COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING 37, 402–419 (1987)

Optimal Quadtree Construction Algorithms*

CLIFFORD A. SHAFFER AND HANAN SAMET

Computer Science Department and Center for Automation Research, University of Maryland, College Park, Maryland 20742

Received December 11, 1985; revised April 4, 1986

An algorithm is presented that builds a quadtree in time proportional to the number of blocks in the image. In particular, this algorithm constructs a linear quadtree from a raster image stored on disk in time proportional to the number of nodes in the output quadtree plus the (relatively minor) amount of time to read the input data. Empirical tests show that for typical 512 × 512 pixel images, the new algorithm results in an order of magnitude or better improvement over traditional algorithms which insert each pixel separately and require a merge routine to form larger nodes. These traditional algorithms have an execution time that is proportional to the number of pixels in the image. Thus, when using the number of insertions into the output quadtree as a metric, our algorithm is optimal to within a constant factor. The algorithm is also adapted to build a pointer-based quadtree, again without a need to merge any nodes. © 1987 Academic Press, Inc.

1. INTRODUCTION

Hierarchical data structures are important representations in the domains of computer vision, robotics, computer graphics, image processing, pattern recognition, and geographic information systems. One such data structure is the quadtree. Today, the term quadtree is used in a general sense to describe a class of data structures whose common property is that they are based on the principle of recursive decomposition of space. In this paper we are concerned with the *region quadtree* as defined by Klinger [6] and will use the term quadtree to refer to it. Figure 1 is an example of a region and its corresponding quadtree. For a comprehensive survey of quadtrees and related hierarchical data structures see [16].

Quadtrees are of interest, in part, because they enable the solution of problems in a manner that focuses the work on the areas where the information is of the greatest density. For many problems, the amount of work that is required is proportional to the number of aggregated units (e.g., blocks) rather than to the actual size of the aggregated units (e.g., the number of pixels in a block). As such they have the potential of leading to execution time efficiency. Nevertheless, building a quadtree from a raster representation requires that every pixel of the raster be examined. Thus most quadtree building algorithms execute in time proportional to the number of pixels in the image. This can be rather costly especially if the image is large and is represented on disk.

In many real world applications, the images may be so large that the space requirements of their quadtree representation exceed the amount of memory that is available. The result is that the images must be stored on disk with portions of the data processed in core as needed. There are two reasons why the traditional pointer-based quadtree structure is considered inappropriate for such applications.

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

0734-189X/87 $3.00
Copyright © 1987 by Academic Press, Inc.
All rights of reproduction in any form reserved.

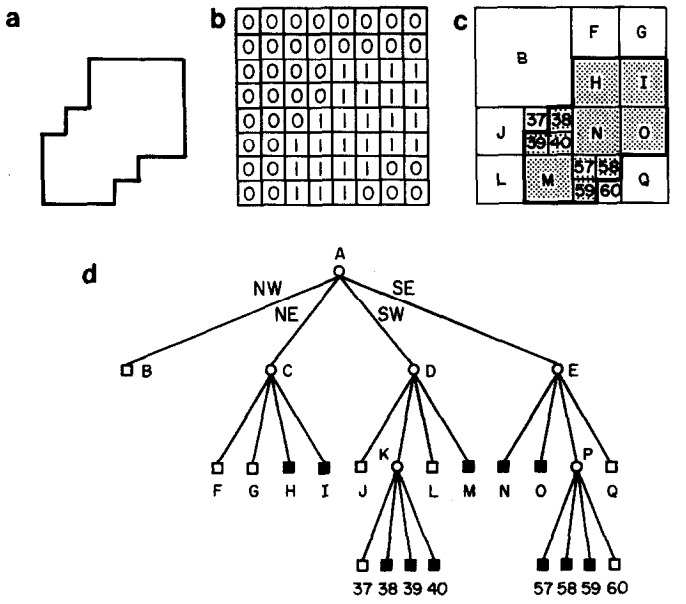

FIG. 1. A region, its binary array, its maximal blocks, and the corresponding quadtree: (a) region; (b) binary array; (c) block decomposition of the region in (a) (Blocks in the region are shaded.); (d) Quadtree representation of the blocks in (c).

First, a large portion of the pointer-based quadtree's storage space is taken up by GRAY nodes and pointers—i.e., a pointer to the node's father and four pointers to its sons. Since I/O time is a major factor in the execution time of disk-based algorithms, it is desirable to reduce the storage space required by the image if possible. Second, individual leaf nodes within a pointer-based quadtree are located by following a chain of pointers from the root to the desired node, which requires many pointer dereferences for large images. As there may be little relationship between the ordering of nodes in the tree and their ordering on the disk, this can lead to an intolerable number of disk accesses when searching and updating the tree.

The *linear quadtree* technique [1, 5] has gained increasing use as it partially alleviates both of these problems. In fact, it is currently being used to store maps in an experimental geographic information system at the University of Maryland [11, 12, 15, 17, 18, 22, 23]. Usage of variants on the linear quadtree is reported by a number of researchers [7, 9, 18]. In the linear quadtree, each leaf node is assigned a unique locational code corresponding to a sequence of directional codes that locate the leaf along a path from the root of the tree. This collection is usually represented as a list whose elements appear in increasing order of locational codes. Such an ordering is useful because it is the same order in which the leaf nodes of a pointer-based quadtree would be visited by a depth-first traversal.

When using the quadtree image representation, many functions such as area and perimeter computation can be performed by traversing the input tree(s), performing a computation at each node, and (for some functions) producing an output tree. For such algorithms, the underlying mechanism for storing the linear quadtree may safely be ignored; such considerations may indeed obscure presentation of the

algorithm. For many other algorithms, however, attention should be focused on utilizing linear quadtrees efficiently from the standpoint of minimizing disk accesses and the time spent making updates. Often, little thought is given to the issues of organizing the linear quadtree or to the cost involved in searching and updating the node list. As an example, consider Gargantini's original algorithm for determining the neighbor of a given node in an image stored as a linear quadtree [5]. This algorithm alters the address of the query node (i.e., its locational code) by changing the appropriate digits of the address to match the address of the desired neighbor in a manner reminiscent of Samet's algorithm for neighbor-finding in pointer-based quadtrees [14]. In actuality, Gargantini's algorithm computes the value of the locational code for the desired neighbor, without taking into account the fact that a search for this locational code must subsequently be made in the node list to locate the actual node. In practice, computing locational codes of nodes takes only a small fraction of the total execution time required by a linear quadtree-based program. Searching and updating the node list usually takes up the vast majority of execution time.

In this paper, we show that the quadtree construction process can be achieved in time proportional to the number of blocks in the image. In particular, we present an algorithm for building a linear quadtree from a raster image stored on disk in time proportional to the number of nodes in the output quadtree plus the (relatively minor) amount of time to read the input data. For typical 512×512 pixel images, the new algorithm results in an order of magnitude or better improvement over traditional algorithms which insert each pixel separately and require a merge routine to form larger nodes. We also adapt this algorithm to build a pointer-based quadtree, again without a need to merge any nodes.

The remainder of this paper is organized as follows. Section 2 reviews the linear quadtree with an emphasis on a particular implementation as well as a cost metric for evaluating algorithms that operate on linear quadtrees. Section 3 presents a new algorithm for building linear quadtrees from raster arrays. Section 4 contains our conclusions.

2. IMPLEMENTING LINEAR QUADTREES

When constructing a linear quadtree, we assume that every pixel in the underlying array of the digitized image has been assigned an address value (i.e., its locational code). The addressing schemes commonly used are variations on that suggested by Morton [8] for use in indexing maps in the Canada Geographic Information System [3]. Such schemes are thus sometimes referred to as *Morton sequencing*. Their application to quadtrees was independently realized by Gargantini [5] and Abel and Smith [1]. Morton sequencing makes use of an addressing scheme which is equivalent to interleaving the bits of the binary representation for the x and y coordinates (each represented by a fixed number of digits) of that pixel. For example, in Fig. 2, a 3 bit binary representation for the row and column coordinates is indicated along the bottom and right sides of an 8×8 array. The locational code of each pixel is formed by bit interleaving such that the y bit precedes the x bit at each position. In Fig. 2, these pixel addresses are represented with base 4 digits (i.e., each x and y bit pair correspond to a single base 4 digit). When the addresses of the pixels are sorted in increasing order, the result is equivalent to a depth-first traversal such that quadrants are visited in the order NW, NE, SW, and SE.

	000	001	010	011	100	101	110	111
000	000	001	010	011	100	101	110	111
001	002	003	012	013	102	103	112	113
010	020	021	030	031	120	121	130	131
011	022	023	032	033	122	123	132	133
100	200	201	210	211	300	301	310	311
101	202	203	212	213	302	303	312	313
110	220	221	230	231	320	321	330	331
111	222	223	232	233	322	323	332	333

FIG. 2. The Morton-code address scheme for labeling pixels.

Given the above method for addressing pixels, we can in turn generate node addresses by declaring that each node will be given the address of the least valued pixel contained within the block it represents. Figure 3 shows the block decomposition for the image in Fig. 1a with each block given the address (in base 4) of the least valued pixel contained within that block. Note that the node in the NW quadrant of the image in Fig. 3 has a 0 value in the first position (indicating a NW branch), all nodes in the NE quadrant have a 1 in the first position, etc. Actually, this method of addressing blocks is inadequate as there is no indication of the block's size. However, this inadequacy can be remedied in a number of ways, one of which is to append the level at which the block is found to the address. Regardless of the method used to address quadtree blocks, the list of quadtree blocks is kept sorted in increasing order of their locational codes.

Given this sorted list of quadtree blocks, some means must be found to organize it so that insertions, deletions, and node searches may be performed efficiently. In addition, it is important that the organization method lend itself to off-line storage of large images. The B-tree [4] is a data structure that meets these requirements.

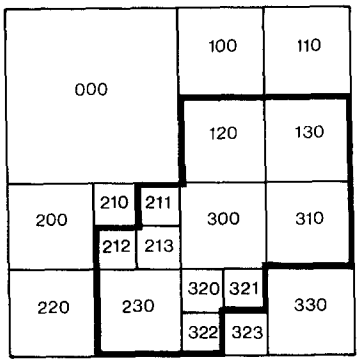

FIG. 3. The Morton-code addresses for the blocks of Fig. 1.

B-trees are very efficient in that the number of accesses necessary to retrieve a given key from secondary storage is kept low. This is partly because the tree is always balanced, and partly because the branching factor is very high. Both Abel [2] and Samet *et al.* [18] use a linear quadtree encoding in conjunction with B-trees to store images.

Since the linear quadtree is disk based, with only a small portion of an image in core at any given instant, the time spent moving segments of the image to and from the disk is an important factor. Comer [4] states that the majority of the time spent in manipulating B-trees is accounted for by I/O. A buffer pool can help reduce the I/O time. The amount of time spent searching for a key within a given B-tree page is also an important factor. For algorithms such as the naive building algorithm (see NAIVE_BUILD in the next section), about 98% of the execution time may be taken up by these parts of the system. Comer uses the number of page fetches as a metric for B-tree analysis which, for a given key search, is related to the number of nodes stored. No more than d pages need be searched in the B-tree, where d is the depth of the tree. This depth will be the same for all keys as the tree is balanced. As each B-tree page must be at least half full, the depth will be about $\log_k N$, where k is the number of keys in a page and N is the number of nodes in the B-tree. For example, in our implementation each B-tree page consists of 1024 8-bit bytes—large enough to contain 120 quadtree nodes. The depth cannot be greater than 4 for a 4096×4096 image even when each node represents a single pixel (i.e., a complete quadtree). Unfortunately, both the amount of time and the number of page fetches spent in a particular quadtree operation (e.g., insertion, deletion, or search) is difficult to compute. This execution time analysis is complicated by the fact that the desired page may be in the buffer pool, with no I/O being necessary. In some cases, insertion of a quadtree node can cause a block to be split many times, thereby causing many additional nodes to be stored in the B-tree. Other insertion operations may cause no such node splitting.

In the remainder of this paper, we will use the following definition for the *insert* function. If the leaf to be inserted, say L, corresponds to several smaller leaves in the node list, then these smaller leaves will be replaced by L (e.g., node A of Fig. 4b). If L corresponds to a leaf of identical size, then the color of the existing leaf will be changed to the value of L; merging may result (e.g., node E in Fig. 4c). If L is part of an existing leaf of the same color, then *no change* will be made in the node list (e.g., the dashed portion of Fig. 4d). If the leaf to be inserted is part of a larger existing leaf of a different color, then the existing leaf will be divided into appropriate quadrants, subquadrants, etc, until a leaf of the same size is created whose color is changed to the value of L (e.g., node $B1$ of Fig. 4e). Thus, a single insert can add or remove many nodes from the quadtree.

Given a disk-based B-tree implementation for linear quadtrees, we would like to find a method for calculating the complexity of an algorithm. A common quadtree analysis metric is the number of nodes of the quadtree that are visited when performing the operation (e.g., [10, 14, 21, 24]). Operations performed on the linear quadtree are quite different from operations performed on pointer-based representations. Many pointer-based algorithms involve a procedure known as *neighbor-finding*. Neighbor-finding in pointer-based quadtrees is done by ascending father links to the nearest common ancestor of the node and its neighbor, then descending the tree to the neighbor. As shown in [14, 21], on the average this cost is constant and

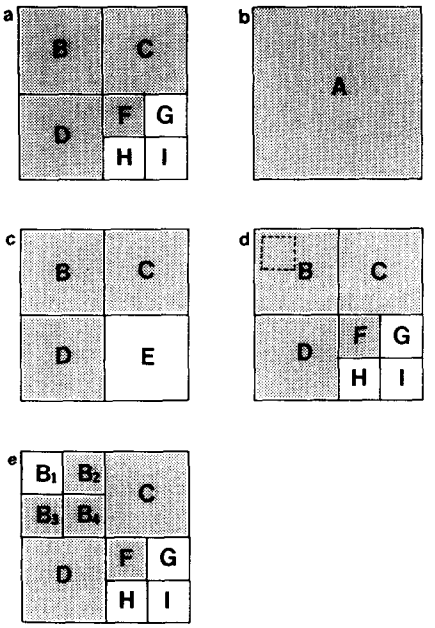

FIG. 4. The effects of node insertion: (a) An example block decomposition; (b) the result of inserting a large BLACK node covering the SE quadrant of (a): merging occurs; (c) the result of inserting a WHITE node at node *F* of (a): *F*, *G*, *H*, and *I* merge; (d) the result of inserting a small BLACK node in the upper left corner of node *B* in (a): no splitting occurs; (c) the result of inserting a small WHITE node into the upper left corner of node *B* in (a): splitting occurs.

hence is not dependent on the depth of the nodes in the tree. In contrast, for linear quadtrees, neighbors are found by computing the address of the desired neighbor, and then locating the actual node in the list; this requires a search. Whether the node is a neighbor of the most recently located node, or unrelated to it, the node finding cost is still that of a single search. Not surprisingly, the two operations which consume the most time when manipulating linear quadtrees are node search and node insertion. Moreover, a node insertion is usually preceded by a node search. Assuming that the cost of inserting a node is constant, a reasonable metric for algorithm complexity is obtained by simply adding the number of node searches to the number of node insertions.

3. AN OPTIMAL RASTER TO QUADTREE ALGORITHM FOR LINEAR QUADTREES

The naive algorithm for converting a raster image to a linear quadtree is to insert individually each pixel of the raster image into the quadtree in raster order. Those pixels making up larger nodes will be merged together by the quadtree insert routine. Previous algorithms presented in the literature [11, 13] have worked on this principle. Attempts at increasing efficiency concentrated on how to improve the insert routine. Procedure NAIVE_BUILD given below demonstrates the naive method. Table 1 contains the execution times of such an algorithm when applied to six test maps. The timings are nearly identical for raster images with the same

TABLE 1
Naive Quadtree Building Algorithm Statistics

Map name	Num nodes	Num inserts	Time (s)
Floodplain	5266	180000	413.2
Topography	24859	180000	429.8
Landuse	28447	180000	436.7
Center	4687	262144	603.8
Pebble	44950	262144	630.1
Stone	31969	262144	629.5

number of pixels (and thus, node inserts), regardless of the number of nodes in the eventual quadtree. In other words, we see that the number of nodes in the output tree has little or no effect on the time required to perform the algorithm. Note that for the naive building algorithm, the amount of time needed to read the picture data is approximately 1% of the time necessary to insert every pixel.

```
procedure NAIVE_BUILD(INPIC,R,C,N,OUTTREE);
/* Build a quadtree in OUTTREE corresponding to input picture INPIC using a
   naive quadtree building algorithm that inserts each pixel individually into the
   quadtree. The input picture has R rows and C columns and the output
   quadtree will be of maximum depth N (i.e., a 2^N × 2^N image). */
begin
 value picture pointer INPIC; /* INPIC points initially to the first row */
 value integer R,C,N;
 reference quadtree pointer OUTTREE;
 row BUFF[1:C];
 integer ROW, COL;

 for ROW ← 1 step 1 until R do
  begin /* Process each row of the picture in sequence */
   GET_ROW(INPIC,BUFF);
   for COL ← 1 step 1 until C do
    INSERT(OUTTREE,MAKE_NODE(COL,ROW,1,BUFF[COL]));
  end;
end;
```

Considering the large number of pixels in the raster representation of an image in comparison to the number of nodes in the quadtree representation for that image, it would be desirable to find an algorithm which can reduce the number of node insertions required. An optimal algorithm would, in the worst case, make a single insertion for each node in the quadtree. Procedure FAST_BUILD given below has this worst-case behavior. It is based on processing the image in raster-scan (top to bottom, left to right) order. An important point is that the largest node for which the current pixel is the first (upper leftmost) pixel will be inserted whenever an insertion is required. Such a policy will avoid the necessity of merging since the upper leftmost pixel of any block is inserted before any other pixel of that block.

This makes it impossible for four sibling blocks to be of the same color since the insert function will not break up a node of color C when an attempt to insert a subquadrant of color C is made.

At any point while the quadtree is being constructed, there is a processed portion of the image (corresponding to those pixels already scanned), and an unprocessed portion. Both the processed and the unprocessed portions of the quadtree have been assigned to nodes. If it were possible to know the current value of all unprocessed pixels in the tree, then it would not be necessary to insert a pixel with color C for which a previous largest-node insertion has already set the containing node for that pixel to C. We say that a node is *active* if at least one, but not all, pixel *covered* by (i.e., contained in) the node has been processed. The optimal quadtree building process must keep track of all these active nodes. Note that there are no active nodes at level 0. Given a $2^n \times 2^n$ image, an upper bound on the number of active nodes is $2^n - 1$ as shown by the following theorem.

THEOREM 1. *Given a $2^n \times 2^n$ image, at any time during a raster-scan building process in which the largest node possible is always inserted, at most $2^n - 1$ nodes will be active.*

Proof. Any given pixel can be covered by at most n active nodes—i.e., a node at each level from 1 to n (corresponding to the root). At any given instant, there can be at most 2^{n-1} active nodes at level 1 (i.e., nodes of size 2×2). This is true because, for any given column, only one node at level 1 will be active, giving at most a solid line of 2×2 active nodes along a row just processed. In a like manner, there will be at most 2^{n-2} active nodes at level 2, and so on with 2^{n-i} active nodes at level i up to a single active node at level n (the root). Thus, there will be at most $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ active nodes. □

Using the above theorem, a quadtree building algorithm which is optimal to within a constant factor is derived below. Assume the existence of a data structure which keeps track of the active quadtree nodes. For each pixel in the raster scan traversal, do the following. If the pixel is the same color as the appropriate active node, do nothing. Otherwise, insert the largest possible node for which this is the first (i.e., upper leftmost) pixel, and (if it is not a 1×1 pixel node) add it to the set of active nodes. Remove any active nodes for which this is the last (lower right) pixel. Procedure FAST_BUILD works in such a fashion. The list of active nodes, referred to as the *active node table*, is represented by an array, called TABLE, with $2^n - 1$ entries to store all potentially active nodes. TABLE is implemented as a table of records of type ACTIVE that contains the relevant information. TABLE[1] stores the root active node (i.e., at level n), TABLE[2] and TABLE[3] store the two active nodes at level $n - 1$, the next four entries store the active nodes at level $n - 2$, etc. Given a pixel in column j, the value of the active node at level i is found at position $2^{n-i} + j/2^i$ in TABLE. Note that shift operations can be used instead of divisions if speed is important.

The only remaining problem is to locate the smallest active node in the table which contains a specified pixel. For a given pixel P in a $2^n \times 2^n$ image, as many as n nodes containing P could be active. Multiple active nodes for a given pixel arise whenever a node is split to accommodate the insertion of a pixel having a color different from that previously associated with the node (e.g., after inserting pixel 3

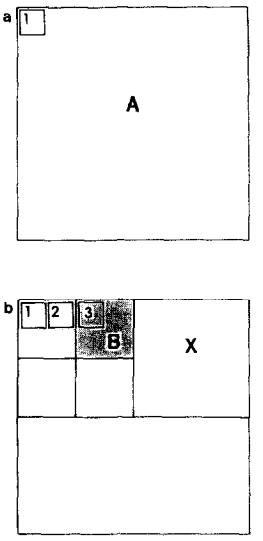

FIG. 5. Node insertion can create multiple active nodes: (a) node A is active after inserting a single pixel of color C; (b) the first two pixels have color C. Pixel 3 has color D. Its insertion creates active node B.

in Fig. 5b). Each pixel will have the color of the smallest active node which contains it, since the smallest active node will be the one most recently inserted. Finding the smallest active node that contains a given pixel can be done by searching, for a given column, from the entry in the table representing the lowest level upwards until the first non-empty entry is found. However, this is time consuming since it might require n steps. Therefore, an additional 1-dimensional array, called LIST and referred to as the *access array*, is maintained to provide a pointer to the current active node in TABLE. LIST contains 2^{n-1} records. While processing a given row, for a pixel in column j, the LIST entry at $j/2$ indicates the entry of TABLE corresponding to the smallest active node containing that pixel. At the beginning of the algorithm, each entry of LIST points to TABLE[1] (i.e., the entry in TABLE corresponding to the root). As active nodes are inserted or completed (and are to be deleted from the active node table), both the active node table and the access array are updated.

```
record ACTIVE
begin
 color VALUE;
 integer DEPTH;
 record ACTIVE pointer FATHER;
end;

procedure FAST_BUILD(INPIC,R,C,N,OUTTREE);
/* Build a quadtree in OUTTREE corresponding to input picture INPIC using a
    quadtree building algorithm that inserts the largest node for which the
    current pixel is the first (i.e., upper leftmost) pixel. The input picture has R
```

rows and C columns and the output quadtree will be of maximum depth N (i.e., a $2^N \times 2^N$ image). Type **row** is an array of type **color** where **color** takes on the values NOCOLOR, BLACK, and WHITE. */

```
begin
 value picture pointer INPIC;
 value integer R,C,N;
 reference quadtree pointer OUTTREE;
 row BUFF[0:C - 1];
 record ACTIVE array TABLE[1:2↑N - 1];
 record ACTIVE pointer array LIST[0:2↑(N - 1) - 1];
 record ACTIVE pointer CURRACT, pointer NEWACT;
 integer ROW,COL,J,DEPTH,XT,YT,T;

 TABLE[1] ← {'WHITE',N,TABLE[1]};
 for J ← 0 step 1 until 2↑(N - 1) - 1 do LIST[J] ← TABLE[1];
 /* Process the picture: */
 for ROW ← 0 step 1 until R - 1 do
  begin
   GET_ROW(INPIC,BUFF); /* Process one row at a time */
   for COL ← 0 step 1 until C - 1 do /* Process each pixel in the row */
    begin
     CURRACT ← LIST[COL/2]; /* Find smallest active node containing
                               pixel */
     if VALUE(CURRACT) ≠ BUFF[COL] then
      begin /* The pixel and the node containing it differ in color */
       /* Calculate depth of largest node for which this is first pixel */
       XT ← COL;YT ← ROW;DEPTH ← 0;
       while even(XT) and even(YT) and (DEPTH < N) do
        begin
         XT ← XT/2; YT ← YT/2; DEPTH ← DEPTH + 1;
        end;
       if DEPTH ≠ 0 then
        begin /* Largest node containing the pixel is larger than 1 × 1 */
         /* Update the active node table and the access array */
         /* NEWACT (a pointer) is set to the appropriate entry in TABLE
            (a record) */
         NEWACT ← address(TABLE[2↑(N - DEPTH) +
                                COL/2↑DEPTH]);
         VALUE(NEWACT) ← BUFF[COL];
         DEPTH(NEWACT) ← DEPTH;
         FATHER(NEWACT) ← CURRACT;
         CURRACT ← NEWACT;
         for J ← COL/2 step 1 until COL/2 + 2↑(DEPTH - 1) - 1 do
           LIST[J] ← NEWACT;
        end;
       INSERT(OUTTREE,
              MAKENODE(COL,ROW,DEPTH,BUFF[COL]));
```

```
        end;
      if mod(ROW + 1,2↑DEPTH(CURRACT)) = 0 and
        mod(COL + 1,2↑DEPTH(CURRACT)) = 0 then
        begin /* The last pixel of one or more active nodes */
        T ← DEPTH(CURRACT);
        while mod(ROW + 1,2↑DEPTH(CURRACT)) = 0 and
              mod(COL + 1,2↑DEPTH(CURRACT)) = 0 do
          begin /* Pop up to father */
          T ← DEPTH(CURRACT);
          CURRACT ← FATHER(CURRACT);
          end;
        for J ← COL/2 step − 1 until COL/2 − 2↑(T − 1) + 1 do
          LIST[J] ← CURRACT; /* Update access array */
        end;
      end;
    end;
  end;
```

Table 2 contains timing results produced by applying an algorithm based on FAST_BUILD to the same test maps as were used in Table 1. As indicated in Table 2, the new algorithm often requires far fewer calls to the insert routine than the number of nodes in the resulting output tree because some calls to insert force node splits to occur, thereby increasing the number of nodes in the tree. This is all accomplished by procedure INSERT whose code is not given here as it is implementation dependent. For example, consider Fig. 4e where node B (from Fig. 4a) is replaced by node $B1$ with a new value along with 3 other nodes ($B2$, $B3$, and $B4$) retaining B's value. However, only one new active node is created ($B1$) as the remaining pixels are still covered by the original active node (B). When the time comes to process the pixels covered by those blocks which are artifacts of the splitting process ($B2$, $B3$, and $B4$), these pixels may have the same value as B, and thus no additional insertion is required. As another example, consider Fig. 5 where the processing of pixel 3 causes the insertion of node B into the quadtree containing a single node resulting in the creation of seven nodes. If the first pixel inserted into node X were to be the same color as the original node (A of Fig. 5a), then no insertion is required.

TABLE 2
Optimal Quadtree Building Algorithm Statistics

Map name	Num nodes	Num inserts	Time (s)
Floodplain	5266	2352	13.8
Topography	24859	12400	51.2
Landuse	28447	14675	56.9
Center	4687	2121	16.1
Pebble	44950	20770	111.0
Stone	31969	14612	70.2

TABLE 3
Trace Table for Active Nodes in Building Example

Pixel	Action	Size	Active nodes by level		
			3	2	1
(0, 0)	insert WHITE node *A*	8 × 8	*A*		
(2, 4)	insert BLACK node *B*	2 × 2	*A*		*B*
(2, 6)	insert BLACK node *C*	2 × 2	*A*		*BC*
(3, 5)	remove *B* from active		*A*		*C*
(3, 7)	remove *C* from active		*A*		
(4, 3)	insert BLACK node *D*	1 × 1	*A*		
(4, 4)	insert BLACK node *E*	4 × 4	*A*	*E*	
(5, 2)	insert BLACK node *F*	1 × 1	*A*	*E*	
(5, 3)	insert BLACK node *G*	1 × 1	*A*	*E*	
(6, 2)	insert BLACK node *H*	2 × 2	*A*	*E*	*H*
(6, 6)	insert WHITE node *I*	2 × 2	*A*	*E*	*HI*
(7, 3)	remove *H* from active		*A*	*EI*	
(7, 5)	insert WHITE node *J*	1 × 1	*A*	*EI*	
(7, 7)	remove *I*, *E*, *A* from active				

As an example of how the new quadtree building algorithm works, let us consider how the quadtree corresponding to the image of Fig. 1 is constructed. Table 3 traces the active nodes at each stage of execution. Each row in Table 3 lists the active nodes after the given pixel has been processed. The pixel identifier (a, b) means that the pixel is in row a and column b relative to an origin at the upper left corner of the image. When the first pixel of the array is processed, the entire quadtree is represented by a single WHITE node (block *A* in Fig. 6a). No other insertions occur while processing rows 0 and 1. When the first BLACK pixel (2, 4) is processed, block *B* of Fig. 6a becomes active. When BLACK pixel (2, 5) is processed, block *B* will be located in the active node table, since it is the smallest active node containing that pixel. When BLACK pixel (2, 6) is processed, block *C* of Fig. 6b becomes active, since only active WHITE block *A* contains it at that point. As row 3 is processed, blocks *B* and *C* are deactivated when their lower right pixels are processed. When pixel (4, 4) is processed, the state is as shown in Fig. 6c. The blocks previously labeled *B* and *C* are not active. Pixel-sized block *D* at (4, 3) is not active since it contains no unprocessed pixels. Blocks *A* and *E* are, therefore, the only active blocks. Figure 6d shows the state of the algorithm when pixel (6, 6) has been processed. Block *H* became active after processing pixel (6, 2). Since the smallest block containing pixel (6, 6) had been BLACK, a new WHITE block has been activated (block *I*). Thus, three active blocks (i.e., *A*, *E*, and *I*) contain pixel (6, 7), with the smallest being block *I*. As the final row is processed, all active nodes will be deactivated once the SW son of *E* has been split to enable the insertion of a WHITE node corresponding to the pixel at (7, 5).

In order to understand why FAST_BUILD is such an improvement over NAIVE_BUILD, let us analyze the cost of both algorithms in terms of the number of insert operations which they perform. NAIVE_BUILD examines each pixel and inserts it into the quadtree. Assuming a cost of *I* for each insert operation, and a

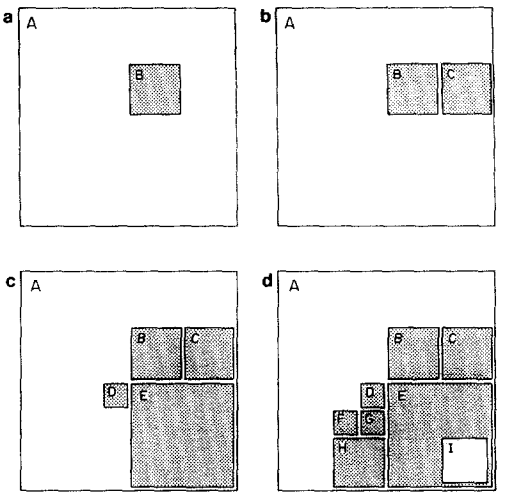

FIG. 6. The construction process for the image of Fig. 1: (a) state after processing pixel (2,4); (b) state after processing pixel (2,6); (c) State after processing pixel (4,4); (d) State after processing pixel (6,6).

cost of c for the time spent examining a pixel, the total cost is then $2^{2n} \cdot (c + I)$. FAST_BUILD must also examine each pixel. However, there will be at most one insert operation for each of the N nodes in the output quadtree. Therefore, the cost of FAST_BUILD is $c \cdot 2^{2n} + I \cdot N$, where c is relatively small in comparison to I, and N is usually small in comparison to 2^{2n}. In other words, the quantity $I \cdot N$ dominates the cost of FAST_BUILD, yet is much less than $I \cdot 2^{2n}$. The result is that using FAST_BUILD reduces the execution time from being O(pixels) to O(nodes). Of course, this is achieved at an increase in storage requirements due to the need to keep track of the active nodes (at most $2^n - 1$ for a $2^n \times 2^n$ image).

The largest-node-insertion technique discussed in this section can be used to improve the pointer based raster-to-quadtree algorithm described in [13]. That algorithm works in a bottom-up manner. It begins with a single pixel representing the first pixel of the raster array. As each pixel of the first row is scanned, the eastern neighbor of the current pixel is located by means of a neighbor finding operation [14, 21]. The southern neighbor of the first pixel of the row is located before a row is processed in order to avoid searching from the root with each new row.

Using the largest-node insertion technique we can devise an analogous top-down algorithm, using neighbor finding, which performs no merging. This can save much processing for images which can be represented with large nodes. Each pixel of the raster image is processed in raster scan order. After processing the first pixel, the quadtree is represented by a leaf node of color C corresponding to the root. As subsequent pixels are processed, if a pixel of a different color, say C', is encountered, then the current node is set to GRAY and given four children with the original value of the parent. The child containing the current pixel becomes the current node. If the current pixel is the first (upper leftmost) pixel of the node, then

TABLE 4
Pointer Quadtree Building Statistics

Map name	Time (s) Old	New
Floodplain	50.3	5.4
Topography	54.0	15.4
Landuse	52.7	17.3
Center	71.0	6.2
Pebble	79.0	27.3
Stone	75.5	20.9

the node value is changed to C'. Otherwise, the split step is repeated until the current pixel becomes the upper leftmost corner of the current node. As the pixels of a given row are processed, it is necessary to keep track of their position within the current node. If the next pixel to be processed is beyond the eastern edge of the current node, then that node's eastern neighbor is located by finding its neighbor. In this way, no unnecessary nodes will be inserted, and no merging needs to be performed. The algorithm, given by procedure RASTER_TO_QUAD, is presented below. Table 4 gives an empirical comparison between the algorithm of [13] (referred to in Table 4 as the "old" algorithm) and RASTER_TO_QUAD (referred to as "new") on our six test images.

```
node pointer procedure RASTER_TO_QUAD(INPIC,R,C,N,OUTTREE);
/* Build a quadtree in OUTTREE corresponding to input picture INPIC using a
   quadtree building algorithm that inserts the largest node for which the
   current pixel is the first (i.e., upper leftmost) pixel. The input picture has R
   rows and C columns and the output quadtree will be of maximum depth N
   (i.e., a 2^N x 2^N image). The algorithm is analogous to FAST_BUILD
   except that the result is a tree instead of a collection of locational codes
   corresponding to the leaf nodes. Type row is an array of type color where
   color takes on the values NOCOLOR, BLACK, and WHITE. */
begin
 value picture pointer INPIC;
 value integer R,C,N;
 reference quadtree pointer OUTTREE;
 row BUFF[0:C - 1];
 node pointer FIRST, TEMP, CURR;
 integer FIRSTX,FIRSTY,FIRSTWID,ROW,COL;
 integer CURRX,CURRY,CURRWID;
 integer I;

 OUTTREE ← FIRST ← CREATEQNODE(NIL,NIL,WHITE);
 FIRSTX ← FIRSTY ← 0;
 FIRSTWID ← 2↑N;
 for ROW ← 0 step 1 until 2↑N - 1 do
```

```
  begin /* Process each row of the picture */
  GET_ROW(INPIC,BUFF); /* Get next row of the input picture */
  if ROW = FIRSTY + FIRSTWID then
   begin /* Moved passed the south side of the current node */
    FIRSTY ← FIRSTY + FIRSTWID;
    GTEQUAL_ADJ_NEIGHBOR(FIRST,'S',TEMP,FIRSTWID);
    FIRST ← TEMP;
    LOCATE(FIRST,FIRSTX,FIRSTY,FIRSTWID,0,ROW);
   end;
  CURR ← FIRST;
  CURRX ← FIRSTX;CURRY ← FIRSTY;CURRWID ← FIRSTWID;
  for COL ← 0 step 1 until 2↑N − 1 do
   begin /* Process each column of the row */
    if COL = CURRX + CURRWID then
     begin /* Moved passed the east side of the current node */
      CURRX ← CURRX + CURRWID;
      GTEQUAL_ADJ_NEIGHBOR(CURR,'E',TEMP,CURRWID);
      CURRY ← (CURRY/CURRWID)*CURRWID;
      CURR ← TEMP;
      LOCATE(CURR,CURRX,CURRY,CURRWID,COL,ROW);
     end;
    if NODETYPE(CURR) ≠ BUFF[COL] then
     begin /* Insert a node */
      TEMP ← CURR;
      INSERT(CURR,CURRX,CURRY,CURRWID,COL,ROW,BUFF[COL]);
      if TEMP = FIRST then /* Has first block in row been altered? */
       begin /* Yes. Update its descriptor */
        FIRST ← CURR;
        FIRSTX ← CURRX;
        FIRSTY ← CURRY;
        FIRSTWID ← CURRWID;
       end;
     end;
   end;
 end;
end;

procedure LOCATE(P,X,Y,WID,COL,ROW);
/* Given a node P of width WID and upper leftmost pixel in column X and row
   Y, find its leaf descendant whose upper leftmost pixel is at column COL and
   row ROW. A pointer to the leaf descendant, its width, and coordinates of its
   upper leftmost pixel are returned in P, WID, X, and Y, respectively. */
begin
 reference node pointer P;
 reference integer X,Y,WID;
 value integer COL,ROW;
 direction S1,S2;

 while NODETYPE(P) = GRAY do
```

```
  begin
   WID ← WID/2;
   if ROW < (Y + WID) then S1 ←'N';
   else
    begin
     S1 ←'S';Y ← Y + WID;
    end;
   if COL < (X + WID) then S2 ←'E';
   else
    begin
     S2 ←'W'; X ← X + WID;
    end;
   P ← SON(P,QUAD[S1,S2]);
  end;
end;

procedure INSERT(P,PX,PY,PWID,C,R,CL);
/* Given a subtree rooted at node P of width PWID and whose upper leftmost
    pixel is at column PX and row PY, insert a node of color CL whose upper
    leftmost pixel is at column C and row R. A pointer to the leaf descendant, its
    width, and the column and row of its upper leftmost pixel are returned in P,
    PWID, PX, and PY, respectively. */
begin
 reference node pointer P;
 reference integer PX,PY,PWID;
 value integer C,R;
 value color CL;
 color TEMPCOL;
 quadrant I;

 TEMPCOL ← NODETYPE(P);
 while(PX ≠ C) or (PY ≠ R) do
  begin
   NODETYPE(P) ←'GRAY';
   for I in {'NW','NE','SW','SE'} do
    CREATEQNODE(P,I,TEMPCOL);
   LOCATE(P,PX,PY,PWID,C,R);
  end;
 NODETYPE(P) ← CL;
end;

procedure GTEQUAL_ADJ_NEIGHBOR(P,D,Q,WID);
/* Return in Q the neighbor of node P, of size greater than or equal to P, in
    horizontal or vertical direction D. WID denotes the width of P and ultimately
    the width of Q. If the neighbor does not exist, then return NIL. */
begin
 value node pointer P;
 value direction D;
 reference node pointer Q;
```

```
reference integer WID;
WID ← WID*2;
if not null(FATHER(P)) and ADJ(D,SONTYPE(P)) then
 /* Find a common ancestor */
 GTEQUAL_ADJ_NEIGHBOR(FATHER(P),D,Q,WID)
else Q ← FATHER(P);
/* Follow the reflected path to locate the neighbor */
if not null(Q) and GRAY(Q) then
 begin
  Q ← SON(Q,REFLECT(D,SONTYPE(P)));
  WID ← WID/2;
 end;
end;
```

4. CONCLUDING REMARKS

As we saw in Section 3, the technique of inserting maximal nodes and maintaining a list of active nodes yields a building algorithm with at most one insert per output tree node. Thus, when using the number of insertions into the output quadtree as a metric, our algorithm is optimal to within a constant factor. Similar techniques can be applied to other quadtree algorithms which create an output tree in a "reasonable" order. By reasonable, we require only that the first (upper leftmost) pixel of every node be inserted first. Such a restriction is satisfied by a depth-first traversal (i.e., node address order), a raster scan order, or any other ordering where all pixels above and to the left of the current pixel have already been processed. Some example tasks to which our methods have already been applied include algorithms for computing set operations (e.g., union, intersection, difference) between two unregistered images represented by quadtrees, as well as algorithms for windowing and matching of unregistered images.

The new quadtree building algorithm achieves its speed up by use of additional storage (i.e., 2^{n-1} records for a $2^n \times 2^n$ image) for the active nodes. This is acceptable for a 2-dimensional image. Hoever, for images of higher dimension, we must be more selective in terms of the information that is stored in the active node arrays. At each level of the tree there are many cases that the elements of the array corresponding to the level are not used. It may be possible to do this by using linked lists to represent the relevant active nodes at each level. A similar technique was used by Samet and Tamminen to enable the efficient computation of connected component labeling for images represented by linear quadtrees. Their 2-dimensional algorithm used arrays [20] and was generalized to arbitrary dimensions through the use of linked lists [19].

ACKNOWLEDGMENT

We have benefited from discussions with Randal C. Nelson.

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 Process.* **24**, No. 1, 1983, 1–13.
2. D. J. Abel, A B^+-tree structure for large quadtrees, *Comput. Vision Graphics Image Process.* **27**, No. 1, 1984, 19–31.

3. M. A. Comeau, *A Coordinate Reference System for Spatial Data Processing*, CLDS Technical Bulletin No. 3, November 1981.
4. D. Comer, The ubiquitous *B*-tree, *ACM Comput. Surveys* **11**, No. 2, 1979, 121–137.
5. I. Gargantini, An effective way to represent quadtrees, *Commun. ACM* **25**, No. 12, 1982, 905–910.
6. A. Klinger, Patterns and search statistics, in *Optimizing Methods in Statistics* (J. S. Rustagi, Ed.), pp. 303–337, Academic Press, New York, 1971.
7. J. P. Lauzon, D. M. Mark, L. Kikuchi, and J. A. Guevara, Two-dimensional run-encoding for quadtree representation, *Comput. Vision Graphics Image Process.* **30**, No. 1, 1985, 56–69.
8. G. M. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*, IBM, Ltd., Ottawa, Canada, 1966.
9. M. A. Oliver and N. E Wiseman, Operations on quadtree encoded images, *Comput. J.* **26**, No. 1, 1983, 83–91.
10. F. Peters, An algorithm for transformations of pictures represented by quadtrees, *Comput. Vision Graphics Image Process.* **32**, No. 3, 1985, 397–403.
11. A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber, *Application of Hierarchical Data Structures to Geographical Information Systems*, Computer Science TR-1197, University of Maryland, College Park, Md., June 1982.
12. A. Rosenfeld, H. Samet, C. Shaffer, and R. E. Webber, *Application of Hierarchical Data Structures to Geographical Information Systems: Phase II*, Computer Science TR-1327, University of Maryland, College Park, Md., September 1983.
13. H. Samet, An algorithm for converting rasters to quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **3**, No. 1, 1981, 93–95.
14. H. Samet, Neighbor finding techniques for images represented by quadtrees, *Comput. Graphics Image Process.* **18**, No. 1, 1982, 37–57.
15. H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber, Quadtree region representation in cartography: Experimental results, *IEEE Trans. Systems Man Cybernet.* **13**, No. 6, 1983, 1148–1154.
16. H. Samet. The quadtree and related hierarchical data structures, *CAM Comput. Surveys* **16**, No. 2, 1984, 187–260.
17. H. Samet, A. Rosenfeld, C. A. Shaffer, R. C. Nelson, and Y-G. Huang, *Application of Hierarchical Data Structures to Geographic Information Systems: Phase III*, Computer Science TR-1457, University of Maryland, College Park, Md., November 1984.
18. H. Samet, A. Rosenfeld, C. A. Shaffer, and R. E. Webber, A geographic information system using quadtrees, *Pattern Recognit.* **17**, No. 6, 1984, 647–656.
19. H. Samet and M. Tamminen, *Efficient Component Labeling of Images of Arbitrary Dimension*, Computer Science TR-1480, University of Maryland, College Park, Md., February 1985.
20. H. Samet and M. Tamminen, Computing geometric properties of images represented by linear quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **7**, No. 2, 1985, 229–240.
21. H. Samet and C. A. Shaffer, A model for the analysis of neighbor finding in pointer-based quadtrees, *IEEE Trans. Pattern Anal. Mach. Intell.* **7**, No. 6, 1985, 717–720.
22. H. Samet, A. Rosenfeld, C. A. Shaffer, R. C. Nelson, Y-G. Huang, and K. Fujimura, *Application of Hierarchical Data Structures to Geographic Information Systems: Phase IV*, Computer Science TR-1578, University of Maryland, College Park, Md., December 1985.
23. C. A Shaffer, H Samet, R. E. Webber, R. C. Nelson, and Y-G Huang, An implementation for a geographic information system based on quadtrees, tutorial lecture notes, SIGGRAPH '85, San Francisco, July 1985.
24. R. E. Webber, *Analysis of Quadtree Algorithms*, Ph.D. thesis, Computer Science Department, University of Maryland, College Park, Md., 1983; Computer Science TR-1376.