RECENT DEVELOPMENTS IN QUADTREE-BASED GEOGRAPHIC INFORMATION SYSTEMS*

Hanan Samet, Clifford A. Shaffer, Randal C. Nelson,
Yuan-geng Huang, Kikuo Fujimura, and Azriel Rosenfeld
Computer Science Department and Center for Automation Research
University of Maryland
College Park, Maryland 20742
USA

Abstract

The status of an ongoing research effort to develop a geographic information system based on a variant of linear quadtrees is described. This system uses quadtree encodings for storing area, point, and line features. Recent enhancements to the system are presented in greater detail. This includes a new hierarchical data structure for storing linear features that enables representing straight lines exactly as well as permitting updates to be performed in a consistent manner. The memory management system was modified to enable the representation of an image as large as 16,384 by 16,384 pixels. Improvements were also made to some basic area map algorithms which yield significant efficiency speedups by reducing node accesses. This includes windowing, set operations with unregistered images, a within function, and an optimal quadtree building algorithm which has an execution time that is proportional to the number of blocks in the image instead of the number of pixels.

*This work was sponsored by the U.S. Army Engineer Topographic Laboratories under contract DAAK-70-81-C-0059.

1. Introduction

Hierarchical data structures are important representations in geographic information systems, as well as in the related domains of computer vision, robotics, computer graphics, image processing, pattern recognition, and computational geometry. The advantage of hierarchical methods is that their use leads to aggregation resulting in algorithms whose execution times are 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). 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. The various elements of the class can be differentiated on the basis of the type of data that they are used to represent, and on the principle guiding the decomposition process. Currently, variants of quadtrees are used for point data, regions, curves, surfaces, and volumes. The decomposition may be into equal-sized parts (termed a regular decomposition), or it may be governed by the input. The parts need not necessarily be disjoint nor must they be at a fixed orientation relative to each other. In the case of spatial data, a representation that is related to the quadtree is the pyramid which is a multiresolution data structure. In contrast, the quadtree is a variable resolution representation. Figure 1 is an example of a region and its corresponding region quadtree. For a recent survey of the use of hierarchical data structures see Samet (1984).

For the past four years, members of the Computer Vision Laboratory at the University of Maryland have been engaged in a research effort to develop a geographic information system based on quadtrees. The project has been conducted in four phases. In this paper we describe the current state of this effort. However, we first review the work that has already been completed. The database used in this study was supplied by the U.S. Army Engineer Topographic Laboratories, Ft. Belvoir, VA. The area data consisted of three registered map overlays representing landuse classes, terrain elevation contours, and a floodplain boundary from a region in Northern California.

In Phase I a quadtree database was built for these maps (Rosenfeld et al., 1982). The overlays were hand-digitized resulting in three arrays of size 400 by 450 pixels. Labels were associated with the pixels in each of the resulting regions, specifying the particular landuse class or elevation range. The regions were subsequently embedded within a 512 by 512 grid and quadtree encoded. The results are shown in Figures 2-4. Algorithms were developed for basic operations on quadtree-represented regions (set-theoretic operations, point-in-region determination, region property computation, and submap generation). The efficiency of these algorithms was studied theoretically and experimentally.

In Phase II, a quadtree-based Geographic Information System was partially implemented, allowing manipulation of images storing area, point and line data (Rosenfeld et al., 1983). This system included a memory management system to allow manipulation of images too large to fit into main memory, a software package to allow users to edit and update images, database management and map manipulation functions, and an English-like query language with which to access the database. We also made use of a geographic survey map for this area, from which we extracted point and line data.

Phase III dealt primarily with enhancements and alterations to this information system package, an evaluation of some of the design decisions, and the collection of empirical results to indicate the utility of the software and to justify the indicated design decisions (Samet et al., 1984). Also included was the first step of an attribute attachment package for storing non-geographic data associated with the map objects, and a survey of appropriate point and linear feature data structures for future investigation.

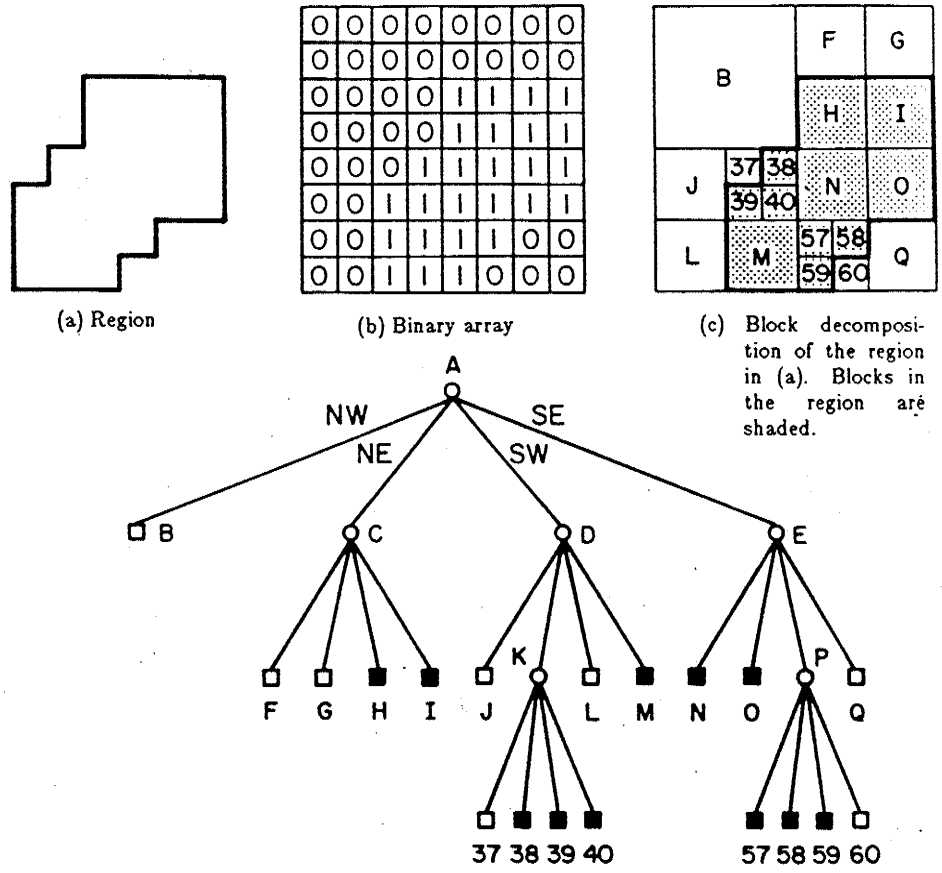

Figure 1. A region, its binary array, its maximal blocks, and the corresponding quadtree.

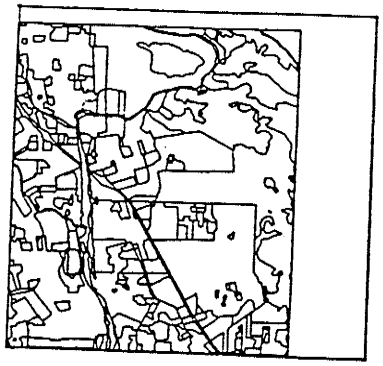

Figure 2. The landuse map.

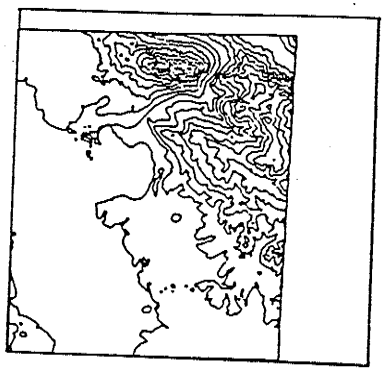

Figure 3. The topography map.

Figure 4. The floodplain map.

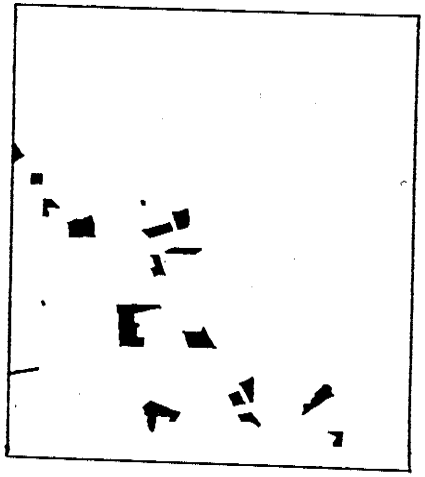

Figure 5. The ACC landuse class map.

Phase IV of the project primarily dealt with developing new structures for storing linear feature data (Samet et al., 1985). The attribute attachment package was extended to point and linear feature data. Existing area map algorithms were improved to yield significant efficiency speedups by reducing node accesses. In this paper we expand further on the developments in Phase IV; however, we first give a brief historical review of the quadtree data structure.

Hierarchical data structures such as the quadtree have an interesting history that reflects their general utility. They are based on the principle of recursive decomposition and early mentions of it did not make much use of their tree formulation. Their use can be traced to several fields. The formulation that is most commonly used today is known as the *region quadtree* and is due to Klinger (1971). Such quadtrees were probably first seen as a way of aggregating blocks of zeros in sparse matrices for use in applications in numerical analysis (Hoare, 1972). They were used by Morton (1966), and undoubtedly others, as a means of indexing into databases containing geographical information. Warnock (1969) applied recursive decomposition in computer graphics, while Hunter and Steiglitz (1979) used the tree representation for the purpose of animation. Variants of the quadtree were also used in robotics in the SRI robot project (Nilsson, 1969), in architecture for space planning (Eastman, 1970), in image understanding (Kelly, 1971), in VLSI design rule checking (Kedem, 1981), and in other fields. There are two other data structures that are related to the region quadtree. The first is the point quadtree (Finkel and Bentley, 1974) which is a generalization of the binary search tree and is used to represent multidimensional point data. It has spawned countless variants (e.g., the k-d tree (Bentley, 1975), etc.) which are used in databases and facilitate search operations. The second is the pyramid (Uhr, 1972; Riseman and Arbib, 1977; Tanimoto and Pavlidis, 1975) which is an exponentially tapering stack of arrays, each one-quarter the size of the previous array. It has found use in such tasks as feature detection, segmentation, etc.

2. The Quadtree Memory Management System

The quadtree memory management system, described in greater detail in Phase II of this project (Rosenfeld et al., 1983), is based on use of a structure termed the *linear quadtree* (Gargantini, 1982). The leaf nodes making up the quadtree of an image are stored in a list. In the variation which we have implemented each leaf node consists of two 32 bit words. The first word contains a 32 bit key which is used to order the node list. It is formed by bit interleaving the binary representation of the x and y coordinates of the pixel in the lower left corner of the block represented by the leaf node. When sorted in ascending order of the value of the key, the node list will be in an order identical to that in which the leaves would have been visited by a depth-first traversal of the original tree. In order to be able to determine the size of the leaf, we must also specify the depth. We use 4 bits of the 32 bit key to denote the depth which means that 14 bits are left for each of the x and y coordinates. Thus an image as large as 16,384 by 16,384 pixels can be represented. Each leaf also contains a 32 bit value field.

One motivation for using a linear quadtree in contrast to a pointer-based quadtree is that it allows for a reduction in storage. In particular, a pointer-based representation requires that we store with each node four pointers to its children and often a pointer to its father. This is in addition to the value field. We have implemented the linear quadtree in conjunction with a disk based memory management system which only needs to maintain a small part of the image in core at one time. In our system, the sorted list of quadtree leaves is stored in a B-tree (Comer, 1979) with a page size of 1024 bytes, capable of holding up to 120 leaves in a page. For more details on the implementation, see the Phase II report (Rosenfeld et al., 1983).

The line representation, described in Section 4, may result in storing more than one item of information with a node. This requires a variable size node implementation. Since the value field of each linear quadtree node consists of just one 32 bit word, we choose to duplicate the

node for each item of information that is associated with the node. This is wasteful of space since the information in the address field is repeated. However, more importantly, this method is compatible with our area and point representations with only minor modifications.

A variable length quadtree node is processed by locating the first record in a B-tree page with the desired address, and then visiting successive records until one with a greater address is encountered. New functions were written for finding the n th record with a given address in the B-tree page, for inserting a record with a given address into the B-tree, and for deleting a record with a given address and specified contents. Manipulating variable-sized nodes using this scheme is efficient since cases where multiple records with a given key are split between B-tree pages are rare. This is true because the average amount of information associated with a quadtree block in our application is small in comparison to that of the B-tree page.

3. Improved Database Functions

A number of existing database functions were significantly improved by being reimplemented using new algorithms. These include the WITHIN function, the raster to quadtree conversion function, and the map windowing function. In addition, the functions that implement set operations (e.g., union and intersection) were extended to work on unregistered images. The new algorithms are described briefly below.

3.1 The Within Function

The WITHIN function generates a map which is BLACK at all pixels within a specified radius of the non-WHITE regions of an input map. It is important for answering queries such as "Find all cities within 5 miles of wheat growing regions". Such a query would be answered by invoking the WITHIN function to operate on a map containing wheat growing regions (i.e., the non-WHITE regions), and then intersecting the result with a map containing cities.

The algorithm that we used previously (Samet et al., 1984) worked by expanding each non-WHITE block of the input image by R units (where R is the radius), and then inserting all of the nodes making up this expanded square into the output quadtree. This leads to many redundant node insertions. In addition, many of the nodes that were inserted were small, and in fact were eventually merged to form larger nodes.

The new algorithm is based on the chessboard distance transform (Samet, 1982). The algorithm does the following for each node of the input image. If the node is non-WHITE, then it is inserted into the output map. If the node is WHITE, and is less than or equal to $(R + 1)/2$ in width, then it must lie entirely within R pixels of a non-WHITE node. This is true because one of its siblings must contain a non-WHITE pixel. Thus, it is made BLACK and inserted into the tree. If the node is WHITE and has a width greater than $(R + 1)/2$, then its chessboard distance transform is computed. In other words, the exact distance from the node's border to the nearest non-WHITE pixel is determined. If this distance is such that the node is completely within radius R of a non-WHITE pixel, then it is inserted as a BLACK node into the output tree. If the node is completely outside the radius, then it is inserted as WHITE. Otherwise, the node is quartered, and the process is recursively reapplied to each quadrant.

The new algorithm is an improvement over the old one for two reasons. First, only large WHITE nodes need excessive computation. Since most nodes in a quadtree are small, very few nodes generate much work. Secondly, although nodes of the input tree may be visited several times when neighboring nodes compute their distance transform values, each block of the output tree is processed exactly once. Table 1 contains a comparison of the two algorithms for the floodplain of Figure 4 and the portion of the landuse class map that only shows class ACC as

BLACK (see Figure 5). The algorithm is applied to the two maps for radius values ranging from 1 to 8 pixels. Notice that the execution time speedups are more than linear.

Table 1. Execution times for the WITHIN function.

Distance	Flood time (secs.)		ACC time (secs.)	
	new algorithm	old algorithm	new algorithm	old algorithm
1	21.4	50.4	25.7	57.8
2	29.2	40.8	32.9	47.9
3	25.4	89.3	30.4	105.5
4	31.5	73.3	35.7	84.7
5	38.7	141.1	42.9	170.1
6	41.6	143.9	43.8	164.5
7	38.3	222.4	48.6	258.8
8	40.1	205.5	43.9	239.8

3.2 An Optimal Quadtree Construction Algorithm

The naive algorithm for converting a raster image to a linear quadtree (or a pointer based quadtree) is to individually insert each pixel of the raster image into the quadtree in raster order. Those pixels making up larger nodes are merged together by the quadtree insert routine. Previous algorithms presented in the literature (Samet, 1981), as well as the one used previously in our system (Rosenfeld et al., 1982), have worked on this principle. Attempts at increasing efficiency concentrated on how to improve the insert routine. Table 2 contains the execution times of the naive algorithm when applied to six test maps. The timings are nearly identical for raster images with the same number of pixels (i.e., 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 image data is approximately 1% of the time necessary to insert every pixel.

Table 2. Naive quadtree building algorithm statistics.

Map Name	Num Nodes	Num Inserts	Time (secs.)
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

An optimal algorithm has been developed that makes a single insertion for each node in the quadtree. It is based on processing the image in raster-scan (top to bottom, left to right) order, always inserting the largest node for which the current pixel is the first (upper leftmost) pixel. Such a policy avoids the necessity of merging since the upper leftmost pixel of any block is inserted before any other pixel of that block. Therefore, it is impossible for four sibling blocks to be of the same color.

At any point during the quadtree building process there is a processed portion of the image and an unprocessed portion. Both the processed and unprocessed portions of the quadtree have been assigned to nodes. We say that a node is *active* if at least one pixel, but not all pix-

covered by the node has been processed. The optimal quadtree building process must keep track of all of these active nodes. Given a $2^n \times 2^n$ image, it has been shown that the number of active nodes is bounded by $2^n - 1$ (Samet et al., 1985). Using these observations, an optimal quadtree building algorithm is outlined below. It assumes 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.

In order to implement such an algorithm we need to keep track of the list of active nodes. This list is represented by a table, say TABLE, with a row for each level of the quadtree (except for level 0 which corresponds to the single pixel level; these nodes cannot be active). Row i of the table contains 2^{n-i} entries, with row n corresponding to the full image. Given a pixel in column j, the value of the active node at row i of the table is found at position $j/2^i$. Note that shift operations can be used instead of divisions if speed is important.

The only remaining problem is how to locate the appropriate active node corresponding to each pixel. In particular, for a given pixel in a $2^n \times 2^n$ image, as many as n active nodes could exist. Multiple active nodes for a given pixel occur whenever a new node is inserted, as illustrated in Figure 6. Each pixel will have the color of the smallest of the active nodes which covers it, since the smallest node will have been the most recently inserted. Finding the smallest active node that contains a given pixel can be done by searching from the lowest level in the table 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, called LIST and referred to as the access array, is maintained to provide an index into TABLE. LIST is of size 2^{n-1} since single-pixel sized nodes need not be stored. For any pixel in column j, the LIST entry at $j/2$ indicates the row of TABLE corresponding to the smallest active node containing the pixel. At the beginning of the algorithm, each entry of LIST points to the entry of TABLE corresponding to the root (i.e., row n for a $2^n \times 2^n$ image). As active nodes are inserted or completed (and are to be deleted from the active node table), the active node table and the access array are updated.

Table 3 contains timing results when the new algorithm is applied to the same test maps as the naive algorithm. As indicated in Table 3, the optimal 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 may cause several node splits to occur thereby increasing the number of nodes in the tree. For example, in Figure 6 inserting node B into the quadtree containing a single node causes 7 nodes to result. If the first pixel inserted into node X happens to be the same color as the original node (A of Figure 6a), then no insertion is required.

In order 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 that they perform. The naive algorithm examines each pixel and inserts it into the quadtree. Assuming a cost of I for each insert operation, and a cost of c for the time spent examining a pixel, 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 cost of the new algorithm 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 the new algorithm. The result is that using the new algorithm reduces the execution time from being O(pixels) to O(nodes). Thus the algorithm is optimal t owithin a constant factor. Of course, this is achieved at an increase in storage requirements due to the need to keep track of the active nodes (approximately 2^{n+1} for a $2^n \times 2^n$ image).

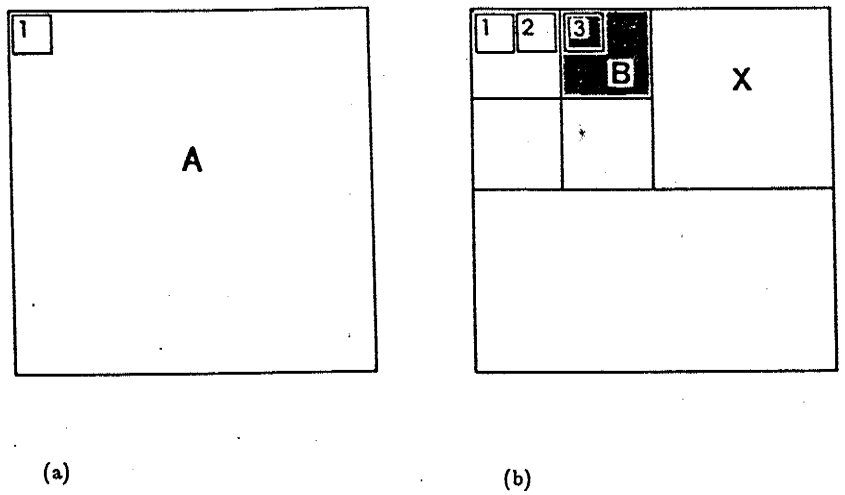

Figure 6. Node insertion can create multiple active nodes. (a) Node A is active after inserting a single pixel of color C. (b) The first two pixels have color C. Pixel 3 has color D. It insertion creates active node B. Node A is still active.

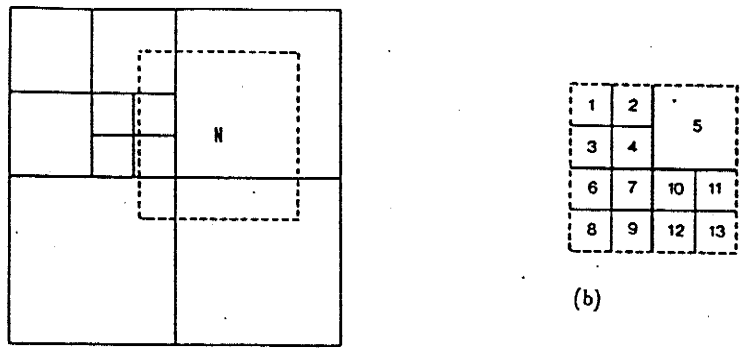

Figure 7. An example of intersection. (a) Node N (shown in dashed lines) from the first input tree is intersected with an image corresponding to the second input tree. It is compared against those nodes which it intersects in the second input tree. (b) The decomposition and order of processing for node N as directed by the image decomposition.

Table 3. Optimal quadtree building algorithm statistics.

Map Name	Num Nodes	Num Inserts	Time (secs.)
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

3.3 Set Operations For Unregistered Maps

In many applications, including geographic information systems, it is desirable to compute set operations on a pair of images. For example, suppose a map is desired of all wheatfields above 100 feet in elevation. This can be is achieved by intersecting a wheatfield map and an elevation map whose pixel values are BLACK if they represent an area whose elevation is above 100 feet. The resulting map would have BLACK pixels wherever the corresponding pixels of the input maps are both BLACK.

In this section we will consider only the case of map intersection - other set operations such as union or difference are handled in an analogous manner. Intersection of quadtrees representing images with the same grid size, same map size, and same origin is accomplished by traversing the two trees in parallel. Each node of the first image is compared with the corresponding node(s) in the second image. On the other hand, very little work has been done on set operations between unregistered quadtrees (i.e., quadtrees which have the same grid size and map size, but differing origins). In particular, the only prior mentions of algorithms for intersecting unregistered quadtrees involved translating one of the images to be in registration with the other, and then performing a registered intersection (Gargantini, 1983). In this section, the principles underlying an optimal algorithm for the intersection of unregistered maps are described. By optimal we mean that each node of the input images is visited only once, and at most one insertion into the output tree is performed for each output tree node.

As with the quadtree building algorithm of Section 3.2, the intersection algorithm maintains a table of the active output tree nodes to minimize insertions into the output tree. We will call this table OUT_TABLE. Unlike the building algorithm, there are two input quadtrees (call them I1 and I2) to be considered as well. The basic algorithm is as follows. I1 is processed in depth-first traversal order. For each node N of I1, the various nodes of I2 which cover N are located. Starting with the upper left pixel of N, the node of I2 which covers that pixel is located. Next, the largest block contained within both nodes is computed. The set function is evaluated on the values of these two nodes, and OUT_TABLE is queried to determine if the new node should be inserted. This step is repeated on subsequent portions of N in depth-first order until all pixels of N are processed. Figure 7 provides an example.

In order to minimize the number of node searches made in the tree I2, a second set of tables will be used to keep track of the active nodes of I2. OUT_TABLE is easily implemented, since nodes will always be inserted in depth-first order (matching the progress made in tree I1). During the traversal of the output tree, the second, third, and fourth subquadrants of a block at level i will not be processed until the previous subquadrants are completed (e.g., the SW subquadrant will not be processed until the NW and NE subquadrants are complete). Thus, at most one node at each level of the tree can be active. This means that for a $2^n \times 2^n$ image, a table of only n entries is needed to represent the active nodes. Each entry of OUT_TABLE

contains the location and value of the current active node at the corresponding level, along with a field to indicate the quadrant relative to its father in which the node lies. In addition, a variable is needed to keep track of the current depth.

The final requirement for the non-registered set function algorithm is a method for keeping track of the active nodes of the second input tree. Consider the border of the nodes of I1 which have been processed at any given instant. Since these nodes are processed in depth-first traversal order, the border will be in the form of a staircase (see Figure 8). The active border, as it crosses an output map of size $2^n \times 2^n$, will form horizontal and vertical segments such that the sums of the horizontal and vertical segments will each be 2^n pixels in length. The active nodes of I2 will be those nodes which, at any given instant, straddle the active border. The active border table for the intersection algorithm is a modification of the active border table used by Samet and Tamminen (1985). It is composed of two arrays each 2^n records wide. Each record contains the location, size, and value of the active node at that position.

3.4 Windowing

Interestingly, a variant of the unregistered intersection algorithm described above can also be used to perform windowing. Windowing is the name given to a function which extracts a window from an image. A window is simply any rectangular subsection of the image. Typically, the window will be smaller than the image, but this is not necessarily the case as the window could also be partly off the edge of the image. More importantly, the origin (or lower left corner) of the window could potentially be anywhere in relation to the origin of the input map. This means that large blocks from the input quadtree must be broken up, and possibly recombined into new blocks in the output quadtree.

Shifting an image represented by a quadtree is a special case of the general windowing problem - taking a window equal to or larger than the input image but with a different origin will yield a shifted image. Shifting is important for operations such as finding the quadtree of an image which has the fewest nodes. It can also be used to register two images represented by quadtrees. In order to simplify the following presentation, we will assume a window of size 2^m by 2^m taken from an image of size 2^n by 2^n where $m \leq n$.

In order to see the analogy between windowing and the intersection of two unregistered images, let I1, corresponding to the first image, be a BLACK block with the same size and origin as the window. Let I2, corresponding to the second image, be the image from which the window is extracted. The resulting image would have the size and position of I1, with the value of the corresponding pixel of I2 at each position. The equivalence between windowing and unregistered set intersection should be clear. In fact, the windowing algorithm would be simpler, since a single BLACK node of the appropriate size would take the place of I1 in the algorithm. Such an algorithm is optimal in the sense that it locates (only once) those nodes of the input tree which cover a portion of the window, and performs at most one insert operation for each output node.

4. Representation Of Linear Features

One of the goals of the research effort described here was the development of a uniform representation for data corresponding to regions, points, and vector features. Uniformity facilitates the performance of set operations such as intersecting a vector feature with an area, etc. Use of a linear quadtree for point and region data is well understood; however, this is not the case for vector features. For vector features, a good linear quadtree representation must also have the following three properties. First, it must use a constant (or at least bounded) amount of storage per node. Second, straight line segments should be represented exactly (not in a

digitized representation). Third, updates must be consistent, i.e., when a vector feature is deleted, the data base should be restored to a state identical (not an approximation) to that which would have existed if the deleted vector feature had never been added.

In Phase II of this project, we implemented a variation of the edge quadtree of Shneier (1981) for which we use the term *linear edge quadtree*. In our implementation of the edge quadtree, the leaf nodes of the quadtree are stored as single records in the B-tree. Each node contains three fields; an address, a type, and a value field. The address field describes the size of the node and the coordinates of one of the corners of its corresponding block. The type field indicates whether the node is empty (i.e., WHITE), contains a single vertex, or contains a line segment. The value field of a line segment indicates the coordinates of its intercepts with the borders of its containing node. Vertices are represented by pixel-sized nodes with the degree of the vertex stored in the value field. Unlike Shneier's formulation, a line segment may not end within a node since in our existing implementation the value field is not large enough to contain the location of an interior point as well as the intercepts. Thus endpoints and intersection points are represented by single pixel-sized point nodes. Figure 9 illustrates the linear edge quadtree representation.

A serious drawback of the edge quadtree is its inability to handle the meeting of two or more edges at a single point (i.e., a vertex) except as a pixel corresponding to an edge of minimal length. This means that all vertices are stored at the lowest level in the digitization - i.e., in deep nodes in the tree. Thus, we cannot distinguish vertices from short line segments. Moreover, boundary following as well as deletion of line segments cannot be properly handled in the vicinity of a vertex at which several edges meet.

In order to overcome these shortcomings we developed a new representation termed a *PMR quadtree* which is a variation on the PM quadtree of Samet and Webber (1985). The PMR quadtree makes use of probabilistic splitting and merging rules, one for splitting and one for merging, to dynamically organize the data. The splitting rule is invoked whenever a line segment is added to a node. The node is split *once* into four quadrants if the number of segments it contains exceeds n (4 in our implementation). Note that this rule *does not* guarantee that each subquadrant will contain at most n line segments. The corresponding merging rule is invoked whenever a segment is deleted. The node is merged with its siblings if together they contain fewer than n (4 in our implementation) distinct line segments. The merge operation can be performed more than once. This scheme differs from the other quadtree structures in that the tree for a given data set is not unique, but depends on the history of manipulations applied to the structure. Certain types of analysis are thus more difficult than with uniquely determined structures. On the other hand, this structure allows the decomposition of space to be based directly on the linear feature data stored locally.

The PMR quadtree makes use of variable sized nodes. When the graph represented by the set of line segments is planar (which is the case for polygonal maps and most geographical situations), the average number of segments per node in the PMR decomposition is limited by topological considerations to some small number (for our map data, the average is less than three). This makes an implementation of the node as a list practical. In an application where this is not the case, other splitting rules can ensure that the number of segments in a node does not become too high. For linear quadtrees, where an address is calculated for each quadrant and used to order it in the list, the simplest way of implementing variable node sizes is simply to duplicate the addresses.

The PMR quadtree induces a decomposition of the space that may split a line segment into many portions. Each portion that lies in a quadtree node is termed a *q-edge*. The q-edges that are stored with each node are represented by a pointer to a record corresponding to the entire line segment of which they are a part. This solves the problem of how to accurately

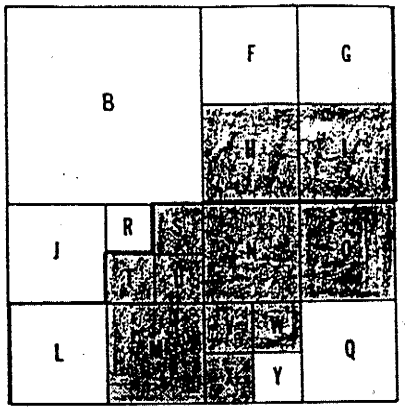

Figure 8. The active border after processing node R in the first input tree.

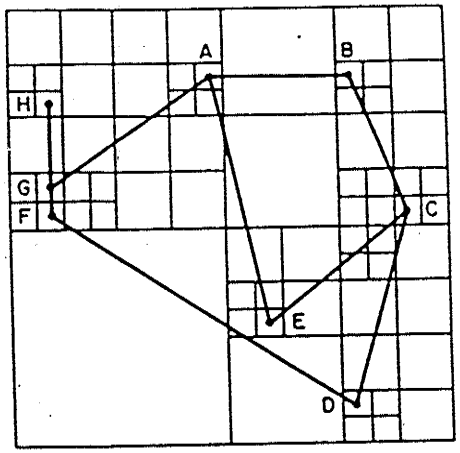

Figure 9. A linear edge quadtree.

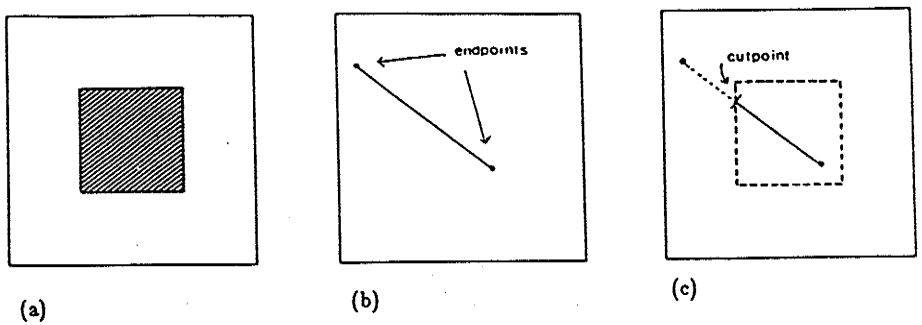

Figure 10. The intersection of (a) a region; and (b) a line segment; producing (c) a fragment with one cut point.

represent the intersection of a q-edge with a quadrant boundary without loss of precision. Since each node containing a q-edge of a given line segment stores the same descriptor, tracking the line from block to block is simple and operations such as deletion can be easily implemented. Note that using a pointer to a record describing each line segment leads to more flexibility since it allows storing an arbitrary amount of information about the line segment without increasing the size of the B-tree record. It also enables this information to be concentrated in one place rather than repeated in every node which refers to the line segment.

We must also address the problem of how to represent a line segment that has been broken into fragments. This situation arises in a geographical application when a line map is intersected with an area. Since the borders of the area may not correspond exactly with the endpoints of the segments defining the line data, certain segments may be cut off (e.g., Figure 10). Such a partial line segment is referred to as a *fragment* and the artificial endpoints produced by such an intersection are referred to as *cut points*. The locations of such cut points must be represented in some fashion. One idea is to introduce an intermediate point at the node intercept. In continuous space, the remaining line segment can then be exactly represented as a new line segment which is collinear with the original one, but has at least one different endpoint. In discrete space however, this is not always possible because the continuous coordinates at the intercept do not, in general, correspond exactly to any coordinates in the discrete space. If the new line segments are represented approximately in the discrete space, then the original information is degraded, and the pieces cannot reliably be rejoined. Note that these were precisely the problems that were encountered with the linear edge quadtree. Moreover, if an intermediate point were to be introduced to produce new segments, then the line segment descriptor would have to be propagated to all quadrants containing the original segment. This is likely to be a very time-consuming operation.

The approach that we took retains the original pointers, and uses the spatial properties of the quadtree to specify what parts of the corresponding segments (i.e., q-edges) are actually present. We observe that even though a node contains a pointer to a line segment, it is not necessarily true that the entire line segment is present as a linear feature. Instead, the line segment descriptor contained in a node is interpreted as only implying the presence of the corresponding q-edge. The original line segment of which the q-edge is a part is termed the *parent segment*. The fragments, therefore, may be represented by a collection of q-edges. The presence or absence of a particular q-edge is completely independent of the presence or absence of those q-edges representing other parts of the line segment. Hence linear features corresponding to partial segments can be represented simply by inserting the appropriate collection of q-edges. Since the original pointers are retained, a linear feature can be broken into pieces and rejoined without loss or degradation of information. Within the quadtree structure, q-edges may represent arbitrary fragments of line segments. Since all bear the same segment descriptor, they are easily recognizable as deriving from the same parent segment. This solves the problem of how to split a line or a map in an easily reversible manner. The use of this principle to represent the linear feature produced by the intersection of Figure 10 is shown in Figure 11.

Properly dealing with entities such as cut points and fragments requires that we modify the splitting and merging rules of the PMR quadtree in the following manner. Nodes are split until no block contains a cut point in its interior, and then once more if a quadrant contains more than four q-edges. The merge condition is invoked both when a q-edge is deleted and when a q-edge is inserted (since a fragment may be inserted which restores a larger piece). Merges occur when there are four or fewer distinct line segments among the siblings and the q-edges are compatible.

In order to evaluate the performance of the PMR quadtree we compared it with three other data structures for handling linear features using the road network of Figure 12. This network has 684 vertices and 764 edges. The three data structures used in the comparison are the

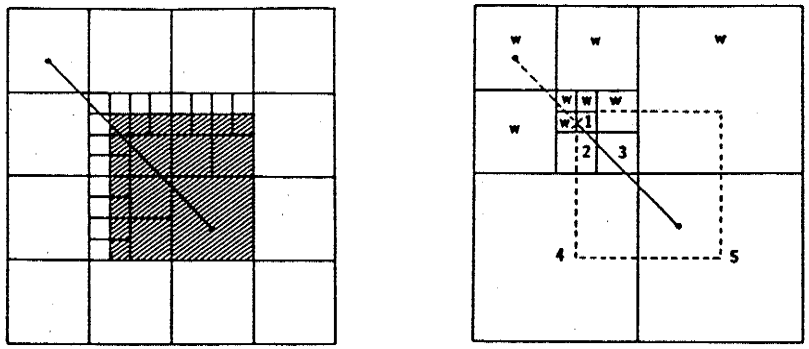

Figure 11. The quadtree representation of a fragment. (a) A region decomposition with a line segment superimposed. (b) The minimal set of five q-edges which make up the fragment of Figure 10.

Figure 12. The road network.

MX quadtree (Hunter and Steiglitz, 1979), the edge quadtree, and the PM_3 quadtree (Samet and Webber, 1985). For a $2^n \times 2^n$ image, the MX quadtree for a collection of line segments treats every pixel that is intersected by a line as BLACK and all remaining pixels as WHITE. Merging is applied to the WHITE pixels to create larger nodes. The MX quadtree, like the edge quadtree, is really not suitable for our applications but it is useful for comparison purposes. The PM_3 quadtree is based on the principle that the space is decomposed until there is only one vertex in each quadrant. In order to deal with cut points and fragments, this decomposition rule is modified by splitting until no block contains a cut point (i.e., all cut points must lie on the boundaries of blocks), and no block contains more than one segment endpoint attached to a q-edge. The PM_3 and the PMR quadtrees are closely related. Table 4 shows the building times for the various quadtrees, the total number of leaf nodes, and the number of q-edges (termed nodes in the table and meaningless for the MX and edge quadtree). Note that the storage requirements for the PMR quadtree are smaller than for the PM_3 quadtree as is the quadtree building time. This is not surprising since the PMR quadtree will not be as deep as the PM_3 quadtree, nor as deep as the MX and edge quadtrees.

Table 4. Building times and sizes for the road network.

Type of Quadtree	Time (secs.)	Leaves	Qnodes
MX	31.5	19699	-----
edge	27.4	7723	-----
PM_3	39.0	3939	2350
PMR	25.8	2078	874

We also compared the performance of the four data structures for line/area intersections. The road network map was intersected with several area maps. Once again, the PMR quadtree required the least amount of space. However, the execution times for the PMR quadtree were slower than those for the PM_3 representation by factors of up to 50% depending on the complexity of the maps. The MX and edge quadtrees were more efficient from the standpoint of execution time than either PM representation. Again, this is not surprising since by having fewer nodes, the PMR quadtree reduces the opportunity for pruning and also has a more complex node structure. To summarize, since the PMR quadtree enables correct execution of the intersection operation, the fact that it is slower than the edge and MX quadtrees for certain operations is irrelevant. The relative slowness of the PMR quadtree with respect to the PM_3 quadtree is a direct result of the time vs: space tradeoff between the two representations.

5. Concluding Remarks

Our goal was to demonstrate the utility of hierarchical data structures for use in the domain of geographic information systems. In order to accomplish this goal we have built a prototype geographic information system which represents images with a linear quadtree. This system is capable of manipulating area, point and linear feature data in a reasonably efficient manner. All of these features are implemented in a consistent manner. Our experience has been that while area and point data are easily handled by an area based-representation, the correct treatment of linear feature data is considerably more difficult. We have developed a new data structure termed a PMR quadtree which meets our requirements and have incorporated it into our system. Future work includes more research into facilities for dealing with attribute data as well as larger images.

References

Bentley, J.L., 1975: Multidimensional Binary Search Trees Used for Associative Searching, *Communications of the ACM 18*, No. 9(September), 509-517.

Comer, D., 1979: The Ubiquitous B-tree, *ACM Computing Surveys 11*, No. 2(June), 121-137.

Eastman, C.M., 1970: Representations for Space Planning, *Communications of the ACM 13*, No. 4(April), 242-250.

Finkel, R.A., and J.L. Bentley, 1974: Quad Trees: a Data Structure for Retrieval on Composite Keys, *Acta Informatica 4*, No. 1, 1-9.

Gargantini, I., 1982: An Effective Way to Represent Quadtrees, *Communications of the ACM 25*, No. 12(December), 905-910.

Gargantini, I., 1983: Translation, Rotation, and Superposition of Linear Quadtrees, *International Journal of Man-Machine Studies 18*, No. 3(March), 253-263.

Hoare, C.A.R., 1972: Notes on Data Structuring. In *Structured Programming*, O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare (Eds.), Academic Press, London, 154.

Hunter, G.M., and K. Steiglitz, 1979: Operations on Images Using Quad Trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, No. 2(April), 145-153.

Kedem, G., 1981: The Quad-CIF Tree: a Data Structure for Hierarchical On-Line Algorithms, TR 91, Computer Science Department, The University of Rochester, Rochester, New York, September.

Kelly, M.D., 1971: Edge Detection in Pictures by Computer Using Planning, *Machine Intelligence 6*, B. Meltzer and D. Michie, Eds., 397-409.

Klinger, A., 1971: Patterns and Search Statistics. In *Optimizing Methods in Statistics*, J.S. Rustagi (Ed.), Academic Press, New York, 303-337.

Morton, G.M., 1966: A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing, IBM Ltd., Ottawa, Canada.

Nilsson, N.J., 1969: A Mobile Automaton: an Application of Artificial Intelligence Techniques, *Proceedings of the First International Joint Conference on Artificial Intelligence*, Washington D.C., 509-520.

Riseman, E.M., and M.A. Arbib, 1977: Computational Techniques in the Visual Segmentation of Static Scenes, *Computer Graphics and Image Processing 6*, No. 3(June), 221-276.

Rosenfeld, A., H. Samet, C. Shaffer, and R.E. Webber, 1982: Application of Hierarchical Data Structures to Geographical Information Systems, Computer Science TR-1197, University of Maryland, College Park, MD, June (see also *IEEE Transactions on Systems, Man, and Cybernetics 13*, 6(November/December 1983), pp. 1148-1154).

Rosenfeld, A., H. Samet, C. Shaffer, and R.E. Webber, 1983: Application of Hierarchical Data Structures to Geographical Information Systems Phase II, Computer Science TR-1327, University of Maryland, College Park, MD, September (see also *Pattern Recognition 17*, 6

(November/December 1984), 647-656).

Samet, H. 1981: An Algorithm for Converting Rasters to Quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 3*, No. 11(January), 93-95.

Samet, H. 1982: Distance Transform for Images Represented by Quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 4*, No. 3(May), 298-303.

Samet, H. 1984: The Quadtree and Related Hierarchical Data Structures, *ACM Computing Surveys 16*, No. 2(June), 187-260.

Samet, H., A. Rosenfeld, C.A. Shaffer, R.C. Nelson, and Y.G. Huang, 1984: Application of Hierarchical Data Structures to Geographical Information Systems Phase III, Computer Science TR-1457, University of Maryland, College Park, MD, November.

Samet, H., and R.E. Webber, 1985: Storing a Collection of Polygons Using Quadtrees, *ACM Transactions on Graphics 4*, 3(July 1985), 182-222.

H. Samet, and and M. Tamminen, 1985: Computing Geometric Properties of Images Represented by Linear Quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, No. 2(March), 229-240.

Samet, H., A. Rosenfeld, C.A. Shaffer, R.C. Nelson, Y.G. Huang, and K. Fujimura, 1985: Application of Hierarchical Data Structures to Geographical Information Systems Phase IV, Computer Science TR-1578, University of Maryland, College Park, MD, December.

Shneier, M., 1981: Two Hierarchical Linear Feature Representations: Edge Pyramids and Edge Quadtrees, *Computer Graphics and Image Processing 17*, No. 3(November), 211-224.

Tanimoto, S., and T. Pavlidis, 1975: A Hierarchical Data Structure for Picture Processing, *Computer Graphics and Image Processing 4*, No. 2(June), 104-119.

Uhr, L., 1972: Layered "Recognition Cone" Networks that Preprocess, Classify, and Describe, *IEEE Transactions on Computers 21*, No. 7(July), 758-768.

Warnock, J.L, 1969: A Hidden Surface Algorithm for Computer Generated Half Tone Pictures, Computer Science Department TR 4-15, University of Utah, Salt Lake City, June.

PROCEEDINGS

SECOND INTERNATIONAL SYMPOSIUM ON SPATIAL DATA HANDLING

July 5–10, 1986
Seattle, Washington
U.S.A.