Recent developments in linear quadtree-based geographic information systems

H Samet, C A Shaffer, R C Nelson, Y-G Huang, K Fujimura and A Rosenfeld

The status of an ongoing research effort to develop a geographic information system based on a variant of the linear quadtree is presented. This system uses quadtree encodings for storing area, point and line features. Recent enhancements to the system are presented in detail. This includes a new hierarchical data structure for storing linear features that represents straight lines exactly and permits 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 × 16 384 pixels. Improvements were also made to some basic area map algorithms which yield significant efficiency speed-ups by reducing node accesses. These include windowing, set operations with unaligned images, a polygon expansion 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.

Keywords: image processing, linear quadtrees, geographic information systems

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 the aggregation of pixels thereby 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 to store 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. A recent survey of the use of hierarchical data structures has been presented by Samet[1].

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 (GIS) based on quadtrees. The project has been conducted in four phases. In this paper we describe the current state of this effort, but first the work that has already been completed is reviewed.

In phase I[2], a quadtree database was built from a set of three map overlays representing land use classes, terrain elevation contours, and a floodplain boundary from a region in Northern California. The overlays were hand digitized resulting in three arrays of 400 × 450 pixels. Labels were associated with the pixels in each of the resulting regions, specifying the particular land use class or elevation range. The regions were subsequently embedded within a 512 × 512 grid and quadtree encoded. The results are shown in Figures 2–4.

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

0262–8856/87/03187–11 $03.00 © 1987 Butterworth & Co. (Publishers) Ltd

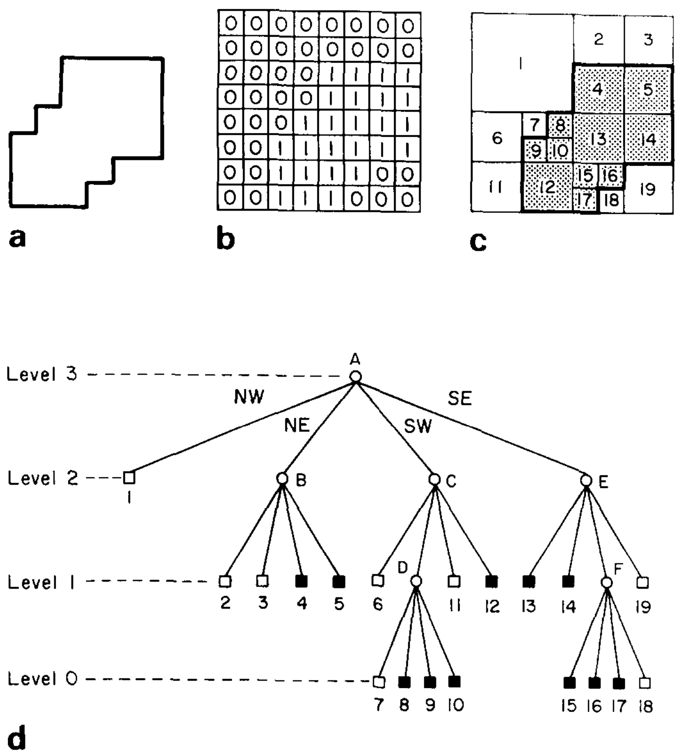


Figure 1. a, region; b, binary array; c, block decompositions of the region; d, quadtree representation of the blocks

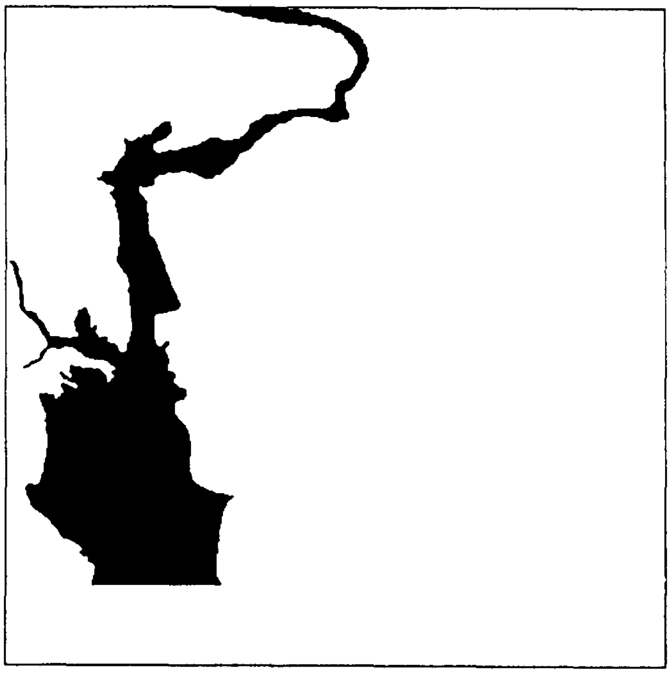


Figure 2. Floodplain map

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[3], a quadtree-based GIS was partially implemented, allowing manipulation of images storing area, point and line data. This implementation included a memory management system to allow manipulation of images too large to fit into main memory, a software

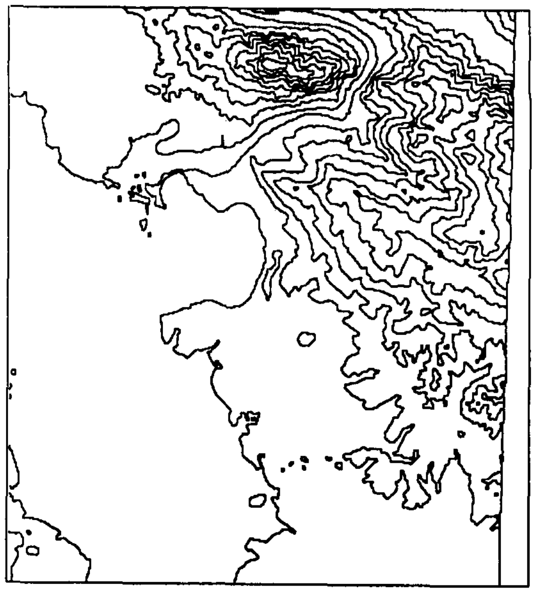


Figure 3. Topography map

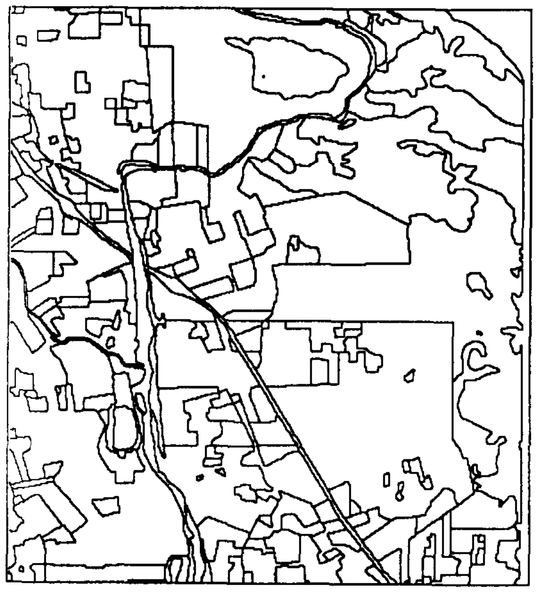


Figure 4. Land use map

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 extended our testbed by extracting point and line data from a geographic survey map of the area used in phase I.

Phase III[4] dealt primarily with enhancements and alterations to the system, an evaluation of some of the design decisions, and the collection of empirical results to indicate the utility of the software as well as to justify the indicated design decisions. Also included was an initial attempt at developing an attribute attachment package for storing non-geographic data associated with geographic objects, and a survey of appropriate point and linear feature data structures for future investigation.

Phase IV[5] of the project provided more extensions to the system. A new structure was developed for storing linear feature data. The attribute attachment package was extended to store attributes of point and linear feature data. Existing area map algorithms were improved to yield significant efficiency speed-ups by reducing node accesses. In this paper we expand further on the developments in phase IV.

QUADTREE MEMORY MANAGEMENT SYSTEM

The quadtree memory management system, described in greater detail in phase II of this project[3], is based on use of a structure termed the 'linear quadtree'[6]. 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 key which is used to order the node list. It is formed by interleaving the bits of the binary representation for the x and y coordinates of the pixel in the upper left corner of the block represented by the leaf node. When sorted in ascending order by 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. To be able to determine the size of the leaf, we must also specify the depth. We use four bits of the 32-bit key to denote the depth. This means that 14 bits are left for each of the x and y coordinates. Thus an image as large as 16 384 × 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. Some implementations also store a pointer to its father. This is in addition to the value field. We have implemented the linear quadtree in conjunction with a disc-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[7] with a page size of 1024 byte, capable of holding up to 120 leaves in a page.

Another consideration for choosing between a pointer-based and a linear quadtree representation is the speed in the execution time of their primitive operations. Current techniques for storing pointer-based trees on disc pages are inefficient in comparison with our linear quadtree/B-tree implementation. However, our characterization of these operations as requiring an explicit disc-based implementation may be criticized on the basis of the availability of a large virtual memory even in relatively small machines. In other words, a large pointer-based quadtree could be maintained in 'core', the operating system controlling the swapping of parts of the tree to disc. This approach trades the ability to explicitly control which pages are swapped for the efficiency of the operating system's swap operation. Empirical tests have shown that when the virtual memory system dominates the amount of time required to perform an operation on the pointer-based quadtree (i.e. for any operation involving large trees), then explicit control of the paging system is much more efficient.

The line representation, described below, may result in associating more than one item of information with a quadtree block. 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 store multiple nodes with the same address field, one node for each item of information that is associated with the block. This is somewhat 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. Functions were written for finding the nth 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.

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 unaligned images. The new algorithms are described briefly below.

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 used to answer 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[4] 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 inserted were small, and were eventually merged to form larger nodes.

The new algorithm is based on a modification of the chessboard distance transform[8]. 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, i.e. the exact distance from the node's centre 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 distance transform calculation is recursively reapplied to each quadrant.

The new algorithm is an improvement over the old one in part because only large white nodes need excessive computation. Since most nodes in a quadtree are small, few nodes generate much work. In addition, there are far fewer duplicate node insertions in the new algorithm. Table 1 contains a comparison of the two algorithms for the floodplain in Figure 2 and the portion of the land use 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 speed-up is more than linear.

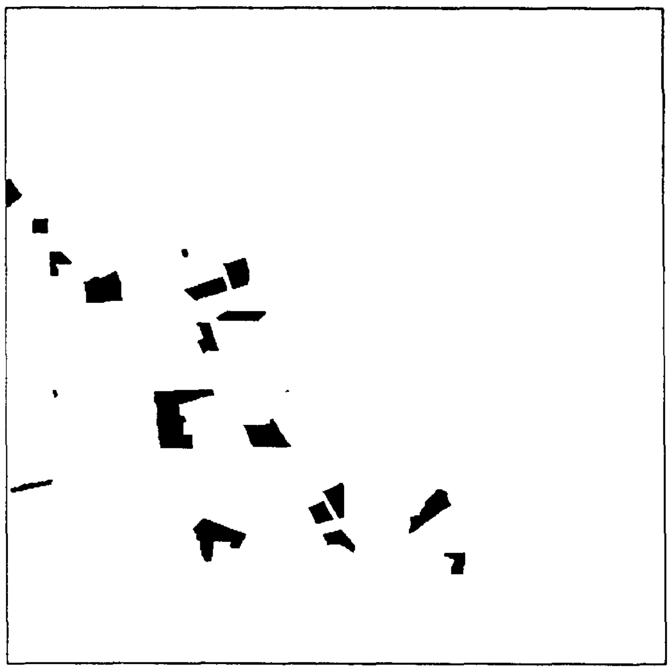


Figure 5. ACC land use class map

An optimal quadtree construction algorithm

The naïve algorithm for converting a raster image to a linear quadtree (or a pointer-based quadtree) is to insert individually 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[9], as well as the one used previously in our system[2], 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 old 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, i.e. the number of nodes in the output tree has

Table 2. Quadtree building algorithm statistics

Map name	No. of nodes	New		Old	
		No. of inserts	Time (s)	No. of inserts	Time (s)
Floodplain	5266	2352	13.8	180000	413.2
Topography	24859	12400	51.2	180000	429.8
Land use	28447	14675	56.9	180000	436.7
Centre	4687	2121	16.1	262144	603.8
Pebble	44950	20770	111.0	262144	630.8
Stone	31969	14612	70.2	262144	629.5

Table 1. Execution times for the Within function

Distance	Flood time (s)		ACC time (s)	
	New algorithm	Old algorithm	New algorithm	Old algorithm
1	16.1	32.9	11.9	15.3
2	21.0	24.1	15.5	12.8
3	19.3	52.9	15.4	27.8
4	23.5	31.5	18.0	19.7
5	35.3	68.9	28.4	39.2
6	37.2	49.4	29.3	31.5
7	29.5	91.1	26.9	53.3
8	30.1	53.3	27.2	36.6
9	44.4	107.5	40.4	63.4
10	43.3	75.8	39.0	48.0
11	57.6	127.3	58.1	87.8
12	50.8	76.6	50.2	53.0
13	69.8	140.8	66.2	86.4
14	66.0	99.4	58.0	67.3
15	57.1	161.4	56.3	106.0
16	44.4	94.6	45.2	68.5

little or no effect on the time required to perform the algorithm. Note that for the old building algorithm the amount of time needed to read the image data is approximately 1% of the time necessary to insert every pixel.

A new and optimal algorithm[10] has been developed that makes a single insertion for each node in the quadtree. 'Optimal' means that in the worst case the number of inserts will be at most the number of nodes in the resulting 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 colour.

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 pixels, 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[10] that the number of active nodes is bounded by $2^n - 1$. 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 colour 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.

To implement the new algorithm, we need to keep track of the list of active nodes. This list is represented by a table, called '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 colour 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, 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,

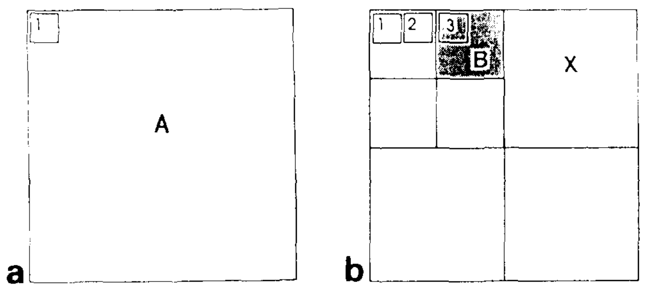


Figure 6. Node insertion can create multiple active nodes. a, node A is active after inserting a single pixel of colour C; b, the first two pixels have colour C; pixel 3 has colour D and its insertion creates active node B; node A is still active

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), Table and List are updated.

Table 2 compares timing results for the new and old algorithms. 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. 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 seven nodes to result. If the first pixel inserted into node X happens to be the same colour as the original node (A of Figure 6a), then no insertion is required.

To understand why the new algorithm is an improvement over the old one, the cost of both algorithms should be analysed in terms of the number of insert operations that they perform. The naïve 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 $2^{2n}.(c + I)$. 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 $c2^{2n} + IN$, where c is relatively small in comparison to I, and N is usually small in comparison to 2^{2n}. In other words, the quantity IN 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 new algorithm is optimal to within a constant factor. Of course, this is achieved at the cost of 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).

Set operations for unaligned 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 30 m in elevation. This can be achieved by intersecting a wheatfield map and an elevation map whose pixel values are non-white if they represent an area whose elevation is above 30 m. The resulting map would have non-white pixels wherever the corresponding pixels of the input maps are both non-white.

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, little work has been done on set operations between unaligned 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 unaligned quadtrees involved translating one of the images to be aligned with the other, and then performing an aligned intersection[11]. In this section, the principles underlying an optimal algorithm for the intersection of unaligned maps are described. 'Optimal' means 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 mode.

As with the quadtree building algorithm above, 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.

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

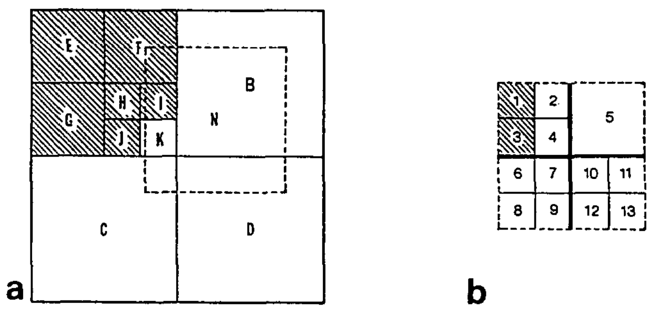


Figure 7. An example of intersection: a, node N (dashed lines) from the first input tree is intersected with an image corresponding to the second input tree and compared against those nodes which it intersects in the second input tree; b, the decomposition and order of processing for node N as directed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

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 unaligned 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[12]. 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.

As an example of how the unaligned intersection algorithm operates, Table 3 shows the contents of the active border tables after processing selected pixels of node N from Figure 7. N is assumed to be I1's first (upper-leftmost) block. In Table 3, records marked with an asterisk indicate a node that has been located in I2 (i.e. a Find operation has been performed). For example, when processing pixel (0,0), no records are initially in the table. The record for node F (the node containing pixel (0,0)) is therefore read into Y__EDGE[0] and X__EDGE[0]. A minus sign in the column marked Y__EDGE (X__EDGE) means that the indicated position in the table did not contain a record covering current pixel (CY,CX) and that the record was copied from X__EDGE (Y__EDGE). A plus sign indicates the record from X__EDGE (Y__EDGE) that was copied to the

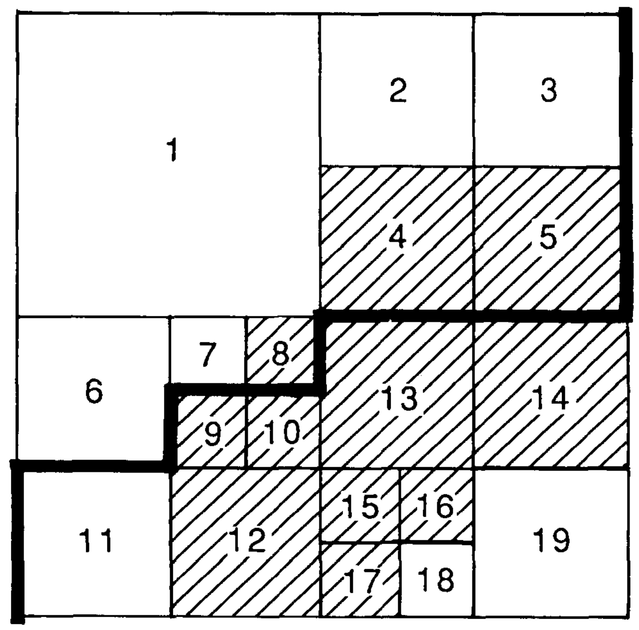


Figure 8. Active border after processing node R in the first input tree I1

Table 3. Trace table for intersection active nodes

Pixel processed	Active node tables	
	Y_EDGE	X_EDGE
(0,0)	0: F*	0: F*
(0,1)	0: B*	0: F
		1: B*
(1,0)	0: B	0: I*
	1: I*	1: B
(1,1)	0: B	0: I
	1: B−	1: B+
(0,2)	0: B+	0: I
	1: B	1: B
		2: B−
(2,0)	0: B	0: K*
	1: B	1: B
	2: K*	2: B
(2,1)	0: B	0: K
	1: B	1: B+
	2: B−	2: B
(3,0)	0: B	0: C*
	1: B	1: B
	2: B	2: B
	3: C*	
(3,1)	0: B	0: C
	1: B	1: D*
	2: B	2: B
	3: D*	
(2,2)	0: B	0: C
	1: B	1: D
	2: B	2: B
	3: D	
(2,3)	0: B	0: C
	1: B	1: D
	2: B+	2: B
	3: D	3: B−
(3,2)	0: B	0: C
	1: B	1: D
	2: B	2: D−
	3: D+	3: B
(3,3)	0: B	0: C
	1: B	1: D
	2: B	2: D
	3: D+	3:D−

* Indicates a node that has been located in I2 (i.e. a Find operation has been performed)
− Indicates position in the table did not contain a record covering current pixel (CY, CX). Record was copied from X_EDGE (Y_EDGE)
+ Indicates the record from X_EDGE (Y_EDGE) that was copied to the position marked with (−) in Y_EDGE (X_EDGE)

position marked with a minus sign in Y_EDGE (X_EDGE) e.g. when processing pixel (2,1) relative to N's origin (labelled as block 7 in Figure 7b), the active border table shown in Table 3 contains the record for block K in Y_EDGE[2] and block B in X_EDGE[1]. Since block B actually corresponds to pixel (2,1), the record for B is copied from X_EDGE[1] to Y_EDGE[2], as indicated by Table 3. Columns two and three are of the format Position:Node where Node indicates the node of I2 being stored at position Position. All coordinates are relative to I1's origin.

Windowing

Interestingly, a variant of the unaligned 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 upper 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 ouput 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 align two images represented by quadtrees. To simplify the following presentation, we will assume a window of size $2^m \times 2^m$ taken from an image of size $2^n \times 2^n$ where $m \leqslant n$.

To see the analogy between windowing and the intersection of two unaligned 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 unaligned 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.

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. Use of a linear quadtree for point and region data is well understood, but this is not the case for vector features. For vector features, a good linear quadtree representation must also have the following three properties. First, straight line segments should be represented exactly (not in a digitized representation). Second, updates must be consistent, i.e. when a vector feature is deleted, the database 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. Third, the structure should allow the efficient performance of primitive operations such as insertion and deletion of vector data elements, and should facilitate the performance of more complex operations such as edge following, intersection with a region and point-in-polygon, though these are somewhat application dependent.

In phase II of this project we implemented a variation of the edge quadtree of Shneier[13], termed '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, as in our area representation. 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.

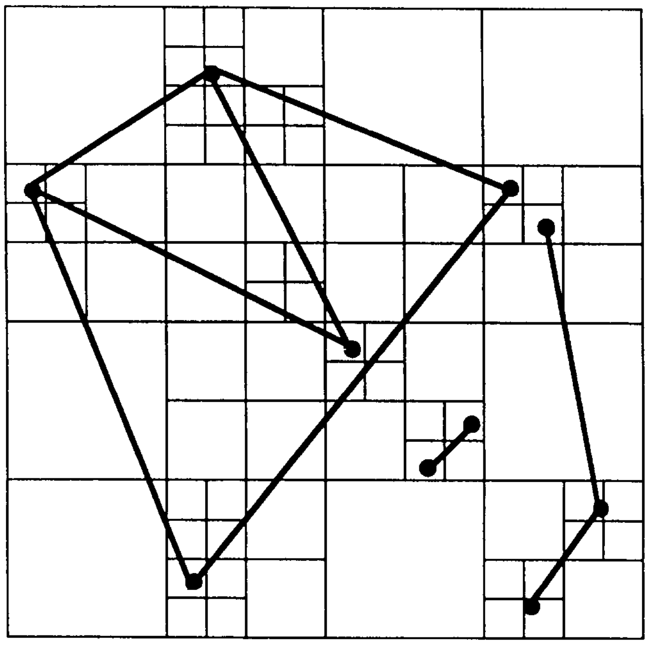


Figure 9. Linear edge quadtree

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, vertices cannot be distinguished from short line segments. Moreover, boundary following and deletion of line segments cannot be handled properly in the vicinity of a vertex at which several edges meet.

To overcome these shortcomings we developed a new representation, termed a 'PMR quadtree', which is a variation on the PM quadtree[14]. The PMR quadtree makes use of probabilistic splitting and merging rules, one for splitting and one for merging, to organize the data dynamically. 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 (four 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 distinct line segments. The merge operation can be performed more than once. This scheme differs from our 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. Figure 10 shows an example of PMR quadtree construction.

The PMR quadtree makes use of variable-size nodes (the implementation of variable-size nodes in our memory management system was described above). 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 practical an implementation of the node as a list. In an application where this is not the case, other splitting rules can ensure that the number of segments in a node

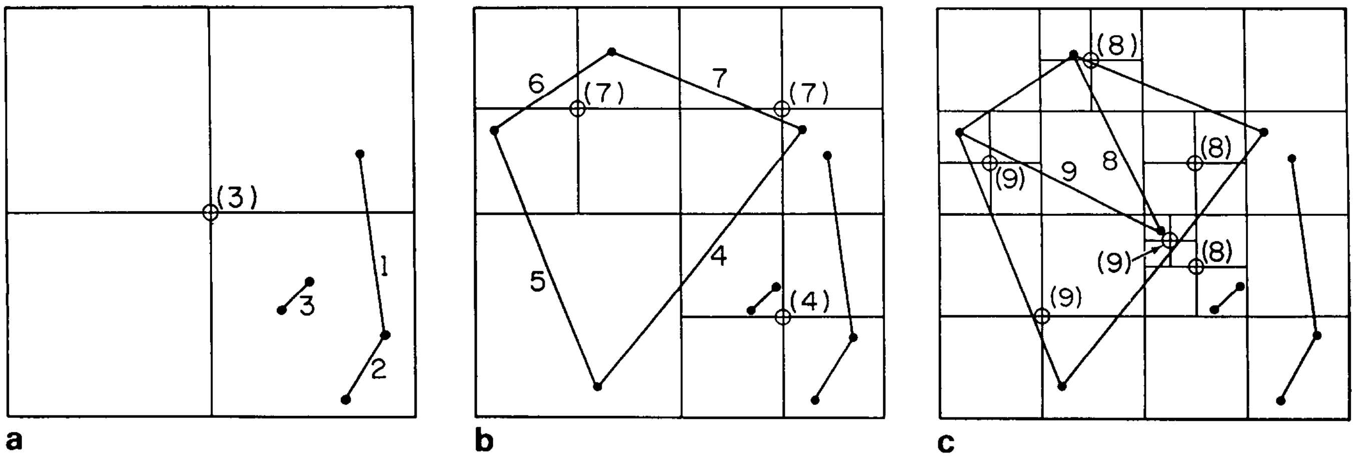


Figure 10. Building a PMR quadtree from the segments of Figure 8 with threshold =2: a, three segments have been inserted causing the plane to be quartered once as indicated by the small circle; b, segments 1–7 inserted causing three blocks to split; c, segments 8 and 9 inserted causing five more blocks to split

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, as described above.

The PMR quadtree induces a decomposition of the space that may split a line segment into many portions. Each portion that lies within 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 represent accurately 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.

The problem of how to represent a line segment that has been broken into fragments must also be addressed. 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 11). 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 with 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 be rejoined reliably. 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 statement. This is likely to be a 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 the q-edges 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 11 is shown in Figure 12.

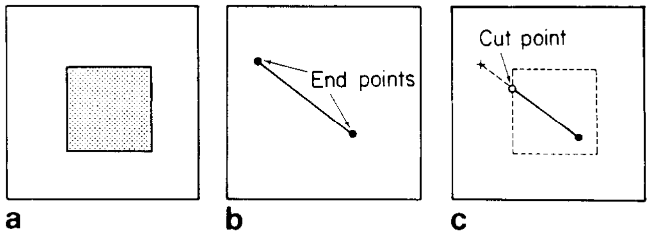
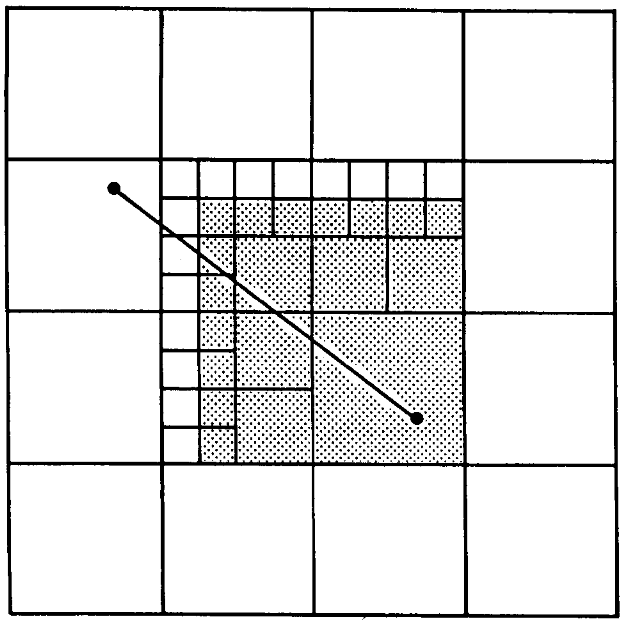


Figure 11. Intersection of: a, a region; b, a line segment, producing c, a fragment with one cut point

Properly dealing with entities such as cut points and fragments requires that the splitting and merging rules of the PMR quadtree are modified 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.

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 13. This network has 684 vertices and 764 edges. The three data structures used in the comparison are the MX quadtree[15], the edge quadtree, and the PM_3 quadtree[14]. 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. 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 q-nodes in the table and meaningless for the MX and edge quadtrees). 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 quad-

a


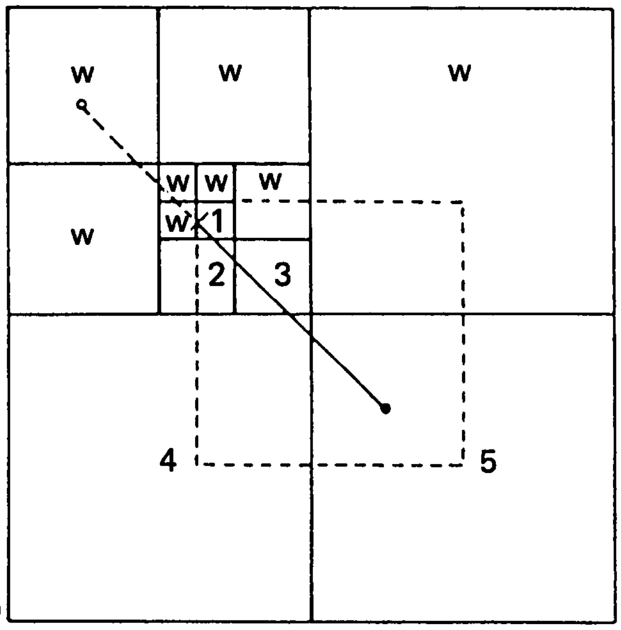
b


Figure 12. 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

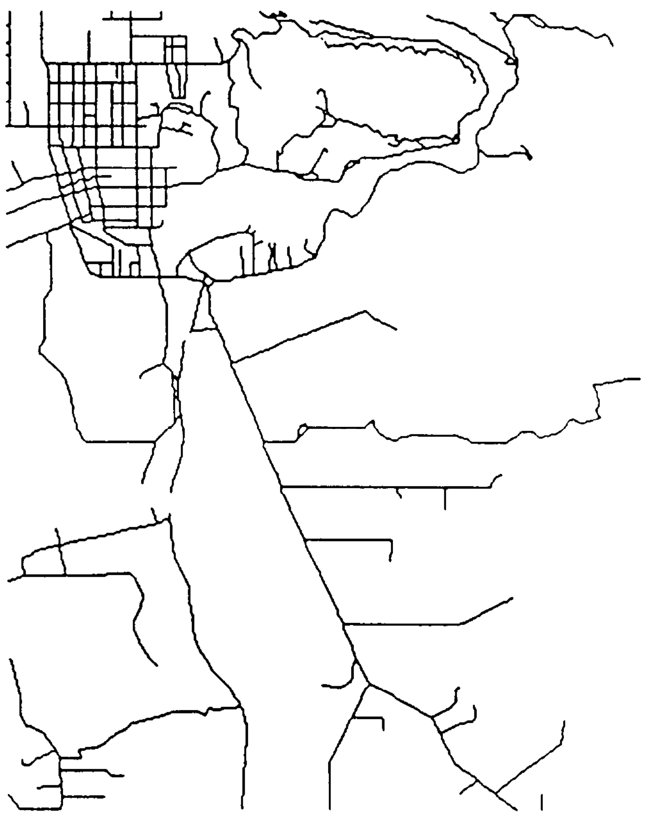


Figure 13. Road network

Table 4. Building times and sizes for the road network

Type of quadtree	Time (s)	Leaves	Q-nodes
MX	31.5	19699	–
Edge	27.4	7723	–
PM_3	39.0	3939	2350
PMR	25.8	2078	874

tree will not be as deep as the PM_3 quadtree, nor as deep as the MX and edge quadtrees.

We also compare 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, depending on the complexity of the maps, the execution times for the PMR quadtree were slower than those for the PM_3 representation by factors of up to 50%. 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-against-space trade-off between the two representations.

CONCLUDING REMARKS

Our goal was to demonstrate the utility of hierarchical data structures for use in the domain of geographic information systems. 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 and faster quadtree memory management systems.

ACKNOWLEDGEMENT

The support of the US Army Engineer Topographic Laboratories under contract DAAK-70-81-C-0059, and of the US National Science Foundation under grant DCR-86-05557, is gratefully acknowledged.

REFERENCES

1 **Samet, H** 'The quadtree and related hierarchical data structures' *ACM Comput. Surveys* Vol 16 No 2 (June 1984) pp 187–260

2 **Rosenfeld, A, Samet H, Shaffer C and Webber R E** 'Application of hierarchical data structures to geographical information systems' *Computer Science TR-1197, University of Maryland, MD, USA* (June 1982); see also *IEEE Trans. Syst. Man Cybernetics* Vol 13 No 6 (November/December 1983) pp 1148–1154

3 **Rosenfeld, A, Samet H, Shaffer C and Webber R E** 'Application of hierarchical data structures to geographical information systems phase II' *Computer Science TR-1327, University of Maryland, MD, USA* (September 1983); see also *Patt. Recogn.* Vol 17 No 6 (November/December 1984) pp 647–656

4 **Samet, H, Rosenfeld A, Shaffer, C A, Nelson, R C and Huang, Y G** 'Application of hierarchical data structures to geographical information systems phase III' *Computer Science TR-1457, University of Maryland, MD USA* (November 1984)

5 **Samet, H, Rosenfeld, A, Shaffer, C A, Nelson, R C, Huang, Y G and Fujimura, K** 'Application of hierarchical data structures to geographical information systems phase IV' *Computer Science TR-1578, University of Maryland, MD, USA* (December 1985)

6 **Gargantini, I** 'An effective way to represent quadtrees' *Comm. ACM* Vol 25 No 12 (December 1982) pp 905–910

7 **Comer, D** 'The ubiquitous B-tree' *ACM Comput. Surveys* Vol 11 No 2 (June 1979) pp 121–137

8 **Samet, H** 'Distance transform for images represented by quadtrees' *IEEE Trans. PAMI* Vol 4 No 3 (May 1982) pp 298-303

9 **Samet, H** 'An algorithm for converting rasters to quadtrees' *IEEE Trans. PAMI* Vol 3 No 11 (January 1981) pp 93–95

10 **Shaffer, C A and Samet, H** 'Optimal quadtree construction algorithms' *Comput. Vision, Graph. Image Process.* Vol 37 No 3 (March 1987) pp 402–419

11 **Gargantini, I** 'Translation, rotation, and superposition of linear quadtrees' *Int. J. Man-Machine Studies* Vol 18 No 3 (March 1983) pp 253–263

12 **Samet, H and Tamminen, M** 'Computing geometric properties of images represented by linear quadtrees' *IEEE Trans. PAMI* Vol 7 No 2 (March 1985) pp 229–240

13 **Shneier, M** 'Two hierarchical linear feature representations: edge pyramids and edge quadtrees' *Comput. Vision, Graph. Image Process.* Vol 17 No 3 (November 1981) pp 211–224

14 **Samet, H and Webber, R E** 'Storing a collection of polygons using quadtrees' *ACM Trans. Graphics* Vol 4 No 3 (July 1985) pp 182–222

15 **Hunter, G M and Steiglitz, K** 'Operations on images using quad trees' *IEEE Trans. PAMI* Vol 1 No 2 (April 1979) pp 145–153