WEBBER [SAME82c]

TR-1162
DAAK 70-81-C-0059

February, 1982

# On Encoding Boundaries with Quadtrees

Hanan Samet
Robert E. Webber

Computer Science Department
and
Computer Vision Laboratory
Computer Science Center
University of Maryland
College Park, MD 20742

## ABSTRACT

A new data structure termed a line quadtree is presented that encodes both a region and its boundary in a hierarchical manner. It is similar to the quadtree with the modification that with each node information is stored as to which of its sides are adjacent to a boundary. The information is available for both terminal and non-terminal nodes. Such a data structure is shown to facilitate a number of basic region processing algorithms including boundary following. In particular, an efficent algorithm is demonstrated for superimposing one map on top of another map when the maps are encoded using the line quadtree.

The support of the Engineering Topographic Laboratories (under contract DAAK 70-81-C-0059) is gratefully acknowledged, as is the help of Janet Salzman in preparing this manuscript and Sherry Palmer for drawing the figures.

## 1. Introduction

Region representation plays an important role in image processing and geographic information systems. There are a number of different data representations in use. They range from point by point representations such as binary arrays and run length coding [12] to vector representations such as chain codes [4] and even hierarchical representations such as quadtrees [8] and strip trees [1]. In fact, variations on hierarchical representations (e.g., medial axis transforms [18]) also exist.

The quadtree [8] is an approach to region representation that is based on the successive subdivision of an image array (for this definition assume that the image consists of foreground, i.e., BLACK, and background, i.e., WHITE) into quadrants. If the region does not cover the entire array (or miss the entire array), we subdivide the array, and repeat this process for each quadrant, each subquadrant, ..., as long as necessary, until we obtain blocks (possibly single pixels) that are entirely contained within the region or entirely disjoint from it. In other words, our partition process results in squares whose sides are powers of two and that are either entirely BLACK or entirely WHITE. For example, the resulting blocks for the region of Figure 1a are shown in Figure 1b. This process can be represented by a tree of degree 4 in which the entire array is the root node, the four sons of a node are its quadrants (labeled in order NW, NE, SW, SE), and the leaf nodes correspond to those blocks for which no further subdivision is necessary. Leaf nodes are said to be BLACK or WHITE depending or whether their corresponding blocks are in the foreground or background while non-leaf nodes are said to be GRAY. The quadtree representation for Figure 1b is shown in Figure 1c.

In this paper, we address the issue of hierarchically representing images which are segmented into a number of different regions rather than mere foreground and background as is done in work relating to quadtrees. In particular, we are interested in a representation that encodes both the region and its border in a hierarchical manner. This is in contrast to the conventional quadtree which only encodes areas in a hierarchical way and the strip tree [1] which only encodes curves in a hierarchical manner. We achieve our goal by introducing a new data structure termed a "line quadtree" and show how it can be used to encode boundaries. Our approach is an iterative one and shows the reader the various design decisions which led to our adoption of a particular definition for the data structure. The ultimate data structure is one that preserves the hierarchical nature of the region quadtree and also introduces a hierarchical boundary representation. We conclude our presentation by showing how the line quadtree facilitates both boundary following algorithms as well as the more traditional postorder tree traversal quadtree algorithms. The latter is done in the context of a algorithm to implement the common problem of superimposing one map on top of another map.

## 2. Line Quadtrees

Two different approaches can be made to the task of defining the quadtree data structure. The first approach is top-down and recursively subdivides a space (to simplify the presentation, we confine the discussion to the Euclidean plane) until some uniformity criteria is met. The second approach is bottom-up and recursively merges portions of a fragmented (digitized) space that meet some compatibility criteria. The second approach is used below.

Planar geometry populates the plane with three distinct entities - the point, the line, and the region. Finkel and Bentley [3] present the application of the quadtree data structure to the problem of storing a set of points. Klinger [8] (also [9,10]) introduced the use of quadtrees to store regions. Such quadtrees are henceforth called region quadtrees. Subsequent work included applications to graphics [5,6,7] as well as interconversions with other representations such as chain codes [2,13], rasters [14,15] and binary arrays [16]. Herein, we present an application of the quadtree data structure to the problem of storing line segments that partition a plane.

Since the partitions bounded by the line segments correspond to regions, the same data structure can be used for region-oriented problems. This is analogous to being able to represent a map by giving each region a separate color or by just drawing the boundary lines. We could view the latter representation as a two color region map where the boundary lines become narrow BLACK regions and the remainder of the map is colored WHITE. This is awkward for two reasons: (1) narrow regions are costly in terms of the number of nodes in the quadtree, and (2) it commits the user to a specific thickness of the boundary line which may be unfortunate when regenerating the picture on an output device.

The bottom-up definition of a region quadtree starts with space fragments, called pixels, that have a single property: its color. The bottom-up definition of a line quadtree, the data structure proposed herein, starts with pixels that record the presence or absence of an edge on each of the four sides of a pixel. The line quadtree is subsequently built by merging the nodes of a complete 4-ary tree where each leaf node corresponds to a pixel.

It should be noted that this is quite different from the edge quadtree of Shneier [19]. Edge quadtrees store (at each leaf) linear approximations of idealized curves with explicit error factors, whereas line quadtrees store an exact representation of a digitized image. Hence, comparing these two quadtrees would be analogous to comparing splines to boundary codes, i.e., an approximation scheme for curves in the real plane vs. an exact encoding of a digital curve.

In order to complete the bottom-up definition of a line quadtree, we must give a compatibility criteria for merging. The scheme that we propose is called the propagated criteria of weak-formedness. Below, this criteria and its associated algorithm are presented as the end result in a successive pattern of refinement from the criteria of identicality (algorithm 1) to the criteria of strong-formedness (algorithm 2) to the criteria of weak-formedness (algorithm 3) to the propagated criteria of weak-formedness (algorithm 4). Note that subroutines common to more than one algorithm are only presented once, i.e., in conjunction with the first relevant algorithm.

Prior to developing the merging criteria, we set forth a number of notational conventions. From a node in a line quadtree data structure, we must be able to access five other nodes. Four of these accessible nodes are the four sons of the original node and are accessed via the function SON.

SON: nodes x quadrants -> nodes

The four quadrants are named NW, NE, SW and SE (e.g., Figure 2). The fifth accessible node is the father of the original node and is accessed via the function FATHER.

FATHER: nodes -> nodes

The nodes of a line quadtree are also interrogated by two predicates, IS_LEAF and EDGE.

IS_LEAF: nodes -> boolean
EDGE: nodes x boundary -> boolean

IS_LEAF is true iff the node has no sons and hence is a terminal node. It should be clear that if a node has one son, then it has all four sons. EDGE is true iff the specified boundary of the node is present (said to be marked SET) and is false when not present (said to be unmarked or marked CLEAR). The four boundaries (at times referred to as sides or directions) are labeled N, E, S, and W (e.g., Figure 2).

Boundaries and quadrants are manipulated with the aid of the auxiliary functions CSIDE, CCSIDE, and QUAD.

CSIDE: boundary -> boundary
CCSIDE: boundary -> boundary
QUAD: boundary x boundary -> quadrant

CSIDE (CCSIDE) returns as its value the boundary that is adjacent clockwise (counter-clockwise) from the boundary indicated by its parameter (e.g., CSIDE(E)=S and CCSIDE(N)=W). QUAD returns as its value the quadrant that contains the corner formed by the boundaries indicated by its parameters provided such a corner exists (e.g., QUAD(N,W)=NW whereas QUAD(N,S) is undefined).

Storage management is facilitated by GET_FROM_AVAIL and RETURN_TO_AVAIL. GET_FROM_AVAIL returns a pointer to a previously inaccessible node. RETURN_TO_AVAIL signals the storage management system that the node referred to by the parameter is no longer being used and that the storage associated with it be recycled.

As with most systems for manipulating geometric data, the intuitions involved in line quadtrees are best motivated pictorially; hence certain pictorial conventions must also be established. In Figure 2, the standard arrangement of boundaries and quadrants is shown. In Figure 3, a map M of six regions is shown. The bold lines indicate the region's boundaries and the light lines indicate the digitization. Note that all maps are surrounded by a border. This maintains the separation of regions when the maps undergo linear transformations. In Figure 4, the complete 4-ary tree that encodes the map M is shown. Note that each square represents a node in the tree. The four largest squares inside a given square represent the four sons of that node. A square that contains no squares represents a leaf. The outermost square represents the tree's root. The EDGE information for a node is stored in the picture by using a bold line on any boundary whose EDGE value is marked SET and using a light line on the other boundaries (i.e., marked CLEAR). Note that algorithms 1 through 3 do not use the EDGE information of interior nodes and thus the interior nodes are drawn under the assumption that all EDGE information is set to false (i.e., marked CLEAR).

The first algorithm, CONDENSE1, embodies the criteria of identicality, i.e., four brothers are merged iff all the nodes corresponding to the blocks represented by the brothers contain identical information in their corresponding EDGE fields. Thus if CONDENSE1 were applied to the 4-ary tree in Figure 4, it would produce the quadtree of Figure 5. Note that only two 4-tuples of brothers were candidates for merging.

Although the criteria of identicality is adequate for region quadtrees, it is inadequate for line quadtrees. Nodes that border edges tend to have as brothers nodes that do not border edges on the same corresponding side. For example, the same edge causes a northern brother to have his southern edge marked SET and a southern brother to have his northern edge marked SET, making it unlikely that the southern brother will also have his southern edge marked SET (at least at levels of digitization that are common in images that do not evidence microlevel texture, e.g., maps in contrast with gray scale photographs). This can be seen by observing the example map in Figure 4. The storage utilization is comparable to that of region quadtrees for line drawings segmented into two regions: one the black line and the other being the clear interior. This claim is asymptotically true with respect to the degree of digitization. Here, we assume that the insignificant number of nodes merged in the "line" region of the region quadtree representation is comparable to the insignificant number of nodes merged that are identical and have some of their edge values marked as SET in the line quadtree representation. In both schemes, the primary effect of the merging criteria occurs at nodes away from the boundaries of the regions represented by the quadtree.

From the above discussion, it follows that a key aspect of the design of a merging criteria is to maximize the probability that it will be met by many nodes of the tree representation of a "typical" image. On the other hand, the allowable merging criteria are restricted by the necessity of having the ability to reconstruct the original image (at least the original digitization of the original image).

We will not attempt to find an "optimal" method of satisfying the above restrictions on allowable merging criteria. Instead, we address the more modest task of finding a merging criteria for line quad-trees that is as reasonable as the standard merging criteria for a region quadtree. Recall that in such a case each region is given a separate color and borders are represented implicitly by the two squares of different color that the border separates. For Figure 3, such a region quadtree would use six colors as is shown in Figure 11.

The next step in finding the above mentioned merging criteria is to consider the criteria of strong-formedness as implemented in CONDENSE2. The criteria of strong-formedness can be stated as follows. If the submap represented by a given subtree corresponds to a square with zero or more entire sides missing, then that subtree can be replaced by a single leaf. Since there are no lines crossing the interior of a drawn square, it follows that a single leaf can never represent a submap that contains more than one region. This can best be understood pictorially. Figure 6 presents the quadtree that is built by applying this merging condition to the 4-ary tree in Figure 4. If we look at node $\alpha$ in Figures 4 and 6, we see the result of the patterns of four brothers being merged to form one pattern. In the NW son of the root of Figure 4, we see the result of this merging process having been performed on two separate levels. Node $\beta$ in Figures 4, 5, and 6 illustrates four sons that could not be merged because they had edges interior to their non-square pattern (although they were merged in CONDENSE1). This is checked by predicate NO_INSIDE_BORDERS in CONDENSE2. In node $\gamma$ of Figure 6, we see four sons that could not be merged because the S border of $\gamma$'s SW son is marked SET but that of $\gamma$'s SE son is marked CLEAR; so that together, they form only a partial border. This is checked by predicate WHOLE_OUTSIDE_BORDERS in CONDENSE2.

Note that procedure SET_EDGES in CONDENSE2 is more complicated than is necessary, i.e., it logically ANDs together two values that have already been tested as equal by WHOLE_OUTSIDE_BORDERS. This permits the same procedure to be used by both CONDENSE3 and CONDENSE4.

Although we note that the number of nodes in Figure 6 is considerably smaller than in Figure 5, we still have not reached the goal of merging all nodes that would be merged if the map were painted with six colors and region quadtrees were used. In particular, note that the SE son of the root in Figure 6 resides entirely inside region M6 of Figure 3 and thus would be represented by one node of color M6 in the mentioned region quadtree, whereas in Figure 6 this node has four offspring.

Consideration of this problem leads to the criteria of weak-formedness. Clearly, in order to be able to merge the above mentioned regions, it is necessary to weaken the criteria of strong-formedness to allow two different edge values to be combined but still to provide for reconstruction of the original map.

CONDENSE3 performs this weakening by eliminating the test for WHOLE_OUTSIDE_BORDERS in CONDENSE2. The combination of the now possibly differing edge values is done via a "logical AND" operation in SET_EDGES. The merging criteria encoded by CONDENSE3 is termed the criteria of weak-formedness. Figure 7 shows the result of applying CONDENSE3 to the complete 4-ary tree of Figure 4. Note that the SE son of the root is now a leaf and also that merging occurred in both the NW son and the SE son of the SW son of the root. Indeed all the hoped-for mergers occurred.

Observing the algorithm, CONDENSE3, it readily becomes apparent that the only criterion for merging is that the "inner" edges must be marked CLEAR. Note that if the inner edges were not marked CLEAR, then by virtue of the fact that line segments forming the edges must partition the space, it follows that the nodes on each side of the edge belong to separate regions. In such a case, the region quadtree would not have merged these brothers either.

Therefore, the number of nodes in the line quadtree is bounded from above by the number of nodes in the corresponding region quadtree. Analogous reasoning about the significance of the absence of inner edges leads to the conclusion that the number of nodes in the line quadtree is equal to the number of nodes in the corresponding region quadtree.

The remaining question with respect to the criteria of weak-formedness is whether or not it is possible to reconstruct the image. The reconstruction is based on the following two premises: 1) if a terminal node's side is marked SET, then there was an edge of that length in that position in the original image; 2) if a terminal node's side is marked CLEAR, but there are segments of the map that coincide with segments of that side, then those edges will be represented by sides being marked SET in the adjacency tree for that side. The adjacency tree [5] of a terminal node x on a given side is a binary tree rooted at the neighbor y (possibly an internal node) on that side and containing all descendants of that neighbor y that have the node x as a neighbor. For example, in Figure 7, we see that a complete description of the W side of the SE son of the root can be obtained by examining the E side of the NE son of the SW son of the root. Since this segment of border does not extend down the entire W side of the SE son of the root, it must turn west (note that all curves are implicitly closed). By turning west, the line prevents the merger of the sons of the SW son of the root, thereby ensuring that one of these unmerged sons could encode it as a side marked SET.

Finally, we are confronted with the recurring issue of what information to store at the internal nodes (analogous to the GRAY nodes of the region quadtree) to speed up our algorithms. This depends on which algorithms are to be speeded up. Our choice is to speed up edge-following algorithms, e.g., the quadtree to boundary code transformation, due to a natural affinity between line quadtrees and line-following algorithms.

At this point, we must be precise as to what constitutes our merging criteria and what information is propagated to the interior nodes. We term our final merging criteria the "propagated criteria of weak-formedness." The propagation criteria is that an edge is propagated upward as marked SET iff at the higher level, it could be followed without reference to the lower nodes that form it, e.g., the south side of M6 in Figure 3. This propagation criteria is encoded by CONDENSE4 with the aid of an additional parameter named CORNERS. In order to see the need for this extra information, examine Figure 8 which results from the execution of CONDENSE4 on the 4-ary tree in Figure 4. Note that for the first time, some of the internal nodes have sides marked SET, e.g., the south side of the root. The key to the need for CORNERS is seen in the W side of the SW son of the ROOT. This side is marked CLEAR because any line following algorithm would have to turn E half way down the segment in order to keep following the border of the same region (e.g., for regions M5 or M6 in Figure 3). However, the two sides that are to be combined to form this W side are both marked SET. Moreover, the information stored in the SW and NW sons of the SW son of the root do not indicate the presence of a T-junction along their western boundary. Hence non-local information must be passed upward from the leaves and this is the role of CORNERS. To summarize, CORNERS indicates whether or not two edges that meet at a corner are both marked SET, e.g., the NE corner of region M1. If CORNERS is true for an interior portion of a segment that is marked SET (e.g., when M5 and M6 meet on the western side of the image), then a T-junction is present, and the segment's other node should be marked CLEAR (e.g., $\alpha$ in Figure 5). This is accomplished by FIX_T_JUNCTION, which is invoked by SET_INTERIOR_NODE, which is in turn invoked by CONDENSE4.

While we are considering edge-following algorithms, we can also take the time to justify the placement of the onus of multiple meaning on the CLEAR edge so that when there is a SET edge, there is no question but that it is SET. In essence, when following an edge, there is a greater benefit in knowing how far ahead we can jump rather than the extent of the absence of a boundary (i.e., the length of the CLEAR edge).

The edge-following algorithms are straight forward extensions of the quadtree-to-boundary-code algorithm. Actually, they are slightly faster because they can sometimes move onward on the basis of information stored at an interior node and not have to examine all the terminal nodes along the path. For example, an edge following algorithm processing the bottom edge of the map of Figure 3, with the aid of the quadtree of Figure 8, would be able to use the fact that the S side of the root is solid and thereby never have to descend to the SE son of the SW son of the SW son of the root to verify the presence of that segment of the edge.

## 3. Postorder Tree Traversal Algorithms

We are also interested in speeding up postorder tree traversal algorithms. Such algorithms are often useful because the worst-case analysis of their execution times does not depend on the depth of the quadtree (i.e., the resolution of the image) as is common to the edge-following algorithms, but rather, depends only on the number of nodes in the quadtree. As an example of such an algorithm, we shall analyze the CROSS_PRODUCT algorithm as described below.

The CROSS_PRODUCT algorithm is the line quadtree analogue of the region quadtree SUPER_POSITION algorithm [5] and hence is of this postorder traversal type. Recall that Hunter was referring to an algorithm for overlapping two images. As an example of its effect, consider Figure 9, where two maps are presented: (1) map A composed of two regions (A1 & A2) and (2) map B composed of two regions (B1 & B2). These maps can be viewed as describing regions uniform with respect to two different variables, A and B. The CROSS_PRODUCT algorithm produces a map (termed AB in Figure 10) of regions uniform in the ordered pair (A,B). It should be noted that if two cells are in separate regions in either map, they will necessarily be in separate regions in the result of the application of CROSS_PRODUCT.

If these maps were being stored in 2-d arrays, then the obvious implementation of the superposition process would be to logically OR the corresponding elements in the two arrays. This is a consequence of the fact that the presence of an edge marked SET in either map implies the presence of the corresponding edge in the resulting set. However, when we logically OR the corresponding elements of the two line quadtrees for maps A and B, we get the line quadtree indicated in Figure 10c. This tree differs from the line quadtree of the map AB shown in Figure 10b at two types of instances which are labeled $\alpha$ and $\beta$. Alpha points to the edge of an internal node that is marked CLEAR due to the presence of a 'T' side in the correct quadtree, whereas the indicated edge is marked SET by the logical OR operation since the corresponding edges in quadtrees A and B are marked SET. Note that this situation occurs in two places in Figure 9. Beta points to the edge of a terminal node that is marked SET in the correct quadtree, but not in the tree generated by the logical OR operation since the corresponding edges in quadtrees A and B are marked CLEAR.

In the following paragraphs, we develop the CROSS_PRODUCT algorithm in a manner analogous to that used for CONDENSE4. Recall that this type of approach is one of a "debug" nature, i.e., we will generate the logical OR tree and then proceed to fix it.

The first iteration of CROSS_PRODUCT is CROSS_PRODUCT1. CROSS_PRODUCT2 generates the same result as CROSS_PRODUCT1, but it has a faster worst-case execution time. Both versions of CROSS_PRODUCT have the following high level structure: (1) generate the logical OR tree using the function NODE_OR; (2) correct the values of the terminal nodes using procedure CONSISTENT; and (3) correct the values of the internal nodes using procedure CONDENSE4. Note that since cells of separate regions in the array stay in separate regions in the result of CROSS_PRODUCT, no merging occurs during the execution of CONDENSE4. Thus, CONDENSE4 is merely being used for its side-effect of calculating the EDGE values for the internal nodes.

In CROSS_PRODUCT1 we use a version of the NODE_OR algorithm to perform a postorder traversal of both trees in parallel until a leaf node is reached. Having reached a leaf node in one tree, NODE_OR proceeds to copy the remaining subtree of the other tree using COPY and then to perform a logical OR of the sides of the region represented by the subtree with the corresponding sides of the leaf node. This is followed by continuing the traversal while simultaneously patching together the copied subtrees to form the ROOT_RESULT. Hence, the result of the NODE_OR operation is the logical OR of the corresponding parts of the two quadtrees involved. Note that the internal nodes are ignored, since they will be fixed up later by CONDENSE4. Figure 11a shows the result of applying NODE_OR to the maps of Figure 9 disregarding the values of the edges of the internal nodes.

Once the NODE_OR operation has been completed, CROSS_PRODUCT1 removes $\beta$ problems in the leaf nodes of the 4-ary tree by use of CONSISTENT1. This process consists of examining (using HAS_DARK_SIDE) the neighboring adjacency trees (found by GETNEIGHBOR) and insuring that they satisfy the criteria of weak-formedness. The version of GETNEIGHBOR used herein is presented in [13] and uses the FATHER function. The result of applying CONSISTENT1 is to set the leaf nodes to the values indicated in Figure 11b. Next, CONDENSE4 is used to mark the edges of the internal nodes, e.g., Figure 11c is the result of the application of CONDENSE4 to the tree in Figure 11b.

## 4. Analysis

The cost of executing CROSS_PRODUCT1 is the sum of the costs of its three components - i.e., NODE_OR, CONSISTENT1, and CONDENSE4. The cost of NODE_OR is bounded from above by three times the number of nodes in both quadtrees. This can be seen by observing that NODE_OR visits every node at least once while MARK_SIDE visits a disjoint set of adjacency trees chosen from the two quadtrees. Since no node can be in more than two nontrivial adjacency trees [5], it follows that the number of nodes processed by MARK_SIDE is bounded from above by twice the number of nodes in both quadtrees. Hence, as stated earlier, the number of nodes processed by NODE_OR is bounded from above by three times the number of nodes in both quadtrees.

CONDENSE4 is simply a postorder traversal of the quadtree rooted at ROOT_RESULT Since CONSISTENT1 does not alter the number of nodes in that tree, it follows that the number of nodes processed by CONDENSE4 is bounded from above by three times the number of nodes in both quadtrees as this is the bound resulting from CONSISTENT1.

CONSISTENT1 performs a postorder traversal of the quadtree rooted at ROOT_RESULT and at each leaf invokes GET_NEIGHBOR four times. The upper bound on the number of nodes visited as a result of the invocation to GET_NEIGHBOR is twice the depth of the tree (i.e., the log of the resolution of the image). Since the worst-case tree depth is proportional to the number of nodes in the tree, the cost of CONSISTENT1 clearly dominates the cost of CROSS_PRODUCT1 in the worst-case. Note that HAS-DARK-SIDE also processes adjacency trees.

In analyzing the CROSS_PRODUCT algorithm as implemented by CROSS_PRODUCT1, we find that the cost of the algorithm is proportional to the product of the the cost of using the GET_NEIGHBOR algorithm and the number of nodes in both trees. Although on the average, the cost of using GET_NEIGHBOR is a constant [13], in the worst case, it is equal to the tree depth. CROSS_PRODUCT2 presents the algorithm in a form whose worst-case execution time analysis leads to an execution time that is proportional to the number of nodes in both trees. In particular, since only CONSISTENT1 makes use of GET_NEIGHBOR, only it is modified in CROSS_PRODUCT2.

In order to avoid having to use GET_NEIGHBOR to locate a leaf's neighbors, four parameters are added to the invocation of CONSISTENT2. These parameters correspond to the four neighbors of the node ROOT. As a programming convenience, it is assumed that the procedure call places those neighbors in the array NEAR. Since most programming languages restrict formal parameters to simple identifiers, this feature can most easily be simulated by using macros. Initially, CROSS_PRODUCT2 creates a node named NEIGHBOR with all borders marked SET to be the four neighbors of the root ROOT_RESULT since the map is assumed surrounded by a solid boundary. The calculation of the neighbor parameters for the processing of the four sons of the ROOT is done by HELPER. HELPER takes nine parameters and its expression would be quite awkward without the parameter array device. The calculation of neighbors is straightforward and is based on the geometry of the quadtree node arrangement. If a son of a neighbor is needed for the next pass but does not exist, then the original neighbor is reused as his EDGE information on the side in question is still appropriate. Note that the semicolons in procedure calls to HELPER are merely used here to facilitate the readability of parameter groupings and otherwise have no special programming significance.

From the above, it should be clear that we have reduced CONSISTENT2 to a postorder traversal algorithm that separately visits all its adjacency trees. Therefore, its execution time is proportional to three times the number of nodes in ROOT_RESULT. Other postorder traversal algorithms could be modified in a similar manner to bring their worst-case execution times in line with the above results.

## 5. Concluding Remarks

We have presented a data structure for storing maps and their boundaries in a hierarchical manner without excessive waste of storage or having to solve the messy problem of graph coloring. Such coloring would be necessary in order to use region quadtree algorithms of comparable storage frugality. The coloring could be achieved by use of connected component labeling algorithms [17] followed by the linear-time five coloring algorithm of [11] (hence minimizing the number of bits needed for colors). However, this entails a substantial loss of information stored in the internal nodes that can be used by line-following algorithms.

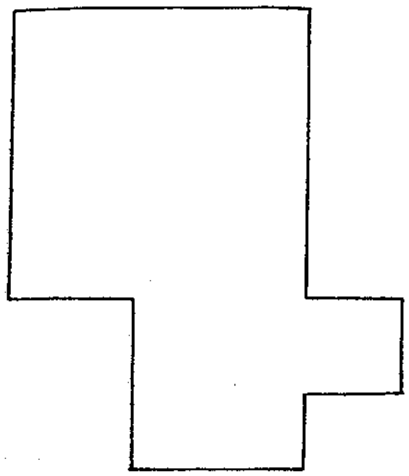
Our approach is predicated on the desire to be able to determine and represent in a hierarchical manner both the areas and the borders of the regions comprising the maps. Thus our techniques are more applicable to a decomposition of a map into counties, states, etc. rather than contour lines, point data such as cities, or roads and rivers. As mentioned earlier, these problems would be more suitably attacked by use of other data structures such as point space quadtrees [3] for cities and strip trees [1] for roads and rivers.

We have also detailed algorithms for the construction of the line quadtree data structure as well as its manipulations. Our presentation was an iterative one so that the reader could gain insights for the rationale behind our ultimate choice of a data structure. Once the data structure was chosen, we demonstrated how it could be used to achieve some typical operations such as border following and map superposition. These algorithms showed that the worst case execution time for line quadtree algorithms is asymptotically equivalent to that obtained for region quadtree algorithms.

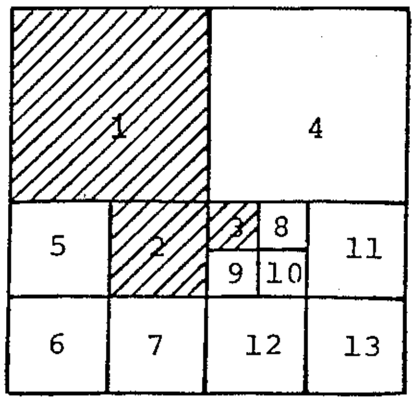
It should be noted that the line quadtree, as proposed in this paper, is not the unique solution to the problems presented. Variations of the line quadtree are a subject for future research. For example, instead of storing values for all four sides at each node, one could just store the values of two sides, e.g., N side and E side. As we see in Figure 13, although the nodes for the "two-sided" line quadtrees contain fewer data bits (2 instead of 4) than our standard (i.e., obtained by use of CONDENSE4) line quadtree, the two-sided line quadtree can have more nodes. We claim that it can never have more than twice as many nodes. The net effect of these extra nodes in the average case has yet to be determined.

## 6. References

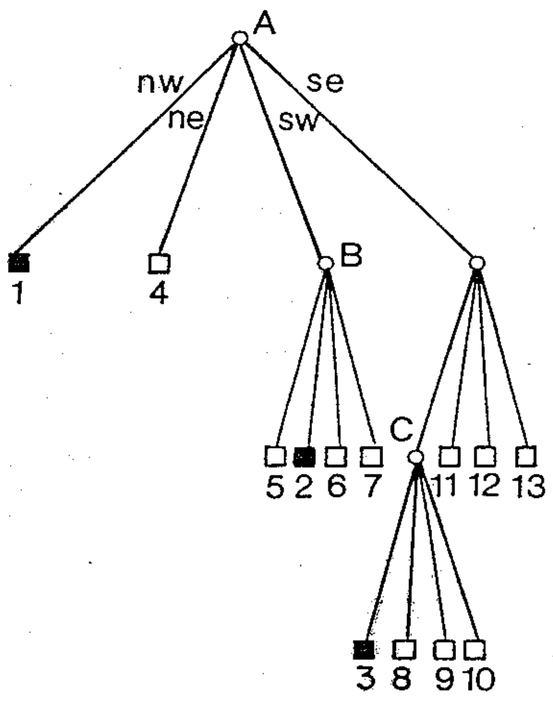
1. D. H. Ballard, Strip trees: a hierarchical representation for curves, *Communications of the ACM*, May 1981, 310 - 321.
2. C. R. Dyer, A. Rosenfeld, and H. Samet, Region representation: boundary codes from quadtrees, *Communications of the ACM*, March 1980, 171 - 178.
3. R. A. Finkel and J. L. Bentley, Quadtrees: a data structure for retrieval on composite keys, *Acta Informatica 4*, 1974, 1 - 9.
4. H. Freeman, Computer processing of line-drawing images, *ACM Computing Surveys 6*, 1974, 57 - 97.
5. G. M. Hunter, Efficient Computation and Data Structures for Graphics, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey, 1978.
6. G. M. Hunter and K. Steiglitz, Operations on images using quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 1979, 145 - 153.
7. G. M. Hunter and K. Steiglitz, Linear transformations of pictures represented by quadtrees, *Computer Graphics and Image Processing 10*, 1979, 289 - 296.
8. A. Klinger, Patterns and search statistics, in *Optimizing Methods in Statistics*, J. S. Rustagi (Ed.), Academic Press, New York, 1971.
9. A. Klinger and C. R. Dyer, Experiments in picture representation using regular decomposition, *Computer Graphics and Image Processing 5*, 1976, 68 - 105.
10. A. Klinger and M. L. Rhodes, Organization and access of image data by areas, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 1979, 50 - 60.
11. D. Matula, Y. Shiloach, and R. Tarjan, Two linear-time algorithms for five-coloring a planar graph, Stanford Technical Report STAN-CS-80-830, November 1980.
12. D. Rutovitz, Data structures for operations on digital images, in *Pictorial Pattern Recognition*, G. C. Cheng et al. (Eds.), Thompson Book Co., Washington D.C., 1968, 105 - 133.
13. H. Samet, Region representation: quadtrees from boundary codes, *Communications of the ACM*, March 1980, 163 - 170.
14. Samet, An algorithm for converting rasters to quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, January 1981, 93 - 95.
15. H. Samet, Algorithms for the conversion of quadtrees to rasters, Computer Science TR-979, University of Maryland, College Park, November 1980.
16. H. Samet, Region representation: quadtrees from binary arrays, *Computer Graphics and Image Processing 13*, 1980, 88 - 93.
17. H. Samet, Connected component labeling using quadtrees, *Journal of the ACM*, July 1981, 487 - 501.
18. H. Samet, A quadtree medial axis transform, Computer Science TR-803, University of Maryland, College Park, August 1979.
19. M. Shneier, Two hierarchical linear feature representations: edge pyramids and edge quadtrees, *Computer Graphics and Image Processing* 17, November 1981, 211 - 224.



a. Sample image



b. Block decomposition of the image in (a)



c. Quadtree representation of the blocks in (b)

Figure 1. An image, its block decomposition, and the corresponding quadtree. The foreqround blocks are shaded and the background blocks are blank.

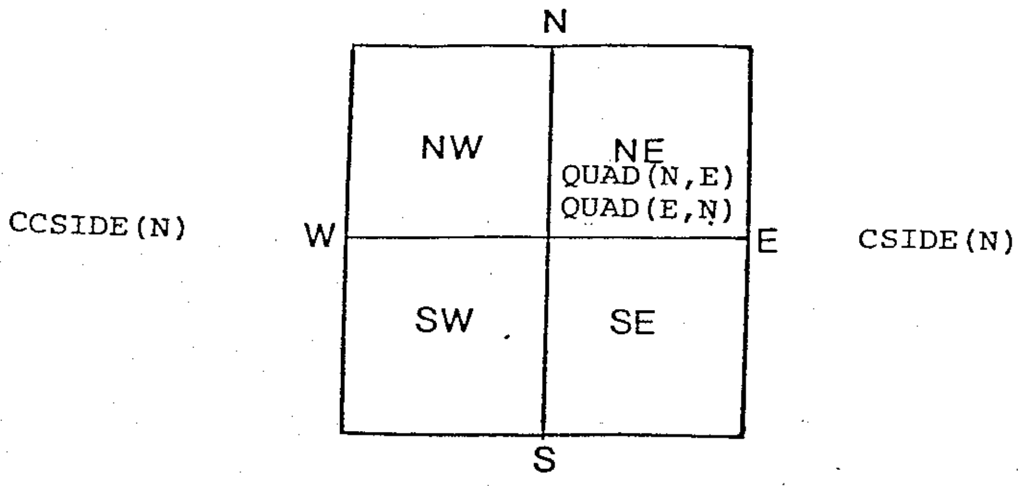


Figure 2. Quadrants and boundaries labeled

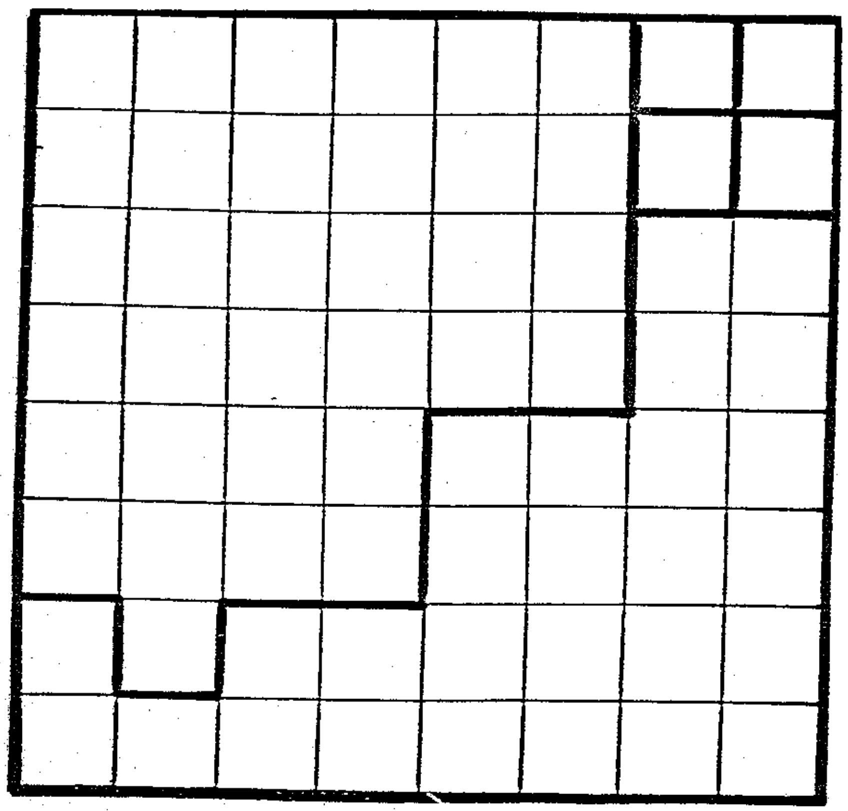


Figure 3. Picture (Map) M on 8x8 array of pixels

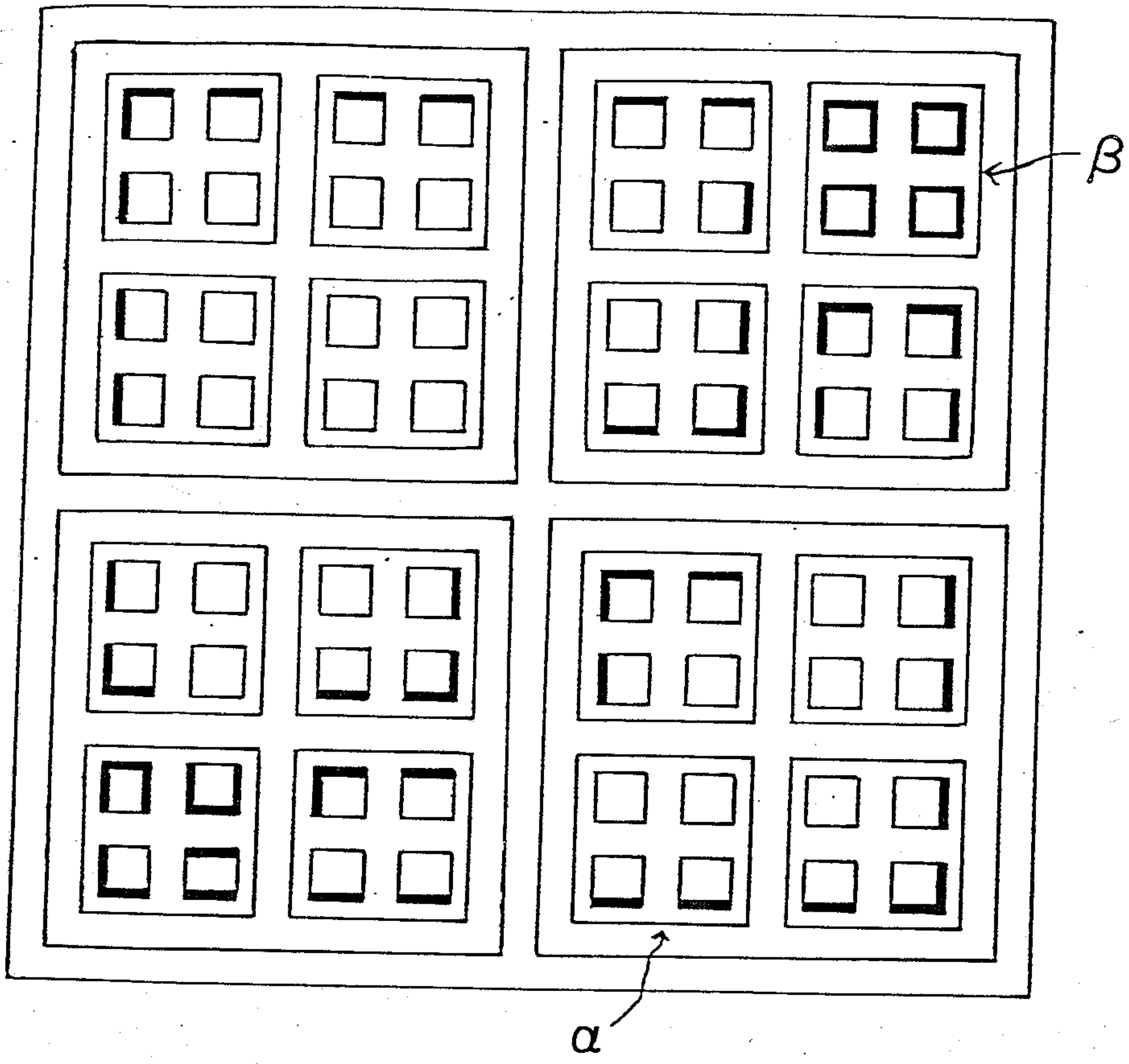


Figure 4. Complete 4-ary tree for Map M

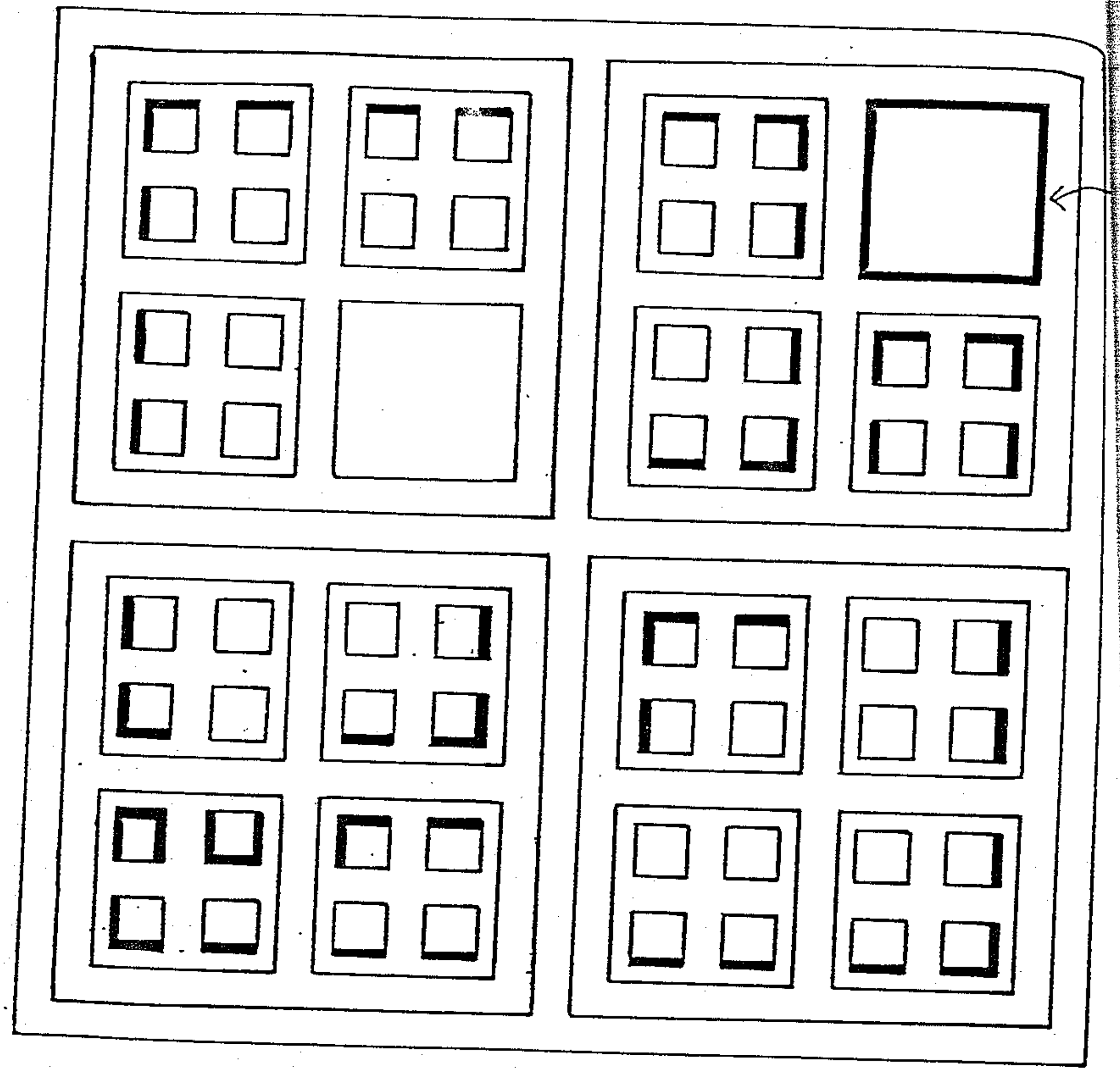


Figure 5. The quadtree for M produced by Algorithm 1

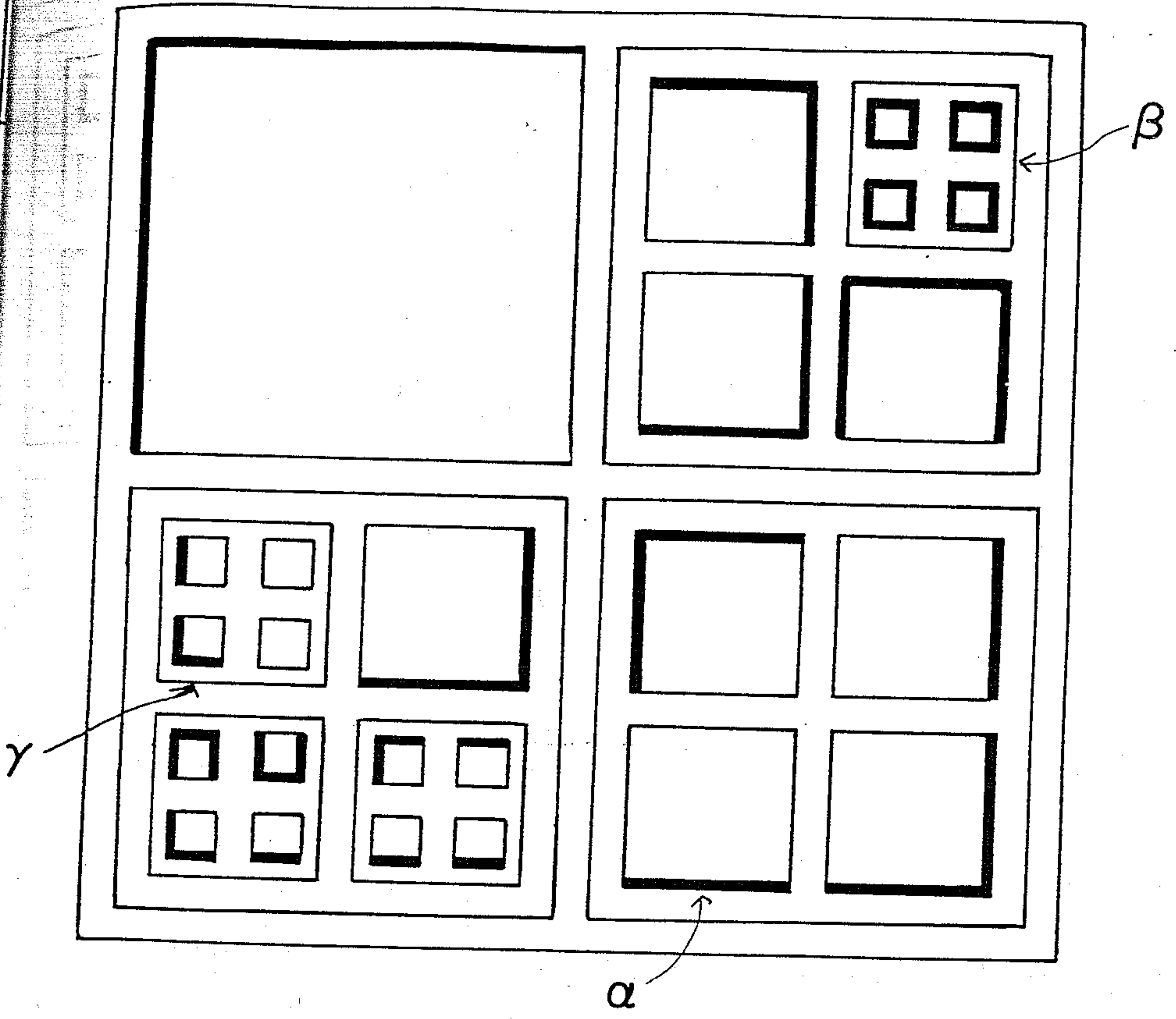


Figure 6. The quadtree for M produced by Algorithm 2

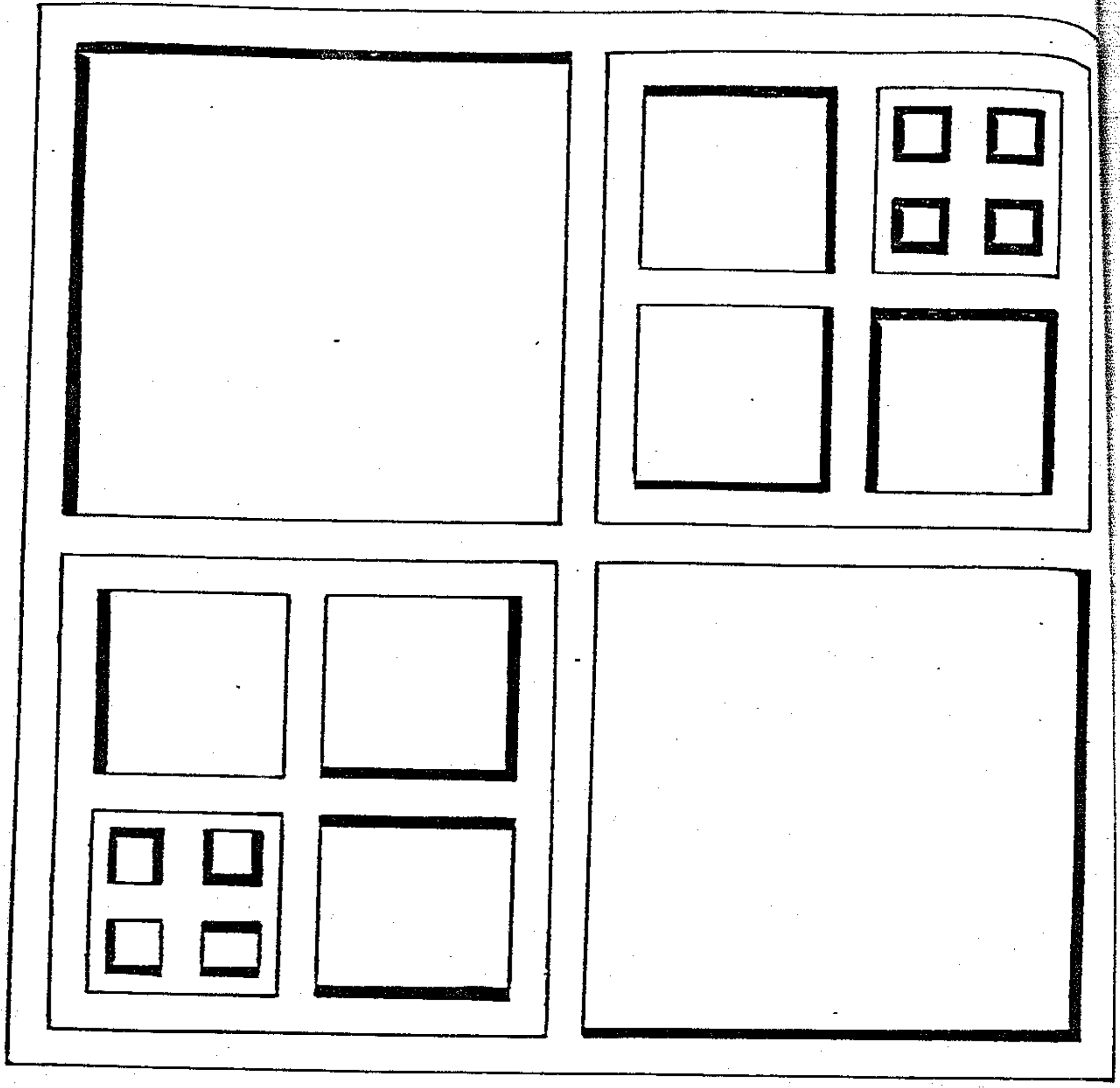


Figure 7. The quadtree for M produced by Algorithm 3

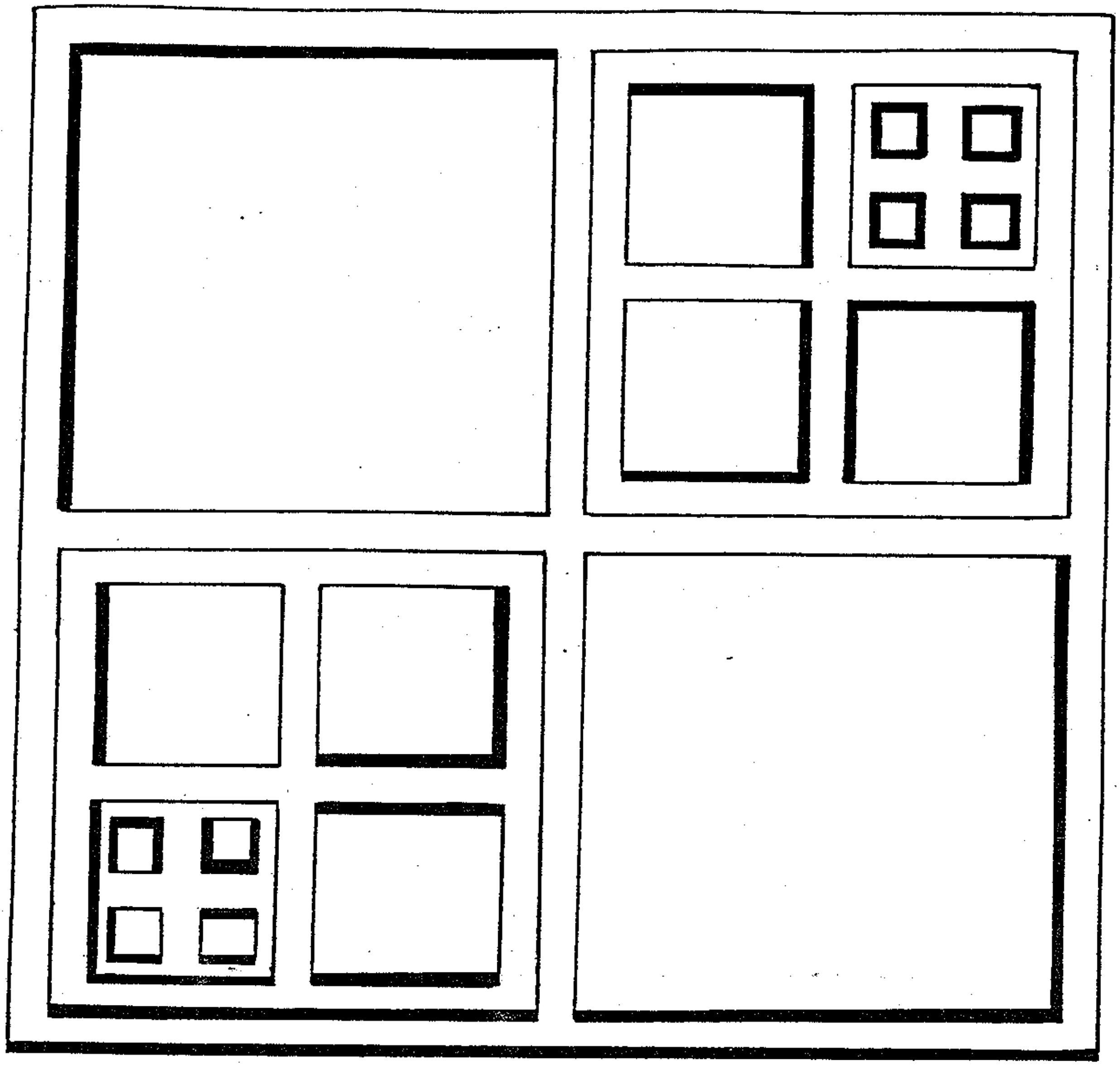


Figure 8. The quadtree for M produced by Algorithm 4

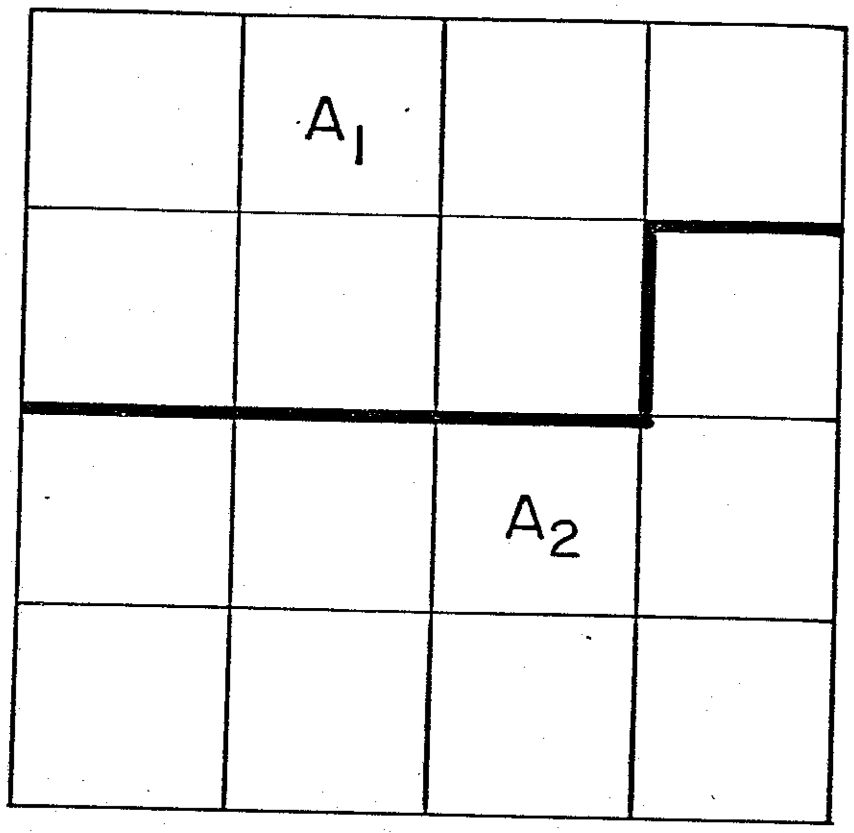


Figure 9a. Map A

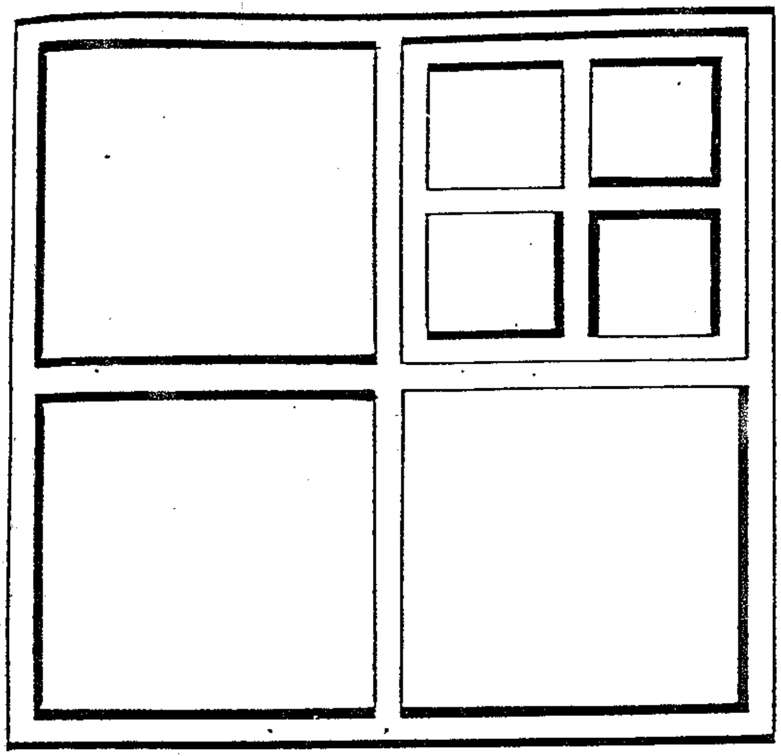


Figure 9b. Corresponding line quadtree for (a)

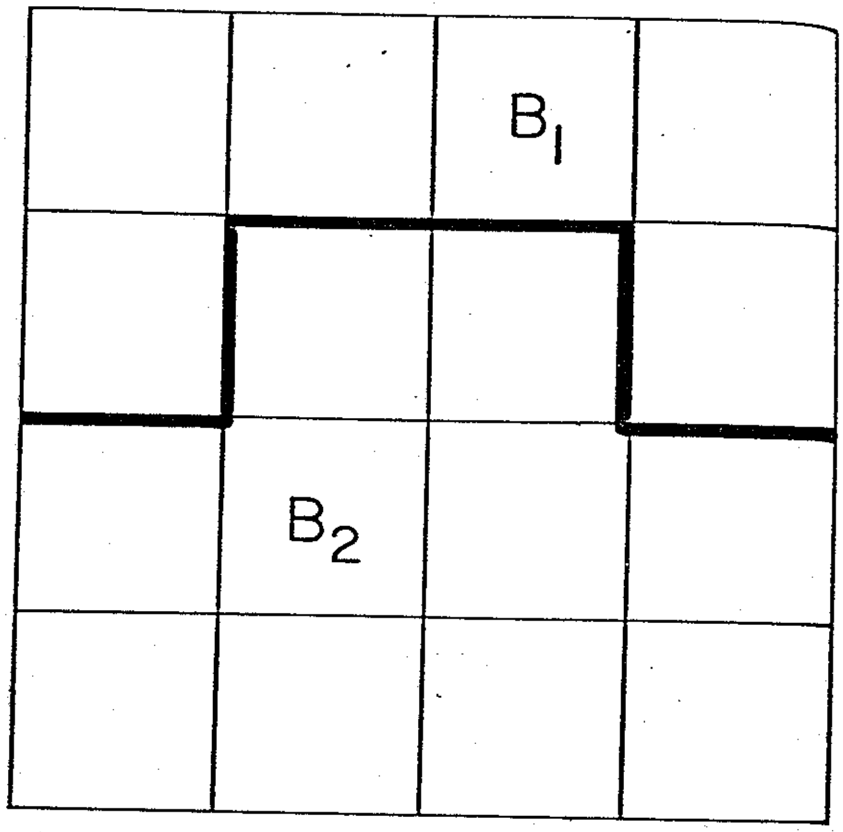


Figure 9c. Map B

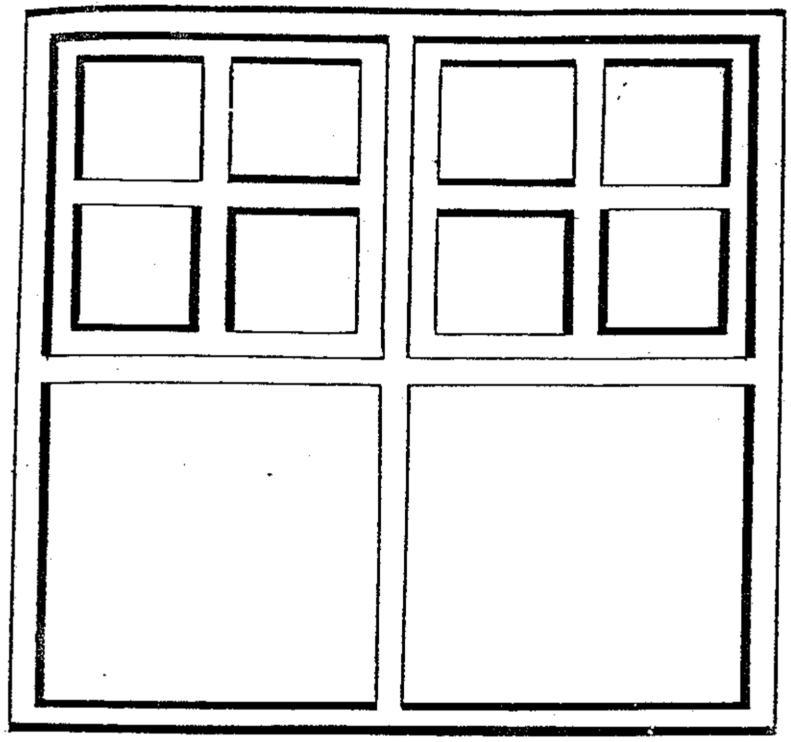


Figure 9d. Corresponding line quadtree for (c)

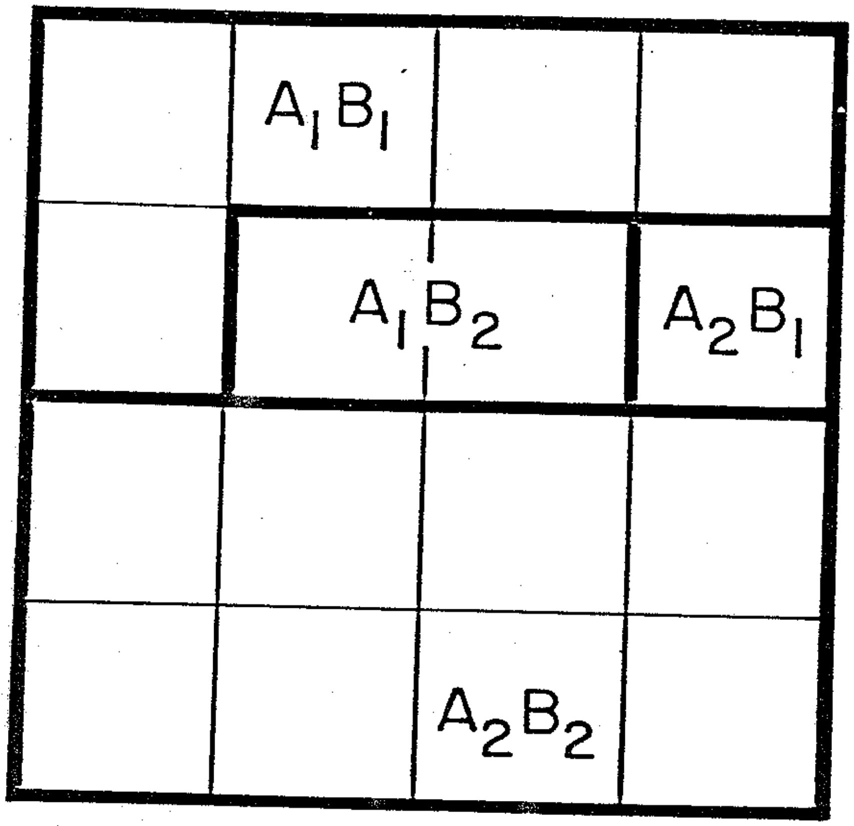


Figure 10a. Map AB

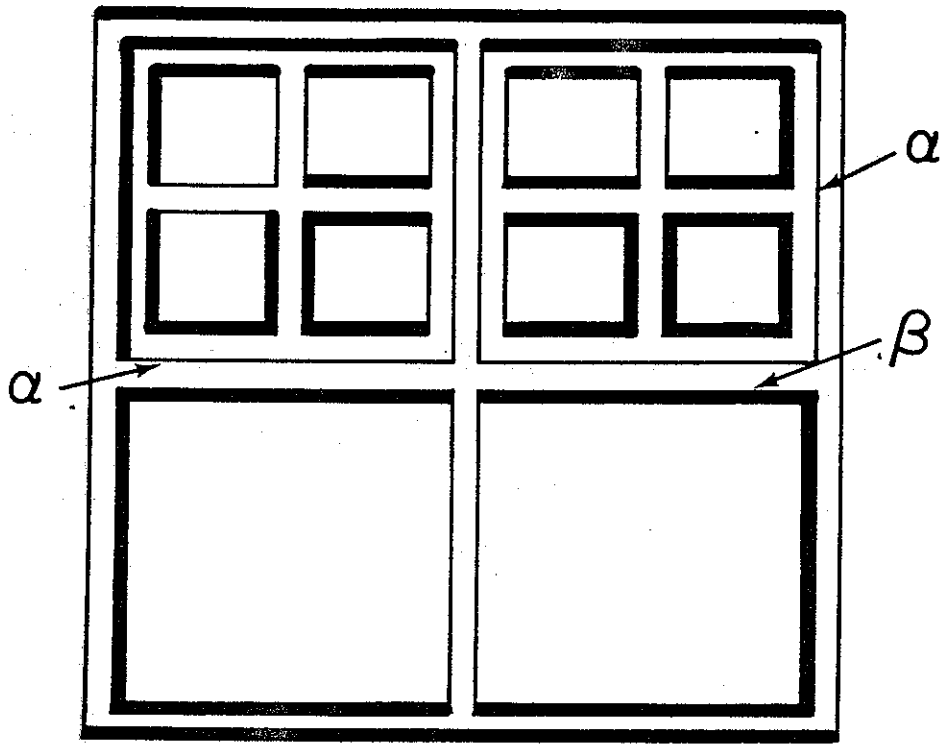


Figure 10b. Corresponding line quadtree for (a)

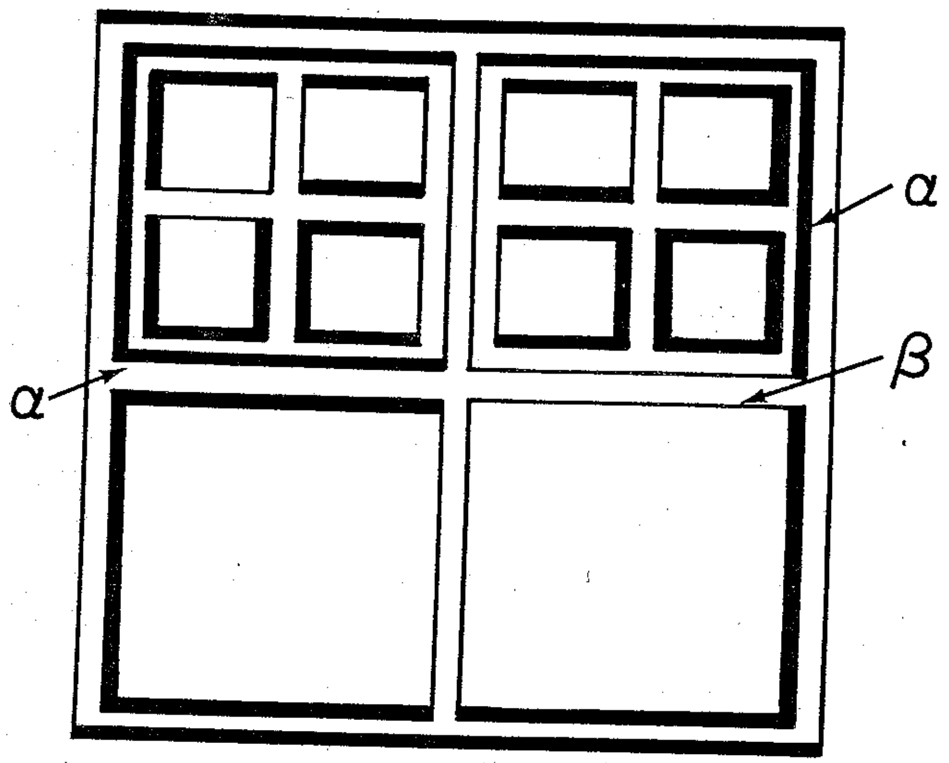
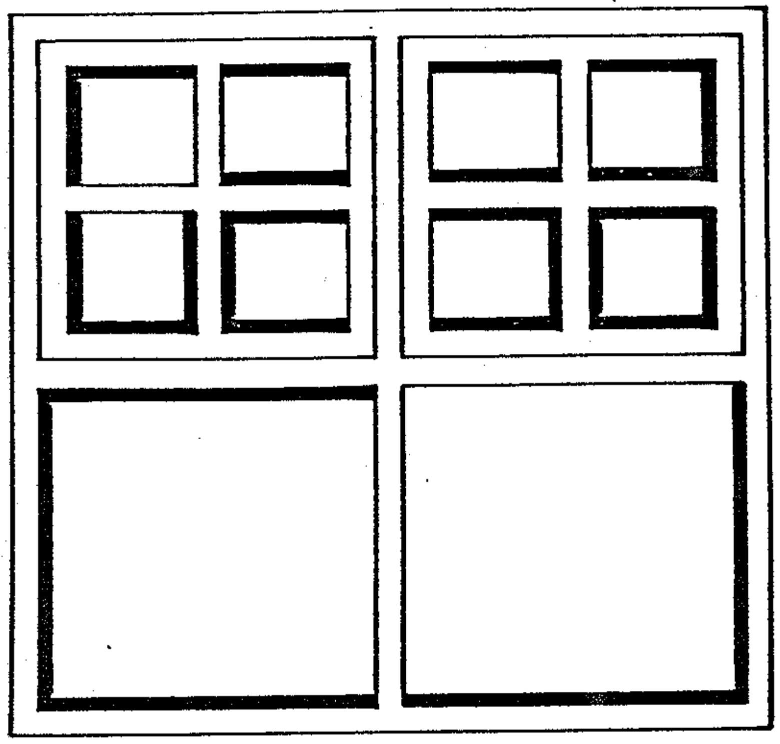
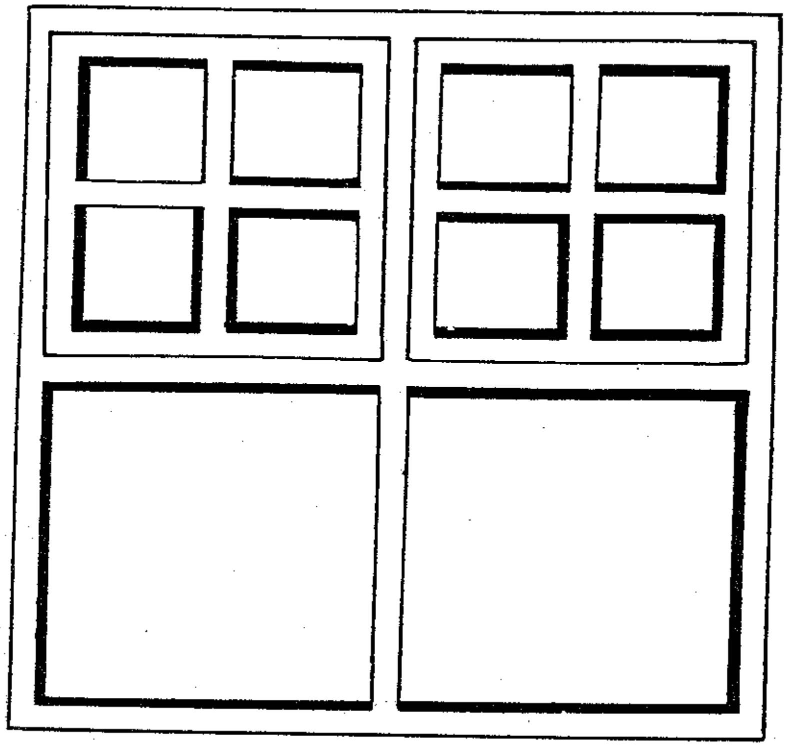


Figure 10c. Result of OR-ing together the two quadtrees for Maps A and B



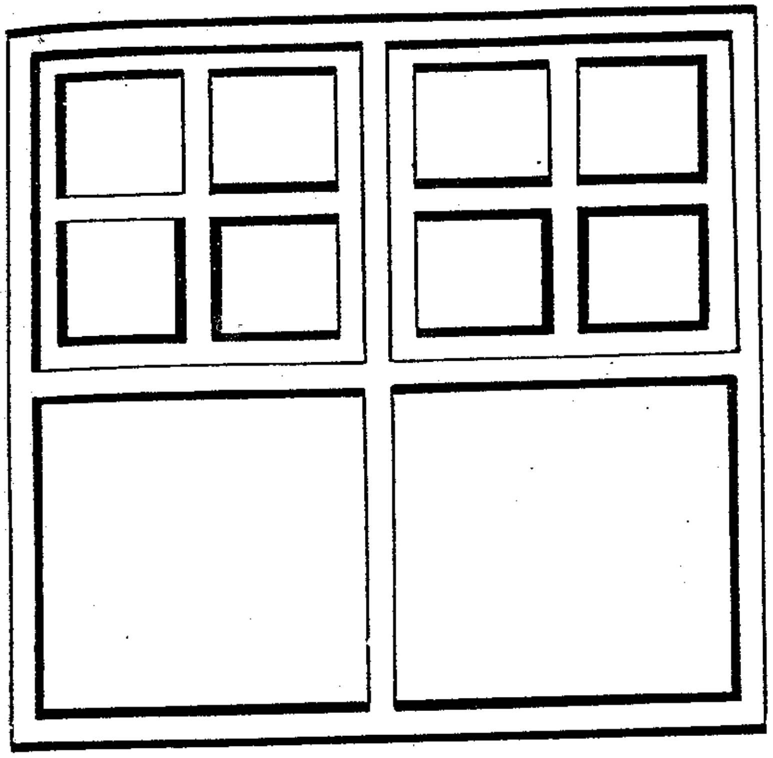
(a) Result of applying NODE_OR to the two quadtrees of Maps A and E

Figure 11. Order of Generation of Map AB



(b) Result of applying CONSISTANT1 to (a)

Figure 11, cont'd.



(c) Result of applying CONDENSE4 to (b)

Figure 11, cont'd.

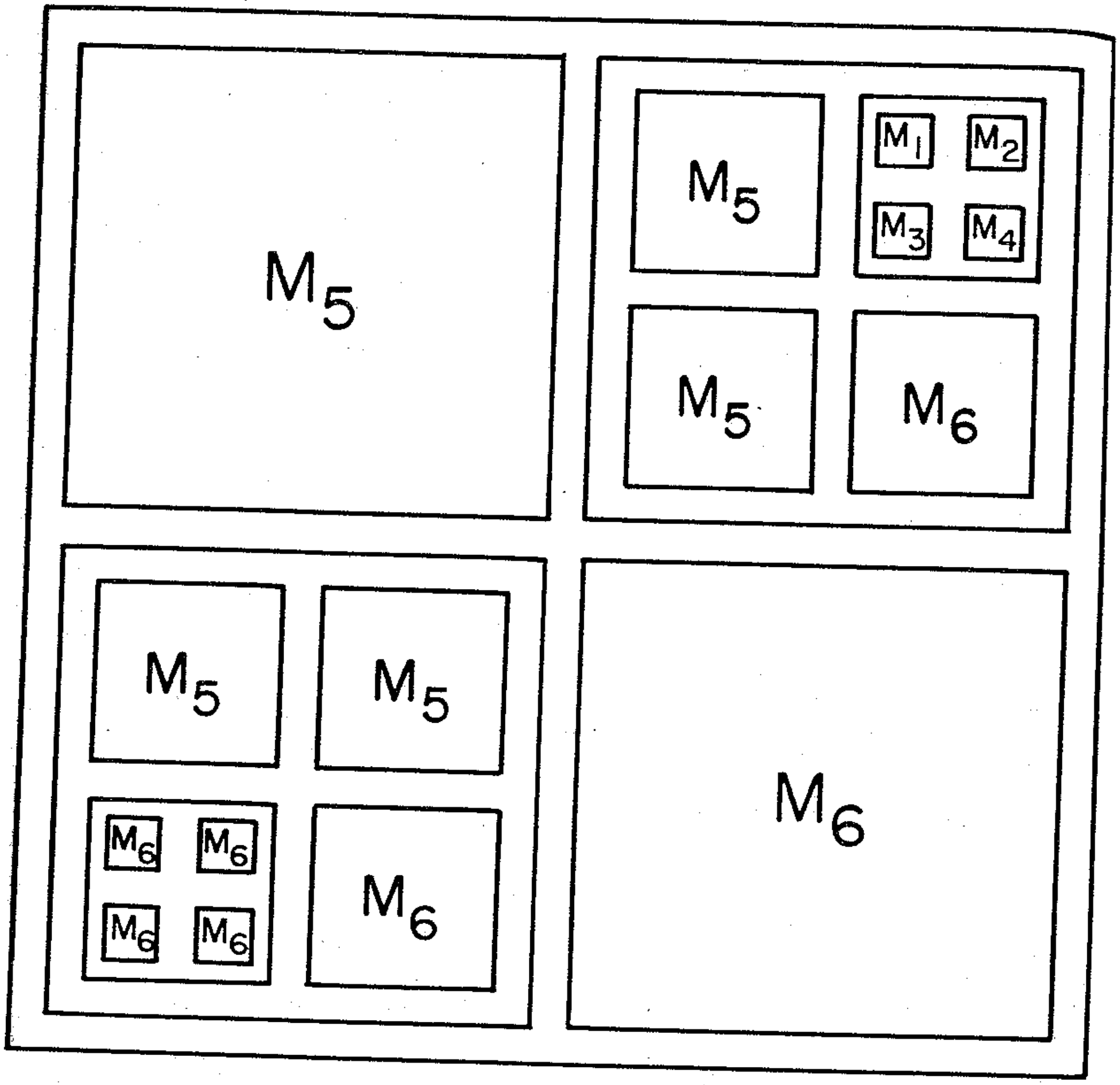


Figure 12. The region quadtree for M

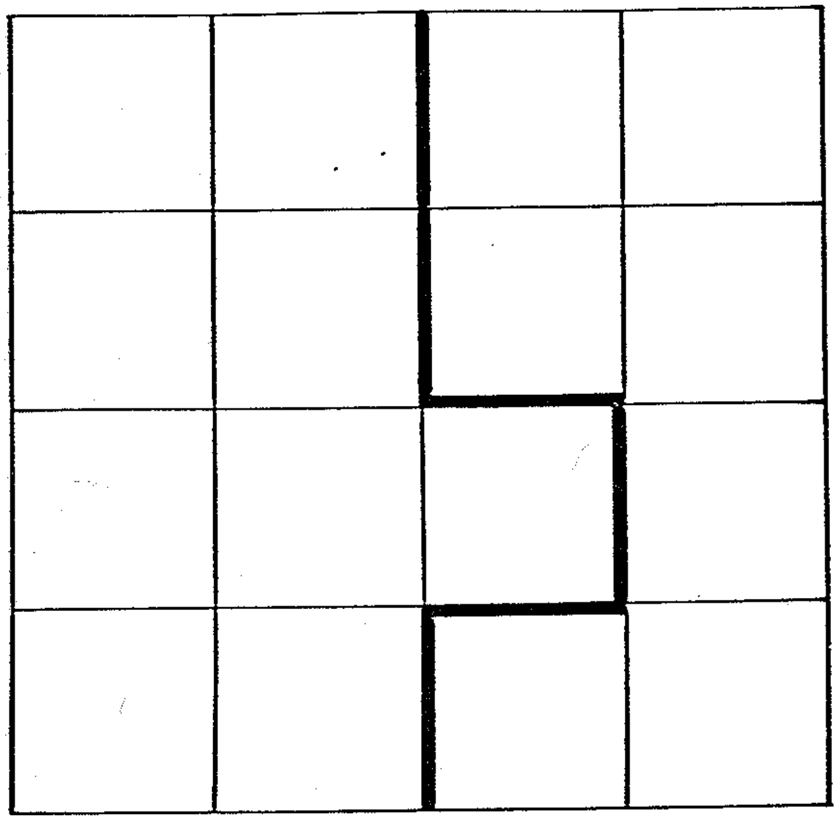


Figure 13a. Map Q

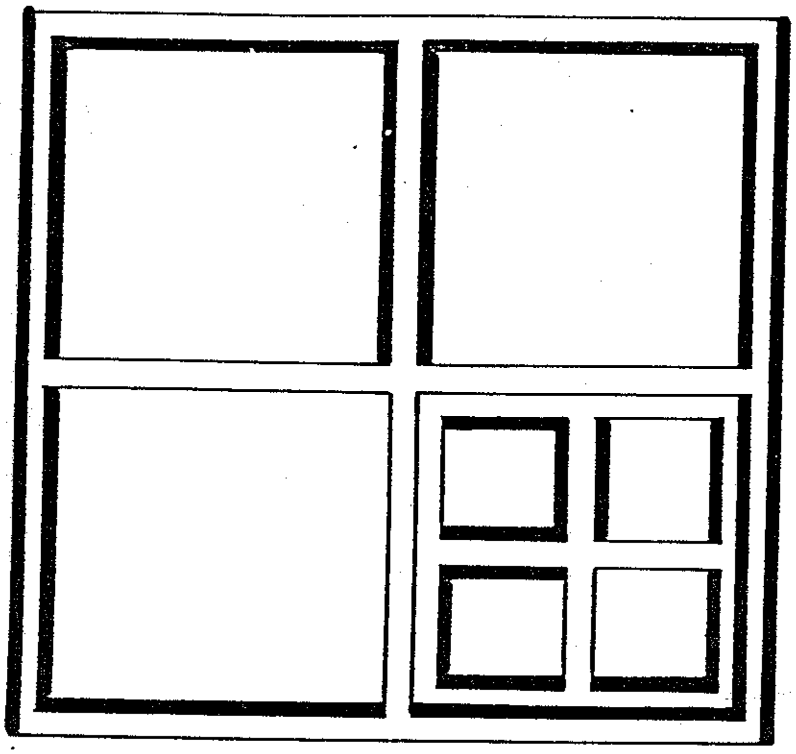


Figure 13b. Corresponding line quadtree for (a)

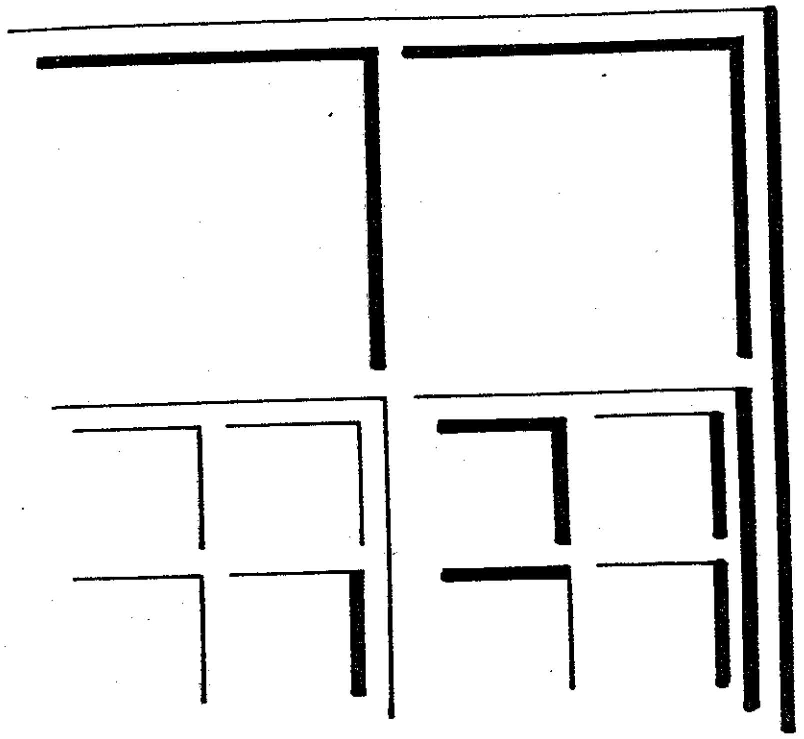


Figure 13c. The alternative two sided line quadtree for (a)

Algorithm 1

```
procedure CONDENSE1 (ROOT);
/* Modify a 4-ary tree rooted at ROOT by merging as many
   nodes as possible under the criteria of identicality
   to produce a quadtree */
begin
  reference node ROOT;
  boundary B;
  if not IS_LEAF (ROOT) then
    begin
      for Q in {NW, NE, SW, SE} do
           CONDENSE1 (SON (ROOT, Q));
      if HAS_LEAF_SONS (ROOT) and HAS_IDENTICAL_SONS (ROOT) then
           begin /* replace ROOT by its NW son since all 4 sons are
                   the same */
             for B in {N, E, S, W} do
               EDGE (ROOT, B) := EDGE (SON (ROOT, NW), B);
             DELETE_SONS (ROOT);
           end;
    end;
end;

boolean procedure HAS_LEAF_SONS (PARENT);
/* Check if all sons of the quadtree rooted at PARENT
   are leaves */
begin
  reference node PARENT;
  return (IS_LEAF (SON (PARENT, NE)) and
          IS_LEAF (SON (PARENT, SE)) and
          IS_LEAF (SON (PARENT, SW)) and
          IS_LEAF (SON (PARENT, NW)));
end;

boolean procedure HAS_IDENTICAL_SONS (PARENT);
/* Check if all the sons of the quadtree rooted at PARENT
   contain identical edge information */
begin
  value node PARENT;
  return (not (SONS_DISAGREE (PARENT, NE, SE) or
               SONS_DISAGREE (PARENT, SE, SW) or
               SONS_DISAGREE (PARENT, SW, NW)));
end;
```

```
boolean procedure NO_INSIDE_BORDERS (PARENT);
/* Check to make sure no edge passes between two leaves
   that are sons of PARENT */
begin
  value node PARENT;
  /* Whenever an edge passes between two adjacent leaves,
     then EDGE of both nodes has value true */
  return (not (EDGE (SON(PARENT, NE), W) or
               EDGE (SON(PARENT, SE), N) or
               EDGE (SON(PARENT, SW), E) or
               EDGE (SON(PARENT, NW), S)));
end;

boolean procedure WHOLE_OUTSIDE_BORDERS (PARENT);
/* Check to make sure each pair of sides that are to be
   merged into one side of the node PARENT agree as to
   whether or not an edge is present */
begin
  value node R;
  return (SAME_EDGE (PARENT, NW, NE, N) and
          SAME_EDGE (PARENT, NE, SE, E) and
          SAME_EDGE (PARENT, SE, SW, S) and
          SAME_EDGE (PARENT, SW, NW, W));
end;

boolean procedure SAME_EDGE (PARENT, SON_1, SON_2, SIDE);
/* Check if SON_1 and SON_2 of PARENT have same EDGE
   value on SIDE */
begin
  value node PARENT;
  value quadrant SON_1, SON_2;
  value boundary SIDE;
  return (EDGE (SON (PARENT, SON_1, SIDE) and
          EDGE (SON (PARENT, SON_2), SIDE));
end;

procedure SET_EDGES (PARENT);
/* Set each edge of the node PARENT to the merger of
   the corresponding edges of its sons */
begin
  value node PARENT;
  boundary B;
  for B in {N, E, S, W} do
   EDGE (PARENT, B) := /* The logical and of the outer
      edges on side B of the appropriate sons of PARENT */
      EDGE (SON (PARENT, QUAD (B, CSIDE (B))), B) and
      EDGE (SON (PARENT, QUAD (B, CCSIDE (B))), B);
end;
```

Algorithm 3

```
procedure   CONDENSE3 (ROOT);
/* Modify a 4-ary tree rooted at ROOT by merging as
   many nodes as possible under the criteria of
   weak-formedness to produce a quadtree */
begin
  reference node R;
  quadrant Q;
  if not IS_LEAF(ROOT) then
    begin
      for Q in {NW, NE, SW, SE} do
           CONDENSE3 (SON(ROOT, Q));
      if HAS_LEAF_SONS(ROOT) and
            NO_INSIDE_BORDERS (ROOT) then
           begin
             SET_EDGES (ROOT);
             DELETE_SONS (ROOT);
           end;
    end;
end;
```

Algorithm 4

```
procedure CONDENSE4 (ROOT, CORNERS);
/* Modify a 4-ary tree rooted at ROOT by merging as many
   nodes as possible under the criteria of weak-formedness
   and propagating edge information to the interior nodes
   of the resulting quadtree. CORNERS is used to pass
   information upward that can not be derived immediately
   from the sons of ROOT */
begin
  reference node ROOT;
  reference boolean array  CORNERS [quadrant];
  boolean array of array SONS_CORNERS [quadrant][quadrant];
  /* The first index corresponds to SON and the second
     index corresponds to a corner of the SON */
  quadrant Q;
  if IS_LEAF (ROOT) then
    SET_CORNERS (CORNERS, ROOT)
  else
    begin
      for Q in {NW, NE, SW, SE} do
          CONDENSE4 (SON(ROOT, Q), SONS_CORNERS[Q]);
      if HAS_LEAF_SONS (ROOT) and
            NO_INSIDE_BORDERS (ROOT) then
          begin
            SET_EDGES (ROOT);
            DELETE_SONS (ROOT);
            SET_CORNERS (CORNERS, ROOT);
          end
      else
          begin
            SET_INTERIOR_NODE (ROOT, SONS_CORNERS);
            for Q in {NW, NE, SW, SE} do
              CORNERS [Q] := SONS_CORNERS [Q][Q];
          end;
    end;
end;

procedure SET_CORNERS (CORNERS, LEAF);
/* Initialize the array CORNERS to indicate the presence
   or absence of touching edges in each of the four
   corners of the regions  encoded by LEAF */
begin
  reference boolean array CORNERS [quadrant];
  value node LEAF;
  boundary B;
  for B in {N, E, S, W] do
    C[QUAD(B, CSIDE(B))] := EDGE (ROOT, B) and
                            EDGE (ROOT, CSIDE(B));
end;
```

```
procedure SET_INTERIOR_NODE (ROOT, SONS_CORNERS);
/* Set the EDGE values of the interior node ROOT using
   knowledge of SONS_CORNERS to avoid setting a
   T-junction edge */
begin
  reference node ROOT;
  value boolean array of array SONS_CORNERS [quadrant][quadrant];
  /* The first index corresponds to the SON and the second
     index corresponds to a corner of the SON */
  SET_EDGES (ROOT);
  FIX_T_JUNCTION (ROOT, SONS_CORNERS);
end;

procedure FIX_T_JUNCTION (ROOT, SONS_CORNERS);
/* Examine the interior sides of the four edges of the
   node ROOT for the presence of T-junctions by use of
   SONS_CORNERS which indicate the presence or absence
   of joining edges for each corner of each son of ROOT.
   If a SET edge forms the top of a T-junction, then it
   is marked CLEAR
begin
  reference node ROOT;
  value boolean array of array SONS_CORNERS [quadrant][quadrant];
  /* The first index corresponds to SON and the second
     index corresponds to a corner of the son */
  boundary B;
  quadrants Q_B_1, Q_B_2;
  for B in {N, E, S, W} do
    begin
      /* Examine and update the two quadrants adjacent
          to side B of the node ROOT */
      Q_B_1 := QUAD (B, CSIDE(B));
      Q_B_2 := QUAD (B, CCSIDE(B));
      if EDGE (ROOT, B) then
          EDGE (ROOT, B) := not (SONS_CORNERS [Q_B_1][Q_B_2] or
                                 SONS_CORNERS [Q_B_2][Q_B_1]);

    end;
end;
```

Algorithm 5

```
node procedure CROSS_PRODUCT_1 (ROOT1, ROOT2);
/* Return the root of the quadtree for the map that
   results from overlaying the maps encoded by the
   quadtrees rooted at ROOT1 and ROOT2 */
begin
  value node ROOT1, ROOT2;
  boolean array CORNERS [quadrant]; /* Set by CONDENSE4 */
  node ROOT_RESULT;
  ROOT_RESULT := NODE_OR (ROOT1, ROOT2);
  CONSISTENT1 (ROOT_RESULT);
  CONDENSE4 (ROOT_RESULT, CORNERS);
  return (ROOT_RESULT);
end;
```

```
 node procedure  NODE_OR(ROOT1, ROOT2);
 /* Generate the 4-ary tree that results from OR-ing
    together the corresponding edge values of the
    leaves of the two  4-ary trees rooted at ROOT1
    and ROOT2 */
 begin
   value node ROOT1, ROOT2;
   node ROOT_RESULT;
   quadrant Q;
   if IS_LEAF(ROOT1) then
     return (COPY_AND_OR (ROOT1, ROOT2))
   else if IS_LEAF (ROOT2) then
     return (COPY_AND_OR (ROOT2, ROOT1))
   else
     begin
       GETFROMAVAIL (ROOT_RESULT);
       for Q in {NW, NE, SW, SE} do
           begin
             SON(ROOT_RESULT, Q) :=
               NODE_OR (SON(ROOT1, Q), SON(ROOT2, Q));
             FATHER (SON(ROOT_RESULT, Q)) := ROOT_RESULT;
           end;
       return (ROOT_RESULT);
     end;
 end;

 node procedure  COPY_AND_OR (LEAF, ROOT);
 /* Copy the 4-ary tree rooted at ROOT and OR the tree's
    sides (accessed via its adjacency trees) with the
    corresponding sides of the LEAF */
 begin
   value node LEAF, ROOT;
   node ROOT_RESULT;
   boundary B;
   ROOT_RESULT := COPY (ROOT);
   for B in {N, E, S, W} do
     begin
       if EDGE (LEAF, B) then
           MARK_SIDE (ROOT_RESULT, B);
     end;
   return (ROOT_RESULT);
 end;
```

```
node procedure  COPY(ROOT);
/* Copy the 4-ary tree rooted at ROOT.  Note that only
   leaf nodes have their edge values copied */
begin
  value node ROOT;
  node ROOT_RESULT;
  quadrant Q;
  boundary B;
  GETFROMAVAIL (ROOT_RESULT);
  if IS_LEAF (ROOT) then
    for B in {N, E, S, W} do
      EDGE (ROOT_RESULT, B) := EDGE (ROOT, B)
  else
    for Q in {NW, NE, SW, SE} do
      begin
          SON(ROOT_RESULT, Q) := COPY(SON(ROOT, Q));
          FATHER (SON(ROOT_RESULT, Q)) := ROOT_RESULT;
      end;
  return(ROOT_RESULT);
end;

procedure  MARK_SIDE (ROOT, SIDE);
/* Set the segments of the side SIDE of the 4-ary tree
   rooted at ROOT */
begin
  reference node ROOT;
  boundary SIDE;
  if IS_LEAF (ROOT) then
    EDGE(ROOT, SIDE) := true
  else
    begin
      /* Set the sides of the two subtrees of the 4-ary
           tree rooted at ROOT adjacent to SIDE */
      MARK_SIDE(SON(ROOT, QUAD(SIDE, CSIDE(SIDE))), SIDE);
      MARK_SIDE(SON(ROOT, QUAD(SIDE, CCSIDE(SIDE)), SIDE);
    end;
end;
```

```
procedure CONSISTENT1(ROOT);
/* Check the common border of neighboring nodes to
   determine whether or not they are consistent with
   the criteria of weak-formedness. When a terminal
   node is marked CLEAR on a side but the bordering
   adjacency tree is solid, then the node is marked SET */
begin
  reference node ROOT;
  quadrant Q;
  boundary B;
  if IS_LEAF(ROOT) then
    for B in {N, E, S, W} do
      begin
          if HAS_DARK_SIDE(GETNEIGHBOR (ROOT, B), UPSIDE(B))
            then EDGE (ROOT, B) := true;
      end
  else
    for Q in {NW, NE, SW, SE} do
      CONSISTENT1 (SON(ROOT, Q));
end;

boolean procedure HAS_DARK_SIDE (ROOT, SIDE);
/* Determine if all segments of the side SIDE of the
   4-ary tree rooted at ROOT are marked SET */
begin
  value node ROOT;
  value boundary SIDE;
  if IS_LEAF (ROOT) then
    return (EDGE(ROOT, SIDE))
  else
    return (HAS_DARK_SIDE (SON(ROOT, QUAD(SIDE, CSIDE(SIDE)))) and
            HAS_DARK_SIDE(SON(ROOT, QUAD(SIDE, CCSIDE(SIDE)))))
end;
```

Algorithm 6

```
node procedure CROSS_PRODUCT_2 (ROOT1, ROOT2);
/* Return the root of the quadtree for the map that results
   from overlaying the maps encoded by the quadtrees rooted
   at ROOT1 and ROOT2 */
begin
  value node ROOT1, ROOT2;
  boolean array CORNERS [quadrant]; /* Set by CONDENSE4 */
  node ROOT_RESULT, NEIGHBOR;
  boundary B;
  ROOT_RESULT := NODE_OR (ROOT1, ROOT2);
  GETFROMAVAIL (NEIGHBOR);
  for B in {N, E, S, W} do
    EDGE (NEIGHBOR, B) := true;
  CONSISTENT2 (ROOT_RESULT, NEIGHBOR, NEIGHBOR, NEIGHBOR, NEIGHBOR);
  /* The map is surrounded by a solid border! */
  CONDENSE4 (ROOT_RESULT, CORNERS);
  return (ROOT_RESULT);
end;
```

```
procedure CONSISTENT2 (ROOT, NEAR [N], NEAR [E], NEAR [S], NEAR [W]);
/* Check the common border of the four neighboring nodes, NEAR,
   of ROOT to determine whether or not they are consistent with
   the criteria of weak-formedness.  When a terminal node is
   marked clear on a side but the bordering adjacency tree is
   solid, then the node is marked SET */
begin
  reference node ROOT;
  value node array NEAR [boundary];
  /* Array NEAR is indexed by variables of the type boundary */
  boundary B;
  if IS_LEAF (ROOT) then
    for B in {N, E, S, W} do
      begin
        if HAS_DARK_SIDE (NEAR [B], OPSIDE (B)) then
          EDGE (ROOT, B):=true
    end
  else
    begin
      HELPER (SON(ROOT, NE); NEAR [N], SE; NEAR [E], NW;
                              ROOT, SE; ROOT, NW);
      HELPER (SON(ROOT, SE); ROOT, NE; NEAR [E], SW;
                              NEAR [S], NE; ROOT, SW);
      HELPER (SON(ROOT, SW); ROOT, NW; ROOT, SE;
                              NEAR [S], NW; NEAR [W], SE);
      HELPER (SON(ROOT, NW); NEAR [N], SW; ROOT, NE;
                              ROOT, SW; NEAR [W], SW);
    end;
end;

procedure HELPER (ROOT; NEAR [N], Q[N]; NEAR[E], Q[E];
                        NEAR[S], Q[S]; NEAR[W], Q[W]);
/* Helps CONSISTENT2 by determining which of its parameter
   list of neighbors have sons and which are leaves.
   NEAR[i] is a neighbor of ROOT in direction i and Q[i] is
   the desired quadrant should NEAR[i] not be a leaf */
begin
  reference node ROOT;
  value node array NEAR[boundary];
  value quadrant array Q[boundary];
  node array PARAMETER [boundary];
  boundary B;
  for B in {N, E, S, W} do
    begin
      if IS_LEAF(NEAR[B] then
          PARAMETER[B]:=NEAR[B]
      else
          PARAMETER[B]:=SON(NEAR[B], Q[B]);
    end;
  CONSISTENT2 (ROOT, PARAMETER[N], PARAMETER[E],
                     PARAMETER[S], PARAMETER[W]);
end;
```