PROCESSING GEOGRAPHIC DATA WITH QUADTREES*

Hanan Samet, Azriel Rosenfeld, Clifford A. Shaffer, and Robert E. Webber**

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

ABSTRACT

We describe the current status of an ongoing research effort to develop a geographic information system based on quadtrees. A linear quadtree encoding was implemented using a B-tree to organize the list of leaves and allow management of trees too large to fit in core memory. Several database query functions have been implemented including set operations, region property computations, map editing functions, and map subset and windowing functions. A user of the system may access the database via an English-like query language.

1. INTRODUCTION

The quadtree representation of regions (e.g., Figure 1), first proposed by Klinger [5] has been the subject of intensive research over the past several years (for an overview, see [9]). Numerous algorithms have been developed for constructing compact quadtree representations, converting between them and other region representations, computing region properties from them, and computing the quadtree representations of Boolean combinations of regions from those of the given regions. Quadtrees have traditionally been implemented as trees which require space for the pointers from a node to its sons. However, pointer-less quadtree representations (e.g. *linear quadtrees* [4]) are superior when working with very large quadtrees. In this case, the set of regions is treated as a collection of leaf nodes. Each leaf is represented by use of a locational code corresponding to a sequence of directional moves that locate the lower left pixel of the leaf along a path from the root of the tree.

In this paper we describe the current status of an ongoing research effort to develop a geographic information system based on a variant of linear quadtrees. Quadtree encodings were constructed for area, point, and line features from maps and overlays representing a small area of Northern California. A memory management system based on B-trees [2] was devised to organize the resulting collection of leaf nodes, allowing for the use of arbitrary sized maps within a restricted amount of core memory. Many database functions were implemented, including map editing capabilities, set operations, and region property functions. Further details about this effort can be found in [7,8].

The database used in the study was supplied by the U.S. Army Engineer Topographic Laboratory, Ft. Belvoir, VA. The area data consisted of three registered map overlays representing landuse classes, terrain elevation contours, and floodplain boundaries, which were hand-digitized at a resolution of 400 by 450 pixels and then embedded within a 512 by 512 grid for quadtree encoding. A geographic survey map for this area supplied point data (house locations) and four sets of line data (a railroad line, a power line, a city border, and a road network).

Note that the variant of the quadtree that we use results in a decomposition of space into equal-sized parts. This is in contrast to the point quadtree [3] and the k-d tree [1] where the decomposition is governed by the input. The advantage of our variant of the quadtree is that different maps will be in registration thereby facilitating set operations such as map overlay.

Our database system can be viewed as being made up of four levels. The lowest level (written in C and discussed in section 2) controls the interface between the disk file used to store the quadtree data and the programs that are used to manipulate the images. The second level, also written in C, implements map editing (discussed in section 3) and other map manipulations (discussed in section 5). The interpretation of quadtree node values as features in a map is discussed in section 4. The third level, written in LISP, controls the interface between the C programs that manipulate the map and our query language. User defined names and data items are maintained at this level. The highest level is an English-like query language also written in LISP and described in section 6.

2. THE QUADTREE MEMORY MANAGEMENT SYSTEM

The quadtree memory management system (henceforth called the kernel) is based on the quadtree encoding scheme illustrated in Figure 2. The key feature of our encoding scheme is that a preorder traversal of the explicit tree will produce the nodes in ascending order of their locational codes. Each node's locational code is formed by interleaving the bits corresponding to the x and y coordinates of the pixel at the lower left corner of the node. We store only the leaf nodes of the tree, sorted in ascending order of this address field. Any pixel contained within a node will have an address greater than that of the leaf's lower left corner, but less than that of the next node in preorder. Therefore, given the address of any pixel and a list of leaves ordered by their addresses, finding the leaf containing that pixel reduces to searching a sorted list.

Given the linear ordering of leaf nodes and the fact that we are storing files containing as many as 30,000-40,000 leaves, we decided to organize the quadtree files using a B-tree structure. The kernel maintains a buffer pool in core and a B-tree in the disk file. The buffer pool need only store that portion of the tree in core for which there is room. We expect that there will be strong locality of reference - i.e. the leaf for which we are presently searching will very likely be near the leaf we last found. Therefore, the buffer pool is maintained on a schedule that replaces the least recently used buffers first. The kernel also controls inserting, deleting and finding quadtree node descriptors in the B-tree structure.

A quadtree node descriptor is composed of two 32-bit words. The first word contains information on the leaf's position and depth in the tree. In particular, it contains a 24 bit field which consists of the address of the pixel at the lower left corner of the node formed by interleaving the bits of its x and y coordinates. The remaining eight bits indicate the depth of the node in the tree. The second word contains information about the data that the node represents. The contents of the second word is not used by the kernel. Thus the kernel is unaware of whether it is manipulating point, line, or area

*The support of the U.S. Army Engineer Topographic Laboratories under Contract DAAK70-81-C-0059 is gratefully acknowledged.

**Department of Computer Science, Rutgers University
Busch Campus, New Brunswick, NJ 08903

data.

A quadtree file is made up of four parts. First, there is the kernel's fixed-size header which contains information about the size of the file and the B-tree structure. Second, there is a fixed-size block for the user's header (further described in Section 3). The third part is a list of comments. These comments are either generated by database functions when a new map is created, or are inserted at the request of the database user. In either case, they serve to document a map. Finally, a quadtree file contains the B-tree pages that contain the quadtree node descriptors.

The bulk of the quadtree file is made up of the B-tree pages. Each page is 512 bytes long (a convenient size for system read and write routines). We store 60 quadtree node descriptors in each page. The remaining space in the page contains information related to the B-tree organization of the database.

3. THE QUADTREE EDITOR

The quadtree editor serves to facilitate the interactive construction and updating of maps stored as quadtrees. Presently, it is a subsystem enterable from the query language, but having its own command language. Rather than forcing the user to think in terms of the tree structure, the editor's tree manipulation commands make references to logical units of the map (e.g., lines, points or polygons). It allows the user to perform such operations as inserting a line or point, changing the value of a specified polygon, 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 enable him to examine all or part of the map at a selected window on a display device. This display is continuously updated as further map manipulation commands are executed. Associated with each map's quadtree representation is a descriptor termed the quadtree header. There exist commands which allow the user to modify this header. The header contains the size of the map, the tree type (area, point, or line), the coordinate of the lower left corner of the map in relation to a global coordinate system. the rotation angle or tilt of the map from the external horizontal, and some information as to the type of data (i.e. topography, landuse, house) that is being stored. A command is also provided to enable the user to insert textual comments for documentation purposes.

When the editor is invoked, the user gives the name of the file to be edited. A temporary disk file is created on which all editing is to be done. Another file is created to store the commands given by the user. These files help protect the user from serious loss due to system crashes or his own errors such as mistyped or unwanted commands. They also enable him to abort the editing session without damaging the original copy. If the file to be edited is an old one, a copy is made in the temporary file. If a new map is to be created, then a default header is installed and the map is initialized to be one WHITE region.

The user of the quadtree editor views the map as a collection of polygons (sets of contiguous pixels with the same value). Each polygon (and hence each node making up the polygon) is a member of a "class". This class could be an elevation range or a landuse type such as "wheatfields." Class information is recorded for each node by use of a value field that is part of the node's descriptor.

Changes to region maps are made by use of the REPLACE, CHANGE, and SPLIT commands as described below. These commands are sufficient for any needed map modification. Line and point maps can be modified via the INSERT and DELETE commands as discussed in Section 4. The REPLACE command is executed by traversing the entire quadtree. Those nodes with the old (class) value have that value replaced by the new.

The CHANGE command is more complicated. It changes the class value of only one polygon; however, other polygons of that class may also exist. Thus, instead of traversing the entire quadtree, a "seed" node inside the polygon is "grown" by examining all of its neighbors until the entire polygon has been processed.

The SPLIT command allows the user to draw an arbitrary line, one pixel wide, of a designated value onto the map. The arbitrary line is specified as a chain code. The intended use of the command is to split a polygon into two or more separate parts. One of these parts would then become a polygon of the same class as the pixels representing the arbitrary line via subsequent invocation of the CHANGE command. The pixels representing the arbitrary 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. This type of correction could not be applied in any other way with the available command set.

The SPLIT command operates by first inserting a one pixel node into the tree corresponding to the first location given and then following the chaincode inserting nodes as it reads the code. A key feature of our implementation of the SPLIT command is that the user can observe the progress of the chaincode as he is inputting it. When the backspace key is typed, the chaincode is undone by one pixel, allowing for easy error correction

By repeated use of the three commands REPLACE, CHANGE, and SPLIT, it is possible to a make any desired changes to a region map. Clearly this is true since in the worst case the user could construct an entire map from one pixel chaincodes.

4. POINT AND LINE REPRESENTATIONS

Quadtree representations for point and line data were also developed. It should be noted that the same kernel (described in Section 2) is used for manipulating quadtrees of all three data types. When storing area data in region quadtrees, the value of a leaf corresponds to the color of the region that contains the leaf. Since there is no notion of color associated with either point or line data, other interpretations are placed on the information stored in the value portion of the leaf descriptor. The interpretation that a particular routine makes of a leaf's value is dependent on the type of data that is being stored in the quadtree. The database system stores the data type of the map in the user header which was described in Section 3.

The value field of a quadtree node is made up of a single word 32 bits long. For area quadtrees, this is a numeric value which can be interpreted as BLACK/WHITE, a color value, or as the key for a symbolic item such as landuse or elevation classes. In this last case, further information describing the item might be part of the database. For point data, nodes containing data points are interpreted as containing the x coordinate (in the upper half of the word) and the y coordinate (in the lower half of the word) of the point. A single word 32 bits long is sufficient to describe the coordinates of one point, as the kernel limits the tree to a depth of twelve (i.e. a 4096 by 4096 pixel image). Nodes that do not contain a data point are represented by the value WHITE.

Insertion of a point in a point data quadtree works as follows. First, we find the leaf that contains the point's location. If the leaf is empty, then the point's x and y coordinates are entered in the leaf's descriptor. Otherwise, the leaf is split into its four sons, the old leaf's point value is copied into the appropriate son, and insertion is re-attempted. Deletion of a point in a point data quadtree is a matter of finding the leaf that contains the point and then changing that leaf's descriptor to that of an empty leaf. Next, we must check to see if it is possible to merge the new empty leaf with its siblings.

The point data quadtree described above is termed a PR quadtree [9] and is also used in [8]. It differs from the point quadtree of Finkel and Bentley [3], in that the structure of the PR quadtree is independent of the order of point insertion. This is a result of the fact that PR quadtree leaves are always split into four congruent squares (conforming to an area quadtree decomposition). In contrast, the splitting points for the point quadtree are the data points themselves, thereby resulting in four rectangles that are not necessarily equal in size.

To store line data, we developed a variant of the edge quadtree of Shneier [10] restricted by our 32-bit node value field. Non-WHITE edge nodes contain exactly one line segment which intersects two of the node's edges. When inserting a new line into the tree, nodes which would not conform to this requirement are quartered and re-processed as appropriate. When two or more lines intersect,

the point of intersection will never contain only one line segment. Special consideration must therefore be made for single pixel nodes (in this case the intersect point). Such single pixel nodes may also result at the endpoints of a line segment that doesn't begin or end on a larger node's boundary.

The value field of the edge quadtree leaf descriptor has four subfields. The first subfield (one bit) indicates error values. The second subfield (one bit) indicates whether or not the node contains a line segment. The third subfield (two bits) tells for all non-WHITE nodes which son a node is with respect to its father. By setting this field, we guarantee that the leaf will not be automatically merged with its brothers by the kernel's insert routine. As this field is not set when the leaf value is WHITE, four empty quadrants are automatically merged together.

The fourth subfield (28 bits) of the value field of the edge quadtree's leaf descriptor contains different information depending on whether or not the leaf corresponds to a single pixel in the map. If the leaf corresponds to a single pixel, then the fourth subfield indicates how many lines pass through that pixel. Otherwise, non-WHITE nodes of a larger region contain exactly one line segment. We have 14 bits to encode each of the intercepts of the line with the edges of the block in which it is contained. We use two bits to indicate which of the four edges of the block the line intersects. The remaining 12 bits indicate the distance along the edge to the intercept. Thus we are able to handle maps containing blocks as large as 4096 by 4096 pixels.

Insertion and deletion algorithms for edge quadtrees are analogous to those of region or PR quadtrees. Insertion of a second line segment into a region described by a leaf that already contains one line segment causes the leaf to be quartered. The information that was in the original leaf is distributed among the new leaves, and the insertion attempt is repeated. Deletion of line segments is simply a matter of deleting all the information that is specific to that line segment. This means that nodes containing line segments are given the value WHITE and merged with their siblings if possible. Single pixel nodes have the number of lines passing through them decremented (with the value becoming WHITE when the number of lines becomes zero).

5. DATABASE FUNCTIONS

One of the basic functions of a geographic information system is to indicate the name of the class or polygon containing a given point. For some purposes, it is sufficient to describe a polygon by listing any point contained within it, and its class value. At times, it is necessary to be able to determine if two points which have the same class value are indeed within the same polygon. For this situation, the user can invoke a function which creates a unique polygon descriptor from a point. This function uses a modified version of the polygon-seed function used by the CHANGE function of Section 3. It examines all of the nodes in the polygon and determines which node has the lowest address. The pixel at the lower left corner of this node is used to describe the polygon. This is an expensive algorithm and should only be used when absolutely necessary.

The database language allows the formation of a map that corresponds to the extraction of a set of polygons from another map. This is achieved by the SUBSET function to which the user gives a list of classes and polygons. The SUBSET function first traverses the input tree, placing in the output tree any nodes whose value is that of a class on the list. Then for each polygon on the list, the polygon-seed function is performed, placing all nodes of the polygon into the output tree.

We have implemented functions that compute region properties such as area and perimeter. In addition, we can compute a minimum enclosing rectangle for a given subset of the map as well as extract a square window from the map. A list of all the classes or polygons in a map can be generated. As an example, such a list could be used to compute the area of every polygon on the map.

Point and line maps can also can be used in conjunction with some of these functions, although they may have slightly different definitions. Given the coordinate values of a point, functions are provided to indicate if it lies on a data point or line of the input map. The area of a point map is the number of points contained within it. The area of a line map is the length of the lines within it. A special regionsearch function is provided, similar to the window function, which yields a map containing all of the points within a given radius of a given point from the input point map. The window and enclosing rectangle functions may also be applied to point and line maps.

We have also implemented set operations such as union and intersection. Both union and intersection may be applied to any two maps of the same type (i.e. area, line, or point). In addition, a line or point map may be intersected with an area map, yielding a line or point map containing only those points or lines contained within the non-WHITE regions of the area map.

6. THE QUERY LANGUAGE

The query language provides an English-like keyword-based interface between the database user and the database system. It allows a non-programming oriented user to access the database with a more natural command language than LISP.

The query language is keyword-based. It operates by translating a query into LISP function calls, ignoring any words not in its vocabulary. This has the advantage that the user can insert noise words and phrases (e.g., articles like "the" and "an") to give the command a more natural appearance. This added flexibility is bought at the cost of more obscure error messages resulting from the misspelling of a keyword. In order to allow the user to customize his interface with the database, there are commands that allow keywords to be changed.

Table 1 presents a brief syntax of the query language in its present form. The *Please* command is used to learn about the system. The *Use* command changes the display device usage area. The *Measure* command lets the user indicate whether coordinates will have the referred map's lower left corner as origin, or use the global coordinate system's origin. The *Enter* command allows the user to inform the system of new data files. The *Display* command enables the display of a map on the display device. The *Let*, *Describe*, and *Forget* commands manipulate names of entities or keywords in the system - e.g., to rename items, describe items, or to remove items from the system, respectively. *Let* and *Forget* allow the user to name or forget a data item (e.g. assigning a name to a polygon description) as well as renaming keywords of the query language. *List* returns a list of polygons or classes from a map. *Edit* accesses the quadtree editor. *Move* displays a cursor at a given point on the display device.

One of the key features of the implementation of our query language is the ability to compose functions. Thus, where the *Display* command requires a map, this could be either a map name, or an expression which yields a map. For example, if we want to display the intersection of the landuse class map with the region below 100 feet elevation, it could be done with the following command:

> Display the intersection of land with the map formed from level1 in top on the Grinnell

where 'land' is the name of the landuse map, 'top' is the name of the topography map, 'level1' is the elevation class from 0 to 100 feet, and 'Grinnell' is the name of our display device.

7. CONCLUDING REMARKS

Our experience in developing a geographic information system based on quadtrees demonstrates that such a system is feasible. The potential advantage of using quadtrees, rather than conventional data structures, lies in the efficiency with which many types of queries can be handled. In its current state, our system can handle a wide range of queries. More capabilities will be added in the future. For the operations that we have implemented so far, we never use more than two input maps and one output map at the same time. However, the system places no restrictions on the number of maps which can be entered in the database. The map size is limited by the address space available, which is a function of the node size. In the current implementation, an individual map may not be larger than 4096 by

4096 pixels. Larger regions can be represented by breaking them up into smaller maps.

REFERENCES

1. J.L. Bentley, Multidimensional binary search trees used for associative searching, *Communications of the ACM 18*, 509-517(1975).

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

3. R.A. Finkel and J.L. Bentley, Quad trees: a data structure for retrieval on composite keys, *Acta Informatica 4*, 1-9(1974).

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

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

6. J.A. Orenstein, Multidimensional tries used for associative searching, *Information Processing Letters 14*, 150-157(1982).

7. A. Rosenfeld, H. Samet, C. Shaffer, and R.E. Webber, Application of hierarchical data structures to geographical information systems, Computer Science TR-1197, University of Maryland, College Park, MD (1982).

8. A. Rosenfeld, H. Samet, C. Shaffer, and R.E. Webber, Application of hierarchical data structures to geographical information systems phase II, Computer Science TR-1327, University of Maryland, College Park, MD (1983).

9. H. Samet, The quadtree and related hierarchical data structures, to appear in *ACM Computing Surveys*. See also Computer Science TR-1329, University of Maryland, College Park, MD (1983).

10. M. Shneier, Two hierarchical linear feature representations: edge pyramids and edge quadtrees, *Computer Graphics and Image Processing 17*, 211-224(1981).

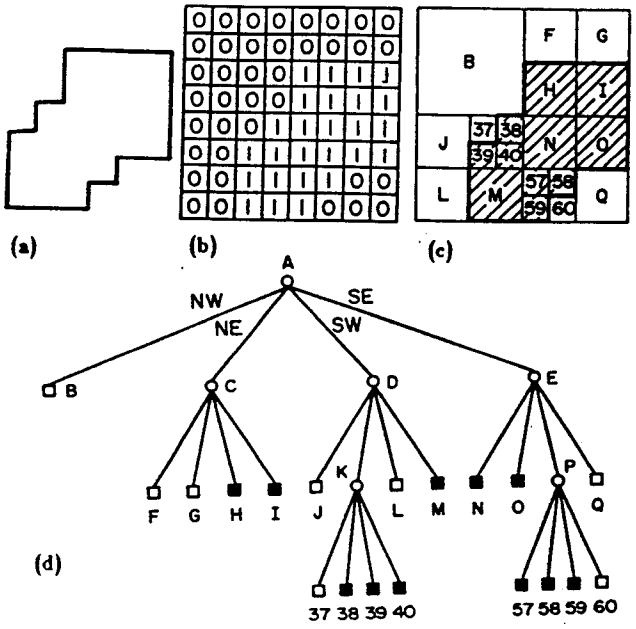


Figure 1. An image (a), its binary array (b), its maximal blocks (c), and the corresponding quadtree (d). Blocks in the image are shaded; background blocks are blank.

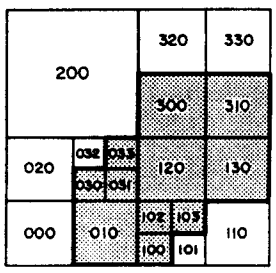
Commands:

```
Please {explain} <syntactic_unit> {}
Use {the Grinnell at} <window> {}
Measure {points from the lower left corner of} map {}
Measure {points from the} global {origin}
Enter <file_name> {into database}
Display <map> {on Grinnell}
Display <map> {on Grinnell starting from} <point> {}
Display {the} value {of} <number> {}
Let <name> {} denote {} <object> {}
Let <name> {} rename {} <map> {}
Describe {the type of this} <name> {}
Forget {about the meaning of this} <key_word> {}
List {all the} classes {on} <map> {}
List {all the} polygons {on} <map> {}
Edit {} <map> {with the database editor}
Move {to} <point> {}
```

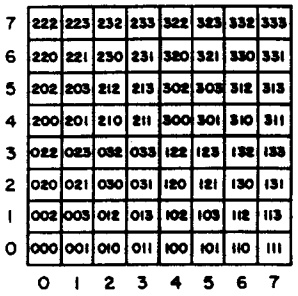
Other syntactic units:

```
<number> ::= {the} area {of} <map>
             {the} perimeter {of} <map>
<point> ::= {where x =} <number> {and y =} <number>
             {the point at the} cursor
<window> ::= <point> {extended} <number> {by} <number>
             {the smallest} window {for} <map>
<map> ::=    {the} intersection {of} <map> {with} <map>
             {the} union {of} <map> {with} <map>
             {the} windowing {of} <map> {with} <window>
             {the} map {formed from} <cplist> {in} <map>
<class> ::= {the} class {of} <poly>
             {the} class {at} <point> {on} <map>
<poly> ::=  {the} polygon {at} <point> {on} <map>
             {the} unique polygon {at} <point> {on} <map>
<cplist> ::= <a list of polygons and classes>
```

Table 1. The syntax of the query language. Words enclosed in curly braces {} are noise words and may be removed or replaced with any other non-keyword. Words enclosed in angle brackets <> are syntactic units, and are replaced by words or phrases matching their definition. In addition to a variable name or integer value which corresponds to the requested syntactic unit in a command, some syntactic units have further definitions as listed above. For example, where <number> is requested, a number may be typed. Alternatively, one of the two definitions given above for <number> may be used (with the first definition resulting in the area of the map, the second definition resulting in the perimeter of the map).



(a) A $2^3 \times 2^3$ grid with each pixel labelled with the (base four) value obtained by interleaving the y and x coordinates of the pixel.



(b) The block decomposition of the image from Figure 9a with each block labelled by the address of its lower left pixel.

Figure 2. An example demonstrating the use of locational codes to address blocks in an image represented by a quadtree.