Computing Geometric Properties of Images Represented by Linear Quadtrees

HANAN SAMET, MEMBER, IEEE, AND MARKKU TAMMINEN, MEMBER, IEEE

Abstract—The region quadtree is a hierarchical data structure that finds use in applications such as image processing, computer graphics, pattern recognition, robotics, and cartography. In order to save space, a number of pointerless quadtree representations (termed linear quadtrees) have been proposed. One representation maintains the nodes in a list ordered according to a preorder traversal of the quadtree. Using such an image representation and a graph definition of a quadtree, a general algorithm to compute geometric image properties such as the perimeter, the Euler number, and the connected components of an image is developed and analyzed. The algorithm differs from the conventional approaches to images represented by quadtrees in that it does not make use of neighbor finding methods that require the location of a nearest common ancestor. Instead, it makes use of a staircase-like data structure to represent the blocks that have been already processed. The worst-case execution time of the algorithm, when used to compute the perimeter, is proportional to the number of leaf nodes in the quadtree, which is optimal. For an image of size $2^n \times 2^n$, the perimeter algorithm requires only four arrays of 2^n positions each for working storage. This makes it well suited to processing linear quadtrees residing in secondary storage. Implementation experience has confirmed its superiority to existing approaches to computing geometric properties for images represented by quadtrees.

Index Terms—Computer graphics, connected component labeling, DF-expressions, Euler number, hierarchical data structures, image processing, linear quadtrees, pattern recognition, perimeter, quadtrees.

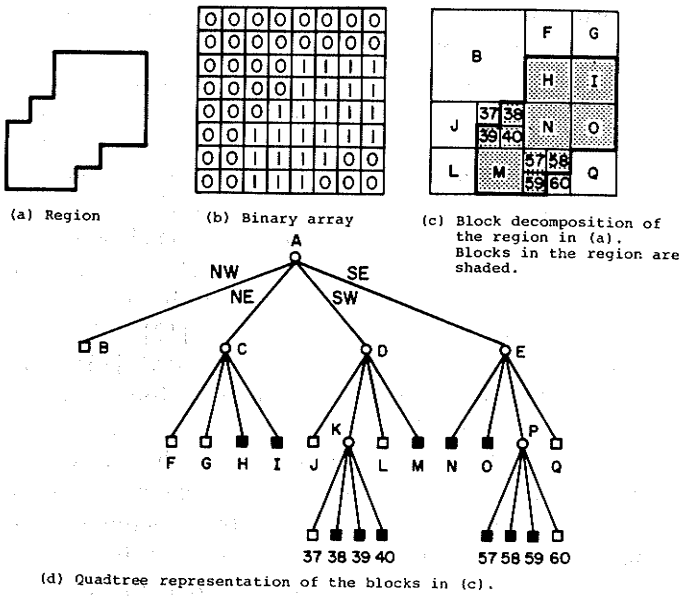

Fig. 1. A region, its binary array, its maximal blocks, and the corresponding quadtree.

I. Introduction

THE region quadtree [11], [7], a hierarchical data structure based on a regular decomposition of space, has been the subject of much research in recent years (see the survey in [22]). It and its variants have been found to be useful in such applications as image processing, computer graphics, pattern recognition, robotics, and cartography. Many algorithms for standard operations in these domains can be expressed as simple tree traversals where at each leaf a computation is performed involving that leaf and some or all of its bordering neighbors [18]. For images represented by quadtrees, algorithms for the computation of perimeter [17] and Euler number [4], as well as connected component labeling [16], have been implemented in this way. The only difference between the tree traversals is in the type of the bordering neighbors that are examined.

As an example of the quadtree, we briefly describe how it is used to represent regions. Consider the region shown in Fig. 1(a), which is represented by the $2^3 \times 2^3$ binary array in Fig. 1(b). Observe that the 1's correspond to picture elements (termed pixels) that are in the region, and the 0's correspond to picture elements that are outside the region. The quadtree approach to region representation is based on the successive subdivision of the array into four equal-size quadrants. If the array does not consist entirely of 1's or entirely of 0's (i.e., the region does not cover the entire array), then we subdivide it into quadrants, subquadrants, etc., until we obtain square blocks (possibly single pixels) that consist entirely of 1's or entirely of 0's; i.e., each block is entirely contained in the region or entirely disjoint from it. As an example, the resulting blocks for the array of Fig. 1(b) are shown in Fig. 1(c). This process is represented by a tree of out degree 4 (i.e., each nonleaf node has four sons). The root node corresponds to the entire array. Each son of the root node represents a quadrant (labeled in order NW, NE, SW, and SE). The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be BLACK or WHITE depending on whether its corresponding block is entirely inside or entirely outside of the represented region. All nonleaf nodes are said to be GRAY. The quadtree representation for Fig. 1(c) is shown in Fig. 1(d). Similar techniques can be used to represent multicolor (or gray scale) images [10].

Manuscript received December 29, 1983. Recommended for acceptance by S. Tanimoto. This work was supported in part by the National Science Foundation under Grant MCS-83-02118 and in part by the Finnish Academy.

H. Samet is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

M. Tamminen is with the Laboratory for Information Processing Science, Helsinki University of Technology, Espoo, Finland.

0162-8828/85/0300-0229$01.00

The quadtree was originally devised as an alternative to the binary array image representation with the goal of saving space by grouping similar regions. However, more importantly, the hierarchical nature of the quadtree also results in savings in execution time. In particular, algorithms for computing basic image processing operations using a quadtree representation have execution times that are only dependent on the number of blocks in the image and not their size (e.g., for connected component labeling see [16]). Nevertheless, in actuality the space required for pointers from a node to its sons is not trivial and processing pointer chains in external storage may be time consuming due to page faults. Consequently, there has been a considerable amount of interest in pointerless quadtree representations. They can be grouped into two categories. The first treats the image as an ordered collection of leaf nodes. Each leaf is represented by a locational code corresponding to a sequence of directional codes to locate the leaf along a path from the root of the tree. Locational codes have been used by a number of researchers to represent quadtrees [12], [27], [5], [14], [2], [1], as well as in other related contexts [13], [3], [26]. The second represents the image in the form of a preorder traversal of the nodes of its quadtree [9], [24].

Pointerless quadtree representations such as those described above are termed *linear quadtrees*.[1] They are attractive because of their relative compactness and their appropriateness for maintenance of the data in external storage. A linear quadtree can be viewed as a sequential file that only supports the operation NEXT( ) which yields, in sequence, the leaf nodes of the tree. Many image operations can be performed by scanning the file, in its natural order, while keeping only one or two nodes in working storage. For example, the computation of moments and set operations (e.g., union, intersection, etc.) are described in [9], [5], and [14].

In this paper, we show how linear quadtrees can be used in conjunction with algorithms for operations that also require inspection of neighbors of leaf nodes. However, we do not have to use neighbor finding methods that are based on locating a nearest common ancestor. Such methods are cumbersome for linear quadtrees and are not explicitly used in [5], [1]. Instead, to examine a particular neighboring node, its key is calculated and the linear quadtree is searched (by a modified binary search) to determine the BLACK leaf node with the given key. If such a leaf node is not found, then the color of the neighbor (WHITE or GRAY) is inferred from the next higher value in the tree (i.e., list). The key of the nearest common ancestor is implicitly determined in calculating the neighbor's key, but a search for the nearest common ancestor is not made. Nevertheless, the binary search is expensive. We present a simple general algorithm for traversing linear quadtree representations and computing geometric image properties such as the perimeter, the Euler number, and the connected components. Our algorithm only assumes that the tree can be traversed in preorder and does not require much internal storage. For example, for an image of size $2^n \times 2^n$, the perimeter algorithm requires only four arrays of 2^n positions each for working storage. We shall show that its worst-case time complexity is $O(N)$ where N is the number of leaf nodes in the quadtree.

The rest of the paper is organized as follows. Section II contains definitions and a description of the notation that we use. Section III presents a graph definition of a quadtree, which we use to give abstract algorithms for the computation of perimeter, connected component labeling, and Euler number. Section IV introduces our implementation with an algorithm for the computation of perimeter using a linear quadtree representation. Section V presents a more general algorithm for the computation of geometric properties using linear quadtrees.

II. Definitions and Notation

Each node of a quadtree corresponds to a block in the original image.[2] Each block has four sides and four corners. At times we speak of sides and corners collectively as directions. Let the four sides of a node's block be called its N, E, S, and W sides. The four corners of a node's block are labeled NW, NE, SW, and SE, with the obvious meaning. Algorithms for the computation of geometric properties require the examination of adjacent nodes. Such nodes are referred to as neighbors. In order to be more precise, given node P and a direction D, we say that Q is the *neighbor* of P in direction D, i.e., *neighbor* $(P, D) = Q$ when both of the following conditions are satisfied.

1) P and Q are adjacent along a side or a corner.

2) If D is a diagonal direction (i.e., NW, NE, SW, or SE), then node Q corresponds to the smallest block that shares the D corner of node P's block. Otherwise (D is one of the horizontal or vertical directions, i.e., N, E, S, or W), the block corresponding to Q is the smallest block (it may be GRAY) of size greater than or equal to the block corresponding to P.

For example, in Fig. 1(d) the E neighbor of node 38 is N; the NE neighbor of node L is 39; and the N neighbor of node M is K. Nodes corresponding to blocks that are adjacent to the border of the image have no neighbors in the direction of the border [e.g., the eastern sides of blocks G, I, O, and Q in Fig. 1(c)]. Note that this definition of neighbor differs slightly from the one given in [18]. In particular, we permit diagonal neighbors to be of smaller size. Two other items are worthy of note. First, the neighbor function does not define a one-to-one correspondence (i.e., a node may be a neighbor in a given direction of several nodes, i.e., in Fig. 1, *neighbor*(J, N) = B, *neighbor*(37, N) = B, and *neighbor*(38, N) = B). Second, the neighbor function is not necessarily symmetric. For example, in Fig. 1, *neighbor*(37, N) = B, but *neighbor*(B, S) = D.

For a quadtree corresponding to a $2^n \times 2^n$ image array, we say that the root is at level n. A node at level 0 corresponds to a single pixel in the image. Each pixel in the array has an x coordinate and a y coordinate between 0 and $2^n - 1$, corresponding to its position in the array. The lower left-hand

[1] We use this term to denote any data structure based on a linear ordering of the nodes of a quadtree. This is somewhat more general than the original definition of Gargantini [5], [6], who stores a list of BLACK leaf nodes with their locational codes to represent the tree structure. The remaining nodes (WHITE and GRAY) can be inferred from a sorted list of the BLACK nodes.

[2] We use the terms block and node interchangeably. The term that is used depends on whether we are referring to a block decomposition [i.e., Fig. 1(c)] or a tree [Fig. 1(d)].

corner of the image is located at (0, 0), and the upper right-hand corner is located at $(2^n - 1, 2^n - 1)$.

In the presentation of our algorithms we use a linear quadtree representation that is based on a preorder tree traversal. Of course, our results are equally applicable to other linear quadtree representations. The traversal yields a string over the alphabet "(," "*B*," and "*W*," corresponding to GRAY, BLACK, and WHITE nodes, respectively. It is due to Kawaguchi and Endo [9], and is called a DF-expression (denoting depth first). For example, the image of Fig. 1 has *(W(WWBB(W(WBBBWB (BB(BBBWW* as its DF-expression when the sons are visited in the order NW, NE, SW, and SE. In order to facilitate the expression of our algorithms, we will visit the sons in the order SW, SE, NW, and NE. Using such an order, the DF-expression for the image of Fig. 1 is *((WBW(BBWB((BWBBWBBW(BBWW*. It should be clear that the original image can be reconstructed from the DF-expression by observing that the degree of each nonterminal (i.e., GRAY) node is always 4.

III. A Graph Definition of Quadtrees

As mentioned in Section I, algorithms for the computation of geometric properties of images represented by quadtrees are implemented as tree traversals. At each node some or all of the adjacent nodes must be inspected, depending on the task being performed. In this section, we give an alternative definition of a quadtree in terms of a graph. Using such a definition facilitates the formulation of a number of geometric property computation algorithms. In particular, it is especially well suited to linear quadtrees since it eliminates the need for the use of neighbor finding techniques based on locating the nearest common ancestor [18] or searching [5], [1]. Instead, our algorithms are expressed in terms of blocks, edges, and vertices.

The partition of space induced by a quadtree may be viewed as an undirected planar graph. The vertices of the graph correspond to the corners of the leaf nodes comprising the quadtree. The edges of the graph are the sides of the leaf nodes. For example, Fig. 2 is the graph representation of the block decomposition in Fig. 1(c). The vertices are labeled *AA-BF*, and the edges are labeled *xa-xy* and *ya-yy*. More precisely, edges are defined as follows: given a leaf P and a side D between vertices V_1 and V_2 of P, there exists an edge between V_1 and V_2 if and only if *neighbor*(P, D) is not GRAY (e.g., in Fig. 1, edge *ye* joins vertices *AN* and *AR* of 39, while there is no edge between vertices *AI* and *AR* of *J*). In order to cope with leaf nodes that are adjacent to the border (e.g., the W side of node *L* in Fig. 1), we say that the image is surrounded by WHITE or BLACK as is appropriate to the task being performed.

It is convenient to associate some additional information with edges and vertices. First, we label edges and vertices as BLACK, WHITE, or GRAY. An edge is defined to be GRAY if its adjacent leaf nodes differ in color, and it is BLACK (WHITE) if its adjacent leaf nodes are both BLACK (WHITE). Similarly, a vertex is said to be BLACK (WHITE) if all edges emanating from it are BLACK (WHITE); otherwise, it is GRAY. For example, in Fig. 2 edge *ys* is BLACK, edge *xf* is WHITE, and edge *ye* is GRAY; vertex *AT* is BLACK, vertex *AI* is WHITE, and vertex *AJ* is GRAY. Second, the width of an

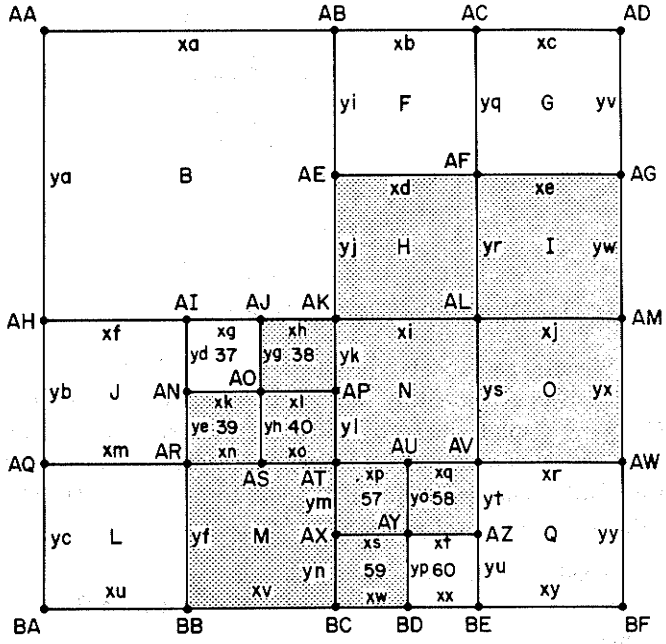

Fig. 2. Graph representation of the quadtree of Fig. 1.

edge is the distance between the two vertices comprising it and is accessed by the use of the function WID.

The description of the relationship between blocks, edges, and vertices is facilitated by the use of the *TOUCH* function. When X is a vertex, the value of *TOUCH* (X) is the set of blocks whose corners meet at X. A similar definition holds for edges. For example, in Fig. 1, $TOUCH(yj) = \{B, H\}$, $TOUCH(AK) = \{B, H, N, 38\}$, and $TOUCH(AE) = \{B, F, H\}$.

The computation of the perimeter of an image represented by a quadtree is a tree traversal where at each BLACK node we look for adjacent WHITE nodes [17]. Using our graph formulation of a quadtree, we merely need to accumulate the sum of the widths of all GRAY edges as shown by procedure PERIMETER given below.

```
integer procedure PERIMETER(Q);
/* Compute the perimeter of quadtree Q. */
begin
  value quadtree Q;
  edge E;
  integer PER;
  PER ← 0;
  foreach E in {edges(Q)} do
    begin
      if GRAY(E) then PER ← PER + WID(E);
    end;
  return (PER);
end;
```

Labeling the connected components of an image represented by a quadtree is a three-step process [16]. The first step is a tree traversal where for each BLACK node we find all adjacent nodes and assign them the same label. If some of the nodes have already been assigned a label, then we note the equivalence. The second step merges all the equivalence pairs that were generated during the first step. The third step performs another

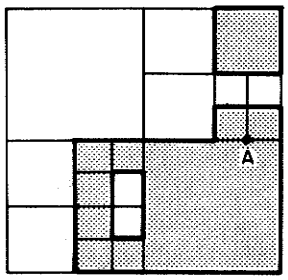

Fig. 3. Sample image containing two connected components.

tree traversal and updates the labels on the nodes to reflect the equivalences generated by the first two steps of the algorithm. Using our graph formulation of the quadtree, step one becomes a simple examination of all the BLACK edges. Steps two and three are unchanged from the original formulation. The algorithm is given below by procedure CCL.

```
procedure CCL(Q);
/* Label the connected components of quadtree Q. */
begin
   value quadtree Q;
   edge E;
   block B;
   foreach E in {edges(Q)} do
      begin /* Step one */
         if BLACK(E) then ASSIGN_SAME_LABEL
            (TOUCH(E));
      end;
   MERGE_EQUIVALENCES; /* Step two */
   foreach B in {blocks(Q)} do
      begin /* Step three */
         if BLACK(B) then UPDATE(B);
      end;
end;
```

Applying the algorithm to Fig. 3 yields two connected components. Note that the first and second steps can be combined to form one step by using the *UNION-FIND* algorithm [25] thereby resulting in a two-step process. However, by separating these tasks we are better able to see the similarity between the algorithms for the different geometric properties.

The Euler number (or genus) of an image is the difference between the number of connected components and the number of holes. Dyer has shown that the Euler number of an image represented by a quadtree is $B - E + V$ [4] where B is the number of BLACK blocks, E is the number of pairs of adjacent BLACK blocks in the horizontal and vertical directions, and V is the number of cases where three or four BLACK blocks share a corner (i.e., there exists a 2×2 block of BLACK pixels such that at least three out of the four pixels are in different BLACK blocks). For example, for Fig. 3, $B = 10$, $E = 10$, and $V = 1$ (contributed by vertex A) yielding a Euler number of 1. The algorithm is analogous to the first step of connected component labeling with the added provision that in order to determine V, for each BLACK block, the leaf nodes surrounding its SE corner must be examined as must vertices on its S and W sides. For example, when processing node N of Fig. 2 we inspect the leaves surrounding vertices AT, AP, and AU. Using our graph formulation of the quadtree, we merely need to accumulate the number of BLACK blocks, BLACK edges, and BLACK vertices. The algorithm is given below using procedure GENUS.

```
integer procedure GENUS(Q);
/* Compute the genus of quadtree Q. */
begin
   value quadtree Q;
   block B;
   edge E;
   vertex V;
   integer TB, TE, TV;
   TB ← TE ← TV ← 0;
   foreach B in {blocks(Q)} do
      begin
         if BLACK(B) then TB ← TB + 1;
      end;
   foreach E in {edges(Q)} do
      begin
         if BLACK(E) then TE ← TE + 1;
      end;
   foreach V in {vertices(Q)} do
      begin
         if BLACK(V) then TV ← TV + 1;
      end;
   return (TB - TE + TV);
end;
```

IV. Computing Perimeters of Linear Quadtrees

Recall from Sections I and II that a linear quadtree represents an image as a preorder traversal of the leaf nodes comprising it. The traversal of the blocks, edges, and vertices of the graph of a quadtree described in Section II can be implemented by a traversal of the leaves of the tree. In this section, we show how this can be achieved for the task of perimeter computation in such a way that each leaf node is examined exactly once. At any instant, the state of the traversal (i.e., after processing m leaf nodes) can be visualized as a staircase (termed an *active border*). The active border partitions the leaf nodes into two sets: those nodes that have already been visited and those that are to be visited in the future. In particular, given a traversal in the order SW, SE, NW, NE, the part of the image described by the m leaf nodes is to the left of and below the staircase.[3] The active border consists of a set of *active edges*. This set can be decomposed into a set of active horizontal edges and active vertical edges. For example, for Fig. 1, a traversal in the above order (without GRAY nodes) is *L, M, J,* 39, 40, 37, 38, 59, 60, 57, 58, *Q, N, O, B, H, I, F, G*. Fig. 4 shows the active border after processing node 37. In the following, we describe a process that systematically visits the edges of the quadtree without requiring the use of neighbor finding methods that are based on locating the nearest common ancestor [18].

From Section III we know that the computation of the perimeter only requires the accumulation of the widths of all

[3] In fact, any traversal order that forms a staircase can be used with our algorithms once appropriate changes are made. Thus, for instance, the order SW, NW, NE, SE would also be permitted, as would others.

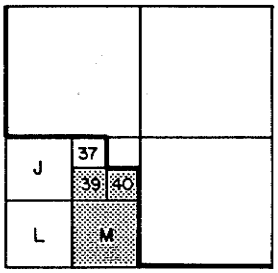

Fig. 4. The active border after processing node 37 in Fig. 1.

GRAY edges. This process is facilitated by the active border which records the color and size (i.e., width) of each leaf node that is adjacent to it. We represent the active border by two arrays of active edges, termed XEDGE and YEDGE, corresponding to the active horizontal and vertical edges and indexed by the x and y coordinates, respectively, of their starting pixels. This enables the cost of accessing the edge data structure to be independent of the number of edges. XEDGE and YEDGE are implemented as arrays of pointers to records of type *edge*. Each *edge* represents an active edge. It has two fields, WID and COL, corresponding to the width and the color of the block adjacent to the side of the edge that has already been processed. Initially, there is one active horizontal edge and one active vertical edge, both of width 2^n and color WHITE at address 0. For example, Fig. 4 shows the active border after processing node 37, while Table I represents the state of the XEDGE and YEDGE arrays.

Perimeter computation is performed by procedures KPERIMETER, PTRAVERSE, and INCREMENT. They are given below using a variant of Algol. Procedure KPERIMETER is invoked with the DF-expression encoding of the nodes in the quadtree. It initializes the active border and then invokes procedure PTRAVERSE, which controls the traversal of the quadtree nodes. For each leaf node, PTRAVERSE calls procedure INCREMENT twice, i.e., once for the southern side of a leaf and once for the western side. INCREMENT augments the perimeter and updates the active border.

In order to see how INCREMENT works, suppose that we are adding leaf node L of width W, color C, and leftmost pixel at x coordinate X. We illustrate the workings of INCREMENT with respect to the southern side of L. Let the southern neighbor of L be Q. Note that at this point Q has already been processed, and so all the relevant data with respect to the edges between L and Q are represented by XEDGE$[X]$, i.e., their color and width. There are three possibilities. First, if Q is of the same size as L, and Q is a leaf node, then we simply check if the edge is GRAY (i.e., L and Q differ in color along their shared side). If yes, then the perimeter is augmented by W. In any case, XEDGE$[X]$ is updated to reflect the new active edge, i.e., the northern side of L. As an example, after processing leaf node 37 in Fig. 1, the perimeter is augmented by 1 corresponding to the width of leaf 37 (assuming an 8×8 pixel array) since the edge between 37 and its southern neighbor, 39, is GRAY. The color field of the active edge is changed from BLACK to WHITE because 37 is a WHITE leaf. The width field remains the same in this case.

TABLE I
XEDGE AND YEDGE AFTER PROCESSING NODE 37 IN FIG. 1

I	XEDGE [I]			YEDGE [I]		
	LEN	COL	BLOCK	LEN	COL	BLOCK
0	2	WHITE	J	2	BLACK	M
1						
2	1	WHITE	37	1	BLACK	40
3	1	BLACK	40	1	WHITE	37
4	4	WHITE	BORDER	4	WHITE	BORDER
5						
6						
7						

Second, if Q is larger than L, then Q must correspond to a leaf node. If the edge between L and Q is GRAY, then the perimeter is augmented by W. In any case, XEDGE$[X]$ is updated to reflect the two new active edges. In particular, XEDGE$[X]$ is set to the northern side of L. In addition, there is a new segment of the active border that corresponds to the part of the active edge associated with Q that has not been replaced by the northern side of L. As an example, after processing leaf node 39 in Fig. 1, the perimeter is unchanged since the edge between it and its southern neighbor, M, is BLACK. The color field of the active edge at XEDGE[2] remains BLACK, but the width field is changed from 2 to 1. Since M is larger than 39, the part of the active border between pixel locations 3 and 4 now forms an active edge. In particular, the color field of XEDGE[3] is set to BLACK, and its width field is set to 1.

Third, if Q is of the same size as L but corresponds to a GRAY node (i.e., WID(XEDGE$[X]$) $< W$), then we must process all of the edges between L and the northernmost descendants of Q. For each such edge that is GRAY, the perimeter is augmented by the width of the adjacent northernmost descendant of Q. In any case, XEDGE$[X]$ is updated to reflect the new active edge, i.e., the northern side of L. As an example, in Fig. 1 the southern neighbor of leaf node B is GRAY node D. After processing B, the perimeter is augmented by 1 corresponding to the width of node 38 since it is the only northernmost descendant of D that has a GRAY edge between it and B. The color field of the active edge (i.e., XEDGE[0]) remains WHITE. The width field of the active edge is set to 4 (i.e., XEDGE[0]).

Procedure INCREMENT must also do some extra work when processing a node adjacent to the border of the image. In particular, it must determine if the edge formed by the block and the border is GRAY (assuming that the image is surrounded by WHITE). For example, after processing node I in Fig. 1 we are at the eastern border of the image. This is detected after the application of INCREMENT to the southern side of I since the x coordinate of the westernmost pixel of I is available for such a test. In this example, the edge is GRAY, and the perimeter is augmented by 2 (i.e., the width of I).

V. A General Geometric Property Computation Algorithm

In Section IV we saw how to compute the perimeter of an image represented by a linear quadtree. This was facilitated

by use of a data structure which was called an *active border*. In this section we show how additional geometric properties such as the Euler number and connected component labeling can be computed for linear quadtrees. This is done in the context of a general algorithm which is able to compute all three geometric properties. In the process we show how the active border data structure is modified to enable the computation of these properties.

The algorithms that we study are of two types: those that are based solely on the inspection of adjacencies at edges and those that also require the inspection of adjacencies at vertices. Computation of perimeter and connected components are of the former type, while the computation of the Euler number is of the latter type. In order to label connected components, we need only add a field to the active border data structure that records the label assigned to the block adjacent to the side of the edge that has already been processed. Thus, the *edge* record type now contains an additional field termed LAB.

The computation of the Euler number requires an additional data structure to keep track of the vertices. To see this, we note that for each leaf, say L, of the preorder traversal, we must determine if the SW vertex of L, say X, is BLACK. This is the last opportunity to examine this vertex since it will never again be adjacent to the active border. This statement is easily verified by observing the staircase nature of the active border and the fact that the tree is traversed in the order SW, SE, NW, NE. In other words, the active border expands in the rightward and upward directions. The active border data structure used for computing perimeter and connected components does not always contain enough information to determine the color of X. In particular, it only keeps track of the colors of the leaf nodes that are adjacent to the border along a side and not at a corner. For example, Fig. 4 and Table I show the active border after processing node 37 in Fig. 1. The next node to be processed is 38, but we do not know the color of the block adjacent to its SW corner. To rectify this situation, we define an additional entity termed an *active vertex*, which corresponds to a vertex on the active border. The set of all such vertices is called the *active vertices*. It is implemented as an array, VRTX, of pointers to records of type *vertex*. A record of type *vertex* has one field, COL, corresponding to the color of the block whose NE corner touches the vertex. For a $2^n \times 2^n$ image array, at any one time, VRTX has a maximum of $2^{n+1}-1$ entries. Given a vertex at (x, y), VRTX$[x - y]$ corresponds to its *vertex* record. Since the line $K = x - y$ can only intersect the active border at one position, we see that no two active vertices may simultaneously have the same VRTX entry.

The general geometric property computation algorithm can be decomposed into two sets of procedures. The first set consists of procedures GEOM_PROPERTY, TRAVERSE, PROCESS_LEAF, PROCESS_SIDE, UPDATE_VERTEX, and UPDATE_EDGE. They are common to the computation of all geometric properties. The actual computation is controlled by procedure GEOM_PROPERTY, which is invoked with the DF-expression encoding of the nodes in the quadtree. It initializes the active border and the active vertices and then invokes procedure TRAVERSE, which controls the traversal of the quadtree nodes. For each leaf node, TRAVERSE calls procedure PROCESS_LEAF to compute the desired geometric property. PROCESS_LEAF invokes procedure PROCESS_SIDE twice: once for the southern border and once for the western border. Procedures UPDATE_VERTEX and UPDATE_EDGE are used to update the active vertices and active border, respectively. TRAVERSE assumes that the linear quadtree is in the form of a DF-expression. However, it should be clear that TRAVERSE could be easily modified to handle other linear quadtree representations. Note that we make use of a record of type *leaf* to represent the leaf nodes. It is an *edge* record with two additional fields, XPOS and YPOS, corresponding to the coordinates of the lower left-hand corner of the leaf.

The second set consists of procedures necessary to compute the geometric property in question and thus may be coded differently depending on the geometric property. It consists of procedures PROCESS_VERTEX, PROCESS_BLOCK, PROCESS_EDGE, PROCESS_EDGES, and PROCESS_BORDER. The actual code for these procedures is given in the Appendix for the computation of perimeter, connected component labeling, and Euler number. In the following we describe briefly their meaning. Note that for a specific geometric property not all of the procedures are used.

Procedure PROCESS_LEAF is the controller of the computation process. For each leaf node, say L, it performs the following three steps. First, the vertex in which the SW corner of L participates as an NE component is now fully specified (i.e., the colors of the blocks touching it are all known), and it is processed by PROCESS_VERTEX. This is useful in the computation of the Euler number. We also use UPDATE_VERTEX to update the VRTX entry corresponding to the NE corner of L (e.g., after processing leaf node G in Fig. 1, the color of the active vertex corresponding to the NE corner of G is changed to WHITE from BLACK, which was the color associated with the previous VRTX entry, i.e., corresponding to the NE corner of H). Second, PROCESS_SIDE is invoked twice: once to process all the edges and vertices on the W side of L and once for those on the S side. Finally, PROCESS_BLOCK is invoked to perform any computation that involves the leaf node alone. For example, when computing the Euler number we must keep track of the number of BLACK leaf nodes.

Procedure PROCESS_SIDE has the same structure as procedure INCREMENT described in Section IV for the computation of perimeter. Suppose that we are adding leaf node L of width W, color C, and leftmost pixel at x coordinate X. Assume that we are processing the southern side of L and let Q be the southern neighbor of L (recall the definition of neighbor in Section II). PROCESS_SIDE must process the edges and vertices between Q and L depending on their relative sizes. This is described below. In addition, it updates the active border for the northern side of the new leaf L by use of UPDATE_EDGE. Finally, it checks to see if the eastern side of the new leaf is on the border, in which case it invokes PROCESS_BORDER to perform any necessary additional computations (e.g., for perimeter).

When processing the edges and vertices lying between Q and L, there are three possibilities. First, if Q is of the same size as

L, and Q is a leaf node, then we simply compute the geometric property for the edge between L and Q using PROCESS_EDGE. Second, if Q is larger than L, then Q must correspond to a leaf node. Again, we compute the geometric property for the edge between L and Q using PROCESS_EDGE. In addition, we invoke UPDATE_VERTEX and UPDATE_EDGE to update the active vertices and active border data structures, respectively, to reflect the additional vertex and the reduction in size of a component of the active border. As an example, after processing leaf node 39 in Fig. 1, we find that the size of the contribution of M to the active border changes from 2 to 1 and lies between pixel positions 3 and 4. Moreover, there is now a vertex at (3, 2), and the set of active vertices is modified accordingly. Third, if Q is of the same size as L but corresponds to a GRAY node, then we compute the geometric property for the edges and vertices between L and the northernmost descendants of Q. This is done by PROCESS_EDGES and PROCESS_VERTEX. Note that this is the last time that we have an opportunity to examine these edges and vertices since they will never again be adjacent to the active border. As an example, in Fig. 1 the southern neighbor of leaf node B is GRAY node D. From Fig. 3 we see that we must compute the geometric property for edges xf, xg, and xh and vertices AI and AJ.

The detailed code for procedures PROCESS_VERTEX, PROCESS_BLOCK, PROCESS_EDGE, PROCESS_EDGES, and PROCESS_BORDER for each geometric property, when they are used, is given in the Appendix. For perimeter computation, PROCESS_EDGE and PROCESS_EDGES check for GRAY edges. Assuming that the image is surrounded by WHITE, PROCESS_BORDER determines if the border edge is GRAY. PROCESS_VERTEX and PROCESS_BLOCK are not used.

For connected component labeling, PROCESS_EDGE and PROCESS_EDGES check for BLACK edges and merge if necessary. PROCESS_VERTEX, PROCESS_BLOCK, and PROCESS_BORDER are not used.

For Euler number computation, PROCESS_EDGE and PROCESS_EDGES check for BLACK edges. PROCESS_VERTEX checks for BLACK vertices. PROCESS_BLOCK checks for BLACK leaf nodes. PROCESS_BORDER is not used.

VI. Concluding Remarks

A general algorithm for the computation of geometric properties for images represented by linear quadtrees has been presented. The key to the success of the algorithm is the use of a staircase-like data structure to represent the active border as the tree is traversed. Use of a graph-theoretic definition of a quadtree simplified the presentation considerably. The algorithm was discussed in the context of a linear quadtree representation in the form of a DF-expression. However, it should be clear that it can be used with other linear quadtree representations. No general algorithms for such tasks using linear quadtrees have existed before.

Computation of the perimeter and the Euler number only requires one pass over the data whereas connected component labeling requires two passes (one for determining equivalences and one for updating; recall that steps one and two can be combined). The execution time of perimeter, Euler number, and the first step of connected component labeling using our general algorithm is proportional to the number of leaf nodes in the image. This is better than results achievable by algorithms based on neighbor finding using a nearest common ancestor. The problem with the latter is that although they have linear average case behavior, in the worst case, even assuming an explicit tree representation, their execution time is proportional to $T \cdot n$ where T is the number of leaf nodes in a $2^n \times 2^n$ image. The worst case for nearest common ancestor neighbor finding methods arises when the nearest common ancestor is the root of the tree. For example, in Fig. 1, to locate the eastern neighbor of node 38 (i.e., N), we must ascend through K, D, and A and descend to N through E. For a more general example of this worst case behavior, see [20]. Note also that Jackins and Tanimoto [8] have devised an $O(T)$ perimeter computation algorithm applicable to explicit quadtree representations. However, their algorithm traverses the quadtree twice, whereas we only need to traverse it once.

Using our algorithms, the amount of extra storage beyond that required for the quadtree nodes is three arrays. We assume a $2^n \times 2^n$ image. Two of the arrays are used for the active border, have a maximum of 2^n entries, and contain three fields per entry (i.e., WID, COL, and LAB). The third array for the active vertices has a maximum of $2^{n+1} - 1$ entries and contains one field per entry. The active border and active vertices data structures could also be implemented as linked lists with a slightly more complex access mechanism. This would have the advantage of not needing to store more entries than there are actual edges and vertices. The quadtree nodes are stored in external memory. However, it is preferable to store the active border and vertices arrays in internal memory.

The algorithms presented in this paper were tried out on an existing geographic information system based on linear quadtrees [15]. Due to the large amount of data, the system is disk based. Use of the new algorithm resulted in an order of magnitude improvement in the execution time of algorithms such as perimeter computation. Aside from the decrease in the computational complexity of the new algorithm, other reasons for the speed-up include the absence of page faults due to the ability to process the nodes of the quadtree in a sequential order. In experiments involving trees stored in central memory, the speed-up was smaller but still considerable (up to sixfold). However, we refrain from giving exact figures due to the great dependence of the performance of the Gargantini encoding scheme [5] on implementation details.

Future work includes an attempt to compute distance transforms [19] and a quadtree medial axis transform [21] using linear quadtrees. This is somewhat more difficult than the algorithms which we have attempted above since we must examine all sides of a leaf node as well as neighbors of neighbors. Most likely, such tasks will require more than one pass over the leaf nodes comprising the linear quadtree. Geometric properties for linear octrees [6] (e.g., connected component labeling, surface area, etc.) could also be computed using similar techniques. However, the active border and active vertices arrays will now get rather large, and it would be preferable to use linked lists.

Such an approach, having the same execution time behavior, is used in [23] with a different graph definition of a quadtree.

Appendix
Algorithms for the Individual Geometric Properties

Perimeter computation:

```
procedure PROCESS_EDGE(L, E);
begin
   value pointer leaf L;
   value pointer edge E;
   global integer PER;
   if COL(L) NEQ COL(E) then /*GRAY edge*/
      PER ← PER + WID(L);
end;

procedure PROCESS_EDGES(L, E);
begin
   value pointer leaf L;
   value pointer edge E;
   global integer PER;
   if COL(L) NEQ COL(E) then /*GRAY edge*/
      PER ← PER + WID(E);
end;

procedure PROCESS_BORDER(L);
begin
   value pointer leaf L;
   global integer PER;
   if COL(L) = BLACK then /*GRAY border edge*/
      PER ← PER + WID(L);
end;
```

Connected component labeling:

```
procedure PROCESS_EDGE(L, E);
begin
   value pointer leaf L;
   value pointer edge E;
   if COL(L) = BLACK and COL(E) = BLACK
   then /*BLACK edge*/
      begin
         if labeled (L) then MERGE(LAB(L), LAB(E))
         else LAB(L) ← LAB(E);
      end;
end;

procedure PROCESS_EDGES(L, E);
begin
   value pointer leaf L;
   value pointer edge E;
   PROCESS_EDGE(L, E);
end;
```

Euler number computation:

```
procedure PROCESS_BLOCK(L);
begin
   value pointer leaf L;
   global integer TB;
   if COL(L) = BLACK then /*BLACK block*/
      TB ← TB + 1;
end;

procedure PROCESS_EDGE(L, E);
begin
   value pointer leaf L;
   value pointer edge E;
   global integer TE;
   if COL(L) = BLACK and COL(E) = BLACK
   then /*BLACK edge*/
      TE ← TE + 1;
end;

procedure PROCESS_EDGES(L, E);
begin
   value pointer leaf L;
   value pointer edge E;
   PROCESS_EDGE(L, E);
end;

procedure PROCESS_VERTEX(C1, C2, C3, C4);
begin
   value color C1, C2, C3, C4;
   global integer TV;
   if C1 = BLACK and C2 = BLACK and C3 = BLACK
   and C4 = BLACK then
      TV ← TV + 1; /*BLACK vertex*/
end;

integer procedure KPERIMETER(M,DF);
/* Compute the perimeter of an M by M (M = 2^n) image
   represented by DF, a preorder traversal of its quadtree.
   Each invocation of NEXT(DF) provides the next element in the list and advances the pointer DF. XEDGE
   and YEDGE are arrays of pointers to records of type
   edge that represent the active edges in the x and y directions, respectively. A record of type edge has two fields,
   WID and COL, corresponding, respectively, to the width
   and the color of the block adjacent to the side of the
   edge that has already been processed. */
begin
   global value integer M;
   global value pointer nodelist DF;
   global pointer edge array XEDGE, YEDGE [0:M-1];
   /* Initialize XEDGE and YEDGE to represent one active
      edge of width M at location 0 and adjacent to WHITE
      blocks: */
   WID(XEDGE[0]) ← WID(YEDGE[0]) ← M;
   COL(XEDGE[0]) ← COL(YEDGE[0]) ← WHITE;
   return (if empty (DF) then 0
           else PTRAVERSE(M,0,0));
end;

integer procedure PTRAVERSE(W,X,Y);
/* Compute the perimeter of a W by W segment of an M
   by M image where DF represents the preorder traversal
   of its quadtree. The lower left corner of the W by W
   segment has x and y coordinates of X and Y, respectively. */
```

```
begin
  value integer W,X,Y;
  global pointer nodelist DF;
  global integer M;
  global pointer edge array XEDGE, YEDGE [0:M-1];
  integer T;
  color P; /*P takes on the values "(," "B," and "W" */
  P ← NEXT(DF);
  if P = "(" then
    begin /*A nonterminal node */
      T ← 0;
      W ← W/2;
      T ← PTRAVERSE(W,X,Y) +
          PTRAVERSE(W,X+W,Y) +
          PTRAVERSE(W,X,Y+W) +
          PTRAVERSE(W,X+W,Y+W);
          /*Process SW, SE, NW,
          and NE sons in order.*/
    end
  else
  T ← INCREMENT(XEDGE, X, W, P) +
      INCREMENT(YEDGE, Y, W, P);
      /*Horizontal contribution and
      vertical contribution*/
  return (T);
end;

integer procedure INCREMENT (E,X,W,C);
/* Compute the contribution to the perimeter of the side
   adjacent to the active border, represented by E, of a
   leaf of color C, side length W, and at position X relative
   to the start of the active border. */
begin
  value pointer edge array E[0:M-1];
  value integer X, W;
  value color C;
  global integer M;
  integer I, PER;
  if W > WID(E[X]) then /*Is the neighbor of the new leaf
                         a GRAY node? */
    begin /*Yes, process the edges adjacent to the new
            leaf*/
      PER ← 0;
      I ← X;
      while I < X+W do
        begin /*Update the perimeter for GRAY edges */
          if C NEQ COL(E[I])
          then PER ← PER + WID(E[I]);
          I ← I + WID(E[I]);
        end;
    end;
  else /*The neighbor is a leaf node*/
    begin
      if C NEQ COL(E[X])
      then PER ← W /*GRAY edge*/
      else PER ← 0;
      if W < WID(E[X])
      then /*Is the neighbor larger? */
        begin /*Yes, update the active border for the
                neighbor */
          WID(E[X+W]) ← WID(E[X]) - W;
          COL(E[X+W]) ← COL(E[X]);
        end;
    end;
  /*Update the active border for the new leaf: */
  WID(E[X]) ← W;
  COL(E[X]) ← C;
  if X+W=M then /*Is one of the other sides of the new
                  leaf on the image border? */
    begin /*Yes, update the perimeter if necessary*/
      if C = BLACK then PER ← PER + W;
    end;
  return (PER);
end;

procedure GEOM_PROPERTY (M,DF);
/* Compute a geometric property of an M by M (M = 2^n)
   image represented by DF, a preorder traversal of its
   quadtree. Each invocation of NEXT(DF) provides the
   next element in the list and advances the pointer DF.
   XEDGE and YEDGE are arrays of pointers to records
   of type edge that represent the active edges in the x and
   y directions, respectively. A record of type edge has
   three fields WID, COL, and LAB, corresponding, re-
   spectively, to the width, color, and label of the block
   adjacent to the side of the edge that has already been
   processed. VRTX is an array of pointers to records of
   type vertex that represent the active vertices. A record
   of type vertex has one field, COL, corresponding to the
   color of the block whose NE corner touches the vertex.
   LL points to the start of a list of the leaf nodes that
   have been processed by TRAVERSE. This is useful
   when there is a need for further processing as in the
   case of connected component labeling.*/
begin
  global value integer M;
  global value pointer nodelist DF;
  global pointer edge array XEDGE, YEDGE [0:M-1];
  global pointer vertex array VRTX [-M+1:M-1];
  global pointer leaflist LL;
  /* Initialize XEDGE and YEDGE to represent one active
     edge of width M at location 0 and adjacent to WHITE
     blocks: */
  WID(XEDGE[0]) ← WID(YEDGE[0]) ← M;
  COL(XEDGE[0]) ← COL(YEDGE[0]) ← WHITE;
  /* Initialize VRTX to represent one active vertex with
     a white block at its SW corner: */
  COL(VRTX[0]) ← WHITE;
  /* Perform any special initialization for the geometric
     property being computed: */
  INITIALIZE_PROPERTY():
  if not (empty (DF)) then TRAVERSE(M,0,0);
  /* Perform any additional steps—e.g., phases 2 and 3 of
     connected component labeling: */
  PERFORM_OTHER_STEPS(LL);
end;
```

```
procedure TRAVERSE(W,X,Y);
/* Compute a geometric property of a W by W segment of
   an M by M image where DF represents the preorder tra-
   versal of its quadtree. The lower left corner of the W
   by W segment has x and y coordinates of X and Y, re-
   spectively. For each leaf node, procedure PROCESS_
   LEAF is invoked to perform the appropriate computa-
   tion. After processing each leaf node, it is added to a
   list, LL, of records of type leaf each of which has five
   fields, WID, COL, LAB, XPOS, and YPOS corresponding
   to the width of its side, color, label, and x and y coor-
   dinates of its lower left corner, respectively.*/
begin
  value integer W, X, Y;
  global pointer nodelist DF;
  global integer M;
  global pointer edge array XEDGE, YEDGE[0:M-1];
  global pointer vertex array VRTX [-M+1:M-1];
  global pointer leaflist LL;
  color P; /*P takes on the values "(," "B," and "W"*/
  pointer leaf L;
  P ← NEXT(DF);
  If P = "(" then
    begin /*Nonleaf node*/
      W ← W/2;
      TRAVERSE(W,X,Y);
      TRAVERSE(W,X+W,Y);
      TRAVERSE(W,X,Y+W);
      TRAVERSE(W,X+W,Y+W);
      /*Process SW, SE, NW,
      and NE sons*/
    end
  else
    begin /*Leaf node*/
      L ← createnode (leaf);
      WID(L) ← W;
      COL(L) ← P;
      LAB(L) ← "unknown";
      XPOS(L) ← X;
      YPOS(L) ← Y;
      PROCESS_LEAF(L);
      addtolist (LL,L);
    end;
end

procedure PROCESS_LEAF(L);
/* Determine the contribution of leaf L to the geometric
   property being computed. Update the active border and
   active vertices data structures to reflect the processing
   of L.*/
begin
  value pointer leaf L;
  global pointer edge array XEDGE, YEDGE[0:M-1];
  global pointer vertex array VRTX[-M+1:M-1];
  global integer M;
  integer W,X,Y;
  color C;
  X ← XPOS(L); /*Unpack some of the fields of the leaf
                 L*/
  Y ← YPOS(L);
  W ← WID(L);
  C ← COL(L);
  /* Process the SW vertex of L, i.e., the colors of all the
     leaf nodes surrounding it are now known and its color
     can be determined:*/
  PROCESS_VERTEX(C,COL(VRTX[X-Y]),
     COL(XEDGE[X]),COL(YEDGE[Y]));
  /* Update the VRTX entry corresponding to the NE
     corner of L:*/
  UPDATE_VERTEX(X-Y,L);
  /* Process all the edges and vertices on the S side of
     L:*/
  PROCESS_SIDE(XEDGE,X,X+W-Y,L);
  /* Process all the edges and vertices on the W side of L:*/
  PROCESS_SIDE(YEDGE,Y,X-Y-W,L);
  /* Process the block corresponding to the leaf L:*/
  PROCESS_BLOCK(L);
end;

procedure PROCESS_SIDE(E,X,V,L);
/* Determine the contribution of a side of the new leaf L
   (side width W and color C) to the geometric property
   being computed. The segment of the active border that
   is adjacent to the side is represented by E starting at en-
   try X. V is the index of the active vertices data structure
   (i.e., VRTX) that corresponds to the vertex at pixel
   X+W. V is used in UPDATE_VERTEX.*/
begin
  value pointer edge array E[0:M-1];
  value integer X,V;
  value pointer leaf L;
  global integer M;
  integer I, VI, W;
  W ← WID(L);
  if W > WID(E[X])
  then /*Is the neighbor of the new leaf a GRAY node?*/
    begin /*Yes, process the edges adjacent to the new
            leaf*/
      PROCESS_EDGES(L,E[X]);
      VI ← X;
      I ← X + WID(E[X]);
      while I < X+W do
        begin /*Compute the geometric property for
                GRAY edges*/
          PROCESS_VERTEX(COL(E[VI]),
            COL(E[I]),COL(L),COL(L));
          PROCESS_EDGES(L,E[I]);
          VI ← I;
          I ← I + WID(E[I]);
        end;
    end
  else /*The neighbor is a leaf node*/
    begin
      PROCESS_EDGE(L,E[X]);
```

```
        if W < WID(E[X])
        then /*Is the neighbor larger? */
          begin /*Yes, update the active border and vertex
                  for the neighbor: */
            UPDATE_EDGE(E[X+W],W,E[X]);
            UPDATE_VERTEX(V,E[X]);
          end;
      end;
  /* Update the active border for the new leaf */
  UPDATE_EDGE(E[X],W,L);
  if X+W = M then /*Is the other side of the new leaf on
                    the image border? */
    PROCESS_BORDER(L);/*Yes, update the geometric prop-
                        erty if necessary*/
    end;


    procedure UPDATE_VERTEX(V,EL);
    /* Set the color of the Vth entry of vertex array VRTX to
       the color associated with edge or leaf EL. The argument
       EL may be a pointer to an edge or a leaf since a leaf is
       an edge record which also has entries for the coordinates
       of its lower left pixel. */
    begin
      value integer V;
      value pointer edge EL;
      global pointer vertex array VRTX[-M+1:M-1];
      global integer M;
      COL(VRTX[V]) ← COL(EL);
    end;


    procedure UPDATE_EDGE(E,D,EL);
    /* Update the active border for edge E to reflect the ad-
       jacency to leaf EL. This procedure is also used to up-
       date the neighbor to a leaf in which case the WID field
       is modified by D and EL is a pointer to an edge. This is
       permissible since a leaf is an edge record which also has
       entries for the coordinates of its lower left pixel. */
    begin
      value pointer edge E,EL;
      value integer D;
      copyfields (E, EL);
      WID(E) ← WID(E) - D;
    end;
```

ACKNOWLEDGMENT

The authors thank Prof. R. Sulonen for his comments, and C. Shaffer and E. Haataja for their assistance in enabling us to make empirical observations.

REFERENCES

[1] D. J. Abel and J. L. Smith, "A data structure and algorithm based on a linear key for a rectangle retrieval problem," *Comput. Vision, Graphics, Image Processing*, vol. 24, no. 1, pp. 1-13, Oct. 1983.

[2] F. W. Burton and J. G. Kollias, "Comment on the explicit quadtree as a structure for computer graphics," *Comput. J.*, vol. 26, no. 2, p. 188, May 1983.

[3] B. G. Cook, "The structural and algorithmic basis of a geographic data base," in *Proc. 1st Int. Advanced Study Symp. Topol. Data Structures Geogr. Inform. Syst.*, G. Dutton, Ed., Harvard Papers on Geographic Information Systems, 1978.

[4] C. R. Dyer, "Computing the Euler number of an image from its quadtree," *Comput. Graphics Image Processing*, vol. 13, no. 3, pp. 270-276, July 1980.

[5] I. Gargantini, "An effective way to represent quadtrees," *Commun. ACM*, vol. 25, no. 12, pp. 905-910, Dec. 1983.

[6] —, "Linear octtrees for fast processing of three dimensional objects," *Comput. Graphics Image Processing*, vol. 20, no. 4, pp. 365-374, Dec. 1982.

[7] G. M. Hunter and K. Steiglitz, "Operations on images using quadtrees," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-1, pp. 145-153, Apr. 1979.

[8] C. Jackins and S. L. Tanimoto, "Quad-trees, oct-trees, and k-trees—A generalized approach to recursive decomposition of Euclidean space," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-5, pp. 533-539, Sept. 1983.

[9] E. Kawaguchi and T. Endo, "On a method of binary picture representation and its application to data compression," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-2, pp. 27-35, Jan. 1980.

[10] E. Kawaguchi, T. Endo, and J. Matsunaga, "Depth-first expression viewed from digital picture processing," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-5, pp. 373-384, July 1983.

[11] A. Klinger, "Patterns and search statistics," in *Optimizing Methods in Statistics*, J. S. Rustagi, Ed. New York: Academic, 1971, pp. 303-337.

[12] A. Klinger and M. L. Rhodes, "Organization and access of image data by areas," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-1, pp. 50-60, Jan. 1979.

[13] G. M. Morton, "A computer oriented geodetic data base and a new technique in file sequencing," IBM Canada, 1966.

[14] M. A. Oliver and N. E. Wiseman, "Operations on quadtree-encoded images," *Comput. J.*, vol. 26, no. 1, pp. 83-91, Feb. 1983.

[15] A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber, "Application of hierarchical data structures to geographical information systems phase II," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR 1327, Sept. 1983.

[16] H. Samet, "Connected component labeling using quadtrees," *J. ACM*, vol. 28, no. 3, pp. 487-501, July 1981.

[17] —, "Computing perimeters of images represented by quadtrees," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-3, pp. 683-687, Nov. 1981.

[18] —, "Neighbor finding techniques for images represented by quadtrees," *Comput. Graphics Image Processing*, vol. 18, no. 1, pp. 37-57, Jan. 1982.

[19] —, "Distance transform for images represented by quadtrees," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-4, pp. 298-303, May 1982.

[20] —, "A top-down quadtree traversal algorithm," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. PAMI-7, pp. 94-98, Jan. 1985.

[21] —, "A quadtree medial axis transform," *Commun. ACM*, vol. 26, no. 9, pp. 680-693, Sept. 1983.

[22] —, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, June 1984.

[23] H. Samet and M. Tamminen, "Efficient image component labeling," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR 1420, June 1984.

[24] M. Tamminen, "Encoding pixel trees," *Comput. Vision, Graphics, Image Processing*, vol. 28, no. 1, pp. 44-57, Oct. 1984.

[25] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, no. 2, pp. 215-225, Apr. 1975.

[26] W. Weber, "Three types of map data structures, their ANDs and NOTs, and a possible OR," in *Proc. 1st Int. Advanced Study Symp. Topol. Data Structures Geog. Inform. Syst.*, G. Dutton, Ed., Harvard Papers on Geographic Information Systems, 1978.

[27] J. R. Woodwark, "The explicit quadtree as a structure for computer graphics," *Comput. J.*, vol. 25, no. 2, pp. 235-238, May 1982.

Hanan Samet (S'70-M'75) received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

In 1975 he joined the University of Maryland, College Park, as an Assistant Professor of Computer Science. In 1980 he became an Associate Professor. His research interests are data structures, image processing, programming languages, artificial intelligence, and database management systems.

Dr. Samet is a member of the Association for Computing Machinery, SIGPLAN, Phi Beta Kappa, and Tau Beta Phi.

Markku Tamminen (M'83) received the M.Sc. degree in applied mathematics in 1966 from the University of Helsinki, Finland, and the Ph.D. degree in computer science from the Helsinki University of Technology in 1982.

He has been with the Laboratory of Information Processing Science, Helsinki University of Technology, since 1980 and is currently Head of a research project on geometric data structures, funded by the Finnish Academy. From 1973 to 1980 he was Chief Mathematician at the Data Center of the Helsinki Metropolitan Area. His research interests include data structures based on address computation, computational geometry, image data structures and algorithms, computer aided design, and the management of spatially referenced data.

Dr. Tamminen is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Operations Research Society of America.