i.e., the neighbor region and the current region are the same region) and a new auxiliary cycle has been detected. In this case a new auxiliary cycle is formed by "pinching off" the portion of the principal cycle between NEIGHBORBRIDGE and CURBRIDGE.

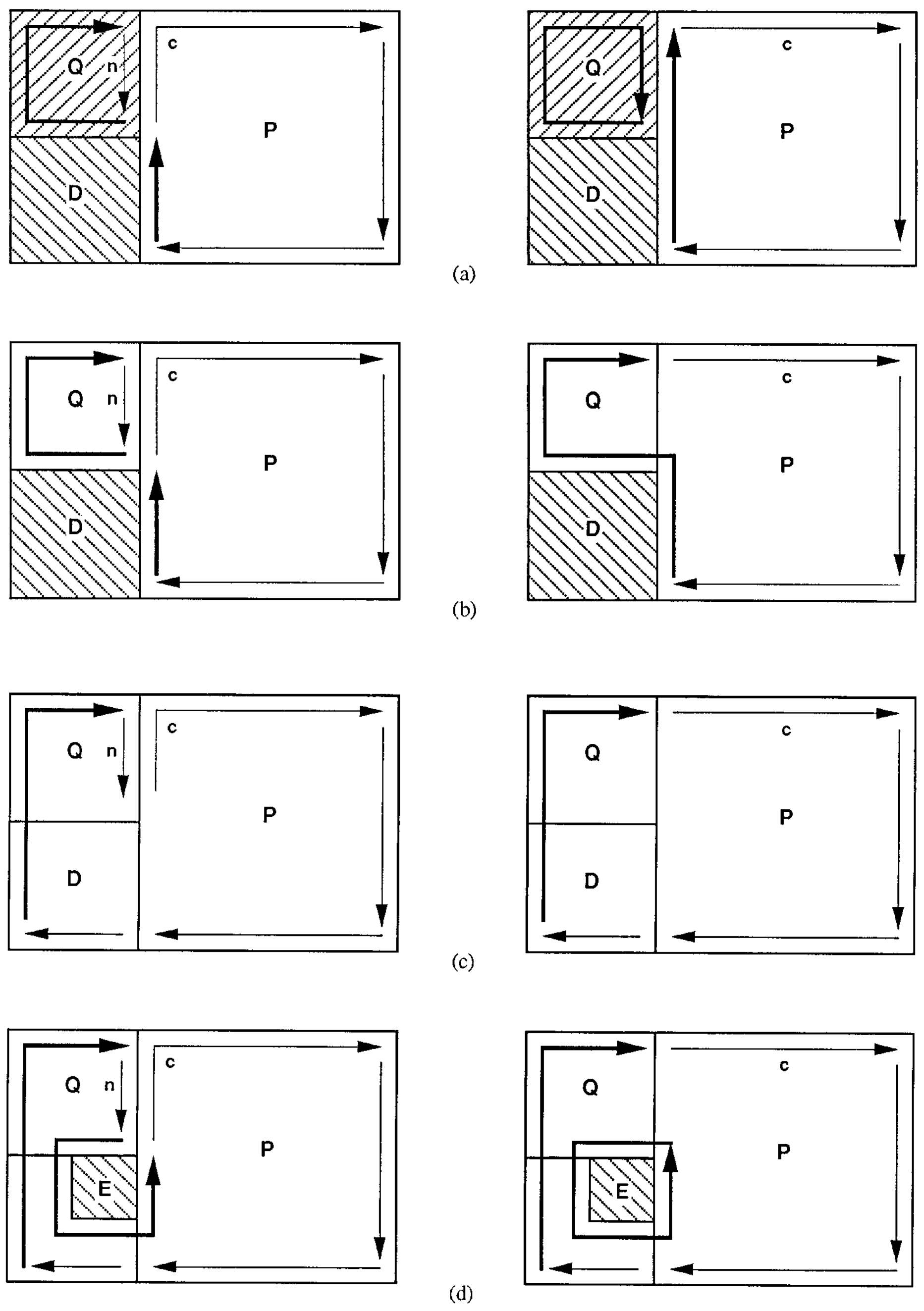

Fig. 4. The four cases that arise in PROCESSLEAFBLOCK. (a) The current region and the neighbor region are different colors. (b) The current region and the neighbor region are the same color and have not previously been merged (i.e., they are different regions). (c) The current region and the neighbor region are the same region and a hole has not been detected. (d) The current region and the neighbor region are the same region and a hole has been detected.

5.4. *Adding Edges to Chain.* Edges are added to chains by the procedure NEWCHAINEDGE. The goal of efficient use of storage, both in program memory and in the output file, dictates that chains must be stored in an efficient manner. For this reason, chains are coalesced as follows. When a new edge is placed at the beginning (resp. end) of a chain, if the new edge and the first (resp. last) edge on the chain are both horizontal or vertical, then the first (resp. last) vertex on the chain is replaced, otherwise a new vertex is added to the chain. Similar considerations apply when two chains are merged.

For example, consider Figure 2. Immediately before block 15 is processed, one of the chains in the boundary of region A is the chain

$$\{(2,8),(2,6),(0,6),(0,0),(12,0),(12,4),(8,4),(8,6)\}.$$

When block 15 is processed, the procedure NEWCHAINEDGE is called to add the edge from $(8,6)$ to $(8,8)$ to this chain. Because this new edge and the last edge on the chain (from $(8,4)$ to $(8,6)$) are both vertical, we *coalesce* these two edges to form a single edge from $(8,4)$ to $(8,8)$ and store the coalesced edge instead.

A more precise statement of the behavior of NEWCHAINEDGE is as follows. When a new edge is added, we know its predecessor and successor along the cycle to which it is being added. If these are both chains, then the new edge will cause the two adjacent chains to be merged. If either the successor or the predecessor (but not both) is a chain, then a new edge is added to that chain. In all these cases we coalesce if possible, as indicated above. If neither the successor nor the predecessor is a chain, then a new chain consisting of a single edge is created.

6. Correctness of the Algorithm.

The correctness of the algorithm follows from the fact that the following properties (P1)–(P7) are preserved by each call to PROCESSLEAFBLOCK.

(P1) The variable CURREG points to the root of the tree of records of type **region** representing the region containing P. This record is called the *primary record* associated with the region.

(P2) The primary record associated with a region (together with its corresponding pointers) contains a complete description of the partial boundary of the region.

(P3) The variable CURBRIDGE points to a bridge corresponding to that portion of the north and west border of P that has not yet been visited. Initially, this bridge corresponds to the entire north and west border. When P has been processed in its entirety, the bridge is null and must be deleted.

(P4) The partial boundary of an active region consists of a collection of cycles. More precisely, it is the boundary of the scanned portion of the region. Some edges of the partial boundary are on the boundary of the scanned portion of the image; these edges are bridges. The remaining edges of the partial boundary belong to chains.

(P5) The principal cycle of an active region may contain both bridges and chains. Let r be the primary record associated with a region. If either COUNT$(r) > 0$ or BORDERCOUNT$(r) > 0$, then the principal cycle of the region contains at least one bridge. Otherwise (i.e., if COUNT$(r) = 0$ and BORDERCOUNT$(r) = 0$), the principal cycle consists only of a chain (and hence the boundary of the region is ready for output). Note that COUNT(r) can be nonzero when BORDERCOUNT(r) $= 0$. Such a situation arises when two regions, r and s, have been merged into r, and the active border elements that refer to s still exist. When a merge occurs, we do not update all entries of the active border, as this would be too time consuming.

(P6) Any auxiliary cycle of an active region consists only of chains. In other words, auxiliary cycles contain no bridges.

(P7) For any record r of type **region**, the fields COUNT(r) and BORDERCOUNT(r) contain, respectively, the number of descendants of r (in the appropriate union-find tree) and the number of **activeborderelement** records whose BRIDGEPTR field references a bridge that in turn points to r.

We first sketch an informal proof that these properties are correct. (P1) and (P2) follow because, when two **region** records are merged in PROCESSLEAFBLOCK, CURREG is assigned to the root of the newly merged tree and the partial boundaries are immediately merged. (P3) and (P4) can be verified by straightforward induction arguments, which we omit. (P5) follows because every active border element is associated with some region. Indeed, if r is the primary record of type **region** associated with a region, any active border element associated with that region is either associated directly with r (in which case BORDERCOUNT(r) > 0) or with some descendant of r (in which case, it follows that r has at least one descendant, so COUNT(r) > 0). (P6) follows from (P4), since once a “hole” is identified, all blocks inside it (and all their neighbors) have been visited, so none of the edges on the boundary of the hole can be on the boundary of the scanned portion of the image (which means they cannot be bridges). Finally, (P7) can be proved by induction, as it is preserved at each place in the code where the union-find structure is altered or the values of the COUNT and BORDERCOUNT fields are modified.

The correctness of the algorithm can now be seen as follows. By (P4), the partial boundary of each region, as maintained in the primary record region, is correct. By (P5), the check described in CHECKFOROUTPUT for determining when a region’s boundary is complete is correct. By (P7), when this check is performed, the requisite fields (COUNT and BORDERCOUNT) have been correctly maintained. Hence each region’s boundary is correctly maintained and is written to the output file when it is available.

7. Analysis of the Algorithm.

The total time required by the algorithm is $O(M\alpha(M))$, where M is the number of quadtree blocks and $\alpha(\cdot)$ is the (extremely slowly growing) inverse of Ackerman’s function. This bound follows from the following, easily verified facts:

1. There are $O(M)$ calls to TRAVERSE and exactly M calls to PROCESSLEAFBLOCK.
2. The number of calls to PROCESSBORDERELEMENT (and the total number of iterations of the **while** loop in PROCESSLEAFBLOCK, over all calls) is $O(M)$. (This follows because each block has four edges, so the total number of active border elements that are created during an entire run of the algorithm is at most $4M$).
3. The total costs of all the union-find processing is $O(M\alpha(M))$. This is because both weight-balancing and path-compression are used [2], 25.

While the $O(\alpha(M))$ overhead is undetectable in practice, it is an interesting theoretical question whether it can be eliminated. By using the age-balancing strategy described in

[7] this overhead can be eliminated for pixel arrays scanned in raster order, but it is open whether the same strategy works for quadtrees.

One of the novel features of our algorithm is that TRAVERSE keeps track of the current position in the active border, so there is never any need to search the active border for the border of a block. By avoiding this search, we eliminate the potential log N factor in the time-complexity that would result from a more naive storage method.

The storage requirements of our algorithm are bounded by the sum of three factors, namely the costs of storing the active border elements, the active regions (i.e., all the records of type **region**), and the partial boundaries. The cost of storing the active border is proportional to the number of active border edges, which is at most $2N$ in an $N \times N$ image and may be significantly less. The maximum number of regions active at any one time is never more than the number of blocks in the active border. The storage required by the partial boundaries is never more than the cost of storing the entire region. Thus, if the algorithm of [8] has enough storage to operate efficiently, our algorithm does also. In the next section we address the question of what to do if the partial boundaries do not all fit in primary storage. (Notice that, in this case, the algorithm of [8] would be forced to swap portions of the quadtree to external storage and would then generate many page faults due to the nonsequential way in which it accesses quadtree blocks.)

8. Conserving Memory.

The storage required by the algorithm is proportional to the length of the active border (which is $O(N)$ in an $N \times N$ image), plus the maximum storage required by the partial boundaries of active regions. This is $\Theta(M)$ in the worst case, where M is the number of quadtree blocks. If, at any time, the maximum storage required by the partial boundaries of active regions is small compared with M, then the total memory required by the algorithm will also be small with respect to M. (This happens, for example, if the map consists of regions of moderate size that do not have long, thin "fingers.") In contrast, the algorithm of [8] always requires $\Theta(M)$ memory (to hold the map).

In the worst case the storage required by the partial boundaries of active regions may be proportional to the number of blocks in the entire quadtree. An example, consisting of a single region with a very long and sinuous boundary, is shown in Figure 5. In this

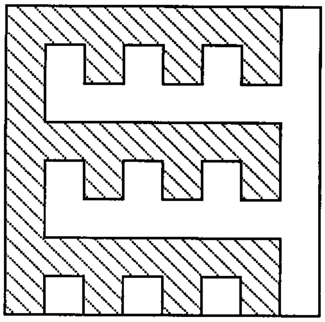

Fig. 5. In this $N \times N$ image, the algorithm requires $\Omega(N^2)$ storage.

case our algorithm as described above would require $\Theta(M)$ memory. Here we briefly describe a modification that permits our algorithm to run with less memory, at the cost of some performance degradation. This modification is only of interest in situations where the size of the map exceeds available memory; in these situations the algorithm of [8], which assumes the map fits in memory, cannot be used.

Only the first two and last vertices of a chain are relevant to the determination of how to coalesce when adding edges to a chain. (Two vertices are necessary because we need to know whether the first or last edge is horizontal or vertical.) Thus long chains may be written (swapped) to an auxiliary file, provided the first two vertices and last two vertices are kept in memory. If this is done, then it is necessary to keep a pointer to the beginning and end of the swapped chain. This can be done by adding a new variant type to the **boundaryelement** record. When a chain is swapped to the auxiliary file, the **boundaryelement** record corresponding to that chain is modified to represent a swapped chain, and all the **vertex** records associated with the portion of the chain that has been swapped may be reused. When a region boundary is output, each swapped chain in the boundary causes a seek operation and some additional reading from the auxiliary file. Thus primary storage cost is reduced, at the cost of some additional input/output.

The above technique suggests the following strategy. Let K be the total amount of available memory. As long as memory usage remains well below K, simply run the algorithm. If memory usage gets within some critical value (say 90% of K), start swapping out long chains. The threshold value for a "long chain" and the exact definition of the critical value will depend on the characteristics of the hardware. This approach permits very large maps with arbitrarily shaped regions to be processed, although performance will degrade as the number of chains that have been swapped out increases.

9. Concluding Remarks.

We have shown how to adapt a variant of the plane-sweep paradigm known as topological sweep to solve geometric problems involving two-dimensional regions when the underlying representation is a region quadtree. The utility of this technique was illustrated by showing how it can be used to extract the boundaries of a map in $O(M)$ space and $O(M\alpha(M))$ time, where M is the number of quadtree blocks in the map, and $\alpha(\cdot)$ is the (extremely slowly growing) inverse of Ackerman's function. The algorithm works for maps that contain multiple regions as well as holes. The algorithm represents a considerable improvement over a previous approach, based on boundary-following, which processes regions one at a time and whose worst-case execution time is proportional to the product of the number of blocks in the map and the resolution of the quadtree (i.e., the maximum level of decomposition). The algorithm works for many different quadtree representations including those where the quadtree is stored in external storage.

Directions for future work include the relaxation of the restriction that the boundaries be rectilinear, as is the case when the map is represented as a PM quadtree [23]. Other applications include its use with quadtree representations of other types of geometric data such as rectangles, points, lines, and even three-dimensional regions. It is an open question whether the algorithm can be improved to get rid of the $\alpha(M)$ factor in the algorithm's execution time.

Appendix. The Boundary Extraction Algorithm

```
recursive procedure TRAVERSE(UPPERLEFT,UPPERRIGHT,PREVLOWERLEFT,XLEFT,
YTOP,SIZE);
  /* Recursive routine to extract all region boundaries in a quadtree stored as a DF-
  expression. On input, UPPERLEFT is the uppermost activeborderelement along the
  left border of the current quadtree block. On output, UPPERRIGHT is the uppermost
  activeborderelement along the right border, and PREVLOWERLEFT is the immediate
  predecessor of the lowermost activeborderelement along the left border of the
  current quadtree block. XLEFT, YTOP, and SIZE describe the current quadtree block's
  location and extent. */
value pointer activeborderelement UPPERLEFT;
reference pointer activeborderelement UPPERRIGHT,PREVLOWERLEFT;
value integer XLEFT,YTOP,SIZE;
pointer activeborderelement PLL,UR,DUMMY;    /* local variables */
color BLOCKCOLOR;
begin    /* TRAVERSE */
  if LEN(UPPERLEFT) > SIZE then SPLIT(UPPERLEFT,SIZE);
  if LEN(NEXT(UPPERLEFT)) > SIZE then SPLIT(NEXT(UPPERLEFT), SIZE);
  BLOCKCOLOR := GET();
  if BLOCKCOLOR = "GRAY" then begin
    TRAVERSE(UPPERLEFT, UR, PLL, XLEFT, YTOP, SIZE/2);    /*NW son*/
    TRAVERSE(UR, UPPERRIGHT, DUMMY, XLEFT + SIZE/2, YTOP, SIZE/2); /*NE son*/
    TRAVERSE(PLL, UR, PREVLOWERLEFT, XLEFT, YTOP + SIZE/2, SIZE/2); /*SW son*/
    TRAVERSE(UR, DUMMY, DUMMY, XLEFT + SIZE/2,
      YTOP + SIZE/2, SIZE/2);    /* SE son */
    end
  else
    PROCESSLEAFBLOCK(UPPERLEFT, UPPERRIGHT, PREVLOWERLEFT, XLEFT, YTOP,
      SIZE, BLOCKCOLOR);
end    /* TRAVERSE */;

procedure PROCESSLEAFBLOCK(UPPERLEFT, UPPERRIGHT, PREVLOWERLEFT, XLEFT,
  YTOP, SIZE, BLOCKCOLOR);
  /* Process a single leaf block, exploring all its adjacencies by first working down
  along the west border and then from left to right along the north border.
  BLOCKCOLOR is the color of the leaf block, otherwise
  parameter definitions are as in TRAVERSE. */
value pointer activeborderelement UPPERLEFT;
reference pointer activeborderelement UPPERRIGHT, PREVLOWERLEFT;
value integer XLEFT, YTOP, SIZE;
value color BLOCKCOLOR;
integer X,Y
pointer region CURREG;
pointer boundaryelement CURBRIDGE, LOWBRIDGE;
pointer activeborderelement E,LOWERLEFT;
```

```
begin   /* PROCESSLEAFBLOCK */
  E := UPPERLEFT;
  Y := YTOP + LEN(E); X := XLEFT;
    while Y < YTOP + SIZE do begin
    E := PREV(E);
    Y := Y + LEN(E);
    end;
  PREVLOWERLEFT := PREV(E);
  LOWERLEFT := E;
  ALLOCATENEWREGION(XLEFT, YTOP, SIZE, BLOCKCOLOR, CURREG, CURBRIDGE,
    LOWBRIDGE);
  repeat
    PROCESSBORDERELEMENT(E, CURBRIDGE, CURREG, X, Y, X, Y − LEN(E));
    Y := Y − LEN(E);
    E := NEXT(E);
  until Y = YTOP;
  repeat
    PROCESSBORDERELEMENT(E, CURBRIDGE, X, Y, X + LEN(E), Y);
    X := X + LEN(E);
    if X < XLEFT + SIZE then E := NEXT(E);
  until X = XLEFT + SIZE;
  UPDATEACTIVEBORDER(LOWERLEFT, E, LOWBRIDGE, UPPERRIGHT, SIZE);
  BORDERCOUNT(CURREG) := BORDERCOUNT(CURREG) + 2;
  FLINK(PLINK(CURBRIDGE)) := FLINK(CURBRIDGE);
  PLINK(FLINK(CURBRIDGE)) := PLINK(CURBRIDGE);
  FREE(CURBRIDGE);
end   /* PROCESSLEAFBLOCK */;

procedure PROCESSBORDERELEMENT(E, CURBRIDGE, CURREG, X, Y, NEWX, NEWY);
  /* Process one border element. CURREG is the surviving region descriptor associated
  with the block being processed. */
reference pointer activeborderelement E;
reference pointer boundaryelement CURBRIDGE;
reference pointer region CURREG;
reference integer X, Y, NEWX, NEWY;
pointer boundaryelement NEIGHBORBRIDGE;
pointer region NEIGHBORREG;
begin   /* PROCESSBORDERELEMENT */
  NEIGHBORBRIDGE := DATA(E);
  NEIGHBORREG := REG(NEIGHBORBRIDGE);
  BORDERCOUNT(NEIGHBORREG) := BORDERCOUNT(NEIGHBORREG) − 1;
  NEIGHBORREG := FIND(NEIGHBORREG);
  if REGCOLOR(NEIGHBORREG) ≠ REGCOLOR(CURREG) then begin   /* Case (a) */
    NEWCHAINEDGE(X, Y, NEWX, NEWY, PLINK(CURBRIDGE), CURBRIDGE);
    NEWCHAINEDGE(NEWX, NEWY, X, Y, PLINK(NEIGHBORBRIDGE),
      FLINK(NEIGHBORBRIDGE));
```

```
        CHECKFOROUTPUT(NEIGHBORREG);
        end
    else begin   /* Cases (b), (c), and (d): regions are the same color */
        if NEIGHBORREG $\neq$ CURREG then begin   /* Case (b) */
            UNION(CURREG, NEIGHBORREG, CURREG, LOSER);
            NEXTCYCLE(AUXCYCLEREAR(CURREG)) := AUXCYCLEFRONT(LOSER);
            FLINK(PLINK(CURBRIDGE)) := FLINK(NEIGHBORBRIDGE);
            PLINK(FLINK(NEIGHBORBRIDGE)) := PLINK(CURBRIDGE);
            end
        else if PLINK(CURBRIDGE) = NEIGHBORBRIDGE then   /* Case (c): no–op */
        else   /* Case (d) */
            NEWAUXCYCLE(FLINK(NEIGHBORBRIDGE), PLINK(CURBRIDGE), CURREG);
        /* Cases (b), (c), and (d) */
        PLINK(CURBRIDGE) := PLINK(NEIGHBORBRIDGE);
        FLINK(PLINK(NEIGBHBORBRIDGE)) := CURBRIDGE;
        FREE(NEIGHBORBRIDGE);
        end;
end   /* PROCESSBORDERELEMENT */;
```

References

[1] 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):1–13, October 1983.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[3] H. S. Baird. Fast algorithms for LSI artworks analysis. *Journal of Design Automation & Fault-Tolerant Computing*, 2:179–209, 1978.

[4] M. Ben-Or. Lower bounds for algebraic computation trees. *Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing*, pp. 80–86, Boston, MA, April 1983.

[5] J. L. Bentley. Algorithms for Klee's rectangle problems. Unpublished manuscript, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1977.

[6] J. L. Bentley and D. Wood. An optimal worst-case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29(7):571–576, July 1980.

[7] M. B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(3):253–280, April 1992. Also see Corrigenda, *Journal of the ACM*, 39(4):985–985, October 1992.

[8] C. R. Dyer, A. Rosenfeld, and H. Samet. Region representation: Boundary codes from quadtrees. *Communications of the ACM*, 23(3):171–179, March 1980.

[9] H. Edelsbrunner. A new approach to rectangle intersections: part I. *International Journal of Computer Mathematics*, 13(3–4):209–219, 1983.

[10] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. EATCS Monographs on Theoretical Computer Science, vol. 10. Springer-Verlag, Berlin, 1987.

[11] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing*, pp. 389–403, Berkeley, CA, May 1986.

[12] H. Freeman. Computer processing of line-drawing images. *ACM Computing Surveys*, 6(1):57–97, March 1974.

[13] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[14] E. Kawaguchi, T. Endo, and J. Matsunaga. Depth-first picture expression viewed from digital picture

processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(4):373–384, July 1983.

[15] D. T. Lee. Maximum clique problem of rectangle graphs. In F. P. Preparata, editor, *Computational Geometry*, pp. 91–107. Advances in Computing Research, vol. 1. JAI Press, Greenwich, CT, 1983.

[16] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[17] J. Nievergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, October 1982.

[18] F. P. Preparata and M. I. Shamos. *Computational Geometry*: *An Introduction*. Springer-Verlag, New York, 1985.

[19] H. Samet. Hierarchical representations of collections of small rectangles. *ACM Computing Surveys*, 20(2):271–309, December 1988.

[20] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[21] H. Samet and M. Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(3):229–240, March 1985.

[22] H. Samet and M. Tamminen. Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):579–586, July 1988.

[23] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985.

[24] M. Tamminen, Encoding pixel trees. *Computer Graphics, Vision, and Image Processing*, 28(1):44–57, October 1984.

[25] R. E. Tarjan, Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.