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 $512 \times 512$ pixel images, the 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. Quadrees and related hierarchical data structures are surveyed in [6]. For many problems, using a quadree 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). Quadrees therefore have the potential for efficient execution time. Nevertheless, raster to quadree conversion requires that every pixel of the raster be examined. For this reason, previously reported quadree 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 quadree is stored on disk. The linear quadtree representation [13] is useful for efficient manipulation of quadrees 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 quadree 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 $512 \times 512$ 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 QUADREES

Before building a linear quadree, 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 interweaving bits of the binary representation for the pixel's $x$ and $y$ coordinates [8] (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 last 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 an 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 [17].

In our application, the linear quadree is disk-based with only...
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. Give 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 nodes in the list) that includes the given pixel, and (if it is not a 1×1 pixel node) it is at 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 $2^n$ 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 nodes split to occur as part of the list of new nodes. In the 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.

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 $2^n (C + I)$. 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 $2^n C + N I$ where $C$ is very small in comparison to $I$, and $N$ is usually small in comparison to $2^n$. In other words, the quantity $I - N$ dominates the cost of the new algorithm, yet is much less than $I 2^n$. 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>
<thead>
<tr>
<th>Pixel</th>
<th>Action</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,0</td>
<td>insert WHITE node A</td>
<td>8x8</td>
</tr>
<tr>
<td>21,2</td>
<td>insert BLACK node B</td>
<td>2x2</td>
</tr>
<tr>
<td>21,6</td>
<td>insert BLACK node C</td>
<td>2x2</td>
</tr>
<tr>
<td>31,6</td>
<td>remove B from active</td>
<td>A</td>
</tr>
<tr>
<td>41,4</td>
<td>insert BLACK node D</td>
<td>1x1</td>
</tr>
<tr>
<td>41,4</td>
<td>insert BLACK node E</td>
<td>1x1</td>
</tr>
<tr>
<td>51,6</td>
<td>insert BLACK node F</td>
<td>1x1</td>
</tr>
<tr>
<td>51,3</td>
<td>insert BLACK node G</td>
<td>1x1</td>
</tr>
<tr>
<td>62,5</td>
<td>insert BLACK node H</td>
<td>2x2</td>
</tr>
<tr>
<td>62,6</td>
<td>insert WHITE node I</td>
<td>2x2</td>
</tr>
<tr>
<td>72,3</td>
<td>remove B from active</td>
<td>A</td>
</tr>
<tr>
<td>72,7</td>
<td>insert WHITE node J</td>
<td>1x1</td>
</tr>
<tr>
<td>82,7</td>
<td>remove I, A from active</td>
<td>E</td>
</tr>
</tbody>
</table>

The largest-node-insertion technique discussed above can be used to improve the pointer-based main-memory 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


Figure 1. Example quadtree block decomposition.

Figure 2. Interleaved pixel addresses for an 8X8 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.

(a) (b)  (c)  (d)

Figure 4. The quadtree construction process for the image of Figure 1. (a) shows the state after processing pixel (2,4); (b) after pixel (5,6); (c) after pixel (4,4); and (d) after pixel (6,6).