Title

Using topological sweep to extract the boundaries of regions in maps represented by region quadtrees

Authors

Dillencourt, Michael B.
Samet, Hanan

Publication Date

1991

Peer reviewed

Using Topological Sweep To Extract the Boundaries of Regions in Maps Represented by Region Quadtrees

Michael B. Dillencourt[1]
Department of Information and Computer Science
University of California
Irvine, California

Hanan Samet[2]
Computer Science Department
University of Maryland
College Park, Maryland

Technical Report 91-01
January, 1991

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.

Keywords and Phrases: computational geometry, topological sweep, plane sweep, region representation, boundary extraction, active borders, region quadtrees, computer graphics, image processing, geographic information systems

[1]The support of the Committee on Research of the University of California, Irvine is gratefully acknowledged.

[2]The support of the National Science Foundation under Grant IRI-88-02457 is gratefully acknowledged.

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 algorithms 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, 20, 21, 27]. 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 [25], 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. 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) is less attractive than one that computes it by exploiting properties of the space in which the region lies.

This distinction is frequently made in the characterization of computer graphics algorithms. In this case, it is one of distinguishing between object space and image space algorithms (e.g., [31]). For example, consider the task of labeling the connected components of an image (e.g., [7]). This can be done using a seed-filling approach (e.g., [22]) or a predetermined approach (e.g., [23, 26, 7]). The former traces connectivity through the entire image, while the latter systematically processes all adjacencies between similarly colored image elements and propagates equivalences via the use of techniques for processing equivalence classes such as the union-find algorithm [33].

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., [27]). This is a flexible representation which is based on sorting geometric data according to their relationship with the space that they occupy. We will 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 this 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 represents a

considerable 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 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 techniques 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[30].

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 giving an overview of our algorithm. Sections 4 and 5 describe the data structures and the new algorithm, while Section 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 pseudo-code for the algorithm.

2 Topological Sweep and Region Quadtrees

Plane sweep is one of the basic paradigms of computational geometry [3, 10, 20, 21, 27]. This technique consists of sweeping a straight line (called the *sweep line*) across the plane through a collection of objects. Without loss of generality, we may assume that a vertical line is being swept from left to right. At certain points, called halting points, a partial solution is computed. After the last halting point is encountered, a complete solution is available. A halting point usually corresponds to the first or last intersection of the sweep line with one of the objects.

The plane sweep technique requires maintaining two sets of data. The first set consists of the set of *halting points*, organized so that they are encountered in the appropriate order. Usually this set is maintained as a list of x-coordinate values sorted in ascending order. The second set consists of the set of *active objects*—i.e., those objects intersected by the current position of the sweep line. The set of active objects is updated at each halting point.

Generally, plane sweep algorithms consist of two phases: a *sort phase*, in which the halting points are sorted, and a *sweep phase*, in which the actual plane sweep is performed. Because sorting is involved, a plane-sweep computation involving n objects requires at least $\Omega(n \log n)$ processing time. In some applications (e.g., computing a convex hull, regularizing a polygonal map [18]), the sweep phase requires time linear (or almost linear) in n, so the total processing time is dominated by the sort phase.

In fact, under certain circumstances, a variant of plane sweep that eliminates the sort phase can be applied. In such a case, the sort phase is replaced by another process

which may be of a lower, equal, or greater complexity. Even when the cost is greater, the process may still be of interest if the preprocessing is amortized over many operations.

This variant, termed *topological sweep* in [11], can be applied when an appropriate combinatorial structure among the input objects is available. In topological sweep, the sweep line is a simple (i.e., non-self-intersecting) curve which need not be a straight line. By using the combinatorial structure to guide the sweep, the sort phase may be eliminated. In [11], topological sweep is used to compute certain properties of arrangements of lines and hyperplanes.

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, 19], and maximum clique in the intersection graph [17]. 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 [21]).

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 regular decomposition, such as the bintree and the region quadtree (e.g., [27]). 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 (non-leaf) 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 NW, NE, SW, SE 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 NW, NE, SW, SE order, then at any instant during the sweep, the sweep line is a "staircase" moving from southeast to northwest. It should be clear that the simple curve that forms the "staircase" is topologically equivalent to a line. The staircase is

termed the *active border*. 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 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" [25, 28, 29].

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 geographic information system (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 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, 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. 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.

The algorithm of [8] is similar to a seed-filling polygon coloring algorithm. It traverses the image in some order until finding a boundary (i.e., an adjacency between two region with a different color). Having found such an adjacency, it then proceeds to follow the border. The border is followed using neighbor finding techniques [24, 26]. The algorithm does not require any storage beyond that needed to hold the image. The execution time

of this algorithm is proportional to the product of the total perimeter of the regions and the average cost of locating a neighbor. The average neighbor-location cost may be as high as the resolution of the image, although for image models that capture the properties of real-world data, the average cost of neighbor-finding is independent of the resolution (i.e., maximum level of decomposition) of the image [24]. It is well-known that the number of blocks in a quadtree representation of a region is proportional to its perimeter [14, 15]. Thus the algorithm of [8], in practice, exhibits execution time that is proportional to the number of blocks in the image. However, its worst-case time is proportional to the product of the number of blocks in the image and the resolution. 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 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 NW, NE, SW, SE. 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 one of the regions to become inactive, in which case its boundary is written to the output file and it is removed from the active region data structure. 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.

The 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, and thus assigning holes to the right region is easy. In addition, there is no computational overhead in locating the appropriate spot in the active border when processing a block. This enables us to avoid the $\log n$ cost factor often associated with the sweep phase of algorithms based on plane-sweep. The fact that equivalences among regions must be

processed results in the $\alpha(M)$ overhead 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 width 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 previous algorithm 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), the previous algorithm [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 many different quadtree representations. Although the region quadtree was originally devised as a means of saving space, this is not always so. In particular, the space required for storing pointers from a node to its sons may be significant. This is especially true in a dynamic environment (in contrast to a static one). Of greater importance is the fact that when the quadtree is represented in external storage, processing pointer chains (e.g., as in neighbor finding which forms the heart of the algorithm in [8]) can be time consuming due to the presence of page faults. Consequently, there has been considerable interest in pointerless quadtree representations and in efficient algorithms for processing them.

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 [16, 32]. This approach is termed a DF-expression (DF standing for depth-first) in [16]. 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. For example, the DF-expression representing the quadtree shown in Figure 2 is

GG11G1121G1121G12G3332G3232G2G2G11222211GG3233G3232G32222,

where we assume that regions A and F have color 1; regions B, C, and E have color 2; and region D has color 3. A pointerless quadtree can be viewed abstractly as a sequential file that supports the single operation GET. Repeated calls to this operation produce, in sequence, the nodes of the tree. 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 [28], 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 doubly linked list of records of type **activeborderelement**, ordered from southwest to northeast.

4.2 The Active Regions

An *active quadtree block* is a block that has been processed but is adjacent to at least one block that has not been processed. 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 contiguous 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 4 regions each 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 light 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 elements:

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 $\{(6,8),(6,6),(4,6),(4,8)\}$;
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 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. Skeletal pseudocode definitions of the record structures are given in Table 1. 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 which makes use of a tree structure where nodes are linked to their fathers but not to their sons. 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 the weight balancing that needs to be done in order for union-find to achieve its execution-time bounds. In particular, when two regions are merged, the COUNT field is use to determine which of the two records of type **region** at the roots of the two trees becomes the root of the new combined tree. Together, the COUNT and BORDERCOUNT fields are used to determine when records of type **region** may be safely reused: r may be 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 is represented by a record of type **boundaryelement**. Such a record always has the two fields FLINK and PLINK. It also has either two additional fields fields FRONT and REAR if it corresponds to a chain, or one additional field REG if it corresponds to a bridge. Let e be a record of type **boundaryelement**. FLINK and PLINK are pointers to the successor and predecessor of e in the circular list of boundary elements comprising the cycle to which e belongs. 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.

```
type region is record
  color REGCOLOR;
  pointer region FATHER;
  integer COUNT, BORDERCOUNT;
  pointer boundaryelement PRINCIPALCYCLE;
  pointer cyclelist AUXCYCLEFRONT,AUXCYCLEREAR;
  end record;

type cyclelist is record
  pointer cyclelist NEXTCYCLE;
  pointer boundaryelement FIRSTBOUNDARYELEMENT;
  end record;


type boundaryelement is record
  pointer boundaryelement FLINK, PLINK;
  case ELEMENTTYPE: (BRIDGETYPE, CHAINTYPE) is
    CHAINTYPE: pointer chain FRONT, REAR;
    BRIDGETYPE: pointer region REG;
    end case;
  end record;

type chain is record
  pointer vertex DATA;
  pointer chain NEXT;
  end record;

type vertex is record
  integer X,Y;
  end record;

type activeborderelement is record
  pointer activeborderelement NEXT, PREV;
  integer LEN;
  Boolean HORIZONTAL;
  pointer boundaryelement DATA;
  end record;
```

Table 1: Definitions of the data structures

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. DATA is a pointer to the record corresponding to the vertex and NEXT points to the next record in the chain. 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.

Elements of the active border are represented by records of type **activeborderelement**. Such a record has five fields, NEXT, PREV, LEN, HORIZONTAL, and DATA. Let e be a pointer to 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. It processes each quadtree block 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 does not exist at any one time. Instead, only the active border and current region structures exist. 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 sub-blocks 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 two appendices.

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))

These relations, in turn, imply 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. If this situation occurs, the left and/or top border element is split as many times as necessary to achieve the desired size. If the block is a gray block, then TRAVERSE is called recursively, for each of the four sons. If the block is a leaf block, 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 upwards 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. Note that the western border could also have been processed simultaneously with the previous step that located LOWERLEFT. However, this would have made PROCESSBORDERELEMENT more complicated. PROCESSBORDERELEMENT, described in Section 5.3, ensures that the following properties hold:

(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 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.

(P3) 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.

(P4) The primary record associated with a region (together with its corresponding pointers) contains a complete description of the partial boundary of the region.

(P5) 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.

(P6) 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 don't update all entries of the active border, as this would be too time consuming.

(P7) Any auxiliary cycle of an active region consists only of chains. In other words, auxiliary cycles contain no bridges.

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 principle cycle of the region containing P. Finally, UPDATEACTIVECHAIN 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. To simplify the following explanation, we introduce some terminology, corresponding to variables in the code. The "current region" (CURREG) is the region containing P (note Property (P1) in Section 5.2). The "neighbor region" (NEIGHBORREG) is the region containing Q. The "neighbor bridge" (NEIGHBORBRIDGE) is the bridge that forms the part of the neighbor regions's partial boundary corresponding to the active border element e.

The 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. In each case, the current bridge (CURBRIDGE) is labeled **c**, and the neighbor bridge is labeled **n**. 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 0.) 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.

5.4 Adding Edges to Chains

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 (respectively, end) of a chain, if the new edge and the first (respectively, last) edge on the chain are both horizontal or vertical, then the first (respectively, 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, it is unnecessary to add a new vertex to the chain. Instead, the last vertex, $(8,6)$, can simply be replaced with the vertex $(8,4)$. In essence, rather than storing the two edges from $(8,4)$ to $(8,6)$ and from $(8,6)$ to $(8,8)$, we *coalesce*

these two edges to form a single edge from $(8,4)$ to $(8,8)$ and store the coalesced edge instead.

The skeletal code for NEWCHAINEDGE appears at the end of Appendix A. 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 via a call to MERGECHAINS. If either the successor or the predecessor (but not both) is a chain, UPDATECHAIN is called to add the new edge to that chain. In all these cases, we coalesce if possible, as indicated above. Finally, if neither the successor nor the predecessor is a chain, then we call ADDCHAIN to create a new chain consisting of a single edge.

6 Correctness of the Algorithm

The correctness of the algorithm follows from the fact that the properties (P1)–(P7) are preserved by each call to PROCESSLEAFBLOCK. We first sketch an informal proof that these properties are correct. (P1) and (P4) 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. (P2) and (P3) can be verified by straightforward induction arguments, which we omit. (P6) 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$(r) > 0$). (P7) follows from (P3), 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, (P5) 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 (P6), the check described in CHECKFOROUTPUT for determining when a region's boundary is complete is correct. By (P5), 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 func-

tion. 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 4 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 cost of all the union-find processing is $O(M\alpha(M))$. This is because both weight-balancing and path-compression are used [2, 33].

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. An alternative approach to maintaining the active border is presented in [28]; this approach uses two arrays of size N to keep track of the active border in an $N \times N$ image. By contrast, our approach stores the active border using an amount of memory proportional to the number of active border elements. Our approach is superior because the number of active border elements in an $N \times N$ image can never exceed $2N$, and in practice it is often much less. Our approach to representing the boundary is similar to the approach for bintrees proposed in [29].

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. As mentioned above, 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

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 is shown in Figure 5. Nevertheless, the algorithm can be made to run in less than $O(M)$ space, where M is the number of quadtree blocks, by adopting the following approach.

Only the first two and last two 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.

To implement this strategy, it is useful to have one additional field, CHAINLENGTH, in the CHAIN variant of the **boundaryelement** record, This field indicates the number of vertices in the chain, and makes it easy to detect long chains. It is perhaps worth noting that with careful implementation, the CHAINLENGTH field may be added without lengthening the **boundaryelement** record. This is because it is never actually necessary to follow the PLINK field of a chain. Thus the definition of the **boundaryelement** record may be modified so that the PLINK field is only associated with the BRIDGE variant. However, if this modification is made, then the code in Appendix A must be modified so that PLINK links are followed or set only when the **boundaryelement** record is indeed a bridge.

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 large images to be processed, although performance will degrade somewhat 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 [Same85]. 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.

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 artwork analysis. *Journal of Design Automation & Fault-Tolerant Computing*, 2:179–209, 1978.

[4] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on the Theory of Computing*, pages 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. To appear in *Journal of the ACM*.

[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*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1987.

[11] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. In *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing*, pages 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] G. M. Hunter. *Efficient computation and data structures for graphics*. PhD thesis, Princeton University, Princeton, NJ, 1978.

[15] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, April 1979.

[16] 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.

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

[18] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM Journal on Computing*, 6(3):594–606, September 1977.

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

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

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

[22] D. F. Rogers. *Procedural Elements for Computer Graphics.* McGraw-Hill, New York, 1985.

[23] A. Rosenfeld and A.C. Kak. *Digital Picture Processing.* Academic Press, New York, NY, second edition, 1982.

[24] H. Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):37–57, January 1982.

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

[26] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

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

[28] 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.

[29] 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.

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

[31] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, March 1974.

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

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

Appendix A: 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,CURREG,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 ≠ 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(NEIGHBORBRIDGE)) := CURBRIDGE;
      FREE(NEIGHBORBRIDGE);
      end;
  end /* PROCESSBORDERELEMENT */;

  procedure UNION(R,S,WINNER,LOSER);
  value pointer region R,S;
  reference pointer region WINNER,LOSER;
  begin /* UNION */
    if COUNT(R) ≥ COUNT(S) then begin
      WINNER := R; LOSER := S;
      end
    else begin
      WINNER := S; LOSER := R;
```

```
      end;
    FATHER(LOSER) := WINNER;
    COUNT(WINNER) := COUNT(WINNER) + COUNT(LOSER) + 1;
 end /* UNION */;

 pointer region FIND(R);
 value pointer region R;
 pointer region R1 := R, R2;
 integer PATHCOUNT := 0;
 begin /* FIND */
   while not NULL(FATHER(R)) do R := FATHER(R);
   while (R1 ≠ R) do begin
     R2 := FATHER(R1);
     /* PATHCOUNT contains value of COUNT(R1) from before start of path compression*/
     COUNT(R2) := COUNT(R2) - PATHCOUNT - 1;
     PATHCOUNT := PATHCOUNT + COUNT(R2) + 1; /* old COUNT(R2) */
     if COUNT(R1) = 0 and BORDERCOUNT(R1) = 0 then begin
       FREE(R1);
       COUNT(R) := COUNT(R)-1;
       end
     else
       FATHER(R1) := R;
       R1 := R2;
     end;
 end /* FIND */;

 procedure NEWCHAINEDGE(X1,Y1,X2,Y2,PRED,SUCC);
   /* Add a new edge from (X1, Y1) to (X2, Y2) */
 value integer X1,Y1,X2,Y2;
 value pointer boundaryelement PRED,SUCC;
 begin /*NEWCHAINEDGE */
   if ELEMENTTYPE(PRED) = CHAINTYPE then
     if ELEMENTTYPE(SUCC) = CHAINTYPE then MERGECHAINS(PRED,SUCC)
     else UPDATECHAIN(X1,Y1,PRED,FALSE)
   else
     if ELEMENTTYPE(SUCC) = CHAINTYPE then UPDATECHAIN(X1,Y1,SUCC,TRUE)
     else ADDCHAIN(X1,Y1,X2,Y2,PRED,SUCC);
 end /* NEWCHAINEDGE */;
```

Appendix B: The uncoded routines in the boundary extraction algorithm

ROUTINES THAT MANIPULATE THE ACTIVE BORDER:

procedure SPLIT(BORDEREDGE,SIZE);
value pointer activeborderelement BORDEREDGE;
value integer SIZE;
/* This routine splits the active border element BORDEREDGE into two parts, BORDEREDGE and NEWBORDEREDGE. It also splits the bridge DATA(BORDEREDGE) into two parts. The split of BORDEREDGE is done in such a way that after the split, LEN(BORDEREDGE) = SIZE. If BORDEREDGE is vertical, NEWBORDEREDGE becomes the *predecessor* of BORDEREDGE on the active border list. If BORDEREDGE is horizontal, NEWBORDEREDGE becomes the *successor* of BORDEREDGE on the active border list. */

procedure UPDATEACTIVEBORDER(LOWERLEFT,RIGHTUPPER,LOWBRIDGE,RIGHT,SIZE);
value pointer activeborderelement LOWERLEFT, RIGHTUPPER;
value pointer boundaryelement LOWBRIDGE;
reference pointer activeborderelement RIGHT;
value integer SIZE;
/* This routine is called to update the active border after a leaf block has been processed. It frees the portion of the active border between LOWERLEFT and RIGHTUPPER (i.e., the left and top border of the leaf block), and replaces this portion with two **activeborderelement** entries, corresponding to the bottom and right border of the leaf block. On entry, LOWBRIDGE points to the boundary element that is the bottom boundary of the leaf block. This information is necessary to correctly set the DATA field in the two new **activeborderelement** entries. On exit, RIGHT points to the active border edge corresponding to the right border of the block. */

ROUTINES THAT MANIPULATE REGIONS:

procedure ALLOCATENEWREGION(XLEFT,YTOP,SIZE,REGCOLOR,CURREG,CURBRIDGE,LOWBRIDGE);
value integer XLEFT, YTOP, SIZE;
value color REGCOLOR;
reference pointer region CURREG;
reference pointer boundaryelement CURBRIDGE,LOWBRIDGE;
/* This routine allocates a region consisting of the leaf block whose location and size are sepecified by XLEFT, YTOP, and SIZE, and whose color is REGCOLOR. A boundary description consisting of one component with three bridges is built. On output, CURREG points to the newly allocated region descriptor; CURBRIDGE is the "floating" bridge described in the text; and LOWBRIDGE is the bridge forming the bottom boundary of the block. */

procedure CHECKFOROUTPUT(REG)
value pointer region REG;
/* This routine checks to see if the region REG can be succesfully output. This can be done if and only both of the following conditions are met: (1) BORDERCOUNT(REG) = 0, and (2) COUNT(REG) = 0. If the region is output, then the region record and all associated cycles and chains are deleted, and their space is available for subsequent reuse. */

ROUTINE THAT MANIPULATES BOUNDARY ELEMENTS AND CYCLES:

procedure NEWAUXCYCLE(START,STOP,REG);
value pointer boundaryelement START,STOP;
value pointer region REG;
/* The chain of boundary elements from START to STOP is made a separate, non-principal component of the region REG. The calling routine is responsible for (1) ensuring that this sequence of boundary elements does, in fact, form a cycle; and (2) delinking START and STOP from the principal boundary component. */

ROUTINES THAT MANIPULATE CHAINS:

procedure ADDCHAIN(X1,Y1,X2,Y2,PRED,SUCC);
value integer X1,Y1,X2,Y2;
value pointer boundaryelement PRED, SUCC;
/* Make a new chain, consisting of the two vertices (X1, Y1) and (X2, Y2), and link it so that it is after PRED and before SUCC. */

procedure UPDATECHAIN(X,Y,CH,ATFRONT);
value integer X,Y;
value pointer boundaryelement CH;
value Boolean ATFRONT;
/* Put the vertex (X,Y) on the front or rear of the chain pointed to by CH, depending on whether ATFRONT is true or false. Do this by replacing the first (last) vertex if possible, otherwise add a new first (last) vertex. */

procedure MERGECHAINS(PRED,SUCC);
value pointer boundaryelement PRED, SUCC;
/* Merge the two chains pointed to by PRED and SUCC into a single chain. If PRED ends with a horizontal (respectively, vertical) edge and SUCC starts with a horizontal (respectively, vertical) edge, coalesce these two edges. */

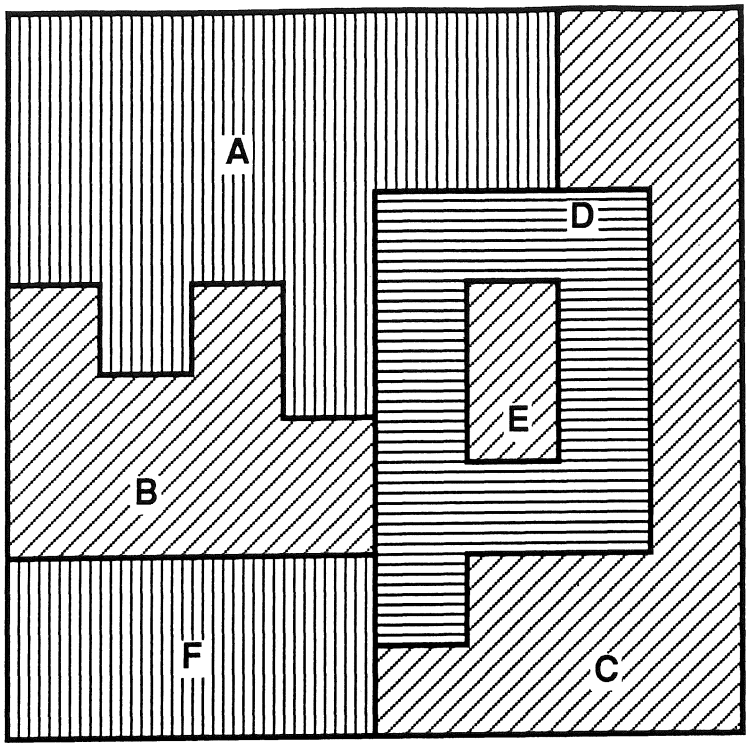


Figure 1: A sample map, consisting of five regions

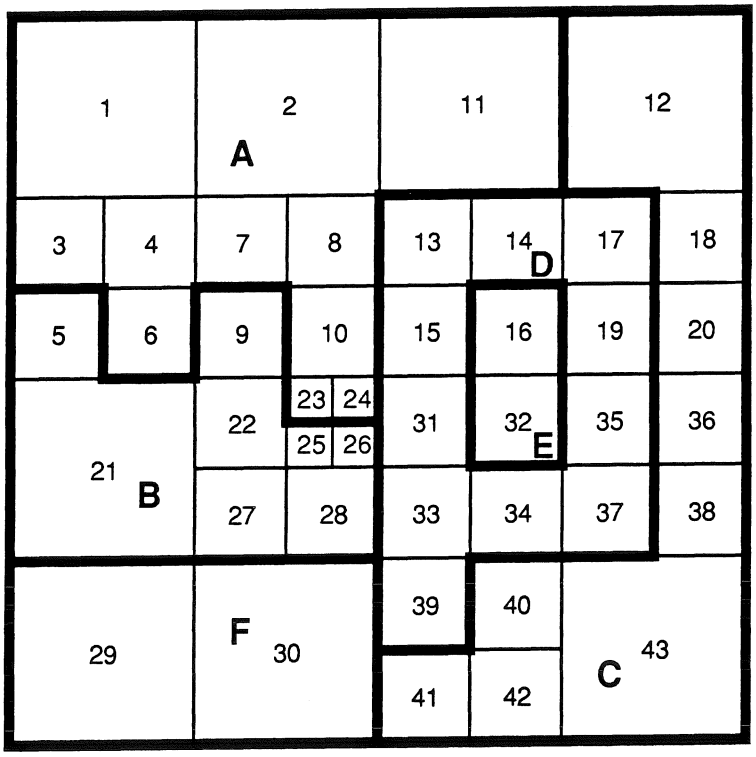
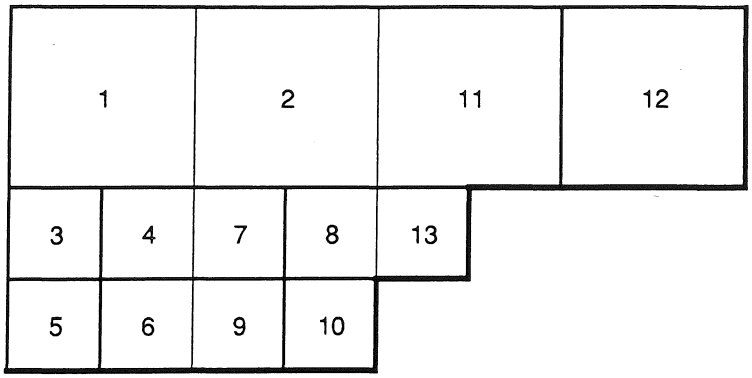
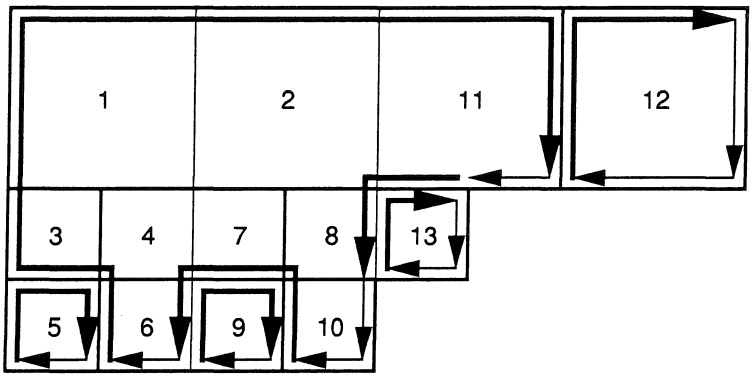


Figure 2: The decomposition of the map of Figure 1 into homogeneous blocks



(a)



(b)

Figure 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.

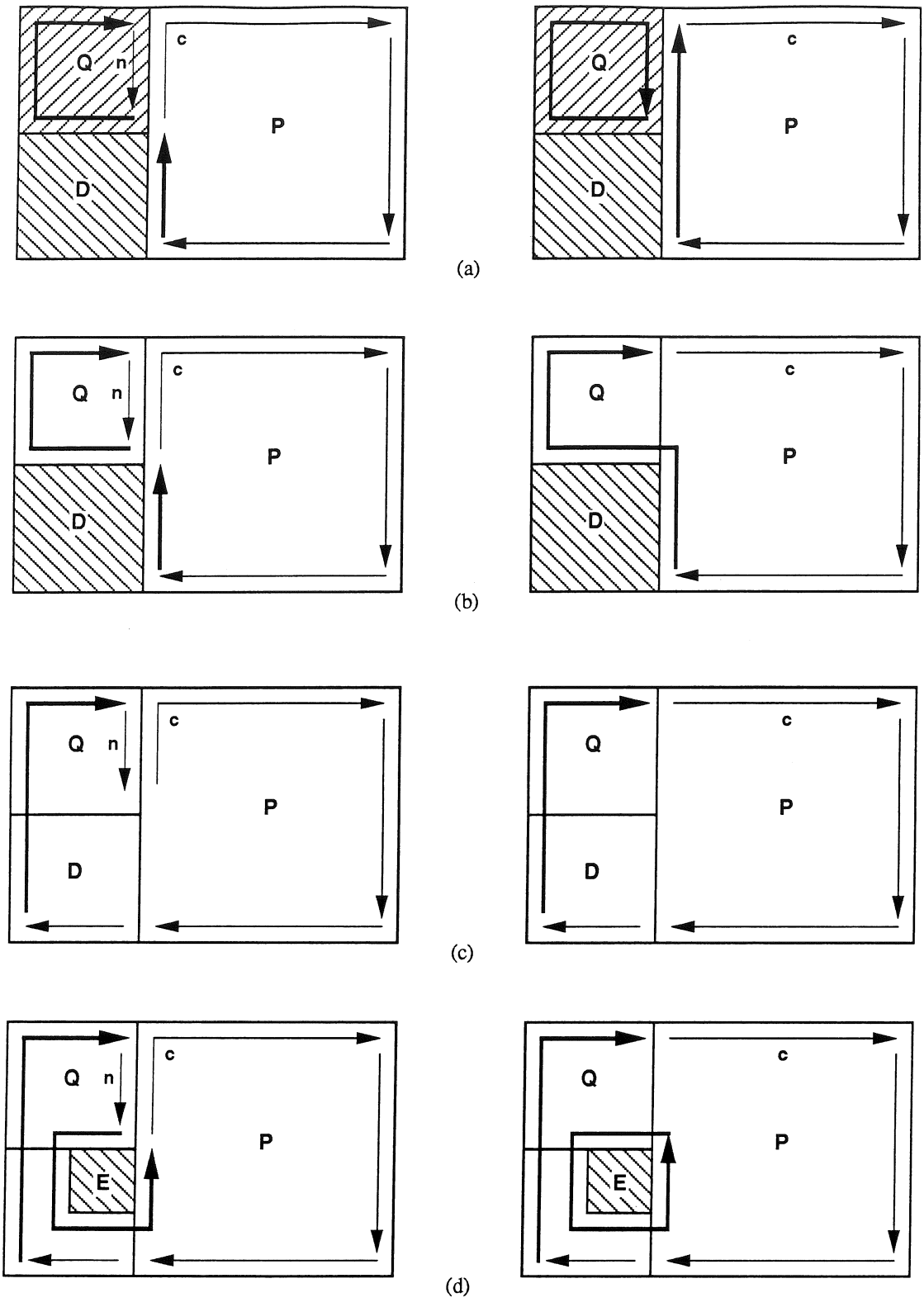


Figure 4: The four cases that arise in PROCESSLEAFBLOCK. (a) The current region and the neighbor region are different colors. (b) The current region and the neighbor region are the same color and have not previously been merged (i.e., they are different regions). (c) The current region and the neighbor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

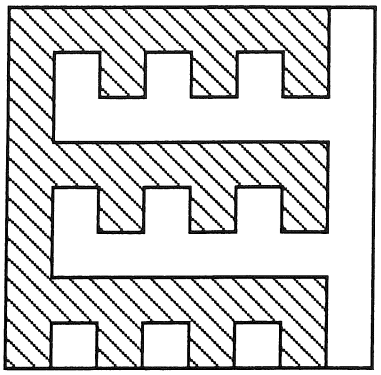


Figure 5: In this $N \times N$ image, the algorithm requires $\Omega(N^2)$ storage