CAR–TR–162
CS–TR–1578

DAAK70–81–C–0059
December, 1985

APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS: PHASE IV

Hanan Samet
Azriel Rosenfeld
Clifford A. Shaffer
Randal C. Nelson
Yuan-Geng Huang
Kikuo Fujimura

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

ABSTRACT

This document is the report for Phase IV of an investigation into the application of hierarchical data structures to geographical information systems. Subjects reported on include new representations for linear feature data, new techniques for reducing node accesses in algorithms for area data, and the extension of our attribute attachment system to point and linear feature data.

The support of the Engineer Topographic Laboratories under Contract DAAK70–81–C–0059 is gratefully acknowledged.

PREFACE

This report documents the research conducted under Phase IV of Contract DAAK70-81-C-0059/P00007. The report was prepared for the U.S. Army Engineer Topographic Laboratories (ETL), Ft. Belvoir, Virginia 22060. The Contracting Officer's Representative was Mr. Joseph A. Rastatter.

This report was prepared by Hanan Samet, Azriel Rosenfeld, Clifford A. Shaffer, Randal C. Nelson, Yuan-Geng Huang, and Kikuo Fujimura. The authors would like to thank Robert E. Webber for his help in computing the storage requirements reported in Section 5.1.

SUMMARY

This document is the final report for Phase IV of an investigation of the application of hierarchical data structures to geographical information systems, conducted under Department of the Army Contract DAAK70-81-C-0059/P00007. The purposes of this investigation were twofold: (1) to construct a geographic information system based on the quadtree hierarchical data structure, and (2) to gather statistics to allow the evaluation of the usefulness of this approach in geographic information system organization.

To accomplish the above objectives, in Phase I of the project a database was built that contained three maps supplied under the terms of the contract. These maps described the floodplain, elevation contours, and landuse classes of a region in California. The map regions were represented in quadtree form, and algorithms were developed for basic operations on quadtree-represented regions (set-theoretic operations, point-in-region determination, region property computation, and submap generation). The efficiency of these algorithms was studied theoretically and experimentally.

In Phase II of the project, a quadtree-based Geographic Information System was partially implemented, allowing manipulation of images which store area, point, and line data. This system included a memory management system to allow manipulation of images too large to fit into main memory, a software package to allow users to edit and update images, database management and map manipulation functions, and an English-like query language with which to access the database.

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

Phase IV of the project primarily dealt with developing new structures for storing linear feature data. The attribute attachment package was extended to point and linear feature data. Existing area map algorithms were improved to yield significant efficiency speedups by reducing node accesses. The efficiency of the linear quadtree was compared to that of the pixel array for computation of several important geographic functions.

The particular tasks reported on in this document are:

(a) *Memory management improvements.* The memory management system has been generalized for operation on a number of machines supporting the UNIX operating system. In the process, the maximum individual map size has been extended to 16,384 by 16,384 pixels. Nodes with duplicate addresses may now be stored, in order to support the new linear feature data structure.

(b) *Database enhancements.* Locating and updating nodes in a disk-based quadtree is a relatively expensive process, compared to in-core manipulations. By altering many

quadtree algorithms to reduce the number of node accesses necessary, significant efficiency improvements have been achieved.

(c) *Attribute attachment.* The attribute attachment package now handles area, point, and linear data maps in a uniform manner.

(d) *New linear feature representation.* The original linear feature representation (based on the edge quadtree) was deemed inadequate for use in the current system. A new implementation was devised, based on the PM quadtree.

(e) *Quadtree/array comparisons.* Many functions performed by the database system were implemented using arrays; timing results for these functions were gathered and compared to timings for our quadtree implementation.

TABLE OF CONTENTS

FIGURES

TABLES

ALGORITHMS

1. Introduction

This document reports on the current status of an ongoing effort to determine the suitability of applying a class of hierarchical data structures known as *quadtrees* [Klin71, Same84a] to the representation of cartographic data. Previous project reports are presented in [Rose82, Rose83, Same84b].

Section 2 describes new work done on the quadtree memory management system. Section 3 describes improvements to the quadtree database system algorithms. Section 4 discusses new work done on the attribute attachment system. Section 5 presents the new linear feature implementation. Section 6 contains our conclusions.

2. Enhancements to the quadtree memory management system

The quadtree memory management system, as described in Phase II of this project, is based on a structure termed the *linear quadtree*. The leaf nodes making up the quadtree of an image are stored in a list. Each leaf contains a 32 bit key; this key is used to order the node list. It is formed by bit interleaving the binary representation of the lower left x and y coordinates of the block represented by the leaf node. When sorted in ascending value of the key, the node list will be in order identical to that in which the leaves would have been visited by a preorder traversal of the original tree. Each leaf also contains a 32 bit value field. The linear quadtree allows for a reduction in storage as compared to pointer-based quadtrees which require each node to store four pointers to the children and a pointer to the father, in addition to the value field. The linear quadtree technique has been implemented in conjunction with a disk based memory management system which maintains only a small part of the image in core at one time. In our system, the sorted list of quadtree leaves is stored in a B^+-tree [Come79] with a page size of 1024 bytes, capable of holding up to 120 leaves in a page. For complete details, see the Phase II report.

In order to implement the line representation described in Section 6 of this report, a variable length node representation is required. This is represented in the linear quadtree by producing B-tree records with duplicate addresses to store separate pieces of information about the same quadtree block. This wastes some space since the information in the address field is repeated, but it avoids updating the pointer fields required by linked lists, or having to maintain an additional field if the address were only stored once for each block. More importantly, however, this method is compatible with our area and point representations with only minor modifications.

A variable length quadtree node is processed by locating the first B-tree record with the desired address, and then visiting successive records until one with a greater address is encountered. New user callable functions are provided for finding the n th B-tree record with a given address, for inserting a record with a given address into the tree, and for deleting a record with a given address and specified contents. The find function operates more efficiently for this implementation than it would were it operating on a linked list. This is because, after locating the first record with a given address, the n th item's position within the B-tree page can be calculated directly. Cases where multiple records with a given key are split between B-tree pages are uncommon since the average amount of information associated with a quadtree block in our application is small in comparison to that of the B-tree page.

A number of modifications were made to the kernel to allow compatibility with a wider variety of UNIX-based computers. Differing 16 and 32 bit word machines have different orderings for the bytes within a word; for example the SUN and the VAX have reversed byte orders. Previously, byte number 4 of the address field (the low order byte on a VAX) was reserved for the depth field. Since the low order byte is system dependent, this definition must be changed for compatibility. All address manipulations are now performed as mask and shift operations on unsigned integer values. This yields the added benefit of releasing 4 extra bits to the coordinate portion of the address since a maximum depth of 16 can be specified by a 4 bit depth field. Quadtrees in our system can now reach a depth of 14 (i.e., the maximum depth which can be represented in the 28 bit coordinate field), corresponding to an image of 16,384 by 16,384 pixels.

3. Enhancements to the quadtree database system

During Phase IV, several existing database functions were significantly improved by the implementation of new algorithms. These include the WITHIN function, the raster to quadtree conversion function, and the map windowing function. In addition, the set functions (e.g., union and intersection) were extended to work on unregistered images.

3.1. The WITHIN algorithm

In Phase III we presented an algorithm to generate a map which is BLACK at all pixels within a specified radius of the non-WHITE regions of an input map. Known as the WITHIN function, it is important for answering queries such as "Find all cities within 5 miles of the wheat growing regions". Such a query would be answered by calling the WITHIN function on a map containing wheat growing regions, and then intersecting the result with a map containing cities.

The algorithm presented in Phase III worked by expanding each non-WHITE block of the input image by R units (where R is the radius), and inserting all the nodes making up this expanded square into the output tree. This leads to many redundant node insertions. In addition, many of the nodes inserted are small, and are eventually merged together to form larger nodes.

A new algorithm is presented here which is based on the distance transform algorithm of Samet [Same82]. The algorithm does the following for each node of the input image. If the node is non-WHITE, it is inserted into the output map. If the node is WHITE, and less than or equal to $(R + 1)/2$ in width, then it must lie entirely within R pixels of a non-WHITE node. This is true because one of its siblings must contain a non-WHITE pixel. Thus, it is made BLACK and inserted into the tree. If the node is WHITE and has a width greater than $(R + 1)/2$, then its distance transform is computed. In other words, the exact distance from the node's border to the nearest non-WHITE pixel is determined. If this distance is such that the node is completely within radius R of a non-WHITE pixel, it is inserted as a BLACK node into the output tree. If the node is completely outside the radius, then it is inserted as WHITE. Otherwise, the node is quartered, and the process is continued for each quadrant.

The new algorithm is an improvement over the old one for two reasons. First, only large WHITE nodes need excessive computation. Since most nodes in a quadtree are small, very few nodes generate much work. Secondly, while input tree nodes may be visited several times when neighboring nodes compute their distance transform value, each block of the output tree is processed exactly once.

Table 3-1 compares execution times for the two algorithms on the images shown in Figures 3-1 and 3-2. The algorithm is computed for radius values of from 1 to 8. Timings for the old algorithm differ from those presented in Phase III due to improvements in the kernel and changes to our computer hardware. The algorithm is as follows.

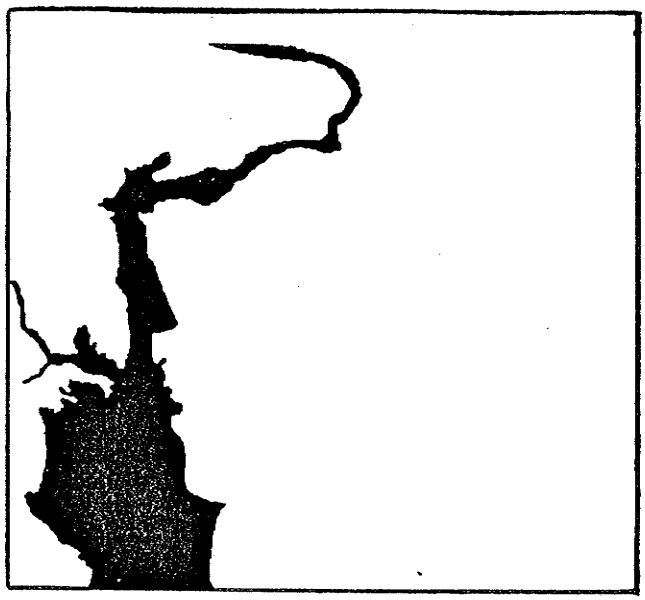


Figure 3-1. The floodplain map.

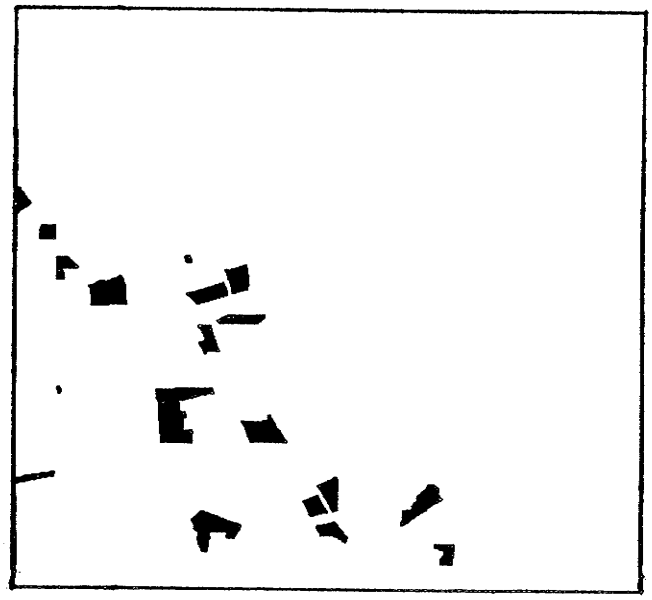


Figure 3-2. The ACC landuse class map.

```
procedure WITHIN(INMAP,OUTMAP,R);
/* Create a map OUTMAP which is BLACK at all pixels within R units of a
   BLACK pixel of INMAP. */
begin
  global pointer map INMAP, OUTMAP;
  global integer R;
  pointer node ND;

  for ND in INMAP do
   begin
    if VALUE(ND) ≠ WHITE then
      INSERT(OUTMAP,ND);
    else if WIDTH_OF(ND) ≤ (R+1)/2 then
     begin /* must be within radius */
      VALUE(ND) ← BLACK;
      INSERT(OUTMAP,ND);
     end;
    else COMPDIST(ND,X_OF(ND),Y_OF(ND),WIDTH_OF(ND));
   end;
end;


procedure COMPDIST(ND,X,Y,WIDTH);
/* Compute the distance transform of node ND which has lower left
   corner (X,Y) and width WIDTH */
begin
  value pointer node ND;
  value integer X,Y,WIDTH;
  global pointer map INTREE, OUTTREE;
  global integer R;
  pointer node Q;
  direction D;
  integer I,C,T;

  Q ← create(node);
  for D in {'N','E','S','W'} do
   begin
    COPY(ND,Q);
    FIND_NEIGHBOR(INTREE,Q,D,CCSIDE[D]);
    if Q ≠ NIL then
      if DEPTH(ND) < DEPTH(Q) then
       begin /* neighbor is GRAY */
        while DEPTH(ND) < DEPTH(Q) do
          Q ← FATHER(Q);
        DIST_NODE(Q,WIDTH_OF(Q),X,Y,WIDTH);
       end
      else if VALUE(Q) ≠ WHITE then
       begin
        T ← SCOMPARE(X_OF(Q),Y_OF(Q),WIDTH_OF(Q),X,Y,WIDTH);
```

```
        if T + WIDTH_OF(ND) ≤ R then  /* node within radius */
         begin
          VALUE(ND) ← BLACK;
          INSERT(OUTTREE,ND);
         end;
        else if T < R then  /* node beyond radius */
          SPLITDIST(X,Y,WIDTH,X_OF(Q),Y_OF(Q),WIDTH_OF(Q));
       end;
     COPY(ND,Q); /* do diagonal neighbor */
     Q ← FIND_DIAG_NEIGHBOR(INTREE,QUAD[D,CSIDE[D]]);
     if Q ≠ NIL then
      begin
       if DEPTH(ND) < DEPTH(Q) then
        begin
         while DEPTH(ND) < DEPTH(Q) do
          Q ← FATHER(Q);
         DIST_NODE(Q,WIDTH_OF(Q),X,Y,WIDTH);
        end;
       else if VALUE(Q) ≠ WHITE then
        begin
         T ← SCOMPARE(X_OF(Q),Y_OF(Q),WIDTH_OF(Q),X,Y,WIDTH);
         if T + WIDTH_OF(ND) ≤ R then /* node within radius */
          begin
           VALUE(ND) ← BLACK;
           INSERT(OUTTREE,ND);
          end;
         else if T < R then /* node beyond radius */
          SPLITDIST(X,Y,WIDTH,X_OF(Q),Y_OF(Q),WIDTH_OF(Q));
        end;
      end;
    end;
end;


procedure DIST_NODE(ND,WIDTH,X,Y,W);
/* Compute the closest distance from node ND with width WIDTH to a
   block with lower left corner (X,Y) and width W.  */
begin
  value pointer node ND;
  value integer WIDTH,X,Y,W;
  pointer node SON;
  pointer node TP;
  integer TEMP;

  SON ← create(node);
  COPY(ND,SON);
  if WIDTH_OF(SON) ≠ 1 then
    FIND(INMAP,SON ← SON_OF(SON,SW));
  else
```

```
    FIND(INMAP,SON);
  if WIDTH_OF(SON = WIDTH) then /* found the leaf node */
    if VALUE(SON) ≠ WHITE then
     begin
      TEMP ← SCOMPARE(X_OF(SON),Y_OF(SON),WIDTH_OF(SON),X,Y,W);
      if TEMP + W ≤ R then
       begin
        VALUE(SON) ← BLACK;
        INSERT(OUTTREE,SON);
       end;
      else if TEMP < R then
        SPLITDIST(X,Y,W,X_OF(SON),Y_OF(SON),WIDTH_OF(SON));
      return;
     end;
  else
   return;
  WIDTH ← WIDTH/2;
  if SCOMPARE(X_OF(ND),Y_OF(ND),WIDTH,X,Y,W) < R then
    DIST_NODE(SON_OF(COPY(ND,SON),SW),WIDTH,X,Y,W);
  if SCOMPARE(X_OF(ND)+WIDTH,Y_OF(ND),WIDTH,X,Y,W) < R then
    DIST_NODE(SON_OF(COPY(ND,SON),SE),WIDTH,X,Y,W);
  if SCOMPARE(X_OF(ND),Y_OF(ND)+WIDTH,WIDTH,X,Y,W) < R then
    DIST_NODE(SON_OF(COPY(ND,SON),NW),WIDTH,X,Y,W);
  if SCOMPARE(X_OF(ND)+WIDTH,Y_OF(ND+WIDTH,WIDTH,X,Y,W) < R then
    DIST_NODE(SON_OF(COPY(ND,SON),NE),WIDTH,X,Y,W);
end;


procedure SPLITDIST(GX,GY,GW,FX,FY,FW)
/* Compute the distance from the quadrants of the block represented
   by GX, GY, and GW to the block represented by FX, FY, and FW. */
begin
  value integer GX,GY,GW,FX,FY,FW;
  integer WIDTH,T;

  WIDTH ← GW/2;
  T ← SCOMPARE(GX,GY,WIDTH,FX,FY,FW);
  if T + WIDTH ≤ R then
    INSERT(OUTTREE,CREATE_NODE(GX,GY,LOG(WIDTH),BLACK));
  else if T < R then
    SPLITDIST(GX,GY,WIDTH,FX,FY,FW);
  T ← SCOMPARE(GX+WIDTH,GY,WIDTH,FX,FY,FW);
  if T + WIDTH ≤ R then
    INSERT(OUTTREE,CREATE_NODE(GX+WIDTH,GY,LOG(WIDTH),BLACK));
  else if T < R then
    SPLITDIST(GX+WIDTH,GY,WIDTH,FX,FY,FW);
  T ← SCOMPARE(GX,GY+WIDTH,WIDTH,FX,FY,FW);
  if T + WIDTH ≤ R then
    INSERT(OUTTREE,CREATE_NODE(GX,GY+WIDTH,LOG(WIDTH),BLACK));
```

```
  else if T < R then
    SPLITDIST(GX,GY+WIDTH,WIDTH,FX,FY,FW);
  T ← SCOMPARE(GX+WIDTH,GY+WIDTH,WIDTH,FX,FY,FW);
  if T + WIDTH ≤ R then
    INSERT(OUTTREE,
        CREATE_NODE(GX+WIDTH,GY+WIDTH,LOG(WIDTH),BLACK));
  else if T < R then
    SPLITDIST(GX+WIDTH,GY+WIDTH,WIDTH,FX,FY,FW);
end;


integer procedure SCOMPARE(X1,Y1,W1,X2,Y2,W2);
/* Find the chessboard distance between two squares (closest approach) */
begin
  value integer X1,Y1,W1,X2,Y2,W2;
  integer XDIST,YDIST;

  if X1 < Y1 then XDIST ← X2 - (X1 + W1);
  else XDIST ← X1 - (X2 + W2);
  if Y1 < Y2 then YDIST ← Y2 - (Y1 + W1);
  else YDIST ← Y1 - (Y2 + W2);
  return(max(XDIST,YDIST));
end;
```

Algorithm 3-1. The new WITHIN algorithm.

Table 3-1. Execution times for the WITHIN function.

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

3.2. A new quadtree building algorithm

The naive algorithm for converting a raster image to a linear quadtree is to insert individually 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. Previous algorithms presented in the literature [Same81, Rose82] have worked on this principle. Attempts at increasing efficiency concentrated on how to improve the insert routine. Algorithm 3-2, encoded by procedure NAIVE_BUILD and given below, demonstrates the naive method. Table 3-2 contains its execution times when applied to six test maps, corresponding to Figure 3-1 and Figures 3-3 to 3-6. The timings are nearly identical for raster images with the same number of pixels (i.e., node inserts), regardless of the number of nodes in the eventual quadtree. In other words, we see that the number of nodes in the output tree has little or no effect on the time required to perform the algorithm. Note that for the naive building algorithm, the amount of time needed to read the picture data is approximately 1% of the time necessary to insert every pixel.

Considering the large number of pixels in the raster representation of an image in comparison 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 insertions required. An optimal algorithm would, in the worst case, make a single insertion for each node in the quadtree. Algorithm 3-3 encoded by procedure OPTIMAL_BUILD and given below has this worst-case behavior. It is based on processing the image in raster-scan (top to bottom, left to right) order, always inserting the largest node for which the current pixel is the first (upper leftmost) pixel. Such a policy will avoid the necessity of merging since the upper leftmost pixel of any block is inserted before any other pixel of that block. Therefore, it is impossible for four sibling blocks to be of the same color.

At any point during the quadtree building process there is a processed portion of the image and an unprocessed portion. Both the processed and unprocessed portions of the quadtree have been assigned to nodes. We say that a node is *active* if at least one, but not all, pixel covered by the node has been processed. The optimal quadtree building process must keep track of all of these active nodes. Given a $2^n \times 2^n$ image, an upper bound on the number of active nodes is $2^n - 1$ as shown by the following theorem.

Theorem: Given a $2^n \times 2^n$ image, at any time during a raster-scan building process in which the largest node possible is always inserted, at most 2^n - 1 nodes will be active.

Proof: Any given pixel can be covered by at most n active nodes - i.e., a node at each level from 1 to n (corresponding to the root). At any given instant, there can be at most 2^{n-1} active nodes at level 1 (i.e., nodes of size 2×2). This is true because, for any given column, only 1 node at level 1 will be active, giving at most a solid line of 2×2 active nodes along a row just processed. In a like manner, there will be at most 2^{n-2} active nodes at level 2, and so on with 2^{n-i} active nodes at level i up to a single active node at level n (the root). Thus, there will be at most $\sum_{i=0}^{i=n-1} 2^i = 2^n$ - 1 active nodes. •

Using the above observation and theorem an optimal quadtree building algorithm is derived below. Assume the existence of a data structure which keeps track of the active

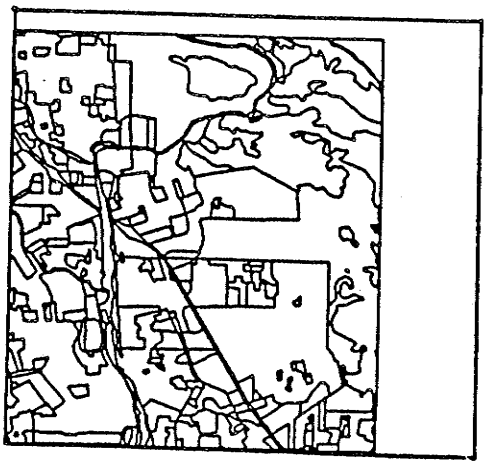


Figure 3-3. The landuse map.

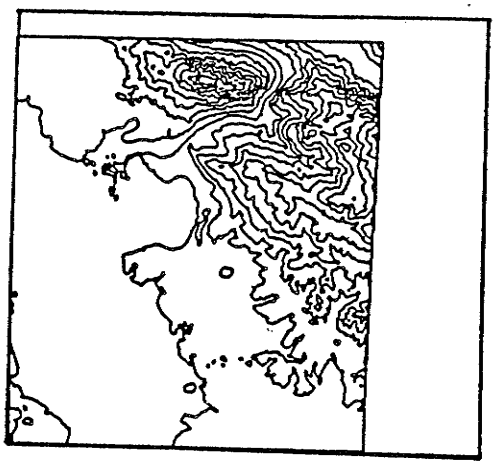


Figure 3-4. The topography map.

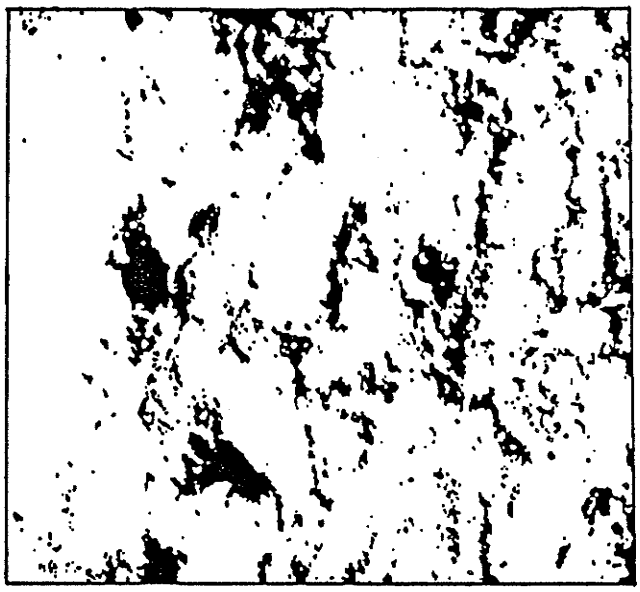


Figure 3-5. The stone image.

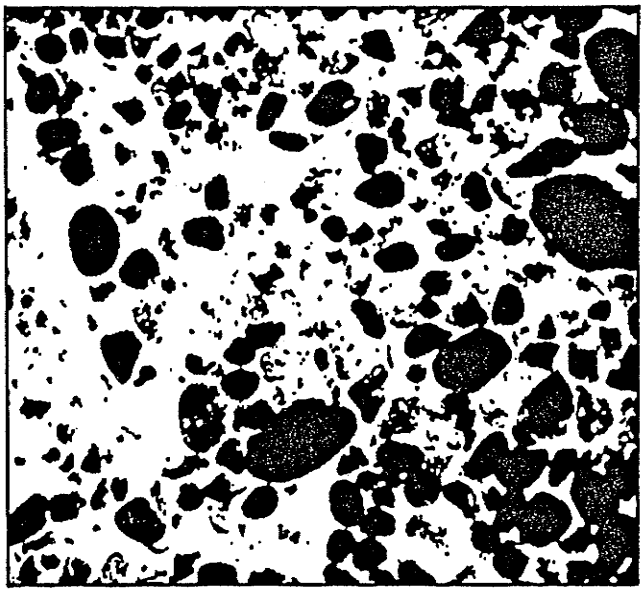


Figure 3-6. The pebble image.

```
procedure NAIVE_BUILD(INPIC,R,C,N,OUTTREE);
/* Build a quadtree in OUTTREE corresponding to input picture INPIC using a naive quad-
   tree building algorithm that inserts each pixel individually into the quadtree. The input
   picture has R rows and C columns and the output quadtree will be of maximum depth N
   (i.e., a 2^N × 2^N image). */
begin
 value pointer picture INPIC; /* INPIC points initially to the first row */
 value integer R,C,N;
 reference pointer quadtree OUTTREE;
 row BUFF[1:C];
 integer ROW, COL;

 for ROW←1 step 1 until R do
  begin /* Process each row of the picture in sequence */
   GET_ROW(INPIC,BUFF);
   for COL←1 step 1 until C do
    INSERT(OUTTREE,MAKE_NODE(COL,ROW,N,BUFF[COL]));
  end;
end;
```

Algorithm 3-2. The naive raster to quadtree algorithm.

Table 3-2. Naive quadtree building algorithm statistics.

Map Name	Num Nodes	Num Inserts	Time (secs.)
Floodplain	5266	180000	413.2
Topography	24859	180000	429.8
Landuse	28447	180000	436.7
Center	4687	262144	603.8
Pebble	44950	262144	630.1
Stone	31969	262144	629.5

quadtree nodes. For each pixel in the raster scan traversal, do the following. If the pixel is the same color as the appropriate active node, do nothing. Otherwise, insert the largest possible node for which this is the first (i.e., upper leftmost) pixel, and (if it is not a 1×1 pixel node) add it to the set of active nodes. Remove any active nodes for which this is the last (lower right) pixel. Algorithm 3-3, encoded by procedure OPTIMAL_BUILD, works in such a fashion. The list of active nodes is represented by a table, called TABLE, with a row for each level of the quadtree (except for level 0 which corresponds to the single pixel level; these nodes cannot be active). Row i of the table contains 2^{n-i} entries, with row n corresponding to the full image. Given a pixel in column j, the value of the active node at row i of the table is found at position $j/2^i$. Note that shift operations can be used instead of divisions if speed is important.

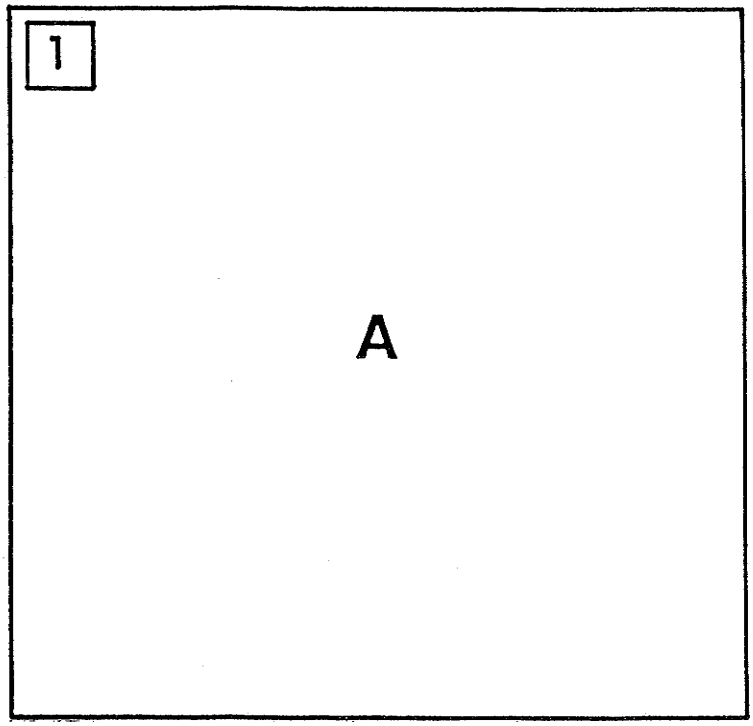
The only remaining problem is to locate the appropriate active node in the table which contains a given pixel. For a given pixel in a $2^n \times 2^n$ image, as many as n active nodes could exist. Multiple active nodes for a given pixel occur whenever a new node is inserted, as illustrated in Figure 3-7. Each pixel will have the color of the smallest of the active nodes which covers it, since the smallest node will have been the most recently inserted. Finding the smallest active node that contains a given pixel can be done by searching from the lowest level in the table upwards until the first non-empty entry is found. However, this is time consuming since it might require n steps. Therefore, an additional one-dimensional array, called LIST and referred to as the access array, is maintained to provide an index into TABLE. LIST is of size 2^{n-1} since single-pixel sized nodes need not be stored. For any pixel in column j, the LIST entry at $j/2$ indicates the row of TABLE corresponding to the smallest active node containing the pixel. At the beginning of the algorithm, each entry of LIST points to the entry of TABLE corresponding to the root (i.e., row n for a $2^n \times 2^n$ image). As active nodes are inserted or completed (and are to be deleted from the active node table), the active node table and the access array are updated.

Table 3-3 contains timing results when Algorithm 3-3 is applied to the same test maps as Algorithm 3-2. As indicated in Table 3-3, the optimal algorithm often requires far fewer calls to the insert routine than the number of nodes in the resulting output tree. This is because some calls to insert may cause several node splits to occur thereby increasing the number of nodes in the tree. This is all accomplished by procedure INSERT whose code is not given here as it is implementation-dependent. For example, consider Figure 3-8e where node B (from Figure 3-8a) is replaced by node B1 with a new value along with three other nodes (B2, B3, and B4) retaining B's value. However, only one new active node is created (B1) as the remaining pixels are still covered by the original active node (B). When it comes time to process the pixels covered by those blocks which are artifacts of the splitting process (B2, B3, and B4)), these pixels may have the same value as B, and thus no additional insertion is required. As another example, in Figure 3-10 inserting node B into the quadtree containing a single node causes seven nodes to result. If the first pixel inserted into node X happens to be the same color as the original node (A of Figure 3-10a), no insertion is required.

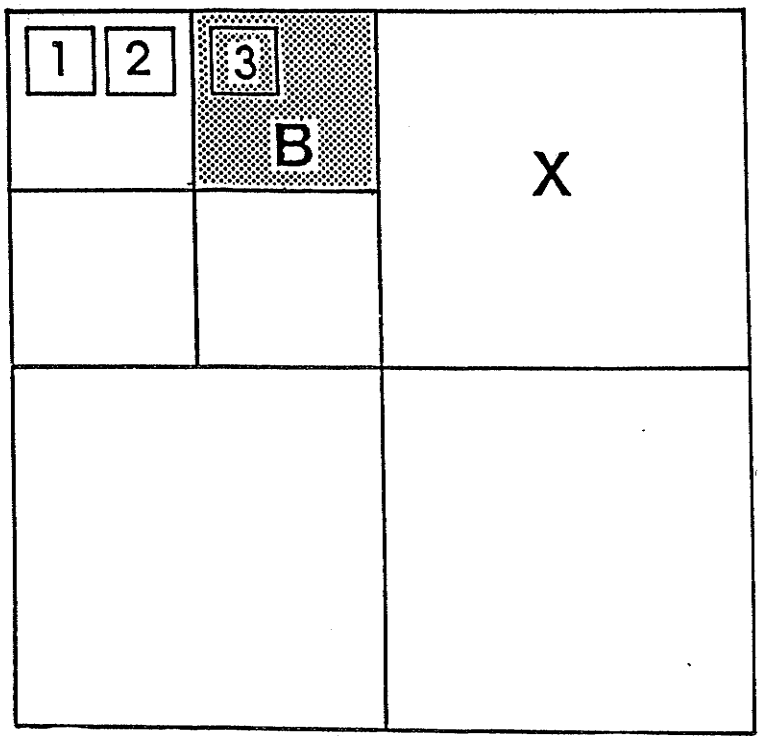
Table 3-3. Optimal quadtree building algorithm statistics.

Map Name	Num Nodes	Num Inserts	Time (secs.)
Floodplain	5266	2352	13.8
Topography	24859	12400	51.2
Landuse	28447	14675	56.9
Center	4687	2121	16.1
Pebble	44950	20770	111.0
Stone	31969	14612	70.2

As an example of how the optimal quadtree building algorithm works let us consider how the quadtree corresponding to the image of Figure 3-9 is constructed. Table 3-4 traces the active nodes at each stage of execution. Each row in Table 3-4 lists the active nodes after the given pixel has been processed. The "+" prefix indicates that the designated node has become active when the corresponding pixel was processed. When the first pixel of the array is

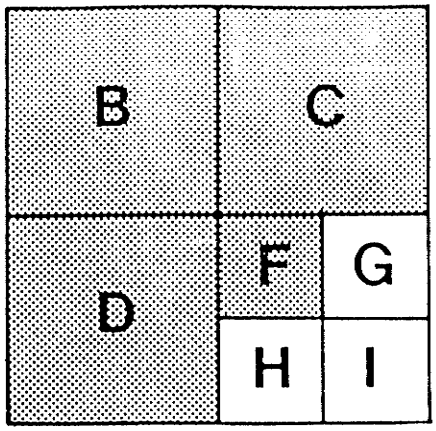


(a) Node A is active after inserting a single pixel of color *C*.

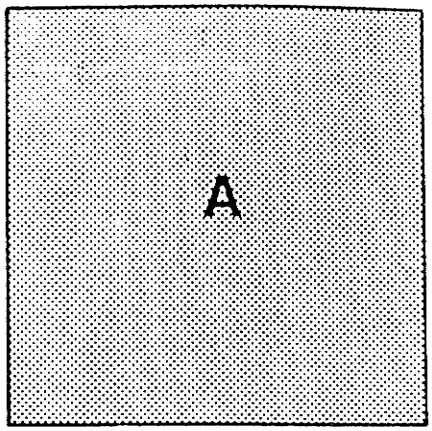


(b) The first two pixels have color *C*. Pixel 3 has color *D*. Its insertion creates active node *B*. Node *A* is still active.

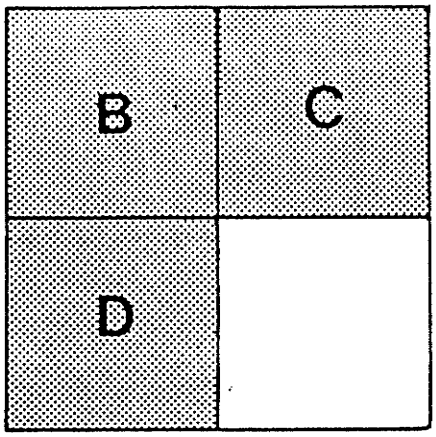
Figure 3-7. Node insertion can create multiple active nodes.



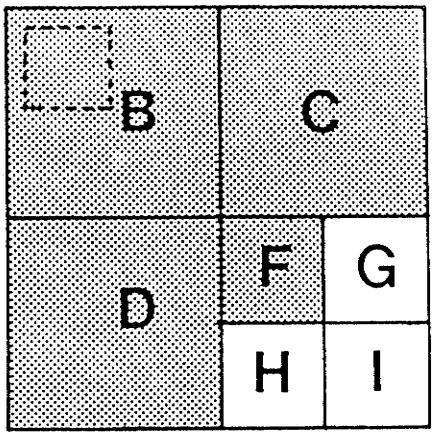
(a) An example of block decomposition.



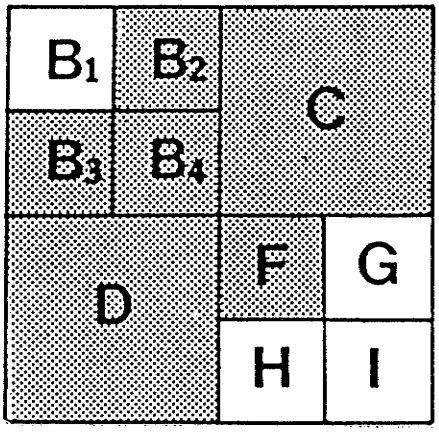
(b) The result of inserting a large BLACK node into (a): merging occurs.



(c) The result of inserting a WHITE node at node F of (a): F, G, H, and I merge.

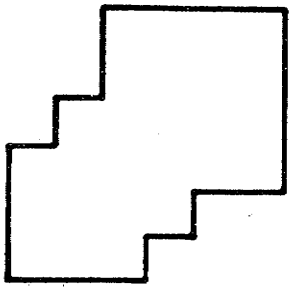


(d) The result of insertng a small BLACK node in the upper left corner of node A in (a): no splitting occurs.

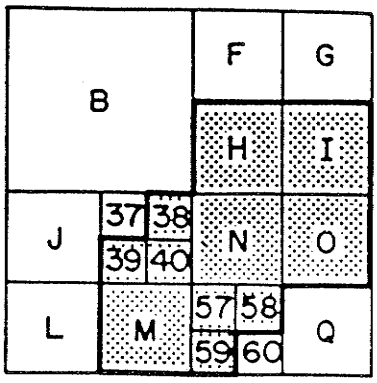


(e) The result of inserting a small WHITE node into the upper left corner of node A in (a): splitting occurs.

Figure 3-8. The effects of node insertion.

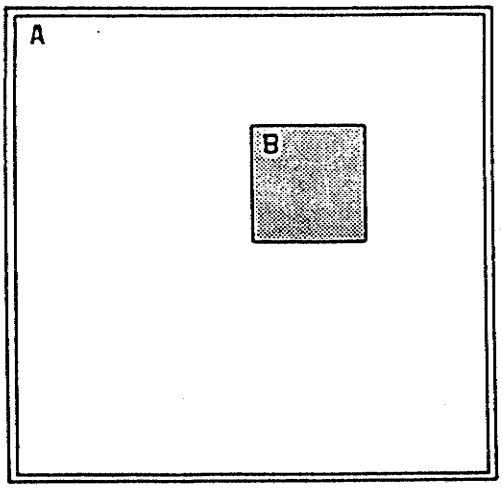


(a) A region

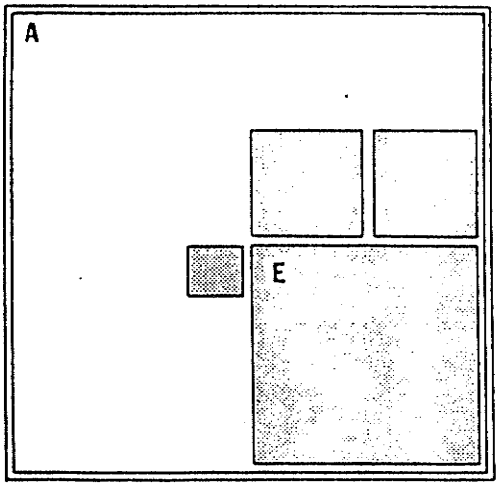


(b) Block decomposition of the region in (a). Blocks in the region are shaded.

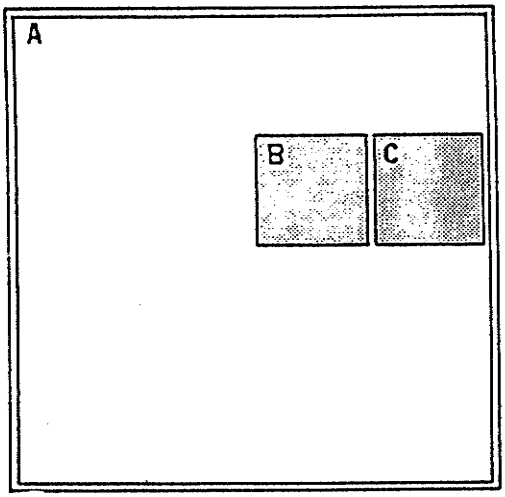
Figure 3-9. A region and its block decomposition.



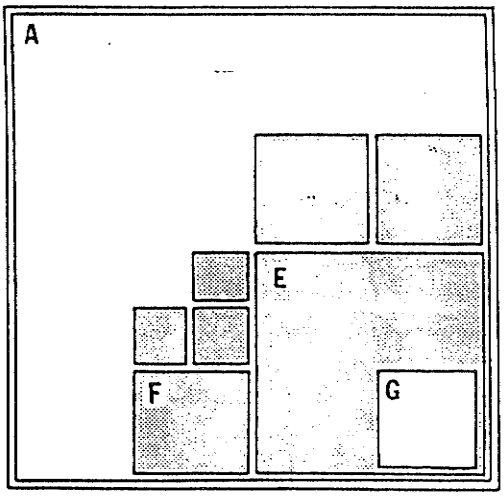
(a) State after processing pixel (2, 4).



(c) State after processing pixel (4, 4).



(b) State after processing pixel (2, 6).



(d) State after processing pixel (6, 6).

Figure 3-10. The construction process for the region in Figure 3-9.

```
procedure OPTIMAL_BUILD(INPIC,R,C,N,OUTTREE);
/* Build a quadtree in OUTTREE corresponding to input picture INPIC using a quadtree
   building algorithm that inserts the largest node for which the current pixel is the first (i.e.,
   upper leftmost) pixel. The input picture has R rows and C columns and the output quad-
   tree will be of maximum depth N (i.e., a 2^N × 2^N image). Type row is an array of type
   color where color takes on the values NOCOLOR, BLACK, and WHITE. */
begin
 value pointer picture INPIC;
 value integer R,C,N;
 reference pointer quadtree OUTTREE;
 row BUFF[0:C-1];
 pointer row array TABLE[1:N];
 integer array LIST[0:2↑(N-1)-1];
 integer ROW,COL,I,J,DEPTH,XT,YT,T;

 /* Allocate space and initialize TABLE for description of active nodes: */
 for I←1 step 1 until N do
  begin
   TABLE[I]←ALLOCATE(row,2↑(N-I));
   for J←0 step 1 until 2↑(N-I)-1 do
    TABLE[I][J]←'NOCOLOR';
  end;
 TABLE[N][0]←'WHITE';
 for J←0 step 1 until 2↑(N-1)-1 do LIST[J]←N;
 /* Process the picture: */
 for ROW←0 step 1 until R-1 do
  begin
   GET_ROW(INPIC,BUFF); /* Process one row at a time */
   for COL←0 step 1 until C-1 do /* Process each pixel in the row */
    begin
     I←LIST[COL/2]; /* Find the smallest active node containing the pixel */
     if TABLE[I][COL/2↑I] NEQ BUFF[COL] then
      begin /* The pixel and the node containing it differ in color */
       /* Calculate the depth of the largest node for which this is the first pixel: */
       XT←COL; YT←ROW; DEPTH←0;
       while EVEN(XT) and EVEN(YT) and (DEPTH<N) do
        begin
         XT←XT/2; YT←YT/2; DEPTH←DEPTH+1;
       end;
       if DEPTH NEQ 0 then
        begin /* The largest node containing the pixel is larger than 1 × 1 */
         /* Update the active node table and the access array */
         TABLE[DEPTH][COL/(2↑DEPTH)] ← BUFF[COL];
         for J←COL/2 step 1 until COL/2+2↑(DEPTH-1)-1 do LIST[J]←DEPTH;
        end;
       INSERT(OUTTREE,MAKENODE(COL,ROW,DEPTH,BUFF[COL]));
       if mod(ROW+1,2↑I)=0 and mod(COL+1,2↑I)=0 then
```

```
          begin /* The last pixel of one or more active nodes */
           while mod(ROW+1,2↑I)=0 and mod(COL+1,2↑I)=0 do
            begin /* Update the active node table */
             TABLE[I][COL/2↑I]←'NOCOLOR'; I←I+1;
            end;
           T←I-1;
           while TABLE[I][COL/(2↑I)]='NOCOLOR' do
            I←I+1; /* Get level of next active node */
           for J←COL/2 step -1 until COL/2-2↑(T-1)+1 do
            LIST[J]←I; /* Update access array */
          end;
        end;
      end;
    end;
end;
```

Algorithm 3-3. The optimal raster to linear quadtree algorithm

---

processed, the entire quadtree is represented by a single WHITE node (block A in Figure 3-10a). No other insertions occur while processing rows 0 and 1. When the first BLACK pixel (2,4) is processed, block B of Figure 3-10a becomes active. When BLACK pixel (2,5) is processed, block B will be located in the active node table, since it is the smallest active node containing that pixel. When BLACK pixel (2,6) is processed, block C of Figure 3-10b becomes active, since only active WHITE block A contains it at that point. As row 3 is processed, blocks B and C are deactivated when their lower right pixels are processed. When pixel (4,4) is processed, the state is as shown in Figure 3-10c. The blocks previously labeled B and C are not active. Pixel-sized block D at (4,3) is not active since it contains no unprocessed pixels. Blocks A and E are, therefore, the only active blocks. Figure 3-10d shows the state of the algorithm when pixel (6,6) has been processed. Block F became active after processing pixel (6,2). Since the smallest block containing pixel (6,6) had been BLACK, a new WHITE block has been activated, block G. Thus, three active blocks (i.e., A, E, and G) contain pixel (6,7), with the smallest being block G. As the final row is processed, all active nodes will be deactivated.

Table 3-4. Trace table for active nodes in building example.			
Pixel	Level 3	Level 2	Level 1
(0,0)	+A (0,0) WHITE		
(2,4)	A (0,0) WHITE		+B (2,4) BLACK
(2,6)	A (0,0) WHITE		B (2,4) BLACK +C (2,6) BLACK
(3,5)	A (0,0) WHITE		C (2,6) BLACK
(3,7)	A (0,0) WHITE		
(4,4)	A (0,0) WHITE	+E (4,4) BLACK	
(6,2)	A (0,0) WHITE	E (4,4) BLACK	+F (6,2) BLACK
(6,6)	A (0,0) WHITE	E (4,4) BLACK	F (6,2) BLACK +G (6,6) WHITE
(7,3)	A (0,0) WHITE	E (4,4) BLACK	G (6,6) WHITE
(7,7)			

In order to understand why Algorithm 3-3 is such an improvement over Algorithm 3-2, let us analyze the cost of both algorithms in terms of the number of insert operations that they perform. Algorithm 3-2 examines each pixel and inserts it into the quadtree. Assuming a cost of I for each insert operation, and a cost of c for the time spent examining a pixel, the total cost is then $2^{2n} \cdot (c + I)$. Algorithm 3-3 must also examine each pixel. However, there will be at most one insert operation for each of the N nodes in the output quadtree. Therefore, the cost of Algorithm 3-3 is $c \cdot 2^{2n} + I \cdot N$ where c is relatively small in comparison to I, and N is usually small in comparison to 2^{2n}. In other words, the quantity $I \cdot N$ dominates the cost of Algorithm 2. The result is that using Algorithm 3-3 reduces the execution time from being O(pixels) to O(nodes). Of course, this is achieved at the cost of an increase in storage requirements due to the need to keep track of the active nodes (approximately 2^{n+1} for a $2^n \times 2^n$ image).

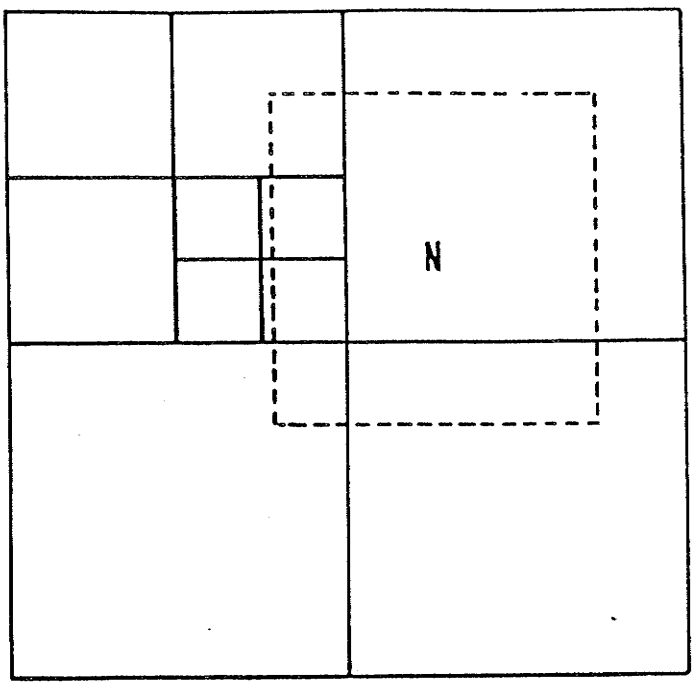
3.3. Set operations for unregistered maps

In many applications, including geographic information systems, it is desirable to compute set operations on a pair of images. For example, suppose a map is desired of all wheatfields above 100 feet in elevation. This can be achieved by intersecting a wheatfield map and an elevation map whose pixel values are BLACK if they represent an area whose elevation is above 100 feet. The resulting map would have BLACK pixels wherever the corresponding pixels of the input maps were both BLACK. In this section we will consider only the case of map intersection, although other set functions such as union or difference can be handled in an analogous manner.

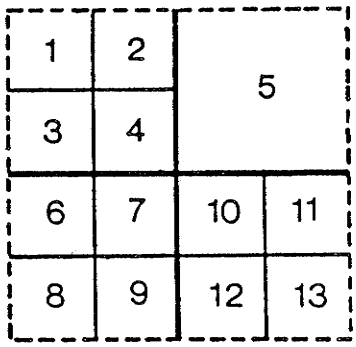
Intersection of quadtrees representing images with the same grid size, same map size, and same origin is quite straightforward. A simple traversal of both trees is performed in parallel; each node of the first image is compared with the corresponding node(s) in the second image. Algorithm 3-4, named REG_INTERSECT illustrates this procedure. On the other hand, very little work has been done on set operations between unregistered quadtrees (i.e., quadtrees which have the same grid size and map size, but differing origins). In particular, the only prior mentions of algorithms for intersecting unregistered quadtrees involved translating one of the images to be in registration with the other, and then performing registered intersection [Garg83]. In this section, an optimal algorithm for unregistered map intersection is presented. By optimal, we mean that each node of the input images is visited only once, and at most one insertion into the output tree is performed for each output tree node.

As with the quadtree building algorithm of Section 3.2, the intersection algorithm maintains a table of the active output tree nodes to minimize insertions into the output tree. We will call this table OUTTABLE. Unlike the building algorithm, there are two input quadtrees (call them I1 and I2) to be considered as well. The basic algorithm is as follows. I1 is processed in depth-first traversal order. For each node N of I1, the various nodes of I2 which cover N are located. In order to minimize the number of node searches made in the tree I2, a second table will be used to keep track of the active nodes of I2 (i.e., those nodes of I2 which have been partially, but not completely, processed). Starting with the upper left pixel of N, the node of I2 which covers that pixel is located. Next, the largest block contained within both nodes is computed. The set function is evaluated on the values of these two nodes, and OUTTABLE is queried to determine if the new node should be inserted. This step is repeated on subsequent portions of N in key order (i.e., Morton sequence order on the pixels of node N) until all pixels of N are processed. Figure 3-11 provides an example.

OUTTABLE is easily implemented, since nodes will always be inserted in Morton sequence order (matching the progress made in tree I1). During the traversal of the output tree, the second, third, and fourth subquadrants of a block at level i will not be processed until the previous subquadrants are completed (e.g., the SW subquadrant will not be processed until the NW and NE subquadrants are complete). Thus, at most one node at each level of the tree can be active; for a 2^n by 2^n image, a table of only n entries is needed to represent the active nodes. Each entry of OUTTABLE contains the location and value of the current active node at the corresponding level, along with a field to indicate the quadrant relative to its father in which the node lies. In addition, a variable is needed to keep track of the current depth.



(a) Node N (shown in dashed lines) is intersected with an image.



(b) The decomposition and order of processing for node N as directed by the image decomposition.

Figure 3-11. An example of intersection. A node N from the first input tree is processed by comparing it against those nodes which it intersects in the second input tree.

```
node procedure REG_INTERSECT(INTREE1,INTREE2)
begin
 value pointer quadtree INTREE1, INTREE2;
 node pointer N;

 if(WHITE(INTREE1) or (WHITE(INTREE2))) then
   return(CREATENODE(WHITE));
 if(BLACK(INTREE1)) then
   return(COPYTREE(INTREE2));
 if(BLACK(INTREE2)) then
   return(COPYTREE(INTREE1));
 N ← CREATENODE(GRAY);
 for I in {'SW','SE','NW','NE'} do
   SON(ND,I) ← INTERSECT(SON(INTREE1,I),SON(INTREE2,I));
 if(WHITE(SON(N,'NW')) and WHITE(SON(N,'NE')) and
     WHITE(SON(N,'SW')) and WHITE(SON(N,'SE'))) then
   return(CREATENODE(WHITE));
 return(N);
end;
```

Algorithm 3-4. Registered intersection of two images represented by quadtrees.

The final requirement for the non-registered set function algorithm is a method for keeping track of the active nodes of I2. First, consider the border of the nodes of I1 which have been processed at any given instant (referred to as the *active border*). Since these nodes are processed in Morton sequence order, the active border will be in the form of a staircase (see Figures 3-12 and 3-13). The active border, as it crosses an output map of size 2^n by 2^n, will form horizontal and vertical segments such that the sums of the horizontal and vertical segments will each be 2^n pixels in length. The active nodes of I2 will be those nodes which, at any given instant, straddle the active border.

The active border table used by the intersection algorithm is a modification of the active border table described by Samet and Tamminen [Same84e]. For their purpose, it was necessary only to store at each position along the border the color of the processed node corresponding to that position, and the length of its border. For the intersection algorithm, we must maintain in the active border table a complete description of each active node. The table will be composed of two node arrays named X_EDGE and Y_EDGE, each 2^n records long. Each record has four fields: X, Y, NODETYPE, and SIZE.

The non-registered intersection algorithm works as follows. For each node N of I1, the function DOSET is called to perform the following. Beginning with (CX, CY) corresponding to the upper left corner, and continuing to the lower right corner, the node will be processed in

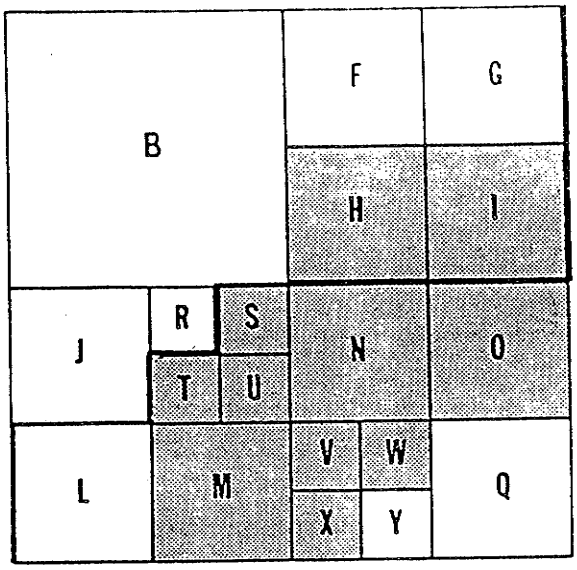


Figure 3-12. The active border during a traversal after processing block R.

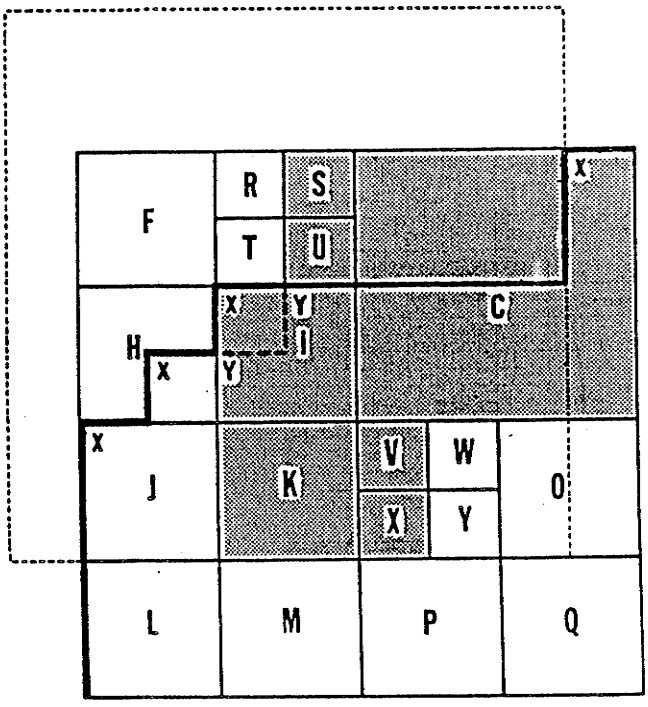


Figure 3-13. The active border (shown in heavy line) after processing block R of Figure 3-12 (shown by dotted lines) as it intersects the second input tree. After node S (of Figure 3-12) is processed, the active border will advance as shown by the dashed line. Note that only sections marked by an "X" are eligible for processing at this time.

pieces, as determined by the size and position of those nodes of I2 corresponding to the pixels covered by N. For each such piece, tables X_EDGE[CX] and Y_EDGE[CY] are checked to determine if the records they store contain (CX, CY). If one table, but not the other, contains the node covering (CX, CY), this record is copied to the deficient table. If neither table contains a record covering (CX, CY), the appropriate node in I2 is located, and the node descriptor containing (CX, CY) is inserted into both tables. This is the only updating of the edge table required by the algorithm to insure that no node of I2 need be located more than once. To understand that this is true, notice the way in which the nodes (or pieces of a node) of I1 are processed. At any given moment, only those pixels in the corner formed by two segments of the active border can be processed (i.e., the X marks in nodes C, H, J, and I of Figure 3-13). Assuming that both X_EDGE[CX] and Y_EDGE[CY] have the correct node descriptor after (CX, CY) has been processed, the new active border will form at most two new angle positions - one at (CX, CY+WIDTH) and the other at (CX+WIDTH, CY) for a processed node portion of width WIDTH (e.g., at positions marked Y in Figure 3-13). When pixel (CX, CY+WIDTH) is processed, X_EDGE[CX] will still contain the same record, since no other pixel with X value CX will be processed in the meantime. Similar reasoning applies to pixel (CX+WIDTH, CY) and Y_EDGE[CY].

Once the node of I2 containing (CX, CY) is located, say N_2, it is compared with node N to generate the largest block with upper left corner (CX,CY) contained in both N and N_2. The set operation (in this case, intersection) is then performed on N and N_2. This value is then compared with the current active node of the output tree, as maintained in OUTTABLE. If the values are the same, no insertion into the output tree is necessary. If they are different, the largest node for which (CX, CY) is the upper-left corner is inserted. Finally, OUTTABLE is updated, along with the values of CX and CY to determine the next part of node N to be processed.

Algorithm 3-5, named INTERSECT, encodes the unregistered intersection algorithm described above. To be precise, this algorithm produces an image whose value is WHITE at those pixels where either the corresponding value of I2 is WHITE or the pixel is beyond the borders of either image; the value is that of the corresponding position in I1 for all other pixels. As an example of the unregistered intersection algorithm, Table 3-5 shows the contents of active border tables X_EDGE and Y_EDGE after processing certain pixels of node N from Figure 3-11. N is assumed to be the first (upper-leftmost) block of map I1. In Table 3-5, those records marked with an asterisk (*) indicate a node which has been located in I2. A minus mark (–) indicates a position which did not contain a record covering current pixel (CX, CY); a plus mark (+) indicates the record from the other table which did contain the current pixel and which was therefore copied. Columns two and three are of the format position: record where the record consists of the X, Y, and WIDTH descriptors for the node from I2 being stored at that position. All coordinates are relative to I1.

```
/* Calculate the intersection of two
   input maps, producing an output map of size and position identical to
   that of the first map. */

pointer node array X_EDGE, Y_EDGE;
integer OFFSETX, OFFSETY;
integer DIFFX[4] ← {0,1,-1,1}; /* difference in x coords moving to quad i from i-1 */
integer DIFFY[4] ← {0,0,1,0}; /* difference in y coords moving to quad i from i-1 */

structure BLOCK
 begin
  integer X_OF, Y_OF, WQUAD, VALUE;
 end;

/* output table - initialize to a single WHITE node */
BLOCK array OUTTAB[0:DEPTH_OF(OUTTREE)] ← {0,0,'SW',WHITE};
/* set to current depth (largest node for current position) in outtab */
integer OUTDEPTH ← DEPTH_OF(OUTTREE);



procedure INTERSECT(IN1TREE,IN2TREE,OUTTREE);
begin
 global pointer quadtree IN1TREE;
 global pointer quadtree IN2TREE;
 global pointer quadtree OUTTREE;
 global node array X_EDGE[WIDTH_OF(IN2TREE)];
 global node array Y_EDGE[WIDTH_OF(IN2TREE)];
 pointer node I;

 OFFSETX = X_OF(IN2TREE) - X_OF(IN1TREE);
 OFFSETY = Y_OF(IN2TREE) - Y_OF(IN1TREE);
 for each I in IN1TREE do
   DOSET(I);
end;


integer procedure DOSET(IN1NODE)
/* For each node of IN1TREE, break it into pieces matching the nodes
   of IN2TREE (active nodes are stored in X_EDGE and Y_EDGE),
   perform SET_FUNC on the node pieces, and insert the result into
   OUTTREE through function NEW_INSERT. */
begin
 value pointer node IN1NODE;
 integer INX, INY, INWID;
 integer STOPX, STOPY;
```

```
pointer node IN2NODE, OUTNODE;
integer CX, CY, MAXDEPTH, VALUE;

CX ← INX ← X_OF(OUTTAB[OUTDEPTH]);
CY ← INY ← Y_OF(OUTTAB[OUTDEPTH]);
INWID ← WIDTH_OF(IN1NODE,WIDTH_OF(IN1TREE));
STOPX ← INX + INWID;
STOPY ← INY + INWID;
while((CX ≠ STOPX) and (CY ≠ STOPY)) do
 begin
  if(((Y_OF(X_EDGE[CX]) + WIDTH_OF(X_EDGE[CX])) ≤ CY) and
    ((X_OF(Y_EDGE[CY]) + WIDTH_OF(Y_EDGE[CY])) ≤ CX)) then
   begin /* locate new node in the tree */
    FIND(IN2TREE,MAKE_NODE(IN2NODE,CX-OFFSETX,CY-OFFSETY,0));
    X_OF(X_EDGE[CX]) ← X_OF(Y_EDGE[CY]) ← X_OF(IN2NODE) + OFFSETX;
    Y_OF(X_EDGE[CX]) ← Y_OF(Y_EDGE[CY]) ← Y_OF(IN2NODE) + OFFSETX;
    VALUE(X_EDGE[CX]) ← VALUE(Y_EDGE[CY]) ← VALUE(IN2NODE);
    WIDTH_OF(X_EDGE[CX]) ← WIDTH_OF(Y_EDGE[CY])
                ← WIDTH_OF(IN2NODE);
   end;
  else if((Y_OF(X_EDGE[CX]) + WIDTH_OF(X_EDGE[CX])) ≤ CY) then
    X_EDGE[CX] ← Y_EDGE[CY];
  else if((X_OF(Y_EDGE[CY]) + WIDTH_OF(Y_EDGE[CY])) ≤ CX) then
    Y_EDGE[CY] ← X_EDGE[CX];

  /* compute the biggest block starting at (CX,CY) contained within both
     input nodes */
  MAXDEPTH ← COMPUTE_DEPTH(CX, CY, STOPX, STOPY,
          X_OF(X_EDGE[CX]) + WIDTH_OF(X_EDGE[CX]),
          Y_OF(X_EDGE[CX]) + WIDTH_OF(X_EDGE[CX]));
  /* attempt to insert the result into the output tree */
  VALUE ← SET_FUNC(VALUE(IN1NODE),VALUE(X_EDGE[CX]));
  if(VALUE(INTAB[INDEPTH]) ≠ VALUE) then
    /* insert node into output file */
    INSERT(OUTTREE,MAKE_NODE(X_OF(OUTTAB[OUTDEPTH]),
                  Y_OF(OUTTAB[OUTDEPTH]),OUTDEPTH));

  /* Update input/output table */
  while(MAXDEPTH > OUTDEPTH) do
   begin
    OUTDEPTH ← OUTDEPTH - 1;
    X_OF(OUTTAB[OUTDEPTH]) ← X_OF(OUTTAB[OUTDEPTH-1]);
    Y_OF(OUTTAB[OUTDEPTH]) ← Y_OF(OUTTAB[OUTDEPTH-1]);
    WQUAD(OUTTAB[OUTDEPTH]) ← 'SW';
    VALUE(OUTTAB[OUTDEPTH]) ← VALUE;
   end;
  while(WQUAD(OUTTAB[OUTDEPTH]) = 'NE')
    OUTDEPTH ← OUTDEPTH + 1;
  WQUAD(OUTTAB[OUTDEPTH]) ← WQUAD(OUTTAB[INDEPTH]) - 1;
```

```
    CX ← X_OF(OUTTAB[OUTDEPTH]) ← X_OF(OUTTAB[OUTDEPTH]) +
         DIFFX[WQUAD(OUTTAB[OUTDEPTH])] * 2↑OUTDEPTH;
    CY ← Y_OF(OUTTAB[OUTDEPTH]) ← Y_OF(OUTTAB[OUTDEPTH]) +
         DIFFY[WQUAD(OUTTAB[OUTDEPTH])] * 2↑OUTDEPTH;
  end;
end;


integer procedure COMPUTE_DEPTH(X1,Y1,X2,Y2,X3,Y3)
/* Compute the depth of the largest node with upper left corner (X1,Y1)
   which does not contain (X2,Y2) or (X3,Y3). */
begin
 value integer X1,Y1,X2,Y2,X3,Y3;
 integer MAX, I;

 MAX ← min(min(X2-X1,Y2-Y1),min(X3-X1,Y3-Y1));
 for I ← 0 step 1 until 2↑I > MAX do;
 /* I now one size too big */
 return(I-1);
end;


integer procedure SET_FUNC(V1,V2)
begin
 integer V1, V2;

 if V1 = WHITE then return(WHITE);
 else return(V2);
end;
```

Algorithm 3-5. Unregistered intersection of two images represented by quadtrees.

Table 3-5. Trace table for active nodes in intersection example.

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

3.4. Windowing

Another function commonly available in geographic information systems 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 off the edge of the image. Most importantly, the origin (or lower left corner) of the window could potentially be anywhere in relation to the origin of the input map. This means that large blocks from the input quadtree must be broken up, and possibly recombined into new blocks in the output quadtree.

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

If windowing is viewed as a set function on two unregistered images, an optimal algorithm can be derived from Algorithm 3-5. By optimal, we mean that each node of the input and ouput trees is located/inserted at most once. In particular, in the worst case a single insertion is performed for each node in the output tree. Let I1 be a BLACK block with the same size and origin as the window. Let I2 be the image from which the window is to be extracted. The resulting image would have the size and position of I1, with the value of the corresponding pixel of I2 at each position. The equivalence between windowing and unregistered set intersection should be clear. In fact, the windowing algorithm would be simpler, since a single BLACK node of the appropriate size would take the place of I1 in the algorithm, and only one call to DOSET would be needed. Otherwise, after modifying procedure SET_FUNC, the algorithm is fundamentally the same. Such an algorithm is optimal in the sense that it locates (only once) those nodes of the input tree which cover a portion of the window, and performs at most one insert operation for each output node. Table 3-6 presents an empirical comparison of the new windowing algorithm with that presented in Phase III.

Size/ Shift	Times Old	New
256		
1	67.4	25.6
2	47.6	22.4
4	35.2	21.7
16	29.8	20.5
64	25.9	16.8
128		
1	13.9	5.3
2	9.7	4.3
4	7.3	4.3
16	6.5	4.3
64	8.8	5.2
64		
1	4.2	1.5
2	3.1	1.4
4	2.3	1.2
16	1.5	0.9
64	2.3	1.6
16		
1	0.5	0.3
2	0.4	0.3
4	0.3	0.3
16	0.2	0.2
64	0.1	0.2

Table 3-6. Windowing timings to compare two algorithms. Size indicates the width and height of the square window. Shift indicates the location of the lower left corner of the window with respect to the input image - i.e., a shift of 2 means that the lower left corner of the window is 2 pixels to the right and above the origin of the input image. Note that a shift by a multiple of a node size will result in no splitting of nodes that size or smaller; thus the shift value affects the difficulty of the operation. Times are measured in seconds.

4. Enhancements to the attribute attachment system

In a geographic information system, it is desirable to associate descriptive information with the geographic objects stored in the maps. This set of descriptive information is referred to as the *attributes* of an object. The ability to store attribute information is provided by our database system. This attribute attachment system is implemented in the LISP programming language, in order to take advantage of the power LISP provides for symbolic manipulation. Functions for editing and manipulating attribute data associated with area maps were described in the Phase III report.

In Phase III, the attribute attachment system had been implemented only with area maps. In Phase IV, we have extended the attribute system to work with point and linear feature data. The capabilities of the system have been enhanced to allow for sharing of attribute data among data objects. Finally, a new function has been added to aid in the development of point data attributes from area maps.

4.1. New features of the attribute attachment system

In many geographic databases, the attribute data forms the vast majority of the information stored. In our system, each geographic object (a polygon, point, or linear feature) is assumed to be a member of an *attribute class* (sometimes simply referred to as a *class*). An attribute class is simply a list of name/value pairs. The name can be any string; the value can be a number or a string. A name/value pair is referred to a an *attribute*. For example, a map might contain polygons representing fields in a farming district. Each field might belong to an attribute class which describes the crop type, average yield for that crop, fertilizer requirements, etc. The user may want certain attribute classes to share attributes, and have identical values. Continuing our example, each field may have an owner, and other specific information. Each wheatfield should still contain all the generic attributes for wheatfields. To help reduce the amount of storage required to store attribute data, the ability to store attribute class names as part of the attribute list has been added. An attribute class, say A , can now store one or more subclass, say B and C, amoung its attributes. A class can store in its attribute list both attributes and class names. Class A would be interpreted as having all attributes which make up B and C as well as any attributes on its own list. In this way we do not need to duplicate information when one class shares attribute values with another. In our example for representing fields, the attribute class for a particular wheatfield would contain the generic wheatfield attribute class in addition to attributes specific to that field. Attribute classes may be viewed as forming an attribute tree. The class name is at the root, those classes listed on its attribute list are at the next level, and their subclasses and attributes are at lower levels. All attributes that belong to any class of the attribute tree are considered to belong also to the class at the root.

Two problems may arise from this implementation. The first problem is attribute value inconsistency. It is possible that a class contains an attribute more than once within its tree, with these multiple occurences storing different values. In this case, the value of each attribute in the class is defined to be that which is located first when the class list is expanded.

The second problem is circularity. A user could define a class, say A , containing attribute class B. Class B could contain class C, which in turn contains class A. A search through this attribute tree would result in an infinite loop. The edit functions have been

modified so as not to allow the user to create such a cycle.

4.2. Point and line map attribute attachment

The attribute attachment system has been extended to associate attribute data with point and linear feature maps. In addition to the attribute class table normally associated with all maps, point and line maps have a second table. This second table, referred to as the *object table,* contains an entry for each point or linear feature in the corresponding map. This table stores for each feature the name of its attribute class which is found in the attribute table.

The object table is maintained for two reasons. First, it is required for quick lookup when features are located in the map. Often, retrieval functions operating on the map image need only to determine the object identifier stored in the object table. The larger attribute table need not be searched in this case. The second reason is to allow sharing of attribute tables between maps. The data stored in the object table is unique to the map; the attribute table may contain information global to a large database of maps.

Area feature maps do not maintain an object table since individual polygons are not typically referenced by "name". Instead, area maps are usually referenced in terms of classes (i.e., those polygons within a given attribute class). However, each point or linear feature is a unique object. The only quick lookup required for area maps is to find the color associated with an attribute class (for display purposes). This is maintained in a special table available to the display functions.

The SUBSET function has been extended to generate subsets of point and linear feature maps based on a set of class names or attribute values. In addition, a new function has been provided which returns a list of points or line features which match a given set of class names or attributes.

4.3. The POINTAREA function

A new function has been implemented to facilitate building attribute tables for point maps. This function, called POINTAREA, takes as input a point map and an area map. It associates with each point in the point map all attributes belonging to the polygon at that point's position in the area map.

The execution of this function is divided into two steps. The first step is done at the C language level. At this level, the two maps are compared. For every point in the point map, the class value of the corresponding position in the area map is determined. If it is not WHITE, the point and its class value are included in a list which is returned to LISP. The second step is done at the LISP language level. The list of points and classes is processed, updating the point attribute table to contain the appropriate attribute values.

5. A new linear feature representation

For reasons explained below, we decided that the linear feature implementation described in Phase II was inadequate. In this section we first present a discussion of the merits of various hierarchical linear feature representations and provide empirical comparisons of their storage requirements. We then present the implementation ultimately decided upon for inclusion in the database system.

5.1. Background and storage requirements

One of the goals of the project was to develop a uniform representation for data corresponding to regions, points, and vector features. Uniformity facilitates the performance of set operations such as intersecting a vector feature with an area, etc. Use of a linear quadtree for point and region data is well understood; however, this is not the case for vector features. For vector features, a good linear quadtree representation must also have the following three properties. First, it should distribute features between nodes in a fairly uniform fashion. Second, straight line segments should be represented exactly (not in a digitized representation). Third, updates must be consistent, i.e., when a vector feature is deleted, the data base should be restored to a state identical (not an approximation) to that which would have existed if the deleted vector feature had never been added.

In Phase II of this project, we implemented a variation of the edge quadtree of Shneier [Shne81]. A serious drawback of the edge quadtree is its inability to handle the meeting of two or more edges at a single point (i.e., a vertex) except as a pixel corresponding to an edge of minimal length. Thus, we cannot distinguish vertices from short line segments. This means that boundary following as well as deletion of line segments cannot be properly handled in the vicinity of a vertex at which more than one edge meets. Another quadtree variant which is closely related to the edge quadtree is the formulation of Hunter and Steiglitz [Hunt79], termed an *MX quadtree* in [Same84a]. It considers the border of a region as separate from either the inside or the outside of that region. Figure 5-2 shows the MX quadtree corresponding to the polygonal map of Figure 5-1. The MX quadtree has problems similar to those of the edge quadtree in handling vertices. Again, a vertex is represented by a single pixel. Thus boundary following and deletion of line segments cannot be properly handled. Worse is the fact that an MX quadtree only yields an approximation of a straight line rather than an exact representation as done by the edge quadtree. Furthermore, note that the edge quadtree in Figure 5-3 contains considerably fewer nodes than the MX quadtree in Figure 5-2.

In our implementation of the edge quadtree, the leaf nodes of the quadtree were stored as single records in the B-tree. Each node contains three fields; an address, a type, and a value field. The address field describes the size of the node and the coordinates of one of the corners of its corresponding block. The type field indicates whether the node is empty (i.e., WHITE), contains a single point, or contains a line segment. The value field of a line segment indicates the coordinates of its intercepts with the borders of its containing node. Vertices are represented by pixel-sized nodes with the degree of the vertex stored in the value field. Unlike Shneier's formulation, a line segment may not end within a node since in the existing implementation the value field is not large enough to contain the location of an interior point as well as the intercepts. Thus endpoints and intersection points are represented by single pixel-sized point nodes. Figure 5-4 illustrates the linear edge quadtree representation. Note the difference in the decomposition of the region containing the vertex H in Figures 5-3 and 5-4.

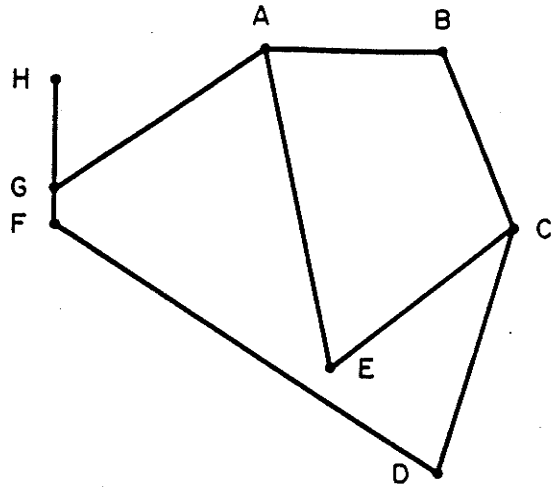


Figure 5-1. A polygonal map.

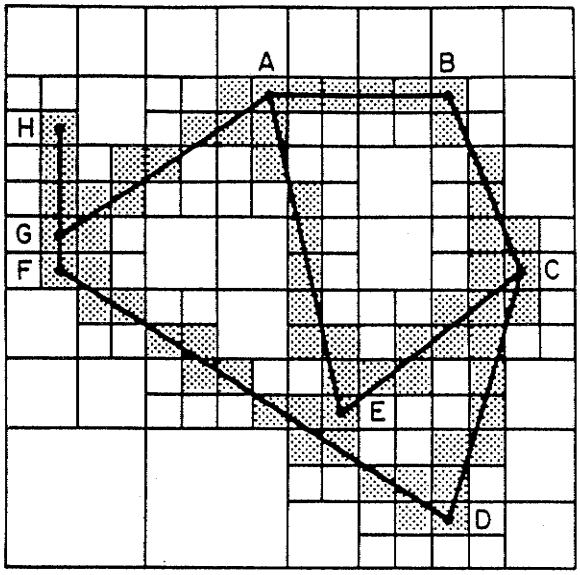


Figure 5-2. The MX quadtree for Figure 5-1.

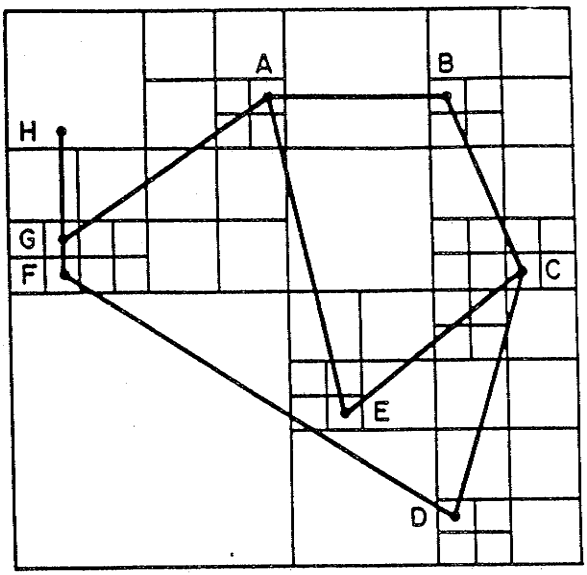


Figure 5-3. The edge quadtree for Figure 5-1.

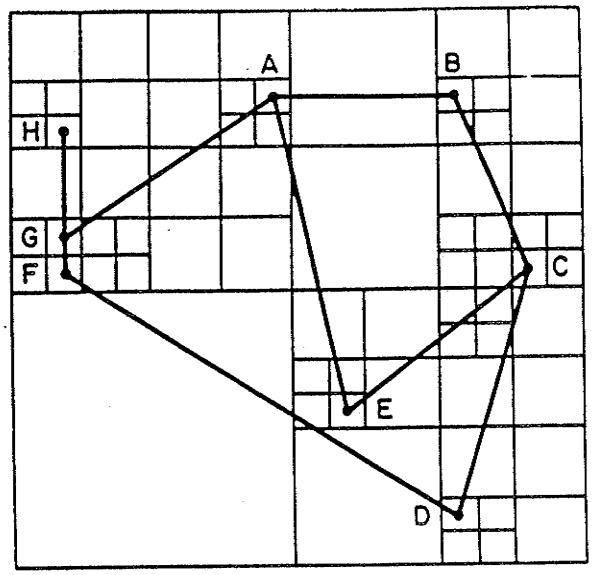


Figure 5-4. The linear edge quadtree for Figure 5-1.

The linear edge quadtree has a number of deficiencies. All vertices and endpoints are stored at the lowest level of digitization, i.e., in nodes deep in the tree. There is no mechanism for following a line segment, as each node describes only that portion of a line segment which is contained within the borders of the node. In particular, given a node that contains a single point, there is no indication as to which of the neighboring nodes are connected to the point by a line segment.

An important criteria for evaluating whether or not a storage representation handles line segments properly is if the successive insertion and removal of the same line segment leaves the map unchanged. Since the edge quadtree nodes store only local information, it is extremely difficult to restore nodes, by merging, which had been split apart by the original insertion. For example, it is not easy to determine the endpoints of the edges emanating from a given vertex. Thus over time, the compactness of the representation could deteriorate until it becomes equivalent to the MX quadtree (i.e., line segments are represented by pixel-sized nodes).

An alternative approach to storing vector feature data is the PM quadtree [Same85a]. The PM quadtree developed from a desire to adapt the PR quadtree to store a polygonal map in a manner which preserves the relationship between edges and vertices. In essence, whenever a group of line segments meet at a common point, those segments can be organized by the linear ordering derived from their orientation. Three variations of the PM quadtree, termed PM_1, PM_2, and PM_3, have been developed.

The PM_1 quadtree is based on a decomposition rule that permits more than one line segment to be stored at a node only if they meet at a vertex that lies within the borders of that node. Figure 5-5 shows the PM_1 quadtree corresponding to the polygonal map of Figure 5-1. From the decomposition of the line segments CD and CE, we observe that the representation of line segments which meet at narrow angles may require a large number of nodes.

The PM_2 quadtree permits more than one line segment to be stored at a node even when the vertex they share is not within the borders of that node. Figure 5-6 shows the PM_2 quadtree that corresponds to Figure 5-1. Note that the PM_2 quadtree requires fewer nodes than the PM_1 quadtree for the same map. However, we observe that when a line segment passes near a vertex that is not incident on it (e.g., segment DF passing near point E in Figure 5-6), it is possible that many nodes may be required to separate them.

The PM_3 quadtree is based on the same decomposition rule as the PR quadtree. All line segments that pass through the node are broken into a fixed number of separate groups. There is one group for all the lines that radiate from the vertex in the node. The remaining line segments are ordered according to the pair of sides of the node's containing block that they intersect. The group of line segments radiating from a vertex is organized by angular orientation and the remaining groups of line segments are organized by their intercepts with the side of the region represented by the node. Figure 5-7 is the PM_3 quadtree that corresponds to Figure 5-1. Note that the block containing vertex E has two line segments intersecting the vertex (i.e., EA and EC), and one line segment (i.e., DF) for the line segment intersecting the south and west boundaries of the block.

Although useful for storing polygonal maps in core, it is not easy to incorporate the PM_2 or PM_3 quadtrees into the fixed-width fields of the linear quadtree disk-based

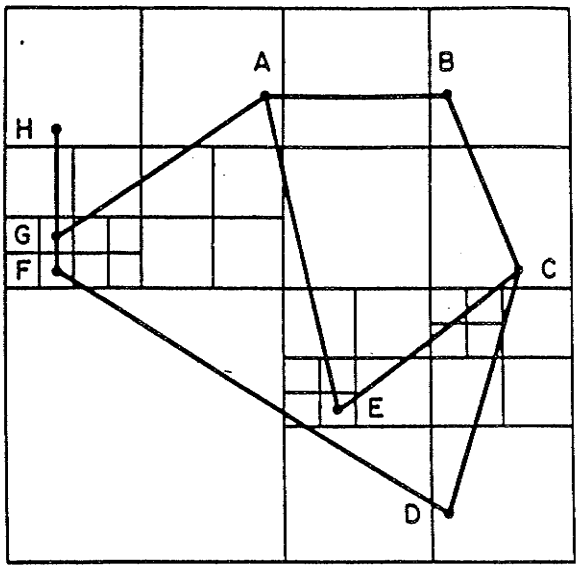


Figure 5-5. The PM_1 quadtree for Figure 5-1.

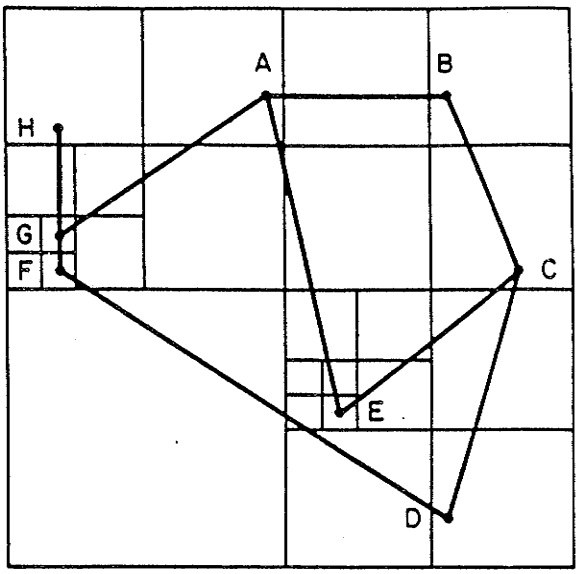


Figure 5-6. The PM_2 quadtree for Figure 5-1.

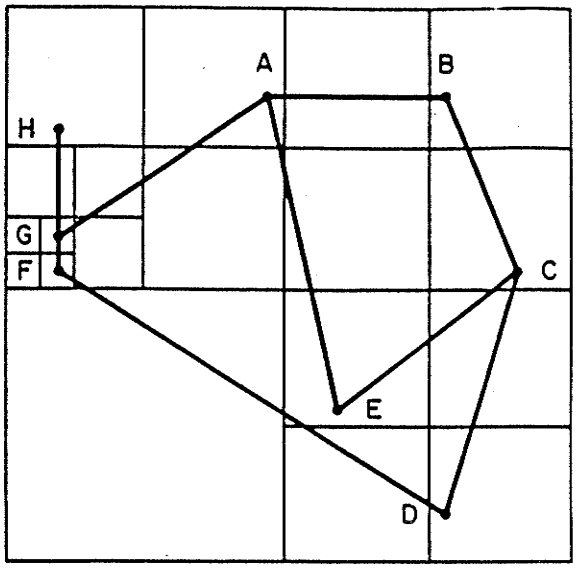


Figure 5-7. The PM_3 quadtree for Figure 5-1.

representation. The problem is that the amount of information stored at a single quadtree node varies widely. For example, in the PM_3 quadtree, a node can contain both a vertex and a set of line segments that do not pass through the vertex. The PM_2 quadtree represents an improvement in the sense that a node either corresponds to a vertex or a set of line segments but not to both. The PM_1 quadtree limits a node further to correspond either to a vertex or to a single line segment. This is more compatible with the node size limitation posed by the linear quadtree. Indeed, the segment quadtree, described below, can be viewed as a linear quadtree adaptation of the PM_1 quadtree.

In the remainder of this section we provide empirical results comparing several linear feature representations. The variants of the PM quadtree as well as the MX and edge quadtree were used to encode three maps (Figures 5-8 through 5-10) chosen from the database described in Phase III. Table 5-1 contains the number of vertices and edges in each of these maps. Note that all of these maps consist of edges whose vertices rest on a 512 by 512 grid that is offset by half a pixel width from the coordinates of the lower left hand corners of the quadtree nodes at depth 9 (later we will consider the impact of this displacement). In other words, the grid of points from which the vertices are drawn corresponds to the centers of pixels.

Table 5-1. Size of the line data sets.

Map	No. of Vertices	No. of Edges
Powerline	15	14
Cityline	64	64
Roadline	684	764

As mentioned above, neither the MX quadtree nor the edge quadtree is really an appropriate representation for polygonal maps since they only correspond to an approximation (or in the case of the MX quadtree, a digitization) of the map, whereas the variants of the PM quadtree represent the maps exactly. Nevertheless, in practice, for the MX quadtree it is natural to consider the approximation that results from representing edges with the same accuracy as the grid. For the 512 by 512 images that we are considering, this means that the MX quadtree is built by truncating the decomposition at depth 9. Similarly, the edge quadtree is also constructed by truncating the decomposition at depth 9.

Tables 5-2 to 5-5 summarize the storage requirements of the various quadtree methods of representing the maps. As we observed before, the PM_1 quadtree will always be the largest of the PM quadtrees - i.e., it will require the most nodes. Therefore, let us consider how it compares with two alternative approaches, the MX and edge quadtrees given in Tables 5-2 and 5-3 respectively. Tables 5-4 and 5-5 contain the data for the different PM quadtrees. Table 5-5 breaks down the leaf count in terms of the different types of nodes and also gives the average number of q-edges for each node type (in parentheses) where it is relevant.

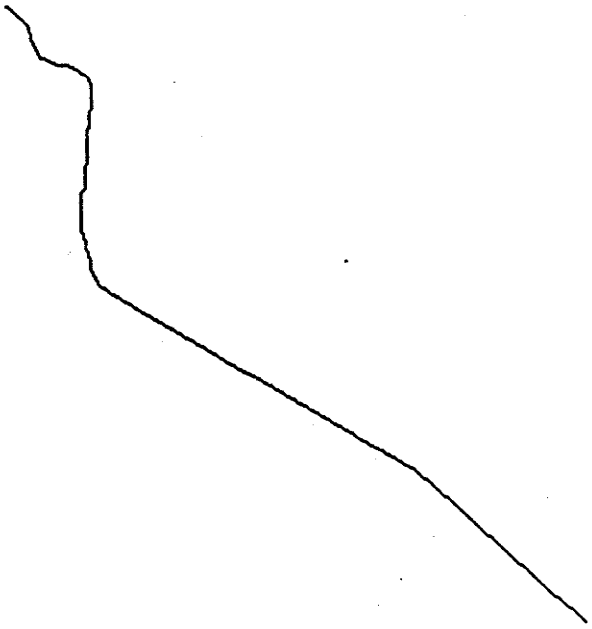


Figure 5-8. The powerline map.

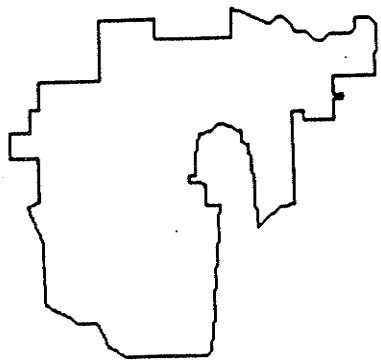


Figure 5-9. The city border map.

Figure 5-10. The road map.

Table 5-2. Size of the MX quadtrees.				
Map	Depth	Leaves	BLACK nodes	WHITE nodes
Powerline	9	1594	521	1073
Cityline	9	2335	782	1553
Roadline	9	19513	7055	12458

Table 5-3: Size of the edge quadtrees.					
Map	Depth	Leaves	Vertex nodes	Line nodes	WHITE nodes
Powerline	9	211	15	68	128
Cityline	9	730	64	219	447
Roadline	9	6658	684	2431	3543

The MX quadtree (see Table 5-2) has the worst performance. In all three examples the MX quadtree is larger than the PM_1 quadtree (see Table 5-4) by at least a factor of 9. More generally, we would expect the number of nodes in the MX quadtree for a polygonal map to be roughly as large as the product of the average edge length and the number of nodes in the corresponding PM_1 quadtree. This can be seen by the following chain of arguments. First, for "typical data", the number of nodes in a PM_1 quadtree of a polygonal map is proportional to the number of vertices in the polygonal map since, typically, the vertex nodes are the lowest nodes in the PM_1 quadtree. In the analysis of quadtrees, a good rule of thumb is that the deepest frequently occurring node type will dominate the size measurement. Second, Hunter [Hunt78, Hunt79] has shown that the number of nodes in an MX quadtree of a polygonal map is proportional to the perimeter of the polygon. Third, we know that polygonal maps are planar maps which means that the number of edges in each map is proportional to the number of vertices in the map. Combining these three arguments with the fact that the perimeter of a map is equal to the product of the number of edges and the average length of an edge leads to the desired result - i.e., typically, the number of nodes in the MX quadtree is on the order of the product of the number of vertices in the PM_1 quadtree and the average edge length (measured in pixels).

Table 5-4. Size of the PM_1, PM_2, and PM_3 quadtrees.									
Map	Depth			Leaves			Q-edges		
	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
Powerline	7	7	7	61	61	61	38	38	38
Cityline	9	8	8	214	208	187	178	176	168
Roadline	13	9	9	2125	1960	1714	2144	2096	1976

Table 5-5. Breakdown of information in Table 5-4 by node type.									
Map	Vertex nodes (average q-edges)			Line nodes (average q-edges)			WHITE nodes		
	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
Powerline	15 (1.9)	15 (1.9)	15 (1.9)	10	10 (1.0)	10 (1.0)	36	36	36
Cityline	64 (2.0)	64 (2.0)	64 (2.1)	50	47 (1.0)	33 (1.0)	100	97	90
Roadline	684 (2.2)	684 (2.2)	684 (2.3)	618	515 (1.1)	360 (1.1)	823	761	670

Now, let us compare the edge quadtree with the PM_1 quadtree. The edge quadtree (see Table 5-3) can be seen to be a definite improvement over the MX quadtree. Considering the trivial (but often typical) maps like Powerline and Cityline, we see that the edge quadtree is about three times as large as the PM_1 quadtree. This can be explained by observing that the average depth of a vertex node in the PM_1 quadtree for each of these two maps was between 6 and 7 (not shown in the tables) whereas the corresponding edge quadtree must represent all of the vertex nodes at depth 9.

The Roadline Map, the most complex map in the data, has an edge quadtree that has three times as many nodes as its corresponding PM_1 quadtree. This might at first appear surprising since the maximum depth of this quadtree is considerably greater than that required by the digitization grid. In this case, the digitization grid requires a depth of 9 while the PM_1 quadtree requires some nodes to be at a depth of 13. Although for this map, the maximum depth of the PM_1 quadtree is greater (i.e., 13) than that of the edge quadtree (i.e., 9), the difference in the number of nodes in the two trees can be explained by examining the distribution of nodes by depth (see Table 5-6). In essence, the average depth of a vertex node is again between 6 and 7 for the PM_1 quadtree while it is 9 for the edge quadtree. The reduction in the average depth of a vertex node in the PM_1 quadtree has a direct effect on the total number of nodes because the decomposition of an edge is identical in the edge and PM_1 quadtrees once the edge has exited the region of the vertex nodes representing its endpoints.

Table 5-6. Distribution of node types by depth for PM quadtrees in the Roadline map.

Depth	Vertex nodes			Line nodes			WHITE nodes		
	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3	PM_1	PM_2	PM_3
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	4	4	4
3	2	2	2	0	0	0	8	8	8
4	10	10	12	4	6	7	38	38	38
5	75	75	82	34	35	29	109	105	101
6	180	180	192	132	127	120	218	212	199
7	224	224	232	204	198	139	237	222	195
8	158	158	135	156	126	59	162	142	104
9	35	35	29	48	23	6	40	30	21
10	0			13			2		
11	0			14			3		
12	0			10			1		
13	0			3			1		

The above discussion leads to the conclusion that the PM_1 quadtree is an improvement over earlier quadtree-based approaches to handling real data. We have seen that the PM_1 quadtree has the desirable property of reducing the average depth at which the dominant node type is located. The PM_2 and PM_3 quadtrees are attempts to further reduce the maximum depth of nodes in a PM_1 quadtree. The PM_2 quadtree has the effect of reducing the maximum depth (see Table 5-6) by virtue of a more compact treatment of the case when close edges that radiate from the same vertex lie in a different node from the vertex. When comparing the data of the PM_1 quadtree columns with the data of the PM_2 quadtree columns of Table 5-4, we observe no change in the Powerline map since it is composed of only obtuse angles. The Cityline map has a few acute angles creating situations where line nodes can be formed containing more than one q-edge, thus causing some of the line nodes to be closer to the root and resulting in a 3% reduction in the number of nodes. The more complicated Roadline map presents more such situations resulting in an 8% reduction in the number of nodes. We note that the PM_2 reduction only affects the depth of the line nodes. Recall that nodes containing a vertex are treated in the same manner in both the PM_1 and PM_2 quadtrees. This observation is reinforced by noting that the vertex node columns in Table 5-6, which show the distribution of node type by depth, are identical.

Comparing the PM_3 quadtree with the PM_1 and PM_2 quadtrees also shows no change in the number of nodes when used on the Powerline map. This is because the Powerline map contains no edges that pass closely to vertices other than their endpoints. This situation occurs a bit more frequently in the Cityline map, resulting in a 12% reduction in the number of nodes. Note that the existence of such situations implies that vertex nodes will be slightly closer to the root in the PM_3 quadtree than in the PM_1 and PM_2 quadtrees. For the Roadline map, the use of the PM_3 quadtree instead of a PM_1 quadtree leads to a 19% reduction in the number of nodes. This is due to the tendency for vertex nodes to occur closer to the root in the PM_3 quadtree than in the PM_1 quadtree and can be seen by examining Table 5-6.

From the above we see that although the differences among the different PM quadtrees can be drastic in principle, for typical cartographic data, the difference in the number of nodes in the various PM quadtrees for a particular map is less pronounced. Thus, for cartographic data, the choice among the different PM quadtrees is dictated more by the problems of implementation rather than by the need to conserve space. However, it should be noted that cartographic data is rather special in that it generally consists of sequences of short edges meeting at obtuse angles. Since the lengths of the edges are often shorter than the distances between the features that the edges are representing, this yields data that tends to bring out the best in each of the types of PM quadtrees with the result that there is little difference between them. For data that is not this simple, the advantages of the PM_2 and PM_3 quadtrees over the PM_1 quadtree should be more pronounced.

Aside from the consideration of the numbers of leaves in the various quadtree implementations, there are two further aspects of storage to be examined: 1) the number of q-edges in the various quadtree nodes and 2) the sensitivity of the PM quadtree representations to slight shifts in the placement of the data.

In Table 5-4 we find that a reduction in the number of q-edges closely parallels the reduction of the number of quadtree leaf nodes across the different PM quadtree implementations. Table 5-5 tabulates the average number of q-edges per node of a particular node type. This is placed in parentheses immediately after the count of the number of leaf nodes of that node type. No averages are given for WHITE nodes and PM_1 quadtree line nodes, as by definition, they have zero and one q-edge, respectively. Investigation of the average number of q-edges per line node shows that it is rare for there to be more than one q-edge per line node. In the case of vertex nodes, we find that the number of q-edges per node is consistently around 2, although it does seem to increase slowly with map complexity. These values tend to indicate that a linked list is usually sufficient to organize the q-edges at a given node. The PM quadtree implementation of the three test maps found at most one node in a given map that had as many as 5 q-edges. Thus not only is the average low, but there also does not seem to be much variance from the average value.

Previously, we stated that one of the motivations for the development of the PM quadtree data structure is that its size is relatively invariant to shifting and rotation. Table 5-7 summarizes the results of some experiments on the effect of minor shifts in positioning the vertices of the Roadline map. The first column, labeled .5, shows the data used to generate Tables 5-2 to 5-6. Recall that to obtain these tables the original data was shifted by adding 0.5 to what were originally integer coordinates on a 512 by 512 grid. The column labeled 0.0 indicates no change in positioning the vertices of the original data and shows significantly higher node counts than the other shifts. This is not surprising since when a vertex lies on the border of a quadtree node, it is inserted in each of the nodes whose border it touches. This can cause further node splits if a quadtree node in which it is inserted already has a vertex in it. However, if the vertices that lie on the borders of quadtree nodes are shifted slightly, then they no longer will share a quadtree node and thus no further decomposition will be required. Once the placement of vertices on the borders of quadtree nodes has been avoided (e.g., by using small shifts), there still remains the secondary effect that vertices close to the border of a quadtree node tend to result in a very small separation between the q-edges in the neighboring quadtree node. This has the greatest effect on the number of nodes in the PM_1 quadtree, while it has no effect on the number of nodes in the PM_3 quadtree. Like the PM_3 quadtree, the PM_2 quadtree is not affected by q-edges whose separation is small because they result from a vertex being

near a quadtree boundary. However, this is not shown so clearly by the entries in Table 5-7 for the PM_2 quadtree since the PM_2 quadtree is susceptible to the digitization effects that result from the process of determining whether or not a line falls within a particular square region. The only digitization effects that can alter the number of leaf nodes in the PM_3 quadtree are those resulting from the process of determining whether or not a vertex lies within a particular square.

Table 5-7. The effect of small shifts on different representations of the Roadline map.

Quadtree type	Numbers of leaves resulting from different shifts					
	.5	.625	.75	0.0	.125	.25
MX	19513	19720	19627	20554	19597	19618
Edge	6658	6709	6757	8611	6760	6727
PM1	2125	2179	2218	2698	2275	2203
PM2	1960	1984	1981	2608	1999	1993
PM3	1714	1714	1714	2434	1714	1714

5.2. New line representation

From the results of Section 5-1, we concluded that the PM structures would best resolve the problems encountered with the edge quadtree, particularly that of representing a set of line segments exactly. The major obstacle to their use is that they require the use of variable size nodes. This reflects what seems to be a general principle: that to adequately represent lineal data in a structure based on spatial decomposition requires the ability to store more than one piece of information about an area. This can be rationalized as follows. For a one item per node representation to work, the amount of information needed to describe an area must decrease as the size of that area is reduced. An intrinsic property of line data, however, is that it allows large amounts of information to be concentrated in a small area, for example, where several lines intersect near a point. Thus it is not surprising that one line per node representations fail to provide an effective representation for linear feature data (as illustrated by our problems near vertices in the linear edge quadtree).

We therefore decided to concentrate on solving the variable node size problem. In the remainder of this discussion, we will refer to the schemes based on variable size nodes as *bucketed structures,* and to the variable size nodes as *buckets.* The usage here is slightly different than that in some other papers (e.g., [Tamm81, Tamm83]) where a bucket refers to a unit of storage of fixed capacity.

The introduction of variable node sizes has important consequences. What was a major difficulty, the problem of representing intersecting lines, is eliminated and a new set of representations, each based on a different splitting rule, is possible. The selection of a splitting rule is itself easier, as almost any scheme that divides up the segments between blocks in a reasonable fashion can be used. The splitting rule can be tailored to the application at hand, and the complexity of a system such as PM_2 becomes justifiable only if it is particularly useful in solving some problem. 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 others. For instance, a rule requiring the node to be quartered if the number of segments in a block exceeds n when a segment is added, in conjunction with a corresponding deletion rule, could be used to dynamically maintain a collection of line segments. We now define a structure termed the PMR quadtree (for PM Random) which uses such a scheme.

The PMR quadtree uses a pair of rules, one for splitting and one for merging, to organize the data dynamically. The splitting rule is invoked whenever a line segment is added to a node. The node is split once into quadrants if the number of segments it contains exceeds n (4 in our implementation). Note that this rule *does not* guarantee that each node will contain at most n line segments. The corresponding merging rule is invoked whenever a segment is deleted. The node is merged with its siblings if together they contain fewer than n (4 in our implementation) distinct line segments. This scheme differs from the other quadtree structures considered here in that the tree for a given data set is not unique, but depends on the history of manipulations applied to the structure. Certain types of analysis are thus more difficult than with uniquely determined structures. On the other hand, this structure allows the decomposition of space to be based directly on the linear feature data stored locally.

The main work reported here involves the implementation and testing of bucketing methods for line storage which could be integrated with the existing representations for point and region data in the geographic information system. Two bucket-based structures were

implemented, utilizing the basic PM notion of storing lineal elements in the quadtree nodes, and making full use of the potential of variable size nodes to limit decomposition. They are based on the PM_3 and PMR quadtrees.

The representations described developed from an attempt to adapt the PM_3 quadtree for use in the geographic data system. The first question is how to implement the variable sized nodes. Since the number of line segments in a node is potentially unbounded, a true variable-length storage scheme must be used. For pointer quadtrees, linked lists are one possibility. A binary tree structure reflecting the position of the segments within the block has also been suggested [Same83]; however, this seems to be unnecessarily complicated for our application. As shown in Section 5-1, the average number of segments intersecting a node can be kept small by the appropriate choice of a splitting rule. When the graph represented by the set of line segments is planar (which is the case for polygonal maps and most geographical situations), the average number of segments per node in the PM_3 decomposition is limited by topological considerations to some small number (for our map data, the average is less than three). This makes a linear search through the list practical. In an application where this is not the case, other splitting rules can ensure that the number of segments in a bucket does not become too high. For linear quadtrees, where an address is calculated for each block and used to order it in the list, the simplest way of implementing variable node sizes is simply to duplicate addresses. This is the method used in the present implementation.

The second question, and one that turns out to be critical to the expressive power of the system, is how to represent the lines intersecting a node. We assume for simplicity that the original data consist of line segments defined by a pair of endpoints digitized to the resolution of the quadtree. One possible implementation is to store in the node a new segment whose endpoints are the intersections of the original segment with the block boundaries, as was done with the linear edge quadtree. This has the single advantage that it is easy to split a line in two (as would be necessary when decomposing a node) by introducing an intermediate point. Since all information is local, we need not be concerned with the effects such a decomposition will have on other nodes through which the segment passes. For the same reason, however, it is hard to track a line from one block to the next. This makes operations such as deletion difficult, and remerging sibling nodes is nearly impossible since there is no way of telling whether or not the pieces originally formed a single line segment. Thus, manipulation of the map can degrade the information it contains just as in the case of the linear edge quadtree. One possible solution is to represent the endpoints of such local segments exactly (with large rationals), or to a very high accuracy (with floating point arithmetic), and to assume that two subsegments connect if their endpoints are identical or match within some tolerance. Such an implementation is difficult to program and is sensitive to error from scale changes.

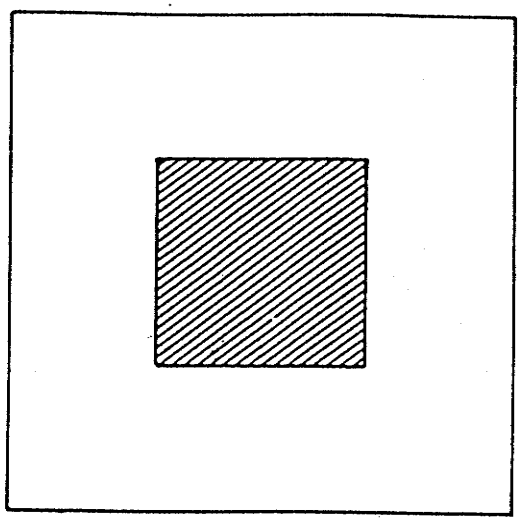
A more viable suggestion is to store a pointer to a record representing the original line segment in each node it intersects. Since each node containing the segment stores the same descriptor, tracking the line from block to block is simple and operations such as deletion can be easily implemented. This approach is also considerably more flexible as it allows the storage of an arbitrary amount of information about the segment without increasing the size of the B-tree record, and permits this information to be concentrated in one place rather than repeated in every node which refers to the segment.

We still have the problem of how to split a segment. In geographical applications, the situation arises when a line map is intersected with an area. Since the borders of the area may

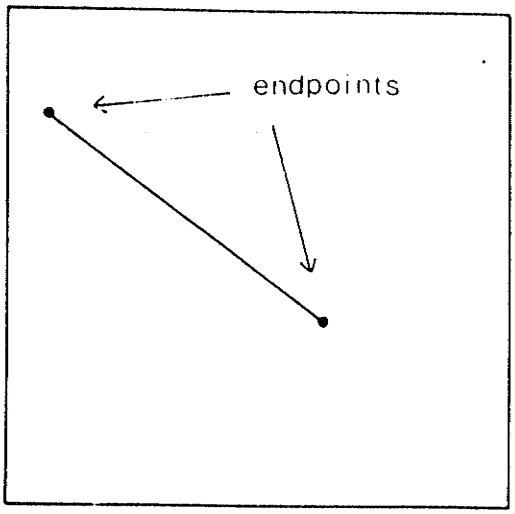
not correspond exactly with the endpoints of the segments defining the line data, certain segments may be cut off. Such an intersection is illustrated in Figure 5-11. The partial line segment produced is referred to as a *fragment* and the artificial endpoints produced by such an intersection are referred to as *cut points*. One way to represent a fragment is to introduce an intermediate point at the node intercept. In continuous space, the remaining line segment can then be exactly represented as a new line segment which is colinear with the original, but has at least one different endpoint. In discrete space, however, this is not always possible because the continuous coordinates at the intercept do not, in general, correspond exactly to any coordinates in the discrete space. If the new line segments are represented approximately in the discrete space, then the original information is degraded, and the pieces cannot reliably be rejoined. These are precisely the problems encountered with the linear edge quadtree. Additionally, if an intermediate point is introduced to produce new segments, then the line segment descriptor must be propagated to all blocks containing the original segment. This is likely to be a very time-consuming operation.

An alternative idea is to retain the original pointers, and use the spatial properties of the quadtree to specify what parts of the corresponding segments are actually present. The underlying insight is that a node may contain a pointer 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 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 fragments, therefore, may be represented by a collection of q-edges. The presence or absence of a particular q-edge is completely independent of the presence or absence of those q-edges representing other parts of the line segment, hence lineal features corresponding to partial segments can be represented simply by inserting the appropriate collection of q-edges. Since the original pointers are retained, a lineal feature can be broken into pieces and rejoined without loss or degradation of information. Within the quadtree structure, q-edges may represent arbitrary fragments of line segments. Since all bear the same segment descriptor, they are easily recognizable as deriving from the same parent segment. This solves the problem of how to split a line or a map in an easily reversible manner. The use of this principle to represent the lineal feature produced by the intersection of Figure 5-11 is shown in Figure 5-12. Below we describe how this idea is incorporated into the PM_3 and PMR linear quadtrees.

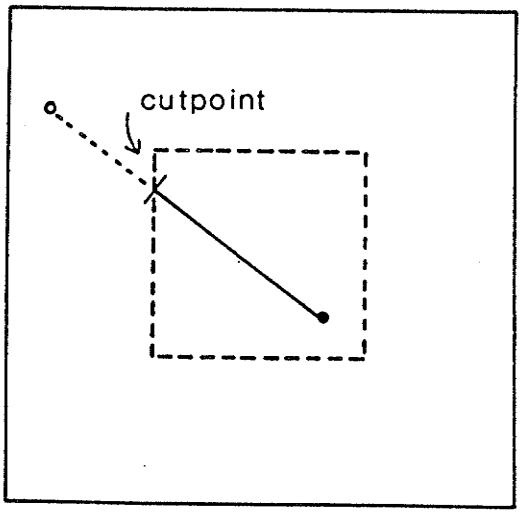
Given a collection of fragments, the basic idea is to use the quadtree decomposition to localize the cut points of the fragments, and simultaneously use the endpoints of the parent segments to induce a PM_3 decomposition. A problem arises however, if we use all such endpoints. Consider the case of a set of fragments which connects to few of the original endpoints (i.e., most of them have been cut off). The remaining fragments may be far away from the original endpoints, but the former are still directing the splitting of blocks. In such a situation it is possible to have a node which contains no q-edges, but must be split into four empty nodes. The situation is illustrated in Figure 5-13. The problem is easily remedied by using only the segment endpoints attached to q-edges to induce PM_3 splitting. The structure can thus be defined by splitting until no block contains a cut point (all cut points must lie on the boundaries between blocks), and no block contains more than one segment endpoint attached to a q-edge. Note that this rule yields the ordinary PM_3 decomposition in the case where all the fragments represent full line segments.



(a) A region.

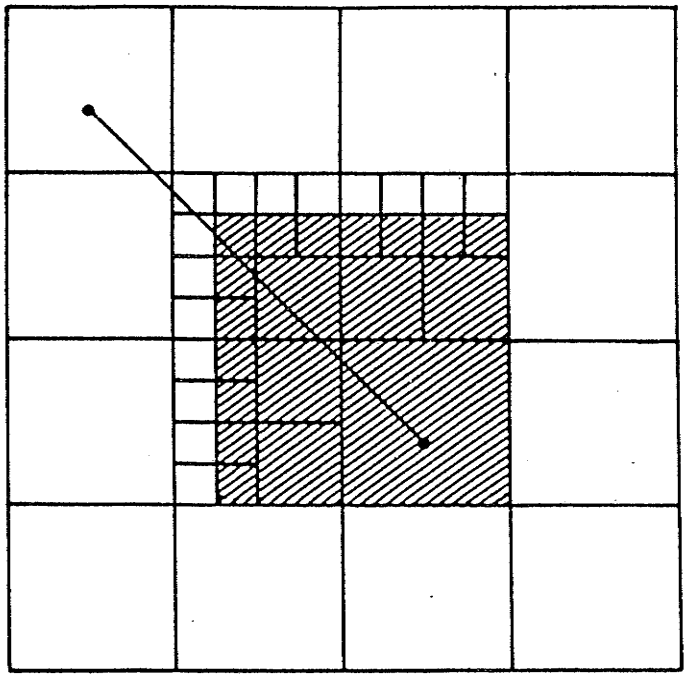


(b) A line segment.

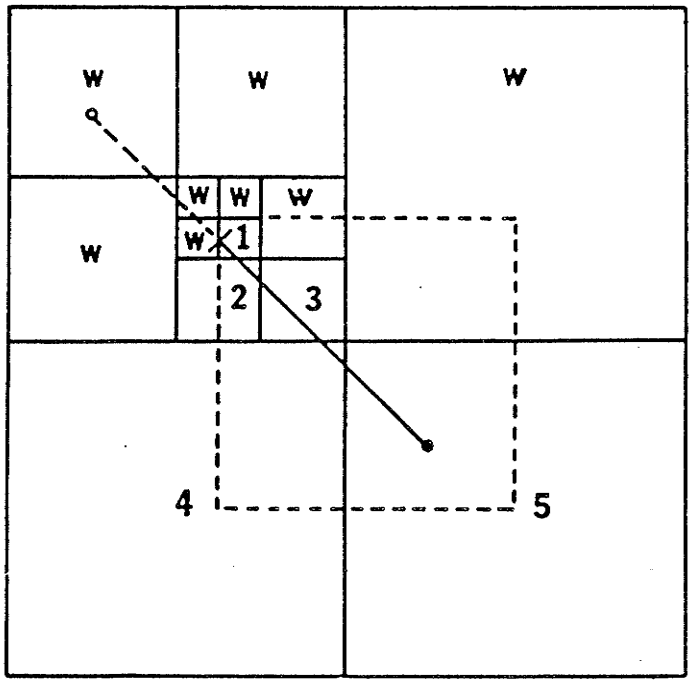


(c) a fragment with one cut point produced by the intersection of the region in (a) with the line segment in (b).

Figure 5-11. Definition of a fragment from the intersection of a segment with a region.



(a) A region quadtree with a line segment superimposed.



(b) A set of 5 q-edges composing fragments of Figure 5-11, and the corresponding quadtree.

Figure 5-12. Representation of the fragment of Figure 5-11 using a collection of q-edges.

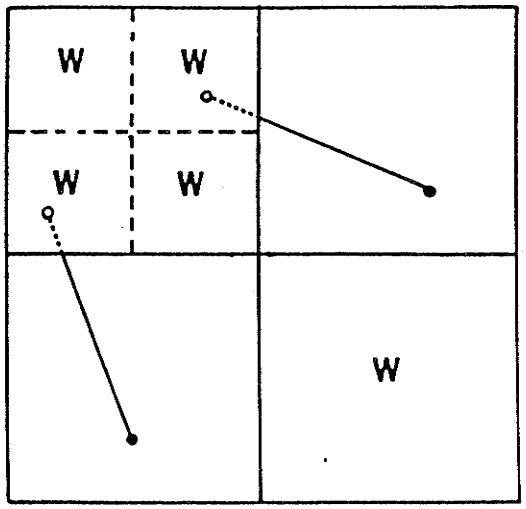


Figure 5-13. Unattached endpoints causing split of a WHITE node.

The above rule is useful as a means of defining the structure, but cannot actually be used to construct a linear PM_3 quadtree because the database has no provision for explicitly referring to general fragments. Instead, the structure is implemented by allowing the insertion and deletion of a restricted set of fragments, namely q-edges. Since these form a set of "building blocks", quadtree representations for arbitrary fragments may be constructed.

Insertion of a q-edge E representing the intersection of line segment L with block B is accomplished as follows. If B corresponds to a leaf node, then E is installed, and the node checked for splitting. If B is subsumed by a leaf node, then that node is split until a leaf node is produced which corresponds to B, and E is then installed. If B corresponds to a gray node, then the q-edges corresponding to the intersection of L with the sons of Q are inserted recursively.

Deletion is accomplished by a similar procedure, where deletion of a q-edge is interpreted as erasing a portion of the parent segment. If B corresponds to a leaf node which contains E, then the reference to E is removed and a check for merging is made on the node and its siblings. If B is subsumed by a leaf node, then that node is split until a leaf node is produced which corresponds to B, and E is deleted. If B corresponds to a gray node, then the q-edges formed by intersecting L with the sons of B are deleted recursively.

The PMR structure is adapted from the PM_3 quadtree by modifying the splitting and merging rules. Nodes are now split until no block contains a cut point in its interior, and then once more if a block contains more than four q-edges. The merge condition is now invoked both when a q-edge is deleted and when one is inserted (since a fragment may be inserted which restores a larger piece). Merges occur when there are four or fewer distinct line segments among the siblings and the q-edges are compatible. The PMR quadtree is implemented in the same way as the PM_3 quadtree just described, except that different splitting and merging rules are used.

The modified PM_3 and PMR quadtrees satisfy all of the properties required for a linear feature representation in our system. With the modifications described in Section 2, the same underlying representation is used for all data types. Insertion and deletion can be performed efficiently - i.e., in time proportional to the number of nodes containing the line segment to be inserted or deleted. Since the line segment is represented in the B-tree by a pointer to an unrestricted line segment descriptor, it can be easily integrated into the attribute attachment system. These line segment descriptors can represent the segment exactly, and each node containing a given line segment stores an identical record. Degradation is therefore avoided, and consistency during insertions and deletions is maintained.

The bucketed q-edge schemes satisfy all the necessary requirements for a linear feature representation. It remains to be seen whether they can be made to perform efficiently in the geographic information system. This is the subject of the next section.

5.3. Empirical results

In order to evaluate the performance of the proposed line representations, four different structures were implemented. They were the MX, edge, PM_3, and PMR quadtrees. The first two, due to the various deficiencies mentioned above, are ultimately unsuitable for the desired application. However, they are still useful for comparison. Specifically, if the performance of the bucketing methods is far worse than that of methods known to be insufficient in other ways, we must ask whether what has been gained is worth the cost. Comparisons were made on the following operations: 1) the time required to build the structure; 2) the sizes of the different representations in our specific implementation; and 3) the time required to perform an intersection with an area image. The three linear feature data sets used in Section 5-1 were again used for these tests.

The building algorithm essentially tests the efficiency of insertion into the structure. Three maps were built for each of the four structures. The results are displayed in Table 5-8. They indicate that none of the methods is overwhelmingly superior in terms of insertion efficiency, but the PMR representation has a definite if irregular lead in most cases. In the case of the road map, which represents the most realistic data set, the MX, edge, and PMR representations are more or less equivalent, and about 30% better than the PM_3. From these results, we see that insertion times for the bucketing methods are comparable to those for the other representations.

As shown in Section 5-1, the PM representations are far superior in storage requirements compared to the MX and edge quadtrees. Node counts are shown in Table 5-8; they differ slightly from those in Tables 5-1 to 5-5 due to different positionings of the vertex within the pixel.

Examination of the data reveals a steady decrease in the required storage from MX to edge to PM_3 to PMR, with the change being approximately a factor of two for each step in the case of the road map (more for the simpler data sets). The PMR representation is at least eight times more efficient in its use of storage than the simple MX in all the cases tested, but both bucketing techniques improve the storage efficiency significantly over the other two techniques. This improvement can be explained by noting that the bucketing methods use one-dimensional primitives which can extend over distances of many pixels rather than the pixel by pixel representation that is used by the MX method exclusively, and by the edge method when the going gets rough. It should be mentioned that these results are actually for pure PM_3 and PMR quadtrees since only complete line segments were inserted in the construction of the maps. For maps containing cut points, the utilization of space would be expected to be less efficient since additional splitting is required to localize the fragment ends.

Table 5-9 gives the sizes of the structures produced by intersecting the road map with various areas. Comparing the sizes of the resulting maps reveals that the improvement in storage requirements is indeed less dramatic. The degree of fragmentation varies, but in the case of intersections with the pebble map and its complement, it is probable that few if any of the original segments are intact. The improvement is still present however, with the PM methods generally using between one half and one quarter of the space of the MX, and significantly less than the edge quadtree. In no case do we see a change in the order of decreasing sizes from MX to PMR, which was noted for the segment data. Finally, we observe

that in all cases, the MX quadtree is by a wide margin the largest structure. This suggests that the MX scheme is fundamentally inefficient in its use of space, at least when the data can be described using high level primitives (e.g., line segments).

The intersection function is a high level geographic computation which involves processing the entire data structure. In the case of the bucket methods, it tests the efficiency of the fragment representation, because the previously intact line segments are now cut where they cross an area boundary. Because the bucket methods allow certain operations not possible with the local methods (e.g., reassembling split lines without data degradation), it is not entirely clear that the different intersection computations are comparable; however, the results may give a general idea of the practicality of high level operations. The intersections were performed using the roadmap as the linear feature data set (the others being too small to provide a reasonable intersection) and three binary maps and their complements as templates. The use of the complements is intended to permit the effects of the size and shape of the templates to be distinguished from the overall efficiency of the different algorithms. This precaution is necessary because the intersection algorithm used with the bucket structures works differently than the one used with the two other methods, and is affected differently by changing the shape of the template. Specifically, the intersection procedure for the first two methods works by inserting into a blank map all linear sections that intersect the template, while the procedure for the bucket structures works by erasing the sections of the line map which do not intersect the template. It turns out that insertion and deletion of q-edges are operations of comparable complexity. For a map produced by erasing portions of a preexisting map, the number of deletions corresponds to the number of insertions necessary to produce the complementary map since the same q-edges are involved. Hence it is more appropriate to compare the intersections of the first two representations with the complementary intersections of the bucket-based representations.

The three templates used are referred to as center, stone, and pebble, and represent a floodplain in register with the road map, and unrelated binary images derived from thresholded photographs of stones and pebbles, respectively. Only the floodplain map has any geographic relevance. The other two were used with the intention of giving the system a more stringent, if less realistic, test. In particular, the degree of fragmentation induced by the pebble map probably exceeds anything that would normally be done in a geographic application.

The results of the test are given in raw form in Table 5-9, and are apparently ambiguous. In some cases the bucketing methods take much longer than the MX and edge schemes, but in others they take less time (though not correspondingly so). This inconsistency is due to the complementary effect of the different intersection algorithms discussed above. Table 5-10 reorganizes the data so that the appropriate complements are compared, and a consistent trend is now apparent. The time needed to perform an intersection generally increases from MX to edge to PM_3 to PMR with the bucketing methods taking somewhere around twice as long as their competitors. Note that the order in which the intersection times increase is the same in which the structure sizes decrease, suggesting that what we are seeing is a time versus space trade-off. In the step from the MX to the edge representation the increase in intersection time is not as sharp as this notion would suggest from the corresponding decrease in size, but this might only indicate that the MX quadtree is not a very efficient representation of the information for any purpose (something which we already suspect). For the other representations, coming up against this limit may suggest that the algorithms are making full use of the information carried in the structure, and that the information carried is not redundant. This

would imply that no drastic improvements will be discovered which will change the relative performance of the intersection algorithms for any of the structures. We suspect this, having tried several radically different approaches in an attempt to reduce the intersection times for the PM structures to at least that of the edge quadtree, all of which had similar performance.

In one sense, the results are disappointing because we have come to expect, since so many quadtree algorithms can be made to run in time proportional to the number of nodes, that a decrease in the size of a structure will imply a corresponding decrease in execution time for operations performed using that structure. There is, however, no reason to expect this to hold across different structures, since the constants of proportionality will certainly differ, as will the amount of work done per node. On the other hand, the increased time is by no means severe enough to damage the value of the representation. The PM structures have capabilities and a certain elegance that the MX and edge quadtrees completely lack. This is worth a certain price. Moreover, if the space versus time tradeoff is authentic, then a PM type structure could be implemented using a different splitting rule, which, at the cost of using more space, would perform intersections as quickly as the edge quadtree.

Table 5-8. Building times and sizes.

Map/Structure	Time	Leaves	Qnodes
powerline			
MX	2.0	1600	-----
edge	0.8	226	-----
PMR	0.7	32	19
PM_3	0.9	82	64
railroad			
MX	2.8	2101	-----
edge	1.0	301	-----
PMR	0.6	35	19
PM_3	0.9	92	70
city			
MX	3.4	2347	-----
edge	2.9	835	-----
PMR	1.9	151	70
PM_3	3.3	310	214
road			
MX	31.5	19699	-----
edge	27.4	7723	-----
PMR	25.8	2078	874
PM_3	39.0	3939	2350

Table 5-9. Intersection times and sizes.

Map/Structure	Time	Leaves	Qnodes
road & center			
MX	5.1	3094	-----
edge	6.3	1759	-----
PMR	15.8	910	775
PM_3	17.0	1019	874
road & centercomp			
MX	21.4	17314	-----
edge	16.2	8320	-----
PMR	8.9	2568	1402
PM_3	8.0	4275	2677
road & stone			
MX	6.5	3397	-----
edge	10.0	2344	-----
PMR	31.0	1853	1651
PM_3	24.8	2011	1774
road & stonecomp			
MX	26.4	17776	-----
edge	22.1	8803	-----
PMR	24.9	3270	2122
PM_3	17.3	4684	3244
road & pebble			
MX	13.6	9022	-----
edge	19.3	5653	-----
PMR	50.0	4034	3370
PM_3	34.7	4530	3760
road & pebblecomp			
MX	20.2	13564	-----
edge	23.2	7459	-----
PMR	45.5	4250	3436
PM_3	32.7	5086	4078

Table 5-10. Reordered intersection times.		
Map	Structure	Time
road & center	MX	5.1
	edge	6.3
road & centercomp	PMR	8.9
	PM_3	8.0
road & centercomp	MX	21.4
	edge	16.2
road & center	PMR	15.8
	PM_3	17.0
road & stone	MX	6.5
	edge	10.0
road & stonecomp	PMR	24.9
	PM_3	17.3
road & stonecomp	MX	26.4
	edge	22.1
road & stone	PMR	31.0
	PM_3	24.8
road & pebble	MX	13.6
	edge	17.3
road & pebblecomp	PMR	45.5
	PM_3	32.7
road & pebblecomp	MX	20.3
	edge	23.2
road & pebble	PMR	50.0
	PM_3	34.7

Table 5-10: Reordered intersection times

6. Conclusions

This project had both specific and general goals. The specific goals dealt with choosing appropriate representations for areas, points, and lines so that responses can be easily made to queries such as "Find all cities with a population in excess of 5000 people in wheat-growing regions within 20 miles of the Mississippi River." For areas we used the region quadtree. For points we used the PR quadtree, a regular decomposition point quadtree. For lines we used the PMR quadtree as discussed in the previous section. Since all of these representations are based on a regular decomposition, queries having the structure of the sample query can be handled by our system.

The general goal of this project was to demonstrate the utility of hierarchical data structures for use in the domain of geographic information systems. We have achieved this goal in that we have produced a prototype geographic system which represents images with a linear quadtree. This system is capable of manipulating area, point, and linear feature data in a reasonably efficient manner. It seems to perform well compared to vector-based systems which make up the majority of geographic systems at this time. Many of these vector-based systems have serious deficiencies due to the extreme difficulty of implementing certain features (most such systems do not do polygon overlays for this reason).

Our experience during this project has been that while area and point data are easily dealt with by an area based representation, dealing with linear feature data is much harder. We have come up with a new solution to this problem which is compatible with our other data representations.

The major deficiency of the system in its present form is the attribute attachment system. Since the main goal of the project was to investigate geographic data representations, relatively little time was available to develop the large database techniques necessary to produce an effective solution to this problem. Clearly, it is a major research topic in itself, and is increasingly being recognized as such. As such techniques become available, it should be an easy matter to combine the geographic representation methods embodied in this system with an attribute database.

7. References

1. [Hunt78] - G.M. Hunter, Efficient computation and data structures for graphics, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

2. [Hunt79] - G.M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 1*, 2(April 1979), 145-153.

3. [Klin71] - A. Klinger, Patterns and Search Statistics, in *Optimizing Methods in Statistics*, J.S. Rustagi, Ed., Academic Press, New York, 1971, 303-337.

4. [Rose82] - 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, June 1982.

5. [Rose83] - 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, September 1983.

6. [Same82] - H. Samet, Distance transform for images represented by quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 4*, 3(May 1982), 298-303.

7. [Same83] - H. Samet and R. E. Webber, Using quadtrees to represent polygonal maps, *Proceedings of Computer Vision and Pattern Recognition 83*, Washington, DC, June 1983, 127-132 (also University of Maryland Computer Science TR-1372).

8. [Same84a] - H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys 16*, 2(June 1984), 187-260 (also University of Maryland Computer Science TR-1329).

9. [Same84b] - H. Samet, A. Rosenfeld, C.A. Shaffer, R.C. Nelson, and Y.-G. Huang, Application of hierarchical data structures to geographical information systems, Phase III, Computer Science TR 1457, University of Maryland, College Park, MD, November 1984.

10. [Same84c] - H. Samet and M. Tamminen, Computing geometric properties of images represented by linear quadtrees, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 2(March 1985), 229-240 (also University of Maryland Computer Science TR-1359).

11. [Shne81] - M. Shneier, Two hierarchical linear feature representations: edge pyramids and edge quadtrees, *Computer Graphics and Image Processing 17*, 3(November 1981), 211-224.

12. [Tamm81] - M. Tamminen, The EXCELL method for efficient geometric access to data, *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series No. 34, Helsinki, 1981.

13. [Tamm83] - M. Tamminen, Performance analysis of cell based geometric file organizations, *Computer Vision, Graphics, and Image Processing 24*, 2(November 1983), 168-181.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ETL-0411	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS (PHASE IV)		5. TYPE OF REPORT & PERIOD COVERED Final Report Nov.1984-Nov.1985 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Hanan Samet Azriel Rosenfeld		8. CONTRACT OR GRANT NUMBER(s) DAAK70-81-C-0059
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Vision Laboratory University of Maryland College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.s. Army Engineer Topographic Laboratories Fort Belvoir, VA 22060-5546		12. REPORT DATE December 1985 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*
Approved for public release; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES
Previous reports completed under this contract are ETL-0301 (AD-A124 196), ETL0337 (AD-A134 999), and ETL-0376 (AD-A152 169).

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
Quadtrees
Hierarchical Data Structures
Geographical Information Systems
Linear Edge Quadtree
PM Quadtree
PMR Quadtree
Edge Quadtree
MX Quadtree

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
This document is a report on Phase IV of an investigation of the application of hierarchical data structures to geographical information systems. It deals primarily with developing new structures for storing linear feature data. The attribute attachment package was extended to point and linear feature data.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*