AN OPTIMAL QUADTREE CONSTRUCTION ALGORITHM*

Clifford A. Shaffer
Hanan Samet

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

ABSTRACT

An algorithm is presented that builds 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 raster. 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.

1. INTRODUCTION

Hierarchical data structures such as the *region quadtree* (e.g., Figure 1) are important representations in many domains. Quadtrees and related hierarchical data structures are surveyed in [6]. For many problems, using a quadtree means that the amount of work required is proportional to the number of aggregated units (e.g., blocks) rather than to the sizes of the aggregated units (e.g., the number of pixels in a block). Quadtrees therefore have the potential for improved execution time efficiency. Nevertheless, raster to quadtree conversion requires that every pixel of the raster be examined. For this reason, previously reported quadtree building algorithms execute in time proportional to the number of pixels in the image. This can be costly, especially as in our case where the image is large and the quadtree is stored on disk. The *linear quadtree* representation [1,3] is useful for efficient manipulation of quadtrees stored on disk. It is currently being used to store images in an experimental geographic information system at the University of Maryland [7].

In this paper, 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 required to read the input data. For typical 512×512 pixel images, the new algorithm results in an order of magnitude or better improvement over naive algorithms that insert each pixel separately and require a merge routine to form larger nodes.

2. IMPLEMENTING LINEAR QUADTREES

Before building a linear quadtree, every pixel in the underlying array of the digitized image is assigned an address value (i.e., its locational code). This address is formed by interleaving the bits of the binary representation for the pixel's x and y coordinates [3] (e.g., Figure 2 where each x and y bit pair is represented by a single base-4 digit). When the pixels' addresses 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. Node addresses are generated by assigning to each node the address of the least valued pixel contained within the block it represents (e.g., the block labels in Figure 1). Note that the block in the NW quadrant of the image has a 0 value in the first position (indicating a NW branch), all blocks in the NE quadrant have a 1 in the first position, etc. The list of blocks is stored in a B-tree [1,7].

In our application, the linear quadtree is disk-based with only a small portion of an image in core at any given instant. Thus, the time spent moving segments of the image to and from the disk is an important factor. According to Comer [2], I/O accounts for the majority of the time spent in manipulating B-trees. A buffer pool helps to 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 discussed in Section 3, those parts of the system may take up about 98% of the execution time.

A common quadtree analysis metric is the number of nodes of the quadtree that are visited when performing an operation. Operations performed on the linear quadtree are different from operations performed on pointer-based representations. Many pointer-based algorithms involve *neighbor-finding* [5], which is done by ascending father links to the nearest common ancestor of the node and its neighbor, then descending the tree to the neighbor. 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, or any arbitrary relation, 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. Assuming a constant cost for inserting a node (an assumption supported by empirical tests), a reasonable metric for algorithm complexity is obtained by simply adding the number of node searches to the number of node insertions.

3. RASTER TO QUADTREE CONVERSION

The naive algorithm for converting a raster to a linear quadtree individually inserts each pixel of the raster into the quadtree in raster order. The quadtree insert routine merges those pixels making up larger nodes. Previous algorithms presented in the literature [4] have worked on this principle. Attempts at increasing efficiency concentrated on how to improve the insert routine. Table 1 contains the execution times and number of node insertions required when such an algorithm (marked as 'old' in the table) is applied to six test maps. The execution times are nearly identical for raster images with the same number of pixels (and thus, node inserts), regardless of the number of nodes in the eventual quadtree. In other words, the number of nodes in the output tree has little or no effect on the time required to execute 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.

Since the number of pixels in the raster representation for our images is large in comparison to the number of nodes in its quadtree representation, 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, perform a single insertion for each node in the quadtree. An algorithm with this worst-case performance is presented in this section. It is based on processing the image in raster-scan (top to bottom, left to right) order and, when necessary, inserting the largest node for which the current pixel is the first (upper leftmost) pixel. Thus, there is no need to merge since 1) the upper leftmost pixel of any block is inserted before any other pixel of that block; and 2) it is impossible for four sibling blocks to be of the same color.

*This work was supported in part by the National Science Foundation under Grant DCR-8605557.

CH2342-4/86/0000/0317$01.00 © 1986 IEEE

Table 1. Quadtree building algorithm statistics.

Map Name	Number Nodes	Number Inserts Old	Number Inserts New	Time (secs.) Old	Time (secs.) New
Flood	5266	180000	2352	413.2	13.8
Top	24859	180000	12400	429.8	51.2
Land	28447	180000	14675	436.7	56.9
Center	4687	262144	2121	603.8	16.1
Pebble	44950	262144	20770	630.1	111.0
Stone	31969	262144	14612	629.5	70.2

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 unprocessed portions of the quadtree are represented by nodes. If it were possible to know the current values of all unprocessed pixels as they are currently represented by the quadtree node list, then it would not be necessary to insert a pixel with color C from which a previous largest-node insertion had already set the containing node for that pixel to color C. We say that a node is ***active*** if at least one, but not all, pixels covered by the node have been processed. The efficient quadtree building algorithm must keep track of all of such active nodes. Given a $2^n \times 2^n$ image, an upper bound on the number of active nodes is 2^n-1. This can be seen by observing that 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. Similarly, 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). Summing results in at most 2^n-1 active nodes.

From these observations an improved quadtree building algorithm can be derived. Assume the existence of a data structure that 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., the 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. The list of active nodes is represented by an array, referred to as the ***active node table***, containing 2^n-1 entries to store all potentially active nodes.

The only remaining problem is to locate the smallest active node in the table that 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 new node is split to accommodate the insertion of a pixel having a color different from that of the current active node (e.g., after inserting pixel 3 in Figure 3b). Each pixel will have the color of the smallest active node that contains it, since the smallest 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 active node 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 one-dimensional array, referred to as *the access array*, is maintained to provide a pointer to the currently active node for that column in the active node table. The access array contains 2^{n-1} records, this being the maximum possible number of active nodes along a given row of 2^n pixels. As active nodes are inserted or completed (and deleted from the active node table), the active node table and the access array are updated.

Table 1 contains execution times of the new algorithm for the same maps used to test the naive algorithm. The new algorithm often requires far fewer calls to the insert routine than the number of nodes in the resulting output tree. This is because some calls to insert force node splits to occur, thereby increasing the number of nodes in the tree. For example, consider Figure 3b where processing 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 is the same color as the original node (A of Figure 3a), it will cause no additional node insertion.

We illustrate the new algorithm by considering how the quadtree of Figure 1 is constructed. Table 2 traces the active nodes at each stage of execution. Each row in Table 2 lists the active nodes after processing the pixel listed in column one. Pixel identifier (a,b) denotes a pixel in row a and column b relative to an origin at the image's upper left corner. When processing the first pixel of the array, the entire quadtree is represented by a single WHITE node (block A in Figure 4a). No other insertions occur while scanning rows 0 and 1. When the first BLACK pixel (2,4) is processed, block B of Figure 4a 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 Figure 4b 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) has been processed, the state is as shown in Figure 4c. The blocks previously labeled B and C are no longer active. Pixel-sized block D at (4,3) does not become active as it contains no unprocessed pixels, and thus only blocks A and E are active at this time. Figure 4d shows the algorithm's state after processing pixel (6,6). Block H became active after processing pixel (6,2). The block corresponding to pixel (6,5) has been inserted, but is not active. Since the smallest block containing pixel (6,6) had been BLACK, a new WHITE block has been activated (block I). Thus, three active blocks (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. A detailed algorithm is given in [8].

To understand why the new algorithm is such an improvement over the old one, let us analyze the cost of both algorithms in terms of the number of insert operations performed. The naive algorithm examines each pixel and inserts it into the quadtree. Denoting an insert operation's cost by I, and the cost for the time spent examining a pixel as c, the total cost is then $2^{2n} \cdot (c+I)$. The new algorithm 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 new algorithm's cost is $c \cdot 2^{2n} + I \cdot N$ where c is very 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 the new algorithm, yet is much less than $I \cdot 2^{2n}$. The result is that using the new algorithm reduces the execution time from O(pixels) to O(nodes). Of course, this is achieved at the cost of a slight increase in storage requirements due to the need to keep track of the active nodes (at most 2^n-1 records for a $2^n \times 2^n$ image). On the other hand, the quadtree's size during construction is likely to be smaller for the new algorithm since no merging need be performed.

Table 2. 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		B C
(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	H I
(7,3)	remove H from active		A	E I	
(7,5)	insert WHITE node J	1×1	A	E I	
(7,7)	remove I, E, A from active				

The largest-node-insertion technique discussed above can be used to improve the pointer based raster-to-quadtree algorithm described in [4]. That algorithm works bottom-up, beginning with a single node representing the raster array's first pixel. As each pixel of the first row is scanned, the current pixel's eastern neighbor is located. Using the largest-node insertion technique we can devise an analogous top-down algorithm which performs no merging. The algorithm also minimizes intermediate storage requirements since no space is needed for nodes that eventually would be merged and removed. 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 value C. The child containing the current pixel becomes the current node. If the current pixel is the node's first (upper-left) pixel, then the node's value is changed to C'. Otherwise the split step is repeated until the current pixel becomes the current node's upper-left corner. If the next pixel to be processed is beyond the current node's eastern edge, then that node's eastern neighbor is located. In this way, no unnecessary nodes are inserted, and no merging is performed.

4. CONCLUDING REMARKS

The techniques used in Section 3 can be applied to other functions that create a linear quadtree in a "reasonable" order. A reasonable order is one in which, for each node, the upper-left pixel is the first pixel to be inserted for that node. This restriction is satisfied by depth-first traversal (i.e., node address order), raster scan order, and 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.

REFERENCES

1. D.J. Abel, A B$^+$-tree structure for large quadtrees, *Computer Vision, Graphics, and Image Processing 27*, 1(July 1984), 19-31.

2. D. Comer, The ubiquitous B-tree, *ACM Computing Surveys 11*, 2(June 1979), 121-137.

3. I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM 25*, 12(December 1982), 905-910.

4. H. Samet, An algorithm for converting rasters to quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 3*, 1(January 1981), 93-95.

5. H. Samet, Neighbor finding techniques for images represented by quadtrees, *Computer Graphics and Image Processing 18*, 1(January 1982), 37-57.

6. H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260.

7. H. Samet, A. Rosenfeld, C.A. Shaffer, and R.E. Webber, A geographic information system using quadtrees, *Pattern Recognition 17*, 6(November/December 1984), 647-656.

8. C.A. Shaffer and H. Samet, An optimal quadtree construction algorithm, to appear in *Computer Vision, Graphics, and Image Processing*.

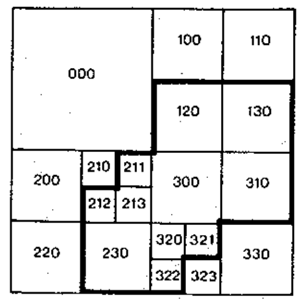

Figure 1. Example quadtree block decomposition.

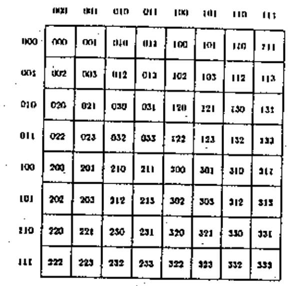

Figure 2. Interleaved pixel addresses for an 8×8 array.

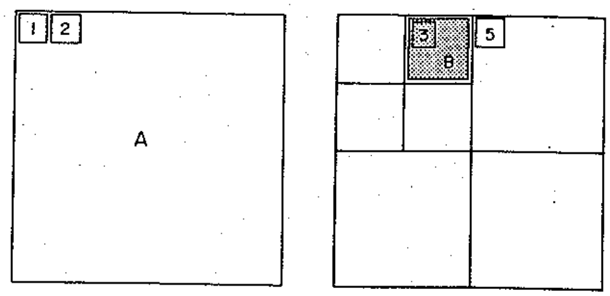

Figure 3. Left: node A is active after inserting a single pixel of color C. Right: the insertion of pixel 3 with color C' causes the creation of active node B when pixels 1 and 2 have color C.

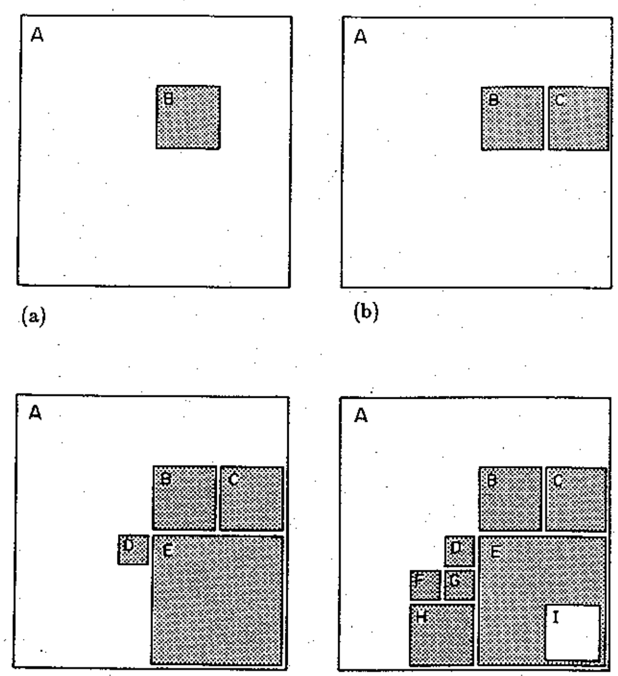

Figure 4. The quadtree construction process for the image of Figure 1. (a) shows the state after processing pixel (2,4); (b) after pixel (2,6); (c) after pixel (4,4); and (d) after pixel (6,6).