INT. J. GEOGRAPHICAL INFORMATION SYSTEMS, 1990, VOL. 4, NO. 2, 103–131

QUILT: a geographic information system based on quadtrees†

CLIFFORD A. SHAFFER

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061, U.S.A.

HANAN SAMET‡

Center for Automation Research and Computer Science Department and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742, U.S.A.

and RANDAL C. NELSON

Computer Science Department, University of Rochester, Rochester, New York 14627, U.S.A.

Abstract. This paper describes QUILT, a prototype geographic information system (GIS) that uses the quadtree data structure as the underlying representation for cartographic data. While QUILT contains many features typically available in a GIS, its primary purpose is to serve as a testbed for the design and testing of new data structures and algorithms for use in computer cartography. Quadtree variants for region, point and line data are implemented using the linear quadtree, organized on disk by a B-tree. QUILT provides a simple attribute attachment system which associates non-spatial data with geographic objects. The user views QUILT as an augmented LISP environment. QUILT's geographic functions include conversion of rasters to and from quadtrees; subset operations to select specified geographic objects; map editing, display, windowing, intersection and union operations; polygon expansion; and computation of geographic object properties such as the centroid, area, perimeter and bounding rectangle for sets of geographic objects.

1. Introduction

The quadtree data structure (Samet 1989, 1990) has been the subject of much research over the past several years. By 1982, numerous algorithms had been developed for constructing compact quadtree representations, converting to and from representations, computing region properties from quadtrees and computing the quadtree representations for Boolean combinations of regions. With this background, it seemed that quadtrees could be ideal for representing maps in a geographic information system (GIS).

This report describes QUILT, a prototype GIS which uses quadtrees as the underlying representation for cartographic data. While it contains many common GIS features, QUILT's primary purpose is as a testbed for new data structures and algorithms for computer cartography. The first, and one of the most complete, of the

† The QUILT software package is ©1985 by the University of Maryland. Franz LISP ©1980 by the Regents of the University of California. UNIX is a trademark of AT&T Bell Laboratories. VAX is a trademark of Digital Equipment Corporation. SUN is a trademark of SUN Microsystems, Inc.

‡ Queries about availability of the QUILT system should be directed to Hanan Samet at the address listed or by electronic mail to hjs@alv.umd.edu.

0269–3798/90

quadtree-based GIS, it has been distributed to many researchers in the U.S.A. and in Europe. The system as described here was essentially completed in early 1986. QUILT is capable of storing region, point and linear feature data. It was originally implemented on a VAX 11/780 running the UNIX BSD operating system and has since been ported to SUN workstations.

QUILT can be separated into four conceptual levels, as illustrated by figure 1. The lowest level, known as the kernel, controls the interface between the disk file used to store the quadtree structure and the programs which manipulate quadtrees. The kernel is written in the C programming language and is described in § 2. The second level, also written in C, contains programs which implement the quadtree algorithms, e.g., set functions, area and perimeter calculations, windowing. The most innovative of these algorithms are described in § 5. Also included in the second level are programs to manipulate the point and line implementations described in §§ 3 and 4 respectively. The third level manipulates logical objects in the database. This level is implemented in LISP since this language is better suited to the manipulation of symbolic information than C. Specifically, the third level contains the attribute database functions described in § 6, and also keeps track of the maps and geographic objects currently known to the system. The top level of QUILT is made up of those LISP functions which comprise the database query language. The user can build complicated queries through combinations of the available functions. Examples of such queries can be found in § 7.

2. The quadtree kernel

Each map in the QUILT system is stored in a separate disk file. A 'map' may contain one thematic type (e.g., land-use class values, topographic contours), a collection of point objects, or a collection of line segments. The memory management system, or kernel, controls the interface between the quadtree structure stored on disk and the GIS. To the kernel, the quadtree structures used for storing area, point and line data are identical. The kernel organizes a quadtree as a sorted list of leaf nodes as described below. Since all routines in QUILT access the disk file solely through the kernel functions, the kernel can be viewed as defining the quadtree as an abstract data type. This is particularly useful to QUILT in its role as an environment for studying data structures. Variant implementations of the kernel may be tested without altering the rest of the system (and vice versa).

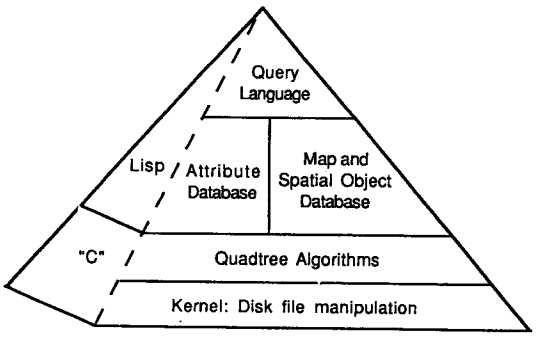


Figure 1. The QUILT software structure.

The region quadtree is usually viewed as a tree of out-degree four which divides a $2^n \times 2^n$ image into equal sized quadrants, subquadrants, and so on, until homogeneous blocks (possibly pixels) are obtained. For example, consider the region shown in figure 2(*a*), which is represented by a $2^3 \times 2^3$ binary array in figure 2(*b*). The blocks resulting from the decomposition of the region quadtree are shown in figure 2(*c*), and the tree in figure 2(*d*).

For many GIS applications, the maps may be so large that the space requirements of their quadtree representation exceed the amount of memory that is available. Therefore, it is necessary to store the map on disk with portions of the data processed in main memory as needed. The quadtree's pointer-based representation is inconvenient for this purpose, since there may be little relationship between the proximity of nodes in the quadtree and their proximity on the disk. This can lead to an intolerable number of disk accesses when manipulating the tree. The linear quadtree (Gargantini 1982, Abel and Smith 1983) has thus become popular since it partially alleviates this problem.

Encoding schemes for linear quadtrees are rooted in the following two observations. First, the method of decomposition used by the region quadtree to break an image into blocks restricts both the position and size of quadtree leaf nodes; furthermore, these blocks must be disjointed and cover the entire image. All that is required to reconstruct the image is a list containing the size, location and value for each of the blocks comprising the image. The second observation is that each block on the list can be

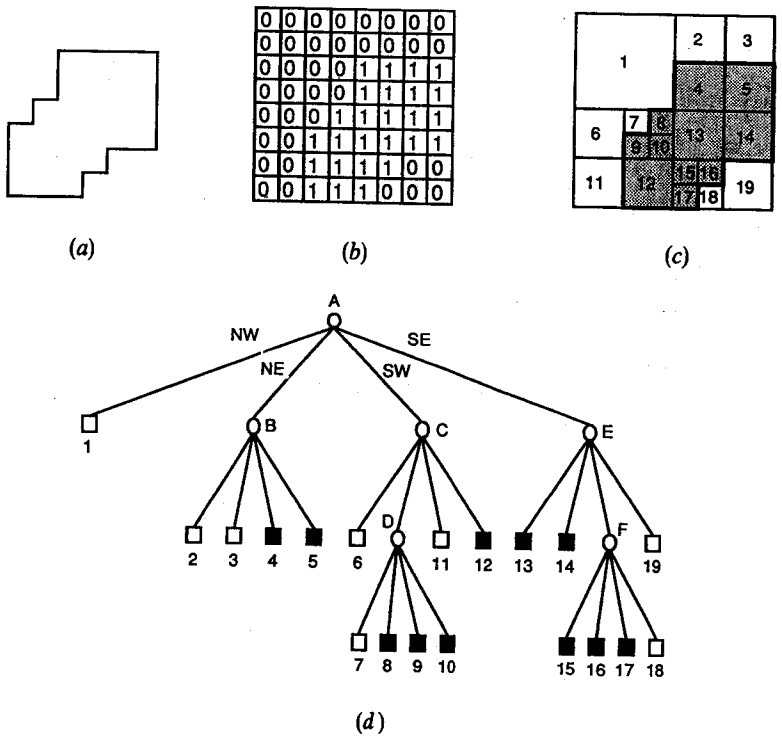


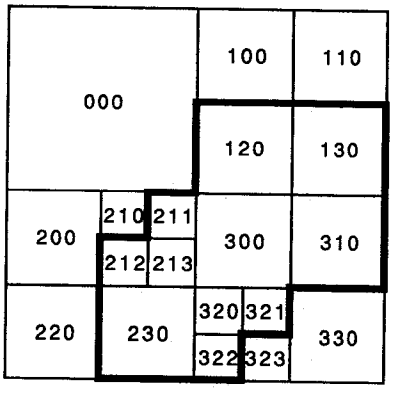
Figure 2. A region, its binary array, its maximal blocks and the corresponding quadtree: (*a*) region, (*b*) binary array and (*c*) block decomposition of the region in (*a*). Blocks in the region are shaded. (*d*) Quadtree representation of the blocks in (*c*).

given a key or address value that, when sorted in ascending order, arranges the blocks in the same order in which the pointer-based quadtree's leaf nodes would be visited by a depth-first traversal of the quadtree.

When constructing a linear quadtree, we assume that every pixel in the underlying array of the digitized image has been assigned an address value (i.e., its locational code). The addressing schemes commonly used are variations on one suggested by Morton (1966) and are thus sometimes referred to as Morton addressing. Morton addresses are created by interleaving the bits of the binary representation for that pixel's x and y coordinates (each represented by a fixed number of digits). For example, figure 3 (*a*) shows the 3-bit binary representation for the row and column coordinates along the top and left sides of an 8×8 array. Each pixel's address is formed by bit interleaving such that the y bit precedes the x bit at each position. These pixel addresses are shown represented by base 4 digits (i.e., each x and y bit pair corresponds to a single base 4

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

(*a*)



(*b*)

Figure 3. (*a*) The Morton code addressing scheme for labelling pixels. (*b*) The Morton code addresses for the blocks of figure 2 (*c*).

digit). When the pixel's addresses are sorted in increasing order, the result is equivalent to a depth-first traversal. (In this report, all quadtrees are assumed to be traversed in order NW, NE, SW, SE; the origin of each image is assumed to be (0,0) appearing at its upper left corner; and pixel coordinates are referred to by (ROW, COL)). Visiting the nodes of the linear quadtree in order of their address values will be referred to as a Morton order traversal. From this method of generating pixel addresses, we can in turn generate node addresses by assigning to each node the address of the least-valued pixel contained within the block that it represents. Figure 3(*b*) shows figure 2(*c*)'s block decomposition with each block given the Morton address of the least-valued pixel contained within block. Note that the block in the NW quadrant of the image in figure 3(*b*) has a 0 value in the first position (meaning that a NW branch was taken to reach that node); all blocks in the NE quadrant have a 1 in the first position, those in the SE quadrant a 2 and and those in the SW quadrant a 3.

In the scheme described thus far, each block has been given a pixel address with no indication of its size (i.e., depth in the tree). Two basic schemes have been proposed to indicate the size. One uses a different symbol, say X, to represent 'don't care' positions at the end of a node address with a fixed number of (base-5) digits. The second solution is to have an address with a variable number of (base-4) digits, and an additional value storing the depth. Alternatively, aspects of these methods can be combined, and the depth value can define how many (base-4) digits of a fixed-length address are to be considered as significant. QUILT adopts this latter representation, storing the address value for each node as separate fixed-length address and depth fields within a single 32-bit word.

The set of pixels covered by each block in the node list will have locational codes numbered consecutively from the block's address value to one less than the address value of the next block in the list. Thus, to find the block containing a specified pixel, it is only necessary to generate the pixel's address by interleaving the x and y coordinates, then search the list for the greatest address less than or equal to that of the query. Node splitting is achieved by locating the desired block and replacing it with its four children (which will be consecutive in the node list). Likewise, to merge four siblings, the corresponding four (consecutive) blocks must be located and replaced with their father.

Given the sorted list of quadtree blocks, some means must be found to organize it so that insertions, deletions, node searches and traversals can be performed efficiently. In addition, it is important that the organization method lend itself to off-line storage of large images. A classic technique for storing large sorted lists is the B-tree (Comer 1979). B-trees are efficient in that the number of accesses necessary to retrieve a given key from secondary storage is kept low. This is partly because the tree is always balanced, and partly because the branching factor is very high. Both Abel (1984) and Samet *et al.* (1984) have proposed using linear quadtree encoding in conjunction with B^+-trees to store images.

QUILT's linear quadtree nodes are composed of two fields: the address field and the value field, each 32 bits long. When storing area data in region quadtrees, the value field contains the leaf's attribute class (see §6), and all pixels within the block are homogeneous. Not all pixels contained within a leaf node of our point and line representations are homogeneous. Thus, other interpretations are placed on the information stored in value portion of the leaf description. The interpretation made by a particular database operation of a leaf's value is dependent on what type of data is being stored in the quadtree. The user keeps track of the data type in the user's header (discussed below). The address field is divided into two subfields. The most significant

28 bits contain the node's Morton code address, obtained by bit-interleaving the x and y coordinates of the upper left corner of the block represented by that node. The remaining 4 bits contain the depth at which the node would appear in the corresponding pointer-based tree (from which its width may be deduced). This depth value also defines how many significant bit-pairs are contained in the Morton code subfield. With a 28-bit code subfield, quadtrees of up to 14 levels with a resolution of 16 384 × 16 384 pixels can be stored. Larger images would require a longer address field.

The B^+-tree is implemented using pages of 1024 8-bit bytes; each page is large enough to hold 120 quadtree (or internal B-tree page) nodes with some additional room for the overhead required to maintain the B-tree. This overhead includes the number of nodes currently located on the page; whether the page is an internal or external node of the B-tree; and the father, in the B-tree, of the page. This page size was selected since our version of UNIX fetches data in blocks of 1024 bytes at each read operation. Each internal B-tree page node contains a pointer to a child page in its data field, with the value of the smallest key from the child page stored in the address field.

Each application function in the system (e.g., intersection or perimeter) makes use of a block of main memory space large enough to hold several B-tree pages. This space forms a buffer pool for storing currently active B-tree pages. When a page is requested, the buffer pool is first searched to determine if it is already in main memory. If the page is not located within the buffer pool, the least recently used page is copied back to the disk (if it has been modified) and replaced with the requested page.

Our quadtree files have four parts: a fixed-length structure containing information needed by the kernel, a fixed-length structure of user-defined information, a list of comments and a list of quadtree leaves. The block of information needed by the kernel, known as the kernel's header, contains information such as the location of the root of the B-tree and the length of the comments list. The user-defined information, known as the user's header, is initialized when the quadtree file is created. The kernel supports the user's header with routines for reading and writing. Typical information stored in the user's header would be the data type (area, point or line), the x and y coordinates of the upper-left corner of the map (with respect to some global coordinate system), and the width and height of the actual map stored within the $2^n \times 2^n$ block represented by the quadtree. The list of comments provides a variable-size disk area where the database user can place text information (by default this includes a list of the function calls used in the creation of the file). Kernel functions are provided to initialize the comment list, read the comments and add comments.

Most of the kernel's code is dedicated to maintaining the list of quadtree leaves. The database system accesses this code through a set of C subroutines and macros. The major distinction between these routines is whether they access the disk file or are utilities for manipulating descriptors of quadtree nodes. Most of these functions return a pointer to a descriptor, thus encouraging the use of functional composition to perform multiple operations conveniently. Functions to manipulate descriptors are provided which do the following: determine the value, depth, x, and y coordinates of the descriptor; create a descriptor by setting its depth, x and y coordinates; set the value of a descriptor; generate the father or son of a descriptor; clear the contents of a descriptor; and copy the contents of a descriptor.

Routines that access the quadtree disk file can be divided into those that query the file and those that modify the file. The most important function that queries the file is one which, given a node descriptor, returns the descriptor of the actual leaf in the quadtree that contains the given key. Sometimes the query leaf covers several leaves in

the actual image; there is no single 'containing' leaf. In this case, the function returns the actual leaf in the quadtree containing the query key (i.e., the pixel in the upper left corner of the query leaf). The Morton order successor of a node, as well as a node's neighbour in a given direction, can also be located. Often we want to apply some function to every leaf in a quadtree. One way to implement this would be to have the function contained in the body of a loop which generates the Morton order successor of the current node at each loop cycle until all nodes have been visited. This method is relatively slow, as the address of the successor of a given leaf address is first calculated and then a search is made for this leaf from the root of the tree. Since we know that we wish to visit each leaf in order, we can take advantage of the B^+-tree structure to find the next leaf in the list. A special routine is provided which takes a function as its argument and applies this function to every leaf in the file in the most efficent manner.

The principal way to modify a quadtree file is to insert a leaf. If the leaf to be inserted is identical to one that is already in the quadtree except for value, then the value in the file is changed. If, as a result, four siblings have the same value, then they are merged. If the leaf to be inserted is represented by more than one leaf already in the quadtree, then the actual leaves are deleted and the new leaf inserted in their place. Thus we can empty a quadtree by inserting a leaf at the root. If a leaf is inserted that is smaller than the leaf actually in the tree at that position, then the existing leaf is split into its four sons (each with the same value as the father) and the insertaion attempt is repeated.

Finally, two routines are needed to monitor the relation between the external operating system, in which the quadtree files persist across many invocations of the database system, and the kernel, whose memory vanishes when the database system exits. The first routine allocates and initializes a portion of main memory that can be used by the routines that manipulate a particular quadtree file. If a new tree is to be created, then appropriate default values are set up. The second routine frees the main memory area that was associated with a quadtree file, after writing to disk all updated B-tree pages remaining in main memory.

3. Representation of point features

The data structure that we have adopted to store point data is the PR quadtree (Orenstein 1982, Samet 1989) (short for point-region quadtree). When stored in the PR quadtree, an image corresponding to a collection of points is decomposed into four equal quadrants if it contains more than point. These are in turn split into subquadrants, sub-subquadrants and so on, until each block contains at most one point. Figure 4 shows the PR quadtree's decomposition for a simple data set.

Our goal is for point and area data structures to appear identical to the kernel. This means that data stored with each node of the point structure must be represented in the 32-bit value field. Since the PR quadtree allows at most one point per quadtree node, it is quite easy to adapt our area representation to point data. The routines that manipulate point data view the 32-bit value field as containing either an index into a table listing the x and y coordinates for all points in the map in the case of data nodes, or a unique value for WHITE in the case of empty nodes. Associated with each point at the LISP level is additional user-specified information such as the name and the attribute class (see § 6) of the point.

Insertion of a point into the PR quadtree works as follows. First, the leaf containing the point's Morton code address is found. If this leaf is WHITE, then the point's index is stored in the leaf's value field. Otherwise, the leaf is split into four sons, the old data

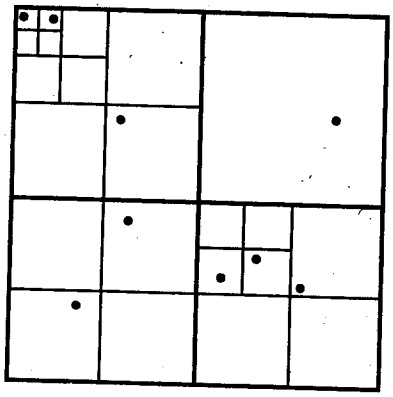


Figure 4. The PR quadtree.

point value is copied to the appropriate son, and insertion is re-attempted. The deletion operation first finds the leaf containing the point and sets the value field to WHITE. Next, the values of the node's siblings are examined. If these siblings contain a single data point between them, all four are merged to form a single node. This process continues until no more merging can take place.

An example data set is shown in figure 5. Each point represents a single house from our test area in the Russian River valley. Functions for querying individual points, finding all the points within a given distance of a given point, and finding all the points intersecting an area map have been implemented.

4. Representation of linear features

Representing linear feature data (also referred to as vector data, and typically represented by line segments) has proven to be a much harder task than representing either area or point data. Several hierarchical structures based on quadtrees have been proposed, including the MX quadtree (Hunter 1978), the line quadtree (Samet and Webber 1984) and the edge quadtree (Shneier 1981). Each of these representations has certain drawbacks, and none has the natural elegance of the adaptations representing points and regions.

For our application, we have determined that a vector representation should have the following properties. First, vectors must be represented precisely rather than as digital approximations. This includes the ability to represent accurately any number of vectors intersecting at a single point. Second, the implementation must allow the data to be updated consistently. For example, insertion and subsequent deletion of a vector should leave the data unchanged. As a more complex example, it should be possible to compute the intersection of a set of vectors with a region and then restore the information to its original state by performing a union with the complement of the original intersection. This operation involves splitting and reassembling parts of vectors. Third, the implementation 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.

The structure used in QUILT to represent linear feature data is referred to as the PMR quadtree (Nelson and Samet 1986 a). It is a variant of the PM quadtree family of representations (Samet and Webber 1985). Each PM quadtree variant uses a

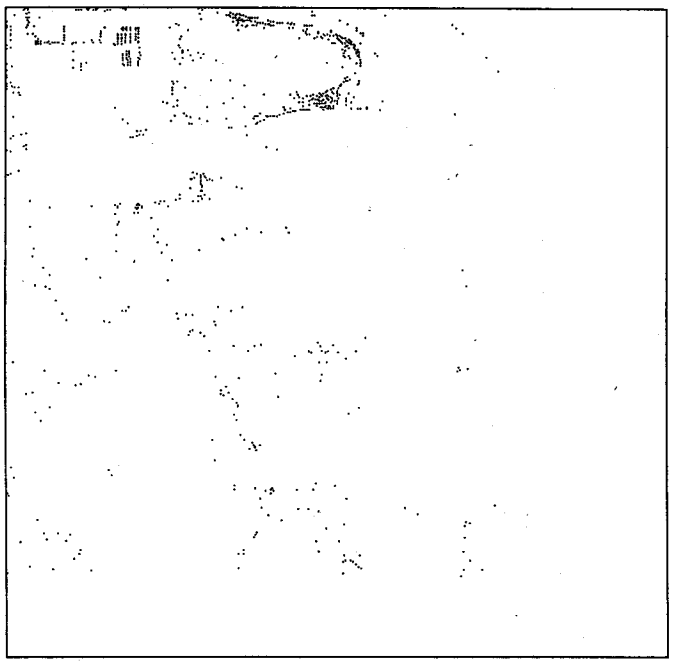


Figure 5. An example of a point data set: house locations from the Russian River valley.

decomposition rule to partition the plane into quadrants recursively, and store with each block all segments passing through it. The PMR quadtree (for PM Random) is based on the observation that any rule that divides the line segments among quadtree blocks in a reasonably uniform fashion can be used as the basis for a PM-like quadtree. In fact, unless it is required by the application, the structure need not be uniquely determined by the data. Probabilistic splitting rules can be used as easily as any other. Note that this does not violate the criteria previously described for an acceptable line representation since, while the structure of the PMR quadtree may change as a result of insertion and subsequent deletion of a line segment, the information content is maintained intact. A detailed description of the PMR quadtree along with comparisons to other quadtree-based linear feature representations can be found in Nelson and Samet (1986 a).

The PMR quadtree uses a pair of rules, one for splitting and one for merging, to organize the data dynamically. A node is split once whenever a line segment is added to a node containing n or more segments (in QUILT, $n=4$). The corresponding merging rule merges the siblings of a node from which a line segment has been removed whenever the number of distinct line segments contained in the node and its siblings is less than, or equal to, n (4). Figure 6 shows the construction of a PMR quadtree with the threshold n equal to two. Note in figure 6 (*b*) that the insertion of segment 7 causes two blocks to split since the capacity of each of these blocks was exceeded by the insertion of segment 7. Since a node is split only once when the insertion of a segment causes the threshold to be exceeded, a resulting node may contain more than n segments. However, except in the unusual case where many segments share an endpoint, the node occupancy is unlikely ever to exceed the threshold by much. This scheme differs from the other PM quadtrees in that the PMR quadtree for a given data set is not unique but depends on the history of manipulations applied to the structure. Certain types of

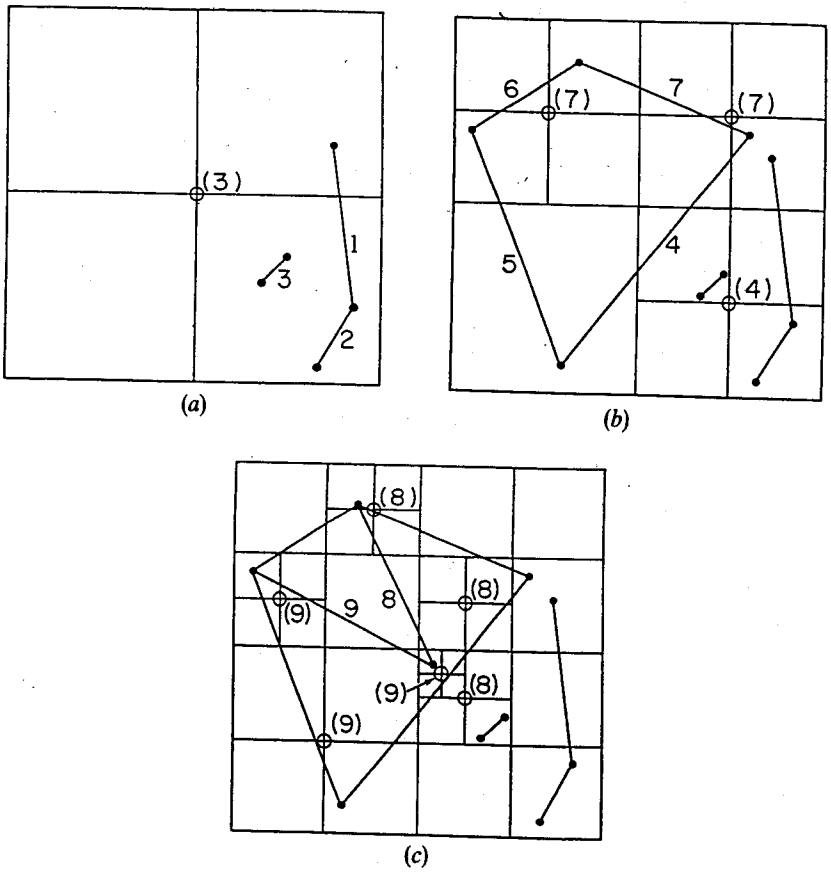


Figure 6. Building a PMR quadtree with node threshold equal to two: (*a*) three segments have been inserted causing the plane to be quartered once as indicated by the small circle; (*b*) segments 4-7 have been inserted causing three blocks to split; (*c*) segments 8 and 9 have inserted causing five more blocks to split.

analysis are thus more difficult than with uniquely determined structures. On the other hand, this structure permits the decomposition of space to be based on local information and is easily implemented. The PMR quadtree was chosen for these reasons.

We now address the problem of how to clip a segment in such a way that the operation may be reversed without data degradation should the missing portion be reinserted. In QUILT, segment truncation arises 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 by clipped. If an intermediate endpoint is introduced to produce new segments (without increasing the resolution of the endpoint representation), then the original information is degraded and the pieces cannot be rejoined reliably.

An alternative solution is to retain the description of the original segment and use the spatial properties of the quadtree to specify what portions of the segment are actually present. The underlying insight is that a node may contain a reference to a segment, even though the entire segment is not present as a lineal feature. Rather, the segment descriptor contained in a node can be interpreted as implying the presence of

just that portion of the segment which intersects the corresponding quadtree block. Such an intersection of a segment with a block will be referred to as a q-edge, and the original segment will be referred to as the parent segment. The presence or absence in the quadtree of a particular q-edge is completely independent of the presence or absence of q-edges representing other parts of the parent segment. Thus lineal features corresponding to partial segments (i.e., fragments) can be represented simply by inserting the appropriate collection of q-edges. Since the original descriptors are retained, a lineal feature can be broken into pieces and rejoined without loss or degradation of information. In PMR quadtree structure, q-edges are combined to represent arbitrary fragments of line segments. Since they 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 main obstacle to use of the PMR quadtree in QUILT is the implementation of variable-sized nodes. Since the number of line segments in a node is potentially unbounded, a true variable-length storage scheme must be used. One property of the PMR quadtree is that, although the maximum number of q-edges occurring in a node is potentially unbounded, the average occupancy remains low. For random vectors, it can be shown that the expected value is less than n if $n > 1$(Nelson and Samet 1986 b). In our empirical tests, with $n = 4$, the average occupancy remained less than 3 in all cases. The low average occupancy makes practical a linear search through the q-edges of a node. For quadtrees, the simplest way of implementing variable (logical) node sizes is to store a separate physical node for each line segment, where each physical node making up a part of the logical node contains a duplicate locational code. This is the method used by QUILT.

Each line segment stored in the PMR quadtree has an associated record in the segment table. The segment table is implemented at the C programming language level. It is organized as an array of records, with each record storing the endpoints and attribute class (see § 6) of the associated line segment. The q-edges that intersect a node are represented by the index in the segment table associated with the parent segment. All q-edges with the same parent share this descriptor, thereby avoiding unnecessary duplication of information. Figure 7 shows two linear feature data sets used in our tests.

5. QUILT functions

From the user's point of view, QUILT is structured into three 'modes'. Most functions are in the standard or default mode. Querying the help function will describe all functions of this mode. The second mode is named edit-map. The functions associated with this mode are described in § 5.1. A user may not accesss any of these functions without first invoking the edit function. Upon entering edit-map mode, the user may not access functions belonging to the other modes until the editing session is finished. This feature is designed to protect the user from damaging a partially-edited map. A query to the function while in edit-map mode will describe only edit-map mode functions.

The third mode is named edit-table. This mode is used to alter an attribute table; the associated functions are described in § 6. Once again, these functions are available to the user only when he has indicated that he wishes to edit an attribute table. When in edit-table mode, only edit-table mode functions are available. Tables 1, 2 and 3 list the QUILT functions available to each mode.

The user's view of the database system is an augmented version of Franz LISP (Foderaro 1980). LISP allows for a flexible query language based on the composition of

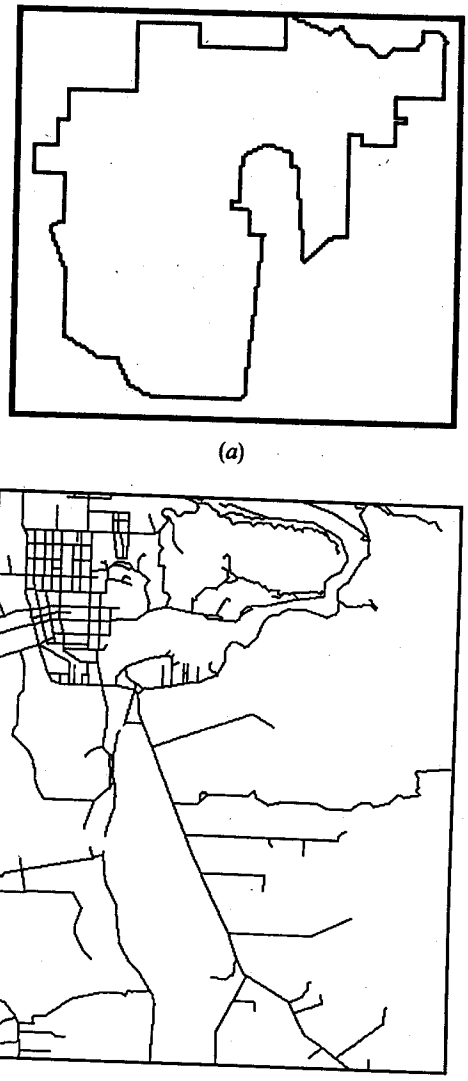


Figure 7. Two linear feature data sets: (*a*) city border map and (*b*) road map.

functions. Most functions in the system yield an object which may be used as the input for other functions. For example, a function which yields a map may be used in place of a map parameter for a second function (e.g., taking the area of the map created by windowing an original input map). Thus, composition of functions plays an important role in our query language. The naive user needs no knowledge of LISP to use the database system; he needs only the names and required parameters of the database functions. For the experienced user, our implementation allows access to a powerful programming language. Queries can easily be generated using the list processing functions of LISP which calculate properties over several maps. Future users will be able to extend the present system easily by writing 'intelligent' queries; as an example, such queries could generate formatted tables of data from a set of maps.

Table 1. Database functions in standard mode

Name	Description
area	Return the area of a map
binary	Change all non-WHITE class polygons to BLACK
build	Build a map
bye	Exit QUILT
centroid	Compute the centroid for the polygons of a map
class=	List all classes satisfying a condition
complement	Generate the complement of a map
cursor	Return the (x, y) coordinates of the cursor
describe	Describe of QUILT object
display	Display a map
dispon	Open a display window
dispoff	Close a display window
edit	Edit a map or table (enter edit mode)
erase	Erase the display window
forget	Forget the binding of a name
genclass	List all attribute classes of objects in a map
genobject	List all objects in a map
handw	Compute the bounding rectangle for all objects in a map
help	Direct the user to the QUILT help system
info	Supply information about a QUILT function
intersect	Perform intersection between two maps
ismap	Make a map known to the database
istable	Make a table known to the database
name	Assign a name to an object
objectat	Return the data object at a specified point
perimeter	Return the length of all edges in a map
pointat	Set cursor to a specified location
printclass	Print attribute information for a map or object
raster	Create raster array for a quadtree
setscale	Set display scale
subset	Take a subset of the objects in a map
union	Create the union of two maps
window	Take a window from a map
within	Expand the polygons of a map

Selected quadtree algorithms are described in this section. The attribute system and its associated functions are described in § 6. Examples of the query language can be found in § 7.

5.1. *The quadtree editor*

Functions in the edit-map mode facilitate interactive construction and updating of maps stored as quadtrees. Rather than force the user to think in terms of the tree structure (e.g., a function to insert nodes) the editor commands make reference to logical units of the map, e.g., lines, points or polygons. The user performs editing operations such as inserting a line or point, changing the value of a specified polygon or collection of polygons, or splitting a specified polygon into more than one piece.

When many changes are to be made, the user may wish to see the effects of each step. Commands are provided to allow him to examine all or part of the map on a display device. The display is automatically updated as further map manipulation commands are executed. Associated with each map's quadtree representation is a descriptor

Table 2. Functions in edit-map mode

Name	Description
abort	Abort editing session
change	Give new value to a specified polygon
comment	Add a new comment to a map
cursor	Return the (x, y) coordinates of the cursor
describe	Describe a QUILT object
display	Display a map
dispon	Open a display window
dispoff	Close a display window
erase	Erase the display window
forget	Forget the binding of a name
head_ed	Edit map header
help	Direct the user to the QUILT help system
info	Supply information about a QUILT function
insert	Insert a point or line
name	Assign a name to an object
objectat	Return the data object at a specified point
pointat	Set cursor to a specified location
printclass	Print attribute information for a map or object
quit	Quit editing, save changes
remove	Remove an object from a point or line map
replace	Replace the value of a class of polygons
setscale	Set display scale
split	Split a polygon with a chaincode

Table 3. Functions in edit-table mode

Name	Description
abort	Abort editing session
addclass	Add classes to the table
cpclass	Copy attributes between classes
delclass	Delete classes from the table
describe	Describe a QUILT object
editclass	Edit the attributes of an attribute class
forget	Forget the binding of a name
help	Direct the user to the QUILT help system
info	Supply information about a QUILT function
istable	Make a table known to the database
name	Assign a name to an object
printclass	Print attribute information for a map or object
quit	Quit editing, save changes

termed the user's header, as mentioned in § 2. The editor allows the the user to modify the individual field values of this header. In addition, commands are provided for the insertion of comments into the quadtree for purposes of documentation.

To begin using the editor, the user calls a function called edit which places the user in edit-map mode. If the file to be edited does not already exist, then the editor asks if a new map is to be created. For new maps the user will also be prompted to indicate the type of map (region, line, or point data). In order to protect the user against errors and

machine crashes, a complete copy of the file to be edited is made and all changes are actually made to the copy. Like many text editors, at termination of an editing session the original copy of the file being edited is stored in a backup file and the updated copy replaces it. Alternatively, the user may abort the editing session, restoring his original file intact.

The pointat and cursor commands are included to assist the user in relating the map coordinates to the displayed image. Without them, choosing coordinates for the beginning of a boundary line or determining the current value of a polygon would be difficult. These functions allow the user to 'point' at a polygon with a trackball or mouse and pass the location of the cursor to the editing functions, respectively, without ever needing to specify actual coordinates.

Area maps are updated by use of the replace, change, and split commands which replace all polygons of a given value with a new value, change the value of a given polygon, or split a polygon into multiple polygons, respectively. Line and point maps are updated by use of the insert and delete commands which insert or delete lines or points; insertion and deletion operations were described in §§ 3 and 4. Each polygon (and hence each node making up the polygon) is considered to be a member of a 'class'. This class could be an elevation range or a land-use type such as 'wheatfields'. The value of each node making up the polygon indicates only the class of which it is a member; node values do not indicate unique polygons. This fact is significant when implementing certain polygon operations, as described below.

The replace command is executed by traversing the entire quadtree. Those nodes with the old class value have that value replaced by the new. For this command it is not necessary to distinguish between polygons of the same class since they are all processed in the same way. The kernel will automatically merge four siblings whose values are identical after processing by the replace command.

The change command is more complicated. This command manipulates only one polygon; however, other polygons of the same class may also exist. The change command is implemented using a variant of the standard graphics seed-fill algorithm. It is a recursive function which takes a node as its parameter. This node is checked to see that it has the old value (i.e., the one to be changed). If not, the function returns. If the node contains the old value, then it is set to the new value and the function is applied recursively to all of the node's neighbours. In this way, all nodes of a polygon will eventually be reached and only nodes in the polygon will have their values changed (since only neighbours of nodes in the desired polygon are ever candidates for processing).

The split command allows the user to impose an arbitrary line, one pixel wide, of a designated class value onto the map. The typical use of the command is to split a polygon into two or more separate parts. One of these parts would then become a polygon with the same class value as the splitting line via subsequent invocation of the change command. The pixels of of the splitting line would then be part of this new polygon. Alternatively, the split command can be used to make slight modifications of only a very few pixels, such as correcting a slightly misplaced border of a polygon. The split command operates by first inserting a node one pixel wide into the tree corresponding to the first coordinate given (typically by means of a cursor) and then following the splitting line, inserting nodes as it processes the line. Owing to the local system environment, QUILT was implemented to create a splitting line either from the keyboard or from a trackball (supplied with the display device supported by our VAX). Use with a mouse is conceptually identical. As the user types the code (or manoeuvres a

mouse or trackball), the chaincode is displayed. Producing an incorrect chaincode was judged to be a very common source of error when the editor was designed. Enabling the user to see the splitting line displayed as he inputs it allows the rapid detection and correction of errors. When the backspace key is typed, the splitting line is backed up by one pixel. This is accomplished by examining the end of a table which contains the chaincode description for the splitting line; both the map and the chaincode table are then updated to reflect the backup

5.2. *Raster to quadtree conversion*

The execution time for most quadtree algorithms in QUILT is proportional to the number of nodes inserted or located from the disk file. Even when building a quadtree from a picture array, the amount of time needed to read the picture data is trivial compared to the time necessary to manipulate the quadtree. The naive algorithm for converting a raster image to a quadtree is to insert each pixel of the raster image into the quadtree in raster order. Those pixels making up larger nodes will be merged together by the quadtree insert routine. In view of the large number of pixels in a raster image compared to the number of nodes in the quadtree representation for that image, it would be desirable to find an algorithm which can reduce the number of node inserts required.

QUILT uses an algorithm which, in the worst case, makes a single insert for each node in the resulting quadtree. This algorithm is based on two observations. The first is that, when processing an image in raster-scan order (top to bottom, left to right), the upper-leftmost pixel of any node is encountered before any other pixels in the node have been seen. Thus, the largest node for which this pixel is the upper left corner may safely be inserted into the tree. This may set the values for pixels in that portion of image that has not yet been processed. The second observation is that a list of 2^n nodes is sufficient to describe the unprocessed portion of the quadtree for a $2^n \times 2^n$ image, allowing decisions about node insertions to be made using information stored in main memory. When a pixel is encountered whose 'predicted' value as stored in the quadtree is different from the true value in the image (i.e., the value of the current pixel), the largest node for which the current pixel forms the upper-leftmost corner should be inserted. This will ensure that no future merging of quadtree nodes will be required. Table 4 shows an empirical comparison of the two algorithms. Shaffer and Samet (1987) give a more complete description of this algorithm and its analysis.

Table 4. Quadtree building algorithm statistics.

Map name	Number of nodes	New		Old	
		Number of inserts	Time (seconds)	Number of inserts	Time (seconds)
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
Center	4687	2121	16·1	262144	603·8
Pebble	44950	20770	111·0	262144	630·8
Stone	31969	14612	70·2	262144	639·5

5.3. *The subset function*

The subset function allows the user to construct a new map composed of an arbitrary collection of attribute classes and polygons from an input map (for the purposes of the subset function, an attribute class is simply a collection of polygons all with the same value). The subset function makes use of the capabilities for attribute attachment described in § 6 to allow the generation of a map containing those polygons of an input map matching some set of attribute values.

The subset function works as follows. First, the list of input objects (i.e., the list of polygons and attribute classes either specified explicitly by the user or returned by the attribute match function class=) is analysed and sorted into a final pair of lists, one for the attribute classes of the input map meeting the user's criteria, the other for specific polygons indicated as being in the subset. Second, each node of the input image is compared to the attribute class list to determine if it is contained in one of the indicated attribute classes. If so, it is inserted into the output map. Lastly, for each polygon on the polygon list, a function similar to the change function described in § 5.1 is executed, inserting each node of the indicated polygon into the output map.

5.4. *The within function*

The within function allows the user to create from a collection of polygons a map that expands the border of each non-WHITE polygon by R units where R is a radius value specified by the user. The result of this operation is known as a buffer zone or corridor. This function is an important step for answering queries such as 'What is the area of all wheat fields within 5 units of the river'? To answer this query, the user would apply the within function to the map containing only the river polygon (which might be obtained from a land-use map by use of the subset function) along with a radius value of 5 units. The resulting map would then be intersected with the map of wheatfields, followed by an area query.

QUILT's within algorithm works as follows. Note that this algorithm computes the chess-board distance metric (where $d(p,q)=\mathrm{MAX}(|p_x|, |p_y-q_y|)$) rather than the usual Euclidean distance metric. Each node of the input map is examined in Morton order. BLACK leaf nodes of width W such that $W>(R+1)/2$ are expanded to form a square of width $W+2R$ centered around the BLACK node. WHITE leaf nodes require no special processing. Collections of small leaf nodes whose aggregate could form a node of width less than or equal to $R+1$ are replaced automatically by a single BLACK node. These aggregates are also expanded by radius R by only processing a subset of their constituent BLACK nodes, and the corresponding BLACK leaf nodes are inserted into the output node list. Figure 8 shows the central region of the floodplain along with the border of the polygon which would result from an expansion by 8 pixels. For complete details on this algorithm, see Ang *et al.* (1988).

5.5. *Quadtree traversal and active borders*

Many quadtree algorithms can be expressed as simple tree traversals where at each leaf a simple computation is performed involving that leaf. Examples of such algorithms are area and centroid calculation, conversion of a multi-coloured image to binary, and the replace function. Another class of algorithms visits each node of the tree and, for each node, performs a computation involving that node and some or all of its neighbours. Such algorithms include perimeter and connected components computation. Finding the neighbour of a node in a linear quadtree implementation is

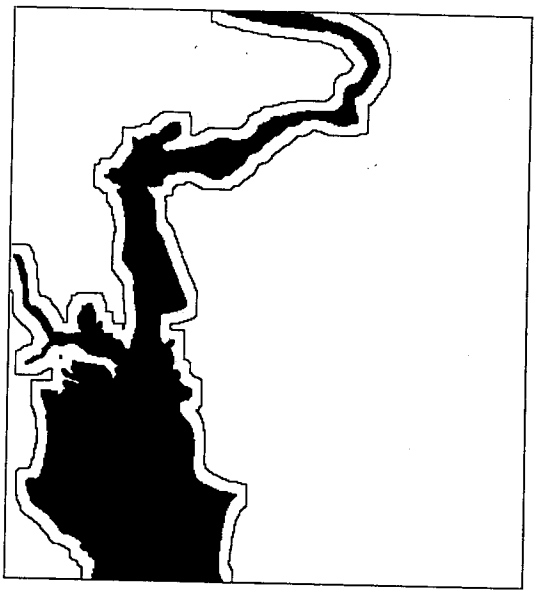


Figure 8. An example of the WITHIN function: expansion of the floodplain by 8 units.

somewhat costly as it involves searching the node list. For some operations, it is possible to maintain tables which contain the value, size and position of those neighbouring nodes which have been seen previously during the traversal, as detailed in Samet and Tamminen (1985a). At any instant, the state of the traversal (i.e., after processing m leaf nodes) can be visualized as a stair-case (termed an active border). Figure 9 shows the active border for the image of figure 2 after processing block 8. To find the value of a North or West neighbour, one needs only to look in the table. The perimeter of an image and connected component labelling can both be computed in this way.

These algorithms make use of two tables, each 2^n records long for a map of size $2^n \times 2^n$. These tables correspond to the vertical and horizontal edges of the active border, respectively. Perimeter computation can be implemented by visiting each node in Morton order. Each node is compared with its Eastern and Northern neighbours. For each node/neighbour pair such that the nodes have different values, the length of the border between them is added to the perimeter value.

5.6. *Set functions and windowing*

Geographic information systems commonly support set operations performed on a pair of images. For example, suppose a map of all wheat fields above 100 feet in elevation is required. This can be produced by intersecting a map of wheatfields and an elevation map whose pixels representing elevations above 100 feet are non-WHITE. The resulting map would have non-WHITE pixels wherever the corresponding pixels of the input maps were both non-WHITE. Set operations on quadtrees representing images with the same grid size, same map size and same origin is straightforward (Hunter 1978). A depth-first traversal of both trees is performed in parallel; each node of the first image is compared with the corresponding node(s) in the second image.

The QUILT system contains an algorithm for unaligned map intersection that visits each node of the input quadtrees only once, and performs at most one insertion into the output quadtree for each output tree node. It takes advantage of the concepts

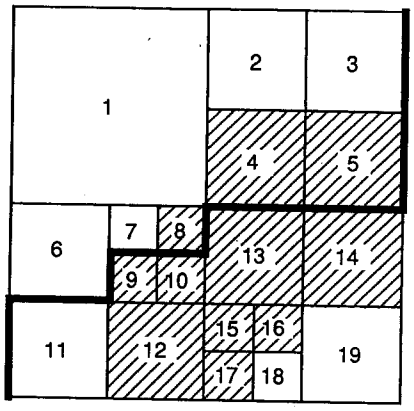


Figure 9. The active border after processing block 8.

of largest node insertion and active borders discussed in §§ 5.2 and 5.4. This algorithm works by traversing one of the input quadtrees, named QA, in Morton sequence order. For each node N of QA, the nodes of the second quadtree, named QU, that cover N are located. The set operation is computed on the value of N and the overlapping blocks of QU, with the resulting sub-blocks sent to the output map in Morton sequence order. Nodes of QU that have been partly (but not completely) processed are maintained in an active border table. With only minor modifications, the intersection procedure can also be used to find the set intersection of two images that are rotated with respect to each other. For complete details on this algorithm, see Shaffer and Samet (1990).

Another function commonly provided by GIS allows the user to extract 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. The window could also be partly beyond the image border. Most importantly, the window's origin (or upper left corner) could potentially be anywhere in relation to the input map's origin. This means that large blocks from the quadtree representation of the image must be broken up and possibly recombined into new blocks in the quadtree representation for the windowed image. 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 representation of an image that has the fewest nodes. It can also be used to align two images represented by quadtrees.

If windowing is viewed as a set function on two unaligned maps, an algorithm requiring at most a single insertion per output tree node can be derived from our intersection algorithm. Let QA be a BLACK rectangle with the same origin as the window and whose size is the least power of 2 greater than, or equal to, the maximum of the window's width and height. Let QU be the image from which the window is extracted. The resulting image would have QA's size and position, with the value of QU's corresponding pixel at each position that falls within the window (pixels beyond QU's border may be set to WHITE). The equivalence between windowing and unaligned set intersection should be clear. In fact, the windowing algorithm is slightly simpler, since a single BLACK node of the appropriate size takes QA's place in the algorithm. The windowing algorithm used in QUILT locates (only once) those input tree nodes that cover a portion of the window, and performs at most one insert

operation for each output node. This is achieved by noting that blocks to be inserted into the output tree are generated in Morton order.

6. Attribute attachment

It is often desirable to associate non-geographic information with geographic objects contained in a map. For example, we may wish to associate with a city its name and population, regardless of whether the city is stored as a data point or as a polygon. Such pieces of data are referred to as attributes. Specifically, in QUILT an individual attribute is a name/value pair. The user may also wish to ask queries about collections of map objects which have some feature or attribute in common, e.g., all cities within a population range. Storing such information, and answering such queries, is done in our system through the attribute attachment functions described in this section.

Each map is associated with what will be referred to as an attribute class table (often shortened to attribute table). This table is actually a separate file which stores information about a collection of attribute classes. An attribute class is simply a list of the name/value pairs making up the associated attributes. An attribute class table is therefore a collection of attribute classes. Any number of maps can share an attribute table. This allows a user to create a database of many maps, say topographic maps, where polygons belonging to a particular attribute class in each map of the set are interpreted in the same way.

Each map file, when created, stores the name of its attribute table. The particular table that is associated with a given map can be changed by the user if desired. As part of the preparation of an image for use in the database system, each quadtree node is given a value indicating to which class it belongs. This value is an offset into the list of attribute classes, e.g., a node storing the value '5' would be a member of the fifth attribute class of the attribute table associated with that map. Information associated with a node from some polygon in an image can be altered by either changing the value of the node so that it becomes a member of another class, or by changing the information stored about that class in the table.

All attribute tables contain the default attribute classes 'white' and 'black' which represent the default background and foreground values, respectively, of a binary image. All attribute classes contain, in addition to any user-supplied information, a default attribute named 'color'. This attribute determines the display colour for the region or object when the map is displayed. The default value for 'color' is the same as the colour value of class 'black'.

The implementation and storage of attribute tables makes use of the symbolic manipulation abilities in the LISP programming language. Each attribute class is represented by a LISP atom and the name/value pairs associated with a class are stored in the atom's property list. The value of an attribute is arbitrary and may be either a string or a numeric value. Since an attribute class is implemented as a LISP atom, individual attribute classes may contain as many name/value pairs (i.e., attributes) as desired. Attribute classes may also refer to other attribute classes. This allows for more efficient use of storage when several classes share information. Cycles within the attribute lists are prevented by the editing functions for attribute tables.

Functions for manipulating attribute tables are listed in table 3. New attribute tables may be created by use of the edit function. Between invocations of QUILT, each attribute class table is stored in a separate disk file. This disk file contains a representation of the LISP structure which maintains the table when it is in main

memory. When a map is entered into the database system, the associated attribute table is entered as well. Attribute tables which are not associated with any maps currently known to the database system can be entered by use of the function istable; they may then be accessed by the user. Copytable copies an attribute table. This may be useful for creating new tables with only a few differences from an old table.

A number of editing functions are available in edit-table mode. Editclass prompts the user to indicate which attributes are to be altered, added or deleted. Delclass removes the named class or classes from the table. It is the responsibility of the map user to ensure that no maps which refer to this attribute table contain nodes indexing the deleted class. Addclass adds the named class or classes to the table. The editclass function would then be used to add attributes. Cpclass copies a class (possibly from another table). The new class will have all the attributes and values of the old class. This function is useful for creating new classes which are nearly identical to old classes, with only a few attributes changed. Printclass prints out information from a table. This function may also be used when not in edit-table mode. Once an attribute table has been created, and associated with a map, the class= function may be used to generate a list of classes from the table which meet some condition or set of conditions. For convenience, the subset function implicitly executes the class= function on any conditional expressions in its parameter list.

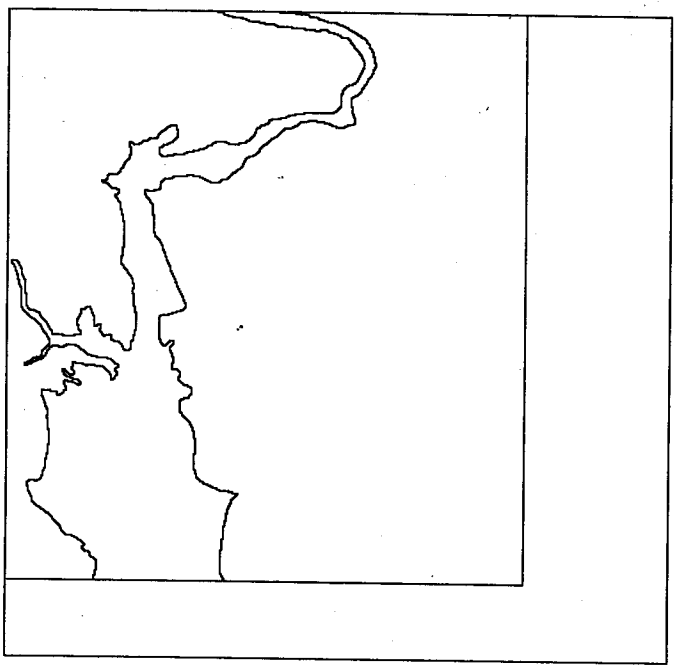
During implementation of QUILT, the primary emphasis was placed on the spatial database and related algorithms. Our attribute system, while quite flexible, is admittedly inefficient for even moderately-sized collections of attribute data. This is partly due to the inefficiency of LISP as a database language. Our system does, however, illustrate many of the features desirable in an attribute database system. Efficient implementation of large-scale attribute databases is a major research topic. We hope to do more work on this area in the future.

7. User interface and query examples

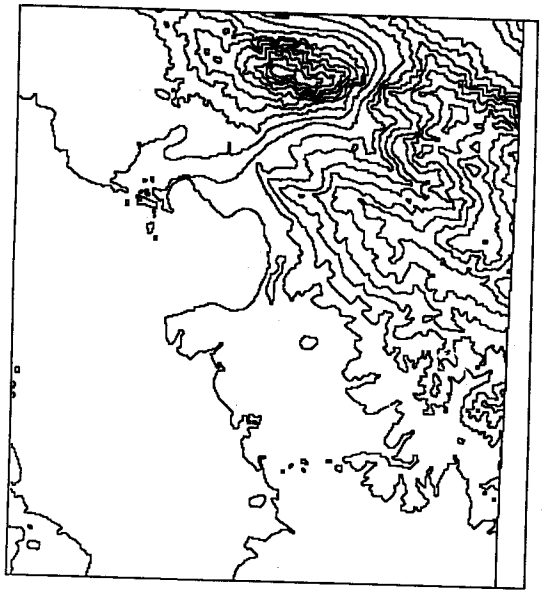
As mentioned before, the attribute attachment system and the user interface are implemented in the LISP programming language. In fact, the user interface is LISP itself, augmented by the GIS functions of the QUILT system. This interface would be likely to be unsatisfactory for use by non-programmers (though no worse than many other GIS currently used). However, as an interface for use by programmers researching the design and implementation of GIS algorithms, it has served us well.

Functional programming has played an important role in our implementation. Not only does the interface support functional programming extensively (as would be expected, since it is LISP) but the kernel was also implemented as a set of composable 'C' functions. At both the kernel and the interface levels, the building blocks provided by the system can easily be put together in many ways. The result is a flexible programming system for the design and testing of algorithms.

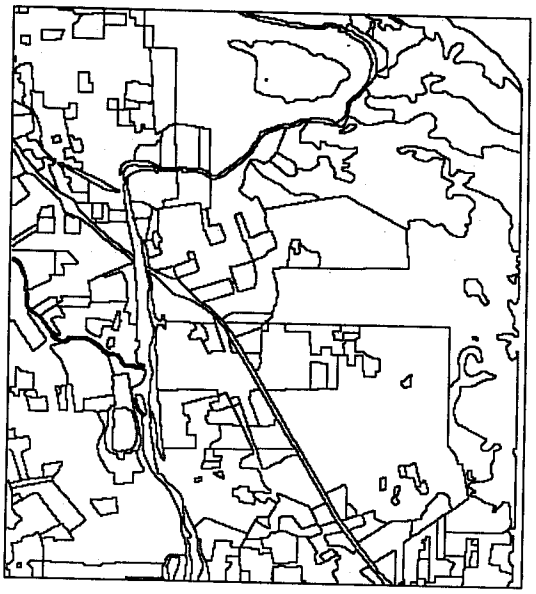
The remainder of this section presents timings from a group of set queries and images resulting from other selected queries. The map test set used throughout this paper was generated from Russian River Valley in Northern California. The point and line data sets in figures 5 and 7 were generated from the US Geological Survey quadrangle map of the area. The region data maps shown in figure 10 were generated from overlays supplied by the US Army Engineer Topographic Laboratories. These region maps are referred to below as flood, top and land respectively.



(a)



(b)



(c)

Figure 10. Three test maps: (a) the Floodplain map, (b) the Topography map and (c) the Land-use map.

7.1. *Set function queries*

A list of 25 set queries, translated into QUILT's query language, is shown in table 5. In addition to demonstrating the use of functional composition in GIS queries, these examples illustrate the subset function and the search abilities of the attribute system in generating maps of polygons with specific attribute values. Table 6 shows statistics on times for these queries when executed on a VAX 11/785 running BSD 4.3 UNIX. Note that these times include both the attribute manipulation and the subset function execution times as well as the set operation times. Typical set operations on maps of comparable complexity to the results of the subset functions reported here take about 5 to 10 seconds of CPU time.

7.2. *Additional examples of queries*

Figure 11 (*a*) demonstrates the use of several database functions. A new map is created containing a specified polygon of the land-use map using the function objectat. Next, the function handw is applied; it returns the descriptor of the smallest enclosing rectangle for this polygon. Finally, a window of the topography map is extracted using this window descriptor. The query appears as follows:

```
(window top (handw (subset land (objectat land 100 400))))
```

Figure 11 (*b*) demonstrates the result of executing multiple intersections. In our implementation, the value of each node in the output map is set to that of the

Table 5. Twenty-five set queries with attribute attachment

```
1.   (intersect (subset land acp) (subset flood inside))
2.   (intersect (subset land acp) (subset flood inside nil))
3.   (intersect (subset land acp) (subset top (> >height 2)))
4.   (intersect (subset land acp)
       (subset top (and (> >height 2) (< <height 6))))
5.   (intersect (subset land acp)
       (subset top (or(< <height 3) (> >height 5))))
6.   (intersect (subset land acp) (subset top (not (= =height 5))))
7.   (union (subset flood inside) (subset land avv nil))
8.   (subset land avv acp nil)
9.   (subset land avv acp)
10.  (union (subset flood inside)
       (intersect (subset land avv nil) (subset top (= =height 1))))
11.  (subset land (= =classname ^u)) (* any name starting with u *)
12a. (intersect (subset land r) top)
12b. (intersect (subset land r) (subset top (> >height 1)))
12c. (intersect (subset land r) (subset top (> >height 2)))
12d. (intersect (subset land r) (subset top (> >height 3)))
12e. (intersect (subset land r) (subset top (> >height 4)))
12f. (intersect (subset land r) subset top (> >height 5)))
12g. (intersect (subset land r) (subset top (> >height 6)))
12h. (intersect (subset land r) (subset top (> >height 7)))
12i  (intersect (subset land r (subset top (> >height 8)))
12j. (intersect (subset land r) (subset top (> >height 9)))
12k. (intersect (subset land r) subset top (> >height 10)))
13.  (intersect top (subset flood inside nil))
14.  (union (subset flood inside) (subset top (= =height 5)))
15.  (union (subset flood inside) subset top (= =height 1)))
```

Table 6. Timings for 25 set function queries. Time in min:sec.

Query number	Area	Elapsed time		CPU time	
		Query time	Cumulative time	Query time	Cumulative time
1.	152	0:21	0:21	0:11·8	0:11·8
2.	26734	0:21	0:42	0:12·9	0:24·7
3.	12311	0:56	1:38	0:33·0	0:57·7
4.	9752	0:38	2:16	0:24·3	1:22·0
5.	16813	0:55	3:11	0:31·7	1:53·7
6.	25386	1:20	4:31	0:41·1	2:34·8
7.	157190	1:22	5:53	0:42·6	3:17·4
8.	118928	0:45	6:38	0:20·8	3:38·2
9.	56571	0:17	6:55	0:10·1	3:48·3
10.	50646	1:34	8:29	0:49·6	4:37·9
11.	37820	0:16	8:45	0:10·6	4:48·5
12a.	20752	0:26	9:11	0:19·0	5:07·5
12b.	20727	1:15	10:26	0:39·7	5:47·2
12c	18518	1:12	11:38	0:32·9	6:20·1
12d.	15591	0:45	12:23	0:26·7	6:46·8
12e.	12211	0:35	12:58	0:22·0	7:08·8
12f.	9442	0:37	13:35	0:18·6	7:27·4
12g.	7416	0:26	14:01	0:15·5	7:42·9
12h.	5532	0:24	14:25	0:13·0	7:55·9
12i.	3198	0:20	14:45	0:10·8	8:06·7
12j.	1108	0:18	15:03	0:8·9	8:15·6
12k.	8	0:17	15:20	0:7·8	8:23·4
13.	142701	0:17	15:37	0:14·2	8:37·6
14.	37194	0:21	15:58	0:12·6	8:50·2
15.	59599	0:20	16:18	0:11·8	9:02·0

corresponding node in the first input map whenever the result of the intersection of the two input maps is not WHITE. Thus, the second input map can be viewed as a template for which portions of the first input map will be preserved. In figure 11 (*b*), the regions below 100 feet in elevation on topography (i.e., below height class 2) are intersected with the central region of the flood plain. This map in turn serves as a template for the land-use map. This query also demonstrates a search of the attribute table associated with map 'top'. 'Height' has been stored as an attribute with a numeric value. Those attribute classes containing a 'height' value less than 2 will be selected during the first phase of the subset function. In the second phase, the map is examined to remove those objects which do not belong to the selected attribute classes. The query is as follows:

(intersect land (intersect center (subset top (≪height 2))))

Figure 11 (*c*) demonstrates the intersection of a point map (houses) and an area map corresponding to the regions in the topography map which are below 100 feet in elevation (i.e., below height class 2). The query

(intersect houses (subset top (≪height 2)))

yields all houses below 100 feet elevation (i.e., below height class 2). Figure 11 (*d*) shows the intersection of a line map (road) with the same area map. The query is as follows:

(intersect road (subset top (≪height 2))).

8. Conclusions and future plans

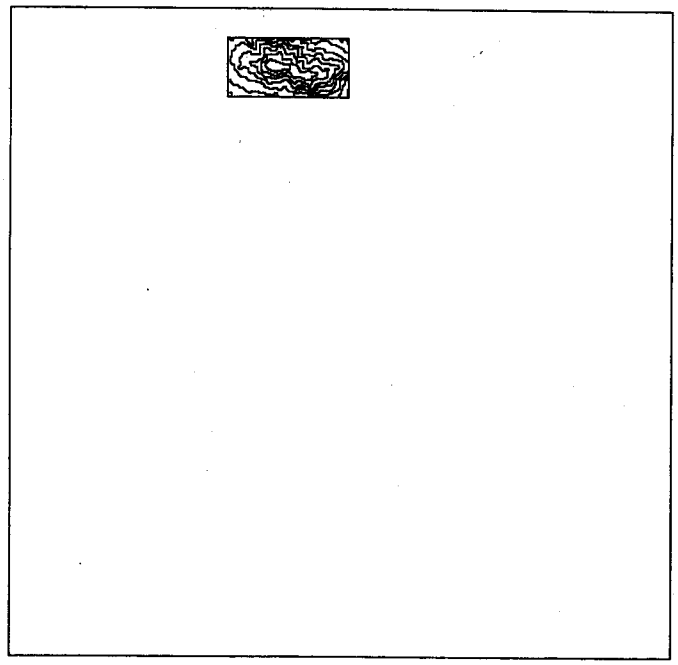
Our initial goal in this project was to demonstrate the feasibility of using quadtree data structures for geographic data. We feel that we have been very successful, demonstrated in part by the extensive current interest in quadtrees and other hierarchical data structures within the GIS research community. Since the inception of the QUILT system, other research groups have also mounted efforts to produce GIS based on hierarchical data structures (for example, Callen *et al.* 1986, Palimaka *et al.* 1986, Smith *et al.* 1987, Abel 1989). QUILT provides a prototype system which handles area, point and line data for small data sets. Many functions such as map editing, set operations, geometric properties and windowing have been implemented efficiently.

In the future, we plan to extend the prototype system in a number of ways. A key issue in geographic databases is the handling of large volumes of data. The quadtree is a data structure of variable resolution whose performance should improve in comparison with the image array as the resolution of the image increases. Our experience has primarily been with 512×512 images. However, we need to study further the effect of increasing the image resolution as well as dealing with a large collection of maps rather than maps of a single area. Empirical tests indicate that the quadtree data structure performs well when compared to the array for the types of maps tested (see Shaffer 1986). Higher resolutions should be even more favourable to the quadtree since every doubling of resolution requires the array to increase by a factor of 4, while the quadtree is expected to increase in size only by a factor of 2 (Shaffer 1986). We need also to compare the quadtree-based representation of regions with quadtree-based vector representations for polygons.

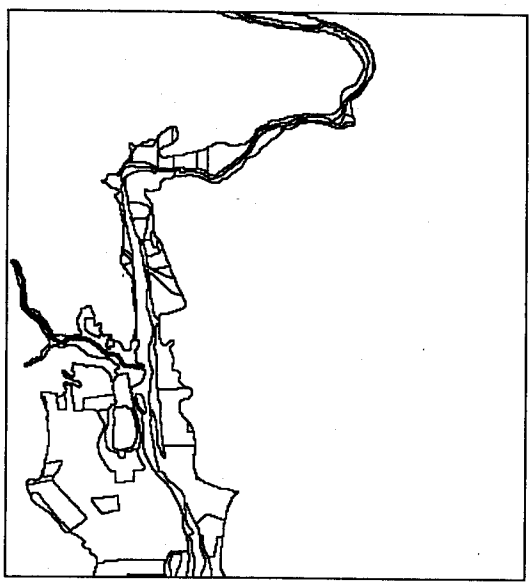
A very large percentage of QUILT's execution time is spent in the kernel, as would be expected of a disk-based file processing application. The algorithms of § 5 attempt to minimize the number of node manipulations required. Several experiments are planned in the hopes of speeding up the kernel itself. These include implementation of the 2DRE representation (Lauzon *et al.* 1984) for area representations; investigation into the use of pointer-based quadtrees stored on disk in pages rather than using the B-tree as an intermediary representation; and use of the Bintree (Samet and Tamminen 1985 b), a close relative of the quadtree which decomposes the image along each axis in turn instead of along all axes at each decomposition.

Our current system of attribute attachment was designed only for quick implementation, as our primary area of interest at the time was map algorithms. The storage of large volumes of relatively unstructured data is an important research topic in its own right. We hope to address this problem in future versions of the system.

The use of large databases involves examining the inter-relationships between sets of maps as well as linking maps which are too large to be handled as individual units by the existing database. This will require the development of a naming convention or, better yet, a directory system which allows a user to locate all requested geographical objects without knowing the names of the maps on which such information resides. Our current system provides many functions for manipulating individual maps, i.e., new maps may easily be generated from individual maps. Set functions allow for the union or intersection of more than one map. Given a window and set of maps, a function should be provided which generates a map containing all information from the set which lies within the window. Another issue typically dealt with by cartographic systems has to do with methods of projection (c.f. Tobler and Chen 1986). Since the earth is a sphere and map representations are flat, some special provisions will be necessary when joining maps. This is particularly true near the poles in most

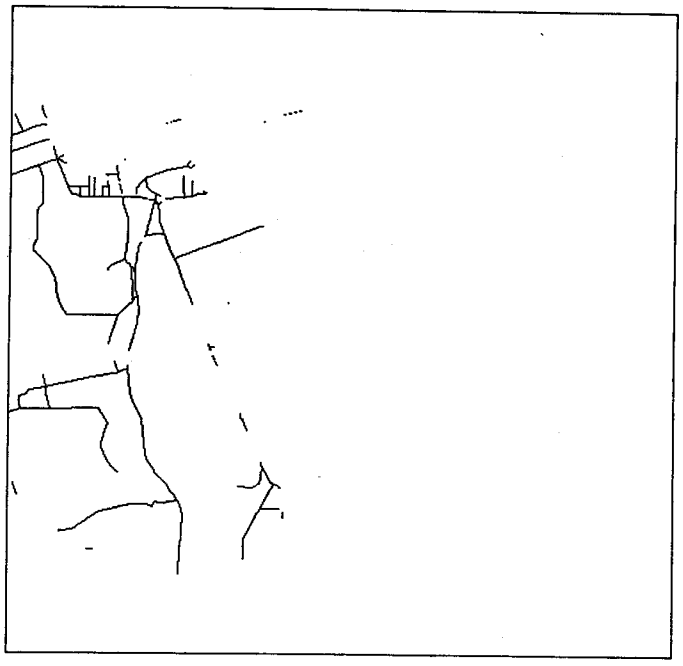


(a)



(b)

(*c*)



(*d*)

Figure 11. Four maps showing query results: (*a*) a windowing example, (*b*) the land-use classes within the floodplain and below 100 feet in elevation, (*c*) the houses below 100 feet in elevation and (*d*) the roads below 100 feet in elevation.

projections. Thus we aim to study the effect of distortions caused by the earth's curvature on our representations. Problems of map projection have not been addressed so far in the existing system.

Acknowledgments

The QUILT system was implemented under the direction of Professors Hanan Samet and Azriel Rosenfeld at the Center for Automation Research, University of Maryland. The major contributors to the system are as follows. The kernel was originally written by Robert E. Webber, with modifications by Randal C. Nelson and Clifford A. Shaffer. Most of the quadtree manipulation algorithms, the point data implementation and the LISP interface functions were written by Clifford A. Shaffer. The map editor was written by Clifford A. Shaffer with modifications by Randal C. Nelson. The line data implementation was written by Randal C. Nelson. The attribute attachment system was written by Yuan-Geng Huang. Many of the quadtree algorithms used were derived from ones originally published by Hanan Samet. Others who contributed to the software effort were Chuan-Heng Ang, Kikuo Fujimura, David Oskard, Jerome R. Smith, Hiow-Tong See, and Richard Aleksandr. QUILT is currently maintained by Walid Aref.

References

ABEL, D. J., 1984, A B^+-tree structure for large quadtrees. *Computer Vision, Graphics and Image Processing*, **27**, 19.

ABEL, D. J., 1989, A spatial database shell for relational database management systems. *Proceedings of the Third International Conference on Foundations of Data Organization and Algorithms*, edited by W. Litwin and H. J. Schek (Paris), pp. 257–261 (see also *Lecture Notes in Computer Science*, 1989 (Berlin: Springer-Verlag), No. 367).

ABEL, D. J., and SMITH, J. L., 1983, A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing*, **24**, 1.

ANG, C.-H., SAMET, H. and SHAFFER, C. A., 1988, Fast region expansion for quadtrees. *Proceedings of the Third International Symposium on Spatial Data Handling held in Sydney, Australia, in August 1988*, pp. 19–37.

CALLEN, M., JAMES, I., MASON, D. C., and QUARMBY, N., 1986, A test-bed for experiments on hierarchical data models in intergrated geographic information systems. In *Spatial Data Processing Using Tesseral Methods*, edited by B. M. Diaz and S. B. M. Bell (Swindon: Natural Environment Research Council).

COMER, D., 1979, The ubiquitous B-tree. *ACM Computing Surveys*, **11**, 121.

FODERARO, J. K., 1980, *The Franz Lisp Manual* (Berkeley: The Regents of the University of California).

GARGANTINI, I., 1982, An effective way to represent quadtrees. *Communications of the ACM*, **25**, 905.

HUNTER, G. M., 1978, *Efficient Computation and Data Structures for Graphics*. Ph.D dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey, U.S.A.

LAUZON, J. P., MARK, D. M., KIKUCHI, L., and GUEVARA, J. A., 1985, Two-dimensional runencoding for quadtree representation. *Computer Vision, Graphics and Image Processing*, **30**, 56.

MORTON, G. M., 1966, A computer oriented geodetic data base and a new technique in file sequencing, IBM Canada.

NELSON, R. C., and SAMET, H., 1986 a, A consistent hierarchical representation for vector data. *Computer Graphics*, **20**, 197.

NELSON, R. C., and SAMET, H., 1986 b, A population analysis of quadtrees with variable node size, Computer Science TR-1740, University of Maryland, College Park, Maryland, U.S.A.

ORENSTEIN, J. A., 1982, Multidimensional tries used for associative searching. *Information Processing Letters*, **14**, 150.

PALIMAKA, J., HALUSTCHAK, O., and WALKER, W., 1986, Integration of a spatial and relational database within a geographic information system. *Proceedings of the 1986 ACSM-ASPRS Annual Convention, Geographic Information Systems*, held in Washington, U.S.A. in March 1986 (Falls Church: American Congress on Surveying and Mapping), pp. 131–140.

SAMET, H., 1989, *Design and Analysis of Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods* (Reading: Addison-Wesley).

SAMET, H., 1990, *Applications of Spatial Data Stuctures: Computer Graphics, Image Processing, and GIS* (Reading: Addison-Wesley).

SAMET, H., ROSENFELD, A., SHAFFER, C. A., and WEBBER, R. E., 1984, A geographic information system using quadtrees. *Pattern Recognition*, **17**, 647.

SAMET, H., and TAMMINEN, M., 1985 a, Computing geometric properties of images represented by linear quadtrees. *I.E.E.E. Transactions on Pattern Analysis and Machine Intelligence*, **7**, 229.

SAMET, H., and TAMMINEN, M., 1985 b, Bintrees, CSG trees, and time. *Computer Graphics*, **19**, 121.

SAMET, H., and WEBBER, R. E., 1984, On encoding boundaries with quadtrees. *I.E.E.E. Transactions on Pattern Analysis and Machine Intelligence*, **6**, 365.

SAMET, H., and WEBBER, R. E., 1985, Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, **4**, 182.

SHAFFER, C. A., 1986, A Comparison of Vectors, Rasters, and Quadtrees for Representing Geographic Data. In *Geologisches Jarhbuch*, Reihe A, Heft 104, edited by R. Vinken (Hannover), pp. 99–115.

SHAFFER, C. A., and SAMET, H., 1987, Optimal quadtree construction algorithms. *Computer Vision, Graphics and Image Processing*, **37**, 402.

SHAFFER, C. A., 1986, A Comparison of Vectors, Rasters, and Quadtrees for Representing *Graphics and Image Processing*, **50**, 29.

SHNEIER, M., 1981, Two hierarchical linear feature representations: edge pyramids and edge quadtrees. *Computer Graphics and Image Processing*, **17**, 211.

SMITH, T., PEUQUET, D., MENON, S., and AGARWAL, P., 1987, A knowledge-based geographical information system. *International Journal of Geographical Information Systems*, **1**, 149.

TOBLER, W., and CHEN, Z. T., 1986, A quadtree for global information storage. *Geographical Analysis*, **18**, 360.