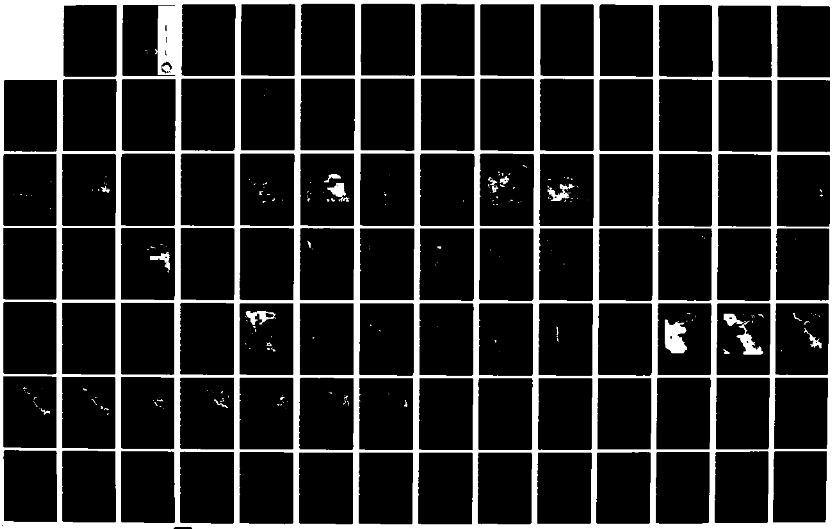
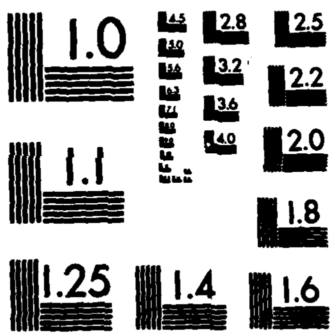
AD-A124 196 APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS(U) MARYLAND UNIV COLLEGE PARK COMPUTER VISION LAB H SAMET ET AL. JUL 82 1/2

UNCLASSIFIED TR-1197 ETL-0301 DAAK70-81-C-0059 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ETL-0301

AD A 124196

Application of hierarchical data structures to geographical information systems

Hanan Samet

Azriel Rosenfeld

Computer Vision Laboratory
University of Maryland
College Park, MD 20742

JULY 1982

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

Prepared for

U.S. ARMY CORPS OF ENGINEERS

ENGINEER TOPOGRAPHIC LABORATORIES

FORT BELVOIR, VIRGINIA 22060

E T L

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER ETL-0301	2. GOVT ACCESSION NO. AD-A124196	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED Contract Report 6. PERFORMING ORG. REPORT NUMBER TR-1197
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 R3205HT09
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Engineer Topographic Laboratories Fort Belvoir, VA 22060		12. REPORT DATE July 1982 13. NUMBER OF PAGES 168
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 unlimited.

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

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Geographical information systems
Data structures
Quadtrees

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document is the final report for an investigation of the application of hierarchical data structures to geographical information systems. 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 to geographic information system organization.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Final Report on
Contract DAAK70-81-C-0059/P00007

APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS

Submitted to:
U.S. Army Engineer
Topographics Laboratories
Fort Belvoir, VA 22060
Attention: Mr. Joseph A. Rastatter

Submitted by:
Computer Vision Laboratory
Computer Science Center
University of Maryland
College Park, MD 20742

Principal Investigators:
Hanan Samet
Azriel Rosenfeld

JULY
~~June 30~~, 1982

PREFACE

This report was produced under 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 Joseph Rastatter.

This report was prepared by Azriel Rosenfeld, Hanan Samet, Cliff Shaffer, and Robert Webber.

SUMMARY

This document is the final report for an investigation of the application of hierarchical data structures to geographical information systems, 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 to geographic information system organization. To accomplish the above objectives, a database was built that contained three maps supplied under the terms of the contract. These maps described the flood plain, elevation contours, and landuse classes of a region in California.

This study report presents the results of the preliminary investigation. It includes analysis of the merits and deficiencies of the various approaches, and provides recommendations for further research.

TABLE OF CONTENTS

FIGURES

FIGURES

TABLES

ALGORITHMS

1. Introduction

This project is concerned with the applicability of a class of hierarchical data structures, known as "quadtrees", to the representation of cartographic data. Section 2 presents a tutorial on quadtree data structures. Section 3 describes the database used, and the process of digitizing and editing it. Section 4 describes the process of quadtree encoding of the data, including algorithms and space/time/acreage tables. Section 5 discusses region analysis and manipulations using quadtrees, including algorithms and tables (time, etc.). The algorithms implemented include set theoretic operations on regions, point-in-region determination, region property measurement, and construction of submaps and merged maps. Section 6 presents a bibliography on quadtrees. The facilities used on the project are described in the Appendix.

2. Tutorial on quadtrees

2.1. Introduction

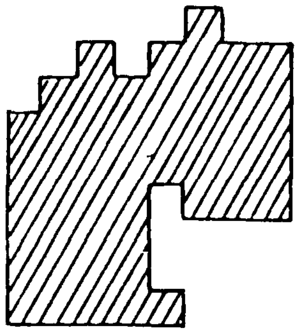
In our discussion we assume that a region is a subset of a 2^n by 2^n array which is viewed as being composed of unit-square pixels. The most common region representations used in image processing are the binary array and the run length representation [1]. The binary array represents region pixels by 1's and non-region pixels by 0's. The run length representation represents each row of the binary array as a sequence of runs of 1's alternating with runs of 0's.

Boundaries of regions are often specified as a sequence of unit vectors in the principal directions. This representation is termed a chain code [2]. For example, letting i represent $90^\circ * i$ (i=0,1,2,3), we have the following sequence as the chain code for the region in Figure 2.1a:

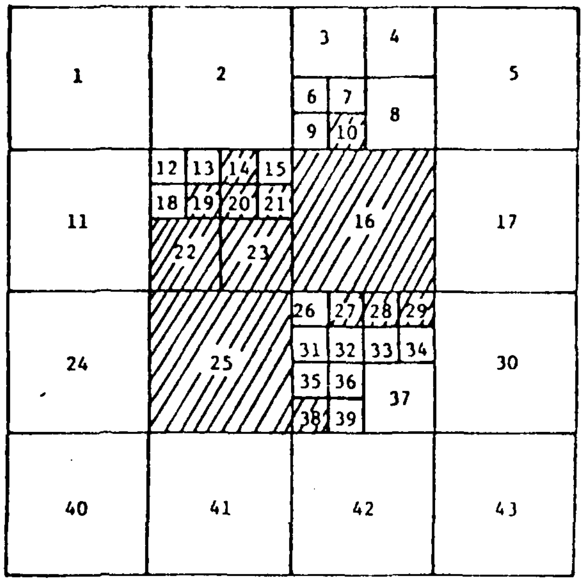
$$030^2 3^5 2^3 123^3 032^5 1^6 0101030101$$

Note that this is a clockwise code which starts at the leftmost of the uppermost border points. Chain codes yield a compact representation; however, they are somewhat inconvenient for performing operations such as set union and intersection. For an alternative boundary representation see the strip trees of Ballard [3].

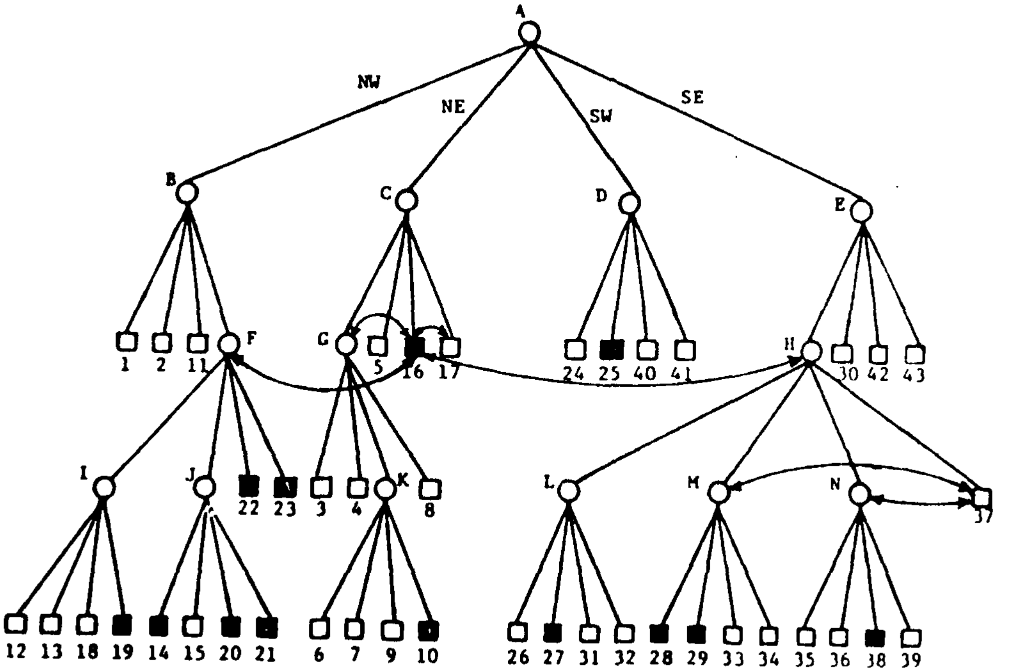
Regions can also be represented by a collection of maximal blocks that are contained in the given region. One such trivial representation is the run length where the blocks are 1 by m rectangles. A more general representation treats the region as a union of maximal blocks (of 1's) of a given shape. The medial axis transform (MAT) [4,5] is the set of points serving as centers of these blocks and their



a. Region



b. Block decomposition of the region in (a).



c. Quadtree representation of the blocks in (b).

Figure 2.1. A region, its maximal blocks, and the corresponding quadtree. Blocks in the region are shaded, background blocks are blank. Horizontal lines indicate ropes.

corresponding radii.

The quadtree is a maximal block representation in which the blocks have standard sizes and positions (i.e., powers of two). It is an approach to region representation which is based on the successive subdivision of an image array into quadrants. If the array does not consist entirely of 1's or entirely of 0's, then we subdivide it into quadrants, subquadrants,... until we obtain blocks (possibly single pixels) that consist of 1's or of 0's, i.e., they are entirely contained in the region or entirely disjoint from it. This process is represented by a tree of out degree 4 (i.e., each non-leaf node has four sons) in which the root node represents the entire array. The four sons of the root node represent the quadrants (labeled in order NW, NE, SW, SE), and the leaf nodes correspond to those blocks of the array for which no further subdivision is necessary. Leaf nodes are said to be "black" or "white" depending on whether their corresponding blocks are entirely within or outside of the region respectively. All non-leaf nodes are said to be "gray". Since the array was assumed to be 2^n by 2^n, the tree height is at most n. As an example, Figure 2.1b is a block decomposition of the region in Figure 2.1a while Figure 2.1c is the corresponding quadtree. Each quadtree node is implemented, storage-wise, as a record with six fields. Five fields contain pointers to the four sons and the father of a node. The sixth field contains type information such as color, etc. Note that the quadtree representation discussed here should not be confused with the quadtree representation of two-dimensional point space data introduced by Finkel and Bentley [6] and also discussed in [7,8] and improved upon in [9].

The quadtree method of region representation is based on a regular decomposition. It has been employed in the domains of computer graphics, scene analysis, architectural design [10], and pattern recognition. In particular, Warnock's [10-13] algorithm for hidden surface elimination is based on such a principle--i.e., it successively subdivides the picture into smaller and smaller squares in the process of searching for areas to be displayed. Application of the quadtree to image representation was proposed by Klinger [14] and further elaborated upon in [15-20]. It is relatively compact [15] and is well suited to operations such as union and intersection [21-23], and detecting various region properties [15,21,22, 24]. Hunter's Ph.D. thesis [21,22,24], in the domain of computer graphics, develops a variety of algorithms (including linear transformations) for the manipulation of a quadtree region representation. In [25-27] variations of the quadtree are applied in three dimensions to represent solid objects and in [28] to more dimensions.

There has been much work recently on the interchangeability between the quadtree and other traditional methods of region representation. Algorithms have been developed for converting a binary array to a quadtree [29], run lengths to a quadtree [30] and a quadtree to run lengths [31], as well as boundary codes to a quadtree [32] and a quadtree to boundary codes [33]. Work has also been done in computing geometric properties such as connected component labeling [34], perimeter [35], Euler number [36], areas and moments [23], as well as a distance transform [37,38]. In addition, the quadtree has been used in image processing applications such as shape approximation [39], edge enhancement [40], image segmentation [41], threshold selection [42], and smoothing [43].

2.2. Preliminaries

In the quadtree representation, by virtue of its tree-like nature, most operations are carried out by techniques which traverse the tree. In fact, many of the operations that we describe can be characterized as having two basic steps. The first step either traverses the quadtree in a specified order or constructs a quadtree. The second step performs a computation at each node which often makes use of its neighboring nodes, i.e., nodes representing image blocks that are adjacent to the given node's block. For examples, see [30-38]. Frequently, these two steps are performed in parallel.

In general, it is preferable to avoid having to use position (i.e., coordinates) and size information when making relative transitions (i.e., locating neighboring nodes) in the quadtree since they involve computation (rather than simply chasing links) and are clumsy when adjacent blocks are of different sizes (e.g., when a neighboring block is larger). Similarly, we do not assume that there are links from a node to its neighbors, because we do not want to use links in excess of four links from a non-leaf node to its sons and the link from a non-root node to its father. Such techniques, described in [44], are used in [30-38] and result in algorithms that only make use of the existing structure of the tree. This is in contrast with the methods of Klinger and Rhodes [19] which make use of size and position information, and those of Hunter and Steiglitz [21, 22,24] which locate neighobrs through the use of explicit links (termed nets and ropes).

Locating neighbors in a given direction is quite straightforward. Given a node corresponding to a specific block in the image, its neighbor in a particular direction (horizontal or vertical) is determined by locating a common ancestor. For example, if we want to find an eastern neighbor, the common ancestor is the first ancestor node which is reached via its NW or SW son. Next, we retrace the path from the common ancestor, but making mirror image moves about the appropriate axis, e.g., to find an eastern or western neighbor, the mirror images of NE and SE are NW and SW, respectively. For example, the eastern neighbor of node 32 in Figure 2.1c is node 33. It is located by ascending the tree until the common ancestor, H, is found. This requires going through a SE link to reach L and a NW link to reach H. Node 33 is now reached by backtracking along the previous path with the appropriate mirror image moves (i.e., going through a NE link to reach M and a SW link to reach 33).

In general, adjacent neighbors need not be of the same size. If they are larger, then only a part of the path to the common ancestor is retraced. If they are smaller, then the retraced path ends at a "gray" node of equal size. Thus a "neighbor" is correctly defined as the smallest adjacent leaf whose corresponding block is of greater than or equal size. If no such node exists, then a gray node of equal size is returned. Note that similar techniques can be used to locate diagonal neighbors (i.e., nodes corresponding to blocks that touch the given node's block at a corner). For example, node 20 in Figure 2.1c is the NW neighbor of node 22. For more details, see [44].

In contrast with our neighbor finding methods is the use of explicit links from a node to its adjacent neighbors in the horizontal and vertical directions reported in [21,22,24]. This is achieved through the use of adjacency trees, "ropes," and "nets." An *adjacency* tree exists whenever a leaf node, say X, has a GRAY neighbor, say Y, of equal size. In such a case, the adjacency tree of X is a binary tree rooted at Y whose nodes consist of all sons of Y (BLACK, WHITE, and GRAY) that are adjacent to X. For example, for node 16 in Figure 2.1, the western neighbor is GRAY node F with an adjacency tree as shown in Figure 2.2. A rope is a link between adjacent nodes of equal size at least one of which is a leaf node. For example, in Figure 2.1, there exists a rope between node 16 and nodes G, 17, H, and F. Similiarly, there exists a rope between node 37 and nodes M and N; however, there does not exist a rope between node L and nodes M and N.

The algorithm for finding a neighbor using a roped quadtree is quite simple. We want a neighbor, say Y, on a given side, say D, of a block, say X. If there is a rope from X on side D, then it leads to the desired neighbor. If no such rope exists, then the desired neighbor must be larger. In such a case, we ascend the tree until encountering a node having a rope on side D, that leads to the desired neighbor. In effect, we have ascended the adjacency tree of Y. For example, to find the eastern neighbor of node 21 in Figure 2.1, we ascend through node J to node F, which has a rope along its eastern side leading to node 16.

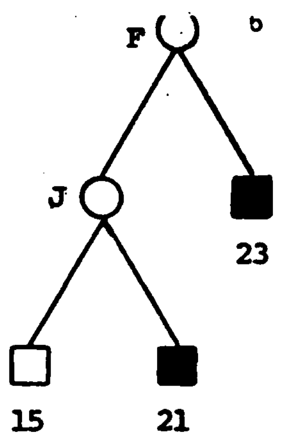


Figure 2.2. Adjacency tree for the western neighbor of node 16 in Figure 2.1.

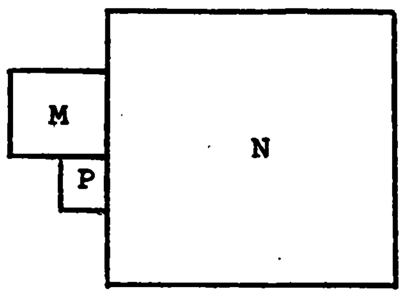


Figure 2.3. Sample pair of blocks illustrating border following.

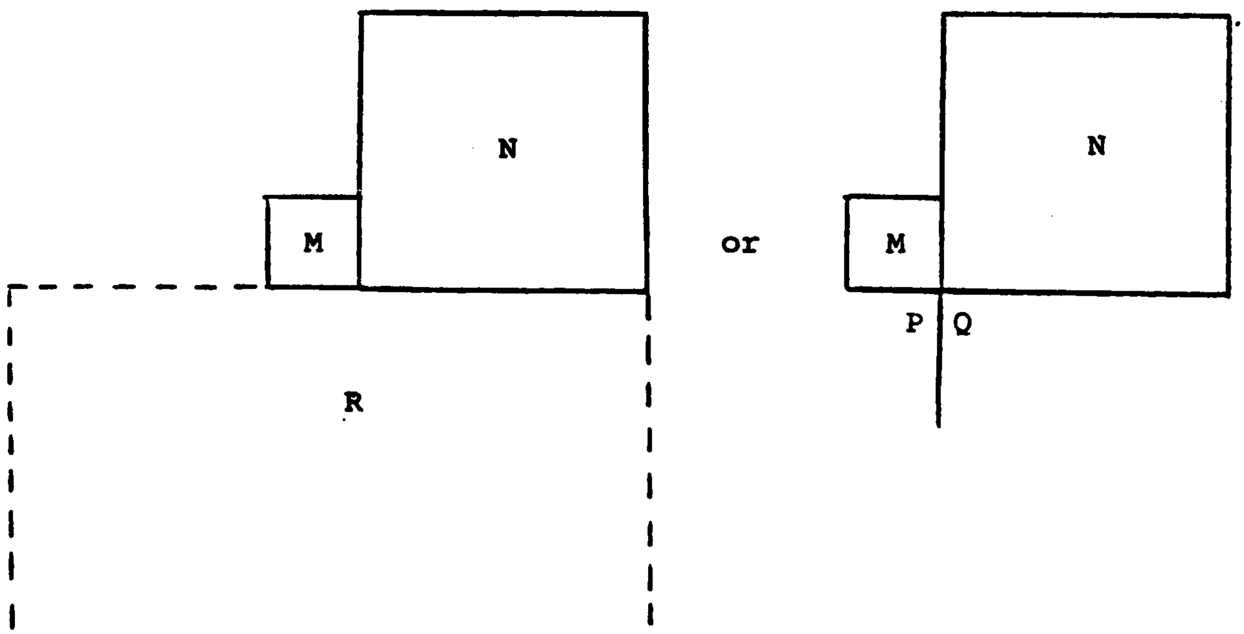


Figure 2.4. Blocks M and N ending at a common corner.

At times it is not convenient to ascend nodes searching for ropes. A data structure named a net is used [21, 22, 24] to obviate this step by linking all leaf nodes to their neighbors regardless of their size. Thus in the previous example there would be a direct link between nodes 21 and 16 along the eastern side of node 21. The advantage of ropes and nets is that the number of links that must be traversed is reduced. However, the disadvantage is that the storage requirements are considerably increased since many additional links are necessary. In contrast, our methods are implemented by algorithms that make use of the existing structure of the tree -- i.e., four links from a nonleaf node to its sons, and a link from a nonroot node to its father.

2.3. Conversion

2.3.1. Quadtrees and Arrays

The definition of a quadtree leads naturally to a "top down" quadtree construction process. This may lead to excessive computation because the process of examining whether a quadtrant contains all 1's or all 0's may cause certain parts of the region to be examined repeatedly by virtue of being composed of a mixture of 1's and 0's. Alternatively, a "bottom-up" method may be employed which scans the picture in the sequence

```
 1  2  5  6  17  18  21  22
 3  4  7  8  19  20  23  24
 9 10 13 14  25  26  29  30
11 12 15 16  27  28  31  32
33 ...
```

where the numbers indicate the sequence in which the pixels are examined. As maximal blocks of 0's or 1's are discovered, corresponding leaf nodes are added along with the necessary ancestor nodes. This is done in such a way that leaf nodes are never created until they are known to be maximal. Thus there is never a need to merge four leaves of the same color and change the color of their common parent from gray to white or black as is appropriate. See [29] for the details of such an algorithm whose execution time is proportional to the number of pixels in the image.

If it is necessary to scan the picture row by row (e.g., when the input is a run length coding) the quadtree construction process is somewhat more complex. We scan the picture a row at a time. For odd-numbered rows, nodes corresponding to the pixel or run values are added for the pixels and attempts are made to discover maximal blocks of 0's or 1's whose size depends on the row number (e.g.,. when processing the fourth row, maximal blocks of maximum size 4-by-4 can be discovered). In such a case merging is said to take place. See [30] for the details of an algorithm that constructs a quadtree from a row by row scan such that at any instance of time a valid quadtree exists. This algorithm has an execution time that is proportional to the number of pixels in the image.

Similarly, for a given quadtree we can output the corresponding binary picture by traversing the tree in such a way that for each row the appropriate blocks are visited and a row of 0's or 1's is output. In essence, we visit each quadtree node once for each row that intersects it (i.e., a node corresponding to a block of size 2^K by 2^K is visited 2^K times). For the details see [31] where an algorithm is described whose execution time depends only on the number of blocks of each size that comprise the image - not on their paticular configuration.

2.3.2. Quadtrees and borders

In order to determine, for a given leaf node M of a quadtree, whether the corresponding block is on the border, we must visit the leaf nodes that correspond to 4-adjacent blocks and check whether they are black or white. For example, to find M's right hand neighbor in Figure 2.3, we use the neighbor finding techniques outlined in Section 2.2. If the neighbor is a leaf node, then its block is at least as large as that of M and so it is M's sole neighbor to the right. Otherwise, the neighbor is the root of a subtree whose leftmost leaf nodes correspond to M's right-hand neighbors. These nodes are found by traversing that subtree.

Let M,N in Figure 2.3 be black and white leaf nodes whose associated blocks are 4-adjacent. Thus the pair M,N defines a common border segment of length 2^K (2^K is the minimum of the side lengths of M and N) which ends at a corner of the smaller of the two blocks (they may both end at a common point as in Figure 2.4). In order to produce a boundary code representation for a region in the image we must determine the next segment along the border whose previous segment lay between M and N. This is achieved by locating the other leaf P whose block touches the end of the segment between M and N. If the M,N segment ends at a corner of both M and N, then we must find the other leaf R or leaves P,Q whose blocks touch that corner (see Figure 2.4) Again, this can be accomplished by using neighbor finding techniques as outlined in Section 2.2.

For the non-common corner case, the next border segment is the common border defined by M and P if P is white, or the common border defined by N and P if P is black. In the common corner case, the pair of blocks defining the next border segment is determined exactly as in the standard "crack following" algorithm [45] for traversing region borders. This process is repeated until we re-encounter the block pair M,N. At this point the entire border has been traversed. The successive border segments constitute a 4-direction chain code, broken up into segments whose lengths are sums of powers of two. The time required for this process is on the order of the number of border nodes times the tree height. For more details see [33].

Using the methods described in the last two paragraphs, we can traverse the quadtree, find all borders, and generate their codes. During this process, we mark each border as we follow it, so that it will not be followed again from a different starting point. Note that the marking process is complicated by the fact that a node's block may be on many different borders.

In order to generate a quadtree from a set of 4-direction chain codes we use a two-step process. First, we trace the boundary in a clockwise direction and construct a quadtree whose black leaf nodes are of a size equal to the unit code length. All the black nodes correspond to blocks on the interior side of the boundary. All remaining nodes are left uncolored. Second, all uncolored nodes are set to black or white as appropriate. This is achieved by traversing the tree, and for each uncolored leaf node, examining its neighbors. The node is colored black unless any of its neighbors is white or is black with a border along the shared boundary. At any stage, merging occurs if the four rows of a non-leaf node are leaves having the same color. The details of the algorithm are given in [32]. The time required is proportional to the product of the perimeter (i.e., the 4-direction chain code length) and the tree height.

2.3.3. Quadtrees of derived sets

Let S be the set of 1's in a given binary array, and let $\bar{S}$ be the complement of S. The quadtree of the complement of S is the same as that of S, with black leaf nodes changed to white and vice versa. To get the quadtree of the union of S and T from those of S and T, we traverse the two trees simultaneously. Where they agree, the new tree is the same and if the two nodes are gray, then their subtrees are traversed. If S has a gray (=nonleaf) node where T has a black node, the new tree gets a black node; if T has a white node there, we copy the subtree of S at that gray node into the new tree. If S has a white node, we copy the subtree of T at the corresponding node. The algorithm for the intersection of S and T is exactly analogous, with the roles of black and white reversed. The time required for these algorithms is proportional to the number of nodes in the smaller of the two trees [23].

2.3.4. Skeletons and medial axis transforms

The medial axis of a region is a subset of its points each of which has a distance from the complement of the region (using a suitably defined distance metric) which is a local maximum. The medial axis transform (MAT) consists of the set of medial axis or "skeleton" points and their associated distance values. The quadtree representation may be rendered even more compact by the use of a skeleton-like representation. Recall that a quadtree is a set of disjoint maximal square blocks having sides whose lengths are powers of 2. We define a quadtree skeleton to be a set of maximal square blocks having sides whose lengths are sums of powers of two. The maximum value (i.e., "chessboard") distance metric [45] is the most appropriate for an image represented by a quadtree. See [37] for the details of its computation for a quadtree; see also [38] for a different quadtree distance transform. A quadtree medial axis transform (QMAT) is a quadtree whose black nodes correspond to members of the quadtree skeleton while all remaining leaf nodes are white. The QMAT has several important properties. First, it results in a partition of the image into a set of possibly non-disjoint squares having sides whose lengths are sums of powers of two rather than, as is the case with quadtrees, a set of disjoint squares having sides of lengths which are powers of two. Second, the QMAT is more compact than the quadtree and has a decreased shift sensitivity. See [46] for the details of a quadtree to QMAT conversion algorithm whose execution time is on the order of the number of nodes in the tree.

2.4. Property measurement

2.4.1. Connected component labeling

Traditionally, connected component labeling is achieved by scanning a binary array row by row from left to right and labeling adjacencies that are discovered to the right and downward. During this process equivalences will be generated. A subsequent pass merges these equivalences and updates the labels of the affected pixels. In the case of the quadtree representation we also scan the image in a sequential manner. However, the sequence's order is dictated by the tree structure - i.e., we traverse the tree in postorder. Whenever a black leaf node is encountered all black nodes that are adjacent to its south and east sides are also visited and are labeled accordingly. Again, equivalences generated during this traversal are subsequently merged and a tree traversal is used to update the labels. The interesting result is that the algorithm's execution time is proportional to the number of pixels. An analgous result is described in the next section. See [34] for the details of an algorithm that labels connected components in time on the order of the number of nodes in the tree plus the product of B·log B where B is the number of black leaf nodes.

2.4.2. Component counting and genus computation

Once the connected components have been labeled, it is trivial to count them, since their number is the same as the number of inequivalent labels. We will next describe a method of determining the number of components minus the number of holes by counting certain types of local patterns in the array; this number, g, is known as the genus or Euler number of the array.

Let V be the number of 1's, E the number of horizontally adjacent pairs of 1's (i.e., 11) and vertically adjacent pairs of 1's, and F the number of two by two arrays of 1's in the array; it is well known [45] that g=V-E+F. This result can be generalized to the case where the array is represented by a quadtree [36]. In fact, let V be the number of black leaf nodes; E the number of pairs of such nodes whose blocks are horizontally or vertically adjacent; and F the number of triples or quadruples of such nodes whose blocks meet at and surround a common point (see Figure 2.5). Then g=V-E+F. These adjacencies can be found (see section 2.3.2) by traversing the tree; the time required is on the order of the number of nodes in the tree.

2.4.3. Area and moments

The area of a region represented by a quadtree can be obtained by summing the areas of the black leaf nodes, i.e., counting 4^k for each such node that represents a 2^k by 2^k block. Similarly, the first x and y moments of the region relative to a given origin can be computed by summing the first moments of these blocks; note that we know the position (and size) of each block from the coordinates of its leaf in the tree. Knowing the area and the first moments gives us the coordinates of the centroid, and we can then compute central moments relative to the centroid as the origin. The time required for any of these computations is proportional to the number of nodes in the tree. Further details on moment computation from quadtrees can be found in [23].

2.4.4. Perimeter

An obvious way of obtaining the perimeter of a region represented by a quadtree is to simply traverse its border and sum the number of steps. However, there is no need to traverse the border segments in order. Instead, we use a method which traverses the tree in postorder and for each black leaf node examines the colors of its neighbors on its four sides. For each white neighbor the length of the corresponding border segment is included in the perimeter.

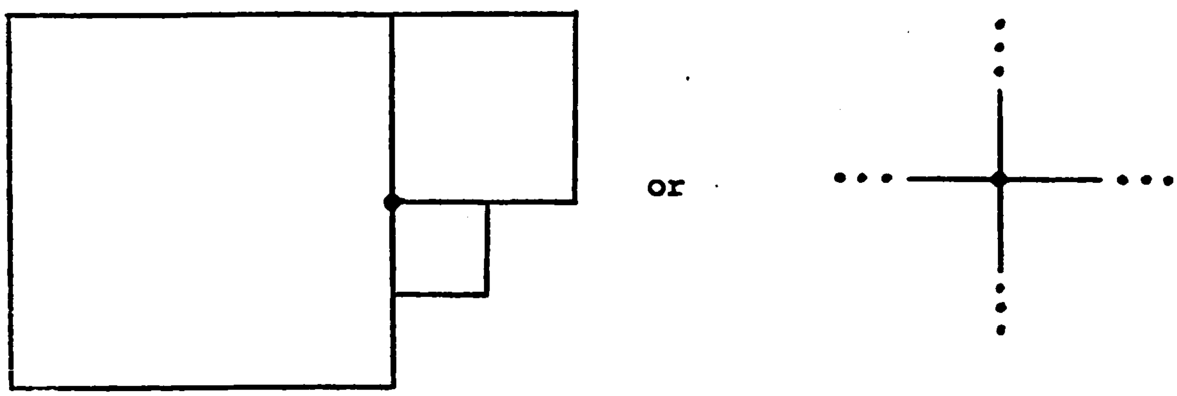


Figure 2.5. Possible configurations of blocks that meet at and surround a common point.

See [35] for the details of such an algorithm which has execution time proportional to the number of nodes in the tree. An even better formulation is reported in [47] which generalizes the concept of perimeter to n dimensions.

2.5. Concluding remarks

We have briefly sketched algorithms for accomplishing traditional region processing operations by use of the quadtree representation. Many of the methods used on the pixel level carry over to the quadtree domain (e.g., connected component labeling, genus, etc.). Because of its compactness, the quadtree permits faster execution of these operations. Often the quadtree algorithms require time proportional to the number of blocks in the image, independent of their size.

The quadtree data structure requires storage for the various links. However, use of neighbor finding techniques rather than ropes a la Hunter [21, 22, 24] is a compromise. In fact, experimental results discussed in the data analysis segment of this report show that the extra storage cost of ropes is not justified by the resulting minor decrease in execution time. This is because the average number of links traversed by neighbor finding methods is 3.5 in contrast with 1.5 for ropes. Nevertheless, there is a possibility that the quadtree may not be efficient spacewise. For example, a checkerboard-like region does not lead to economy of space. The space efficiency of the quadtree is analyzed in [48]. Some savings can be obtained by normalizing the quadtree [49,50] as is also possible by constructing a forest of quadtrees [51] to avoid large regions of WHITE. Storage can also be saved by using a locational code for all BLACK blocks [52]. Gray level quadtrees using a sequence of array codes to economize on storage are reported in [53].

The quadtree is especially useful for point in polygon operations as well as for query operations involving image overlays and set operations. The hierarchical nature enables one to use image approximations. In particular, a breadth-first transmission of an image yields a successively finer image yet enabling the user to have a partial image. Thus the quadtree could be used in browsing through a large image database.

Quadtrees constitute an interesting alternative to the standard methods of digitally representing regions. Their chief disadvantage is that they are not shift-invariant; two regions differing only by a translation may have quite different quadtrees (but see [46]). Thus shape matching from quadtrees is not straightforward. Nevertheless, in other respects, they have many potential advantages. They provide a compact and easily constructed representation from which standard region properties can be efficiently computed. In effect, they are "variable-resolution arrays" in which detail is represented only when it is available, without requiring excessive storage for parts of the image where detail is missing. Their variable-resolution property is superior to trees based on a hexagonal decomposition [54] in that a square can be repeatedly decomposed into smaller squares (as can be done for triangles as well [55]) whereas once the smallest hexagon has been chosen it can not be further decomposed into smaller hexagons. Note that the variance of resolution only applies to the area. For an application of the quadtree concept to borders, as well as area, see the line quadtree of [56].

3. Database, digitization, and editing

3.1. Procedures and results

The data supplied by ETL consisted of three map overlays (Figures 3.1-3) representing land use classes, terrain elevation contours, and flood plain boundaries for a small area of Northern California. These overlays are shown, at a reduced scale, in the figures attached to this section. In the case of the elevation contours, only those at multiples of 100 feet were to be digitized, and for all three overlays, only the portions bounded by the fiducial marks.

Conversion of the data to machine-readable form was carried out as follows: Each overlay was superimposed on a grid (graph paper, 20 boxes to the inch). The boundaries to be digitized were followed by hand and marked on a second sheet of graph paper. Every box on the original graph was copied onto a 2-by-2 block of boxes on the second sheet. This yielded increased resolution and also separated boundary lines which on the original graph would have been in adjacent boxes. This graph was then hand chain-coded and the chain-codes were typed into the computer (see the description of the program "mkbin" for a definition of the chain-code used).

A binary array was created for each of the three overlays in which the pixels that were on a boundary in the original overlay are represented by a value of 1, and all other pixels are represented by a value of 0. A connected component labeling program was then applied to this array yielding an array in which the pixels in each connected region have a unique label. (Pixels of value 1 were regarded as connected even if they were only diagonally adjacent, whereas pixels of value 0 were regarded as connected only if they were horizontally or vertically adjacent.) A lookup table was then created to convert these labels to a consistent label set in which all regions of a given land-use class,or all regions between a given pair of elevation contours, had the same label. At this time, all polygons on the landuse map which either had no label or for which the label was unclear were placed in a special landuse class "unk".

The final data preparation task was to remove the boundary lines separating the regions (those pixels given a value of 1 in the binary array). This was uniformly done by assigning to each boundary pixel the label of its right-hand neighbor, or if this was also a boundary pixel, the label of its neighbor in the row above. The three digital maps (one per overlay) resulting from this processing were 450 pixels high by 400 wide, partitioned into labelled regions with no "black" boundary lines separating them.

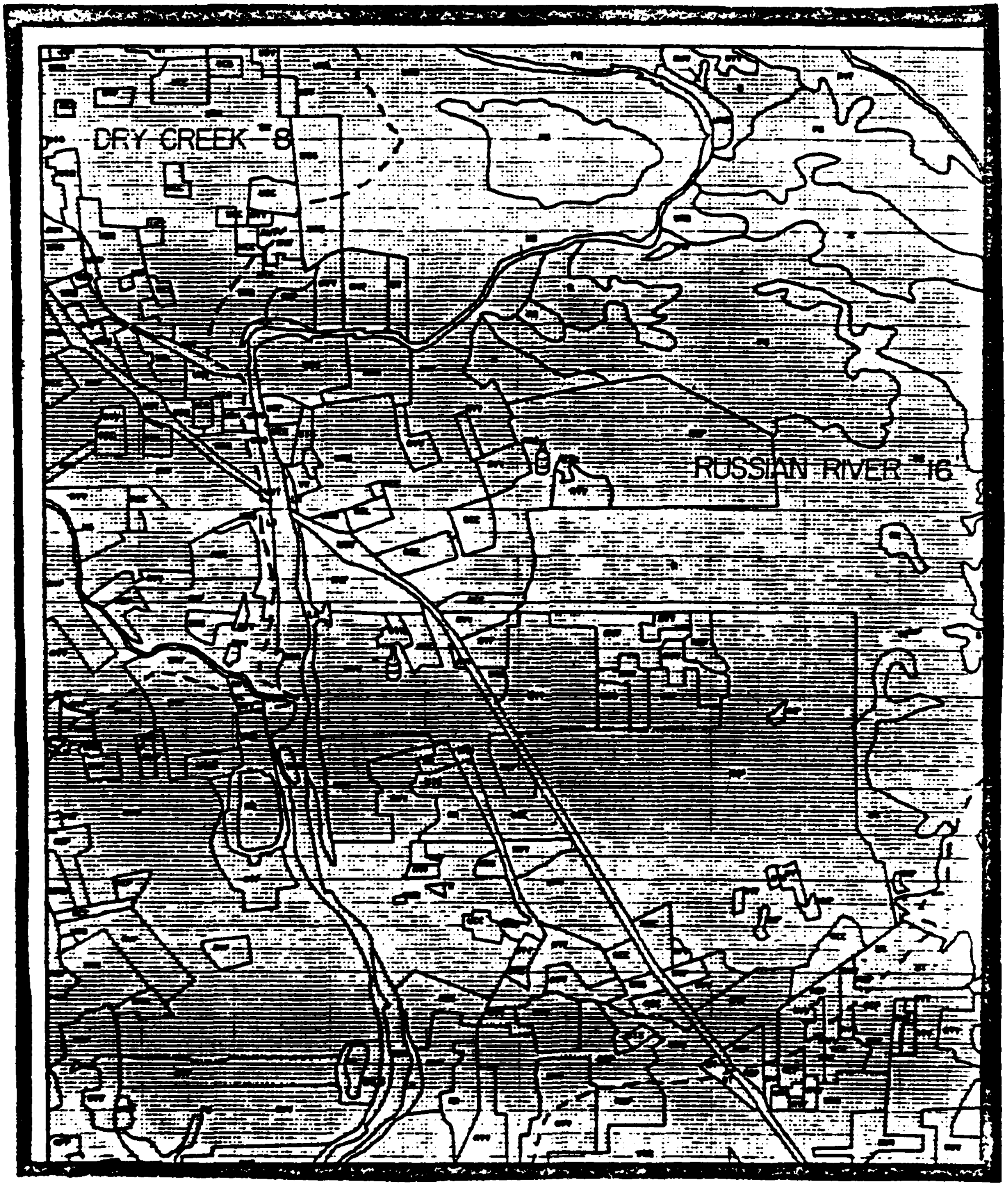


Figure 3.1. Land use classes.

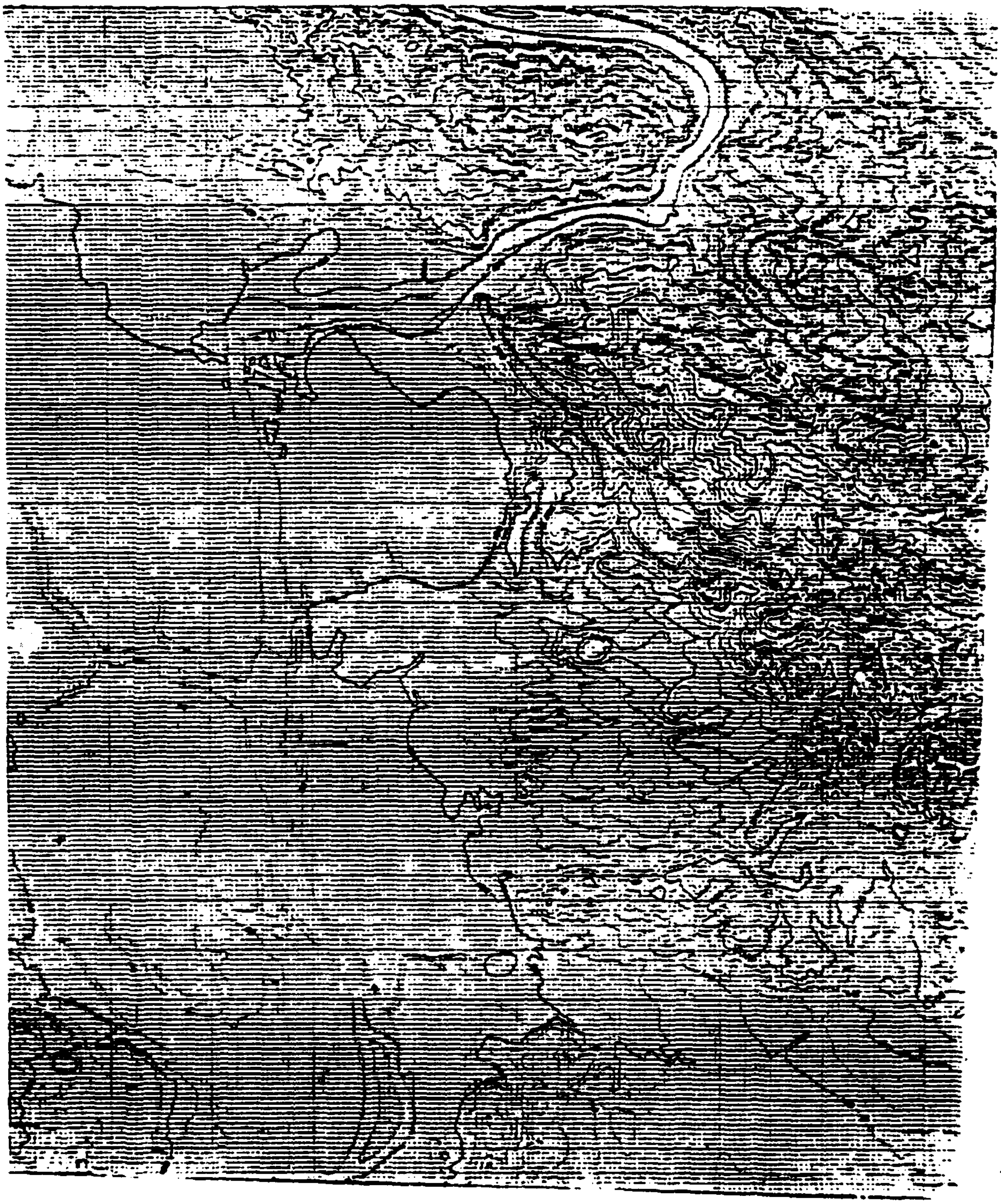


Figure 3.2. Elevation contours.

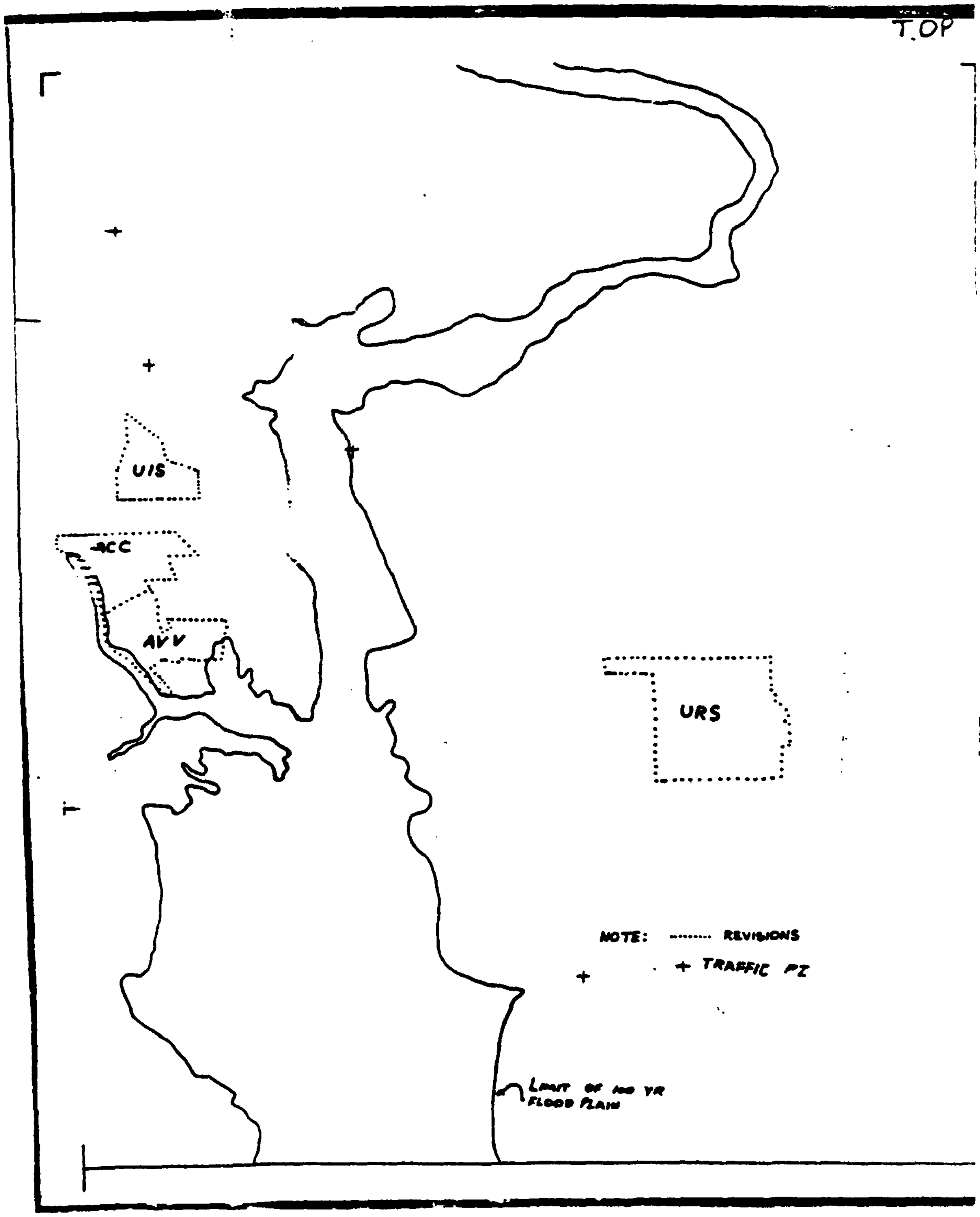


Figure 3.3. Flood plain boundaries.

The resulting maps were then quadtree encoded using the quadtree building algorithm described in the Section 4. For each map, a multi-color quadtree was built giving every region in the map a unique label. A multi-color quadtree refers to a quadtree in which the leaf nodes can have different colors. Thus a multi-color quadtree is an extension of the black and white (or binary) quadtree that is discussed in Section 2. Certain operations, for example union and intersection, are not defined in terms of multi-color quadtrees, but rather are defined in terms of binary quadtrees that are derived from the multi-color quadtrees by considering one of the colors as black (this is usually the color of the object of interest) and the other colors to be white. In addition to the above three multi-color quadtrees, a quadtree was built for each land-use class or elevation level (i.e. regions in the class or elevation are labeled, all other regions are white). Programs were then written to manipulate and display quadtrees, as well as calculate region properties and compute set theoretic operations on the trees. These programs are described in later sections.

The hand digitization process took approximately 100 manhours, including both planning and implementation. This time could be greatly shortened by using coordinate digitizing equipment. Editing of the hand-input data was carried out by visual inspection of the resulting regions to verify that there were no gaps or overlaps. This process, together with a few hand corrections of touching lines, took at most 20 manhours.

Figures 3.4-3.38 show the components of each land use class. Figures 3.39-3.49 show the components of each elevation level, and Figure 3.50 shows the three components of the flood plain map.

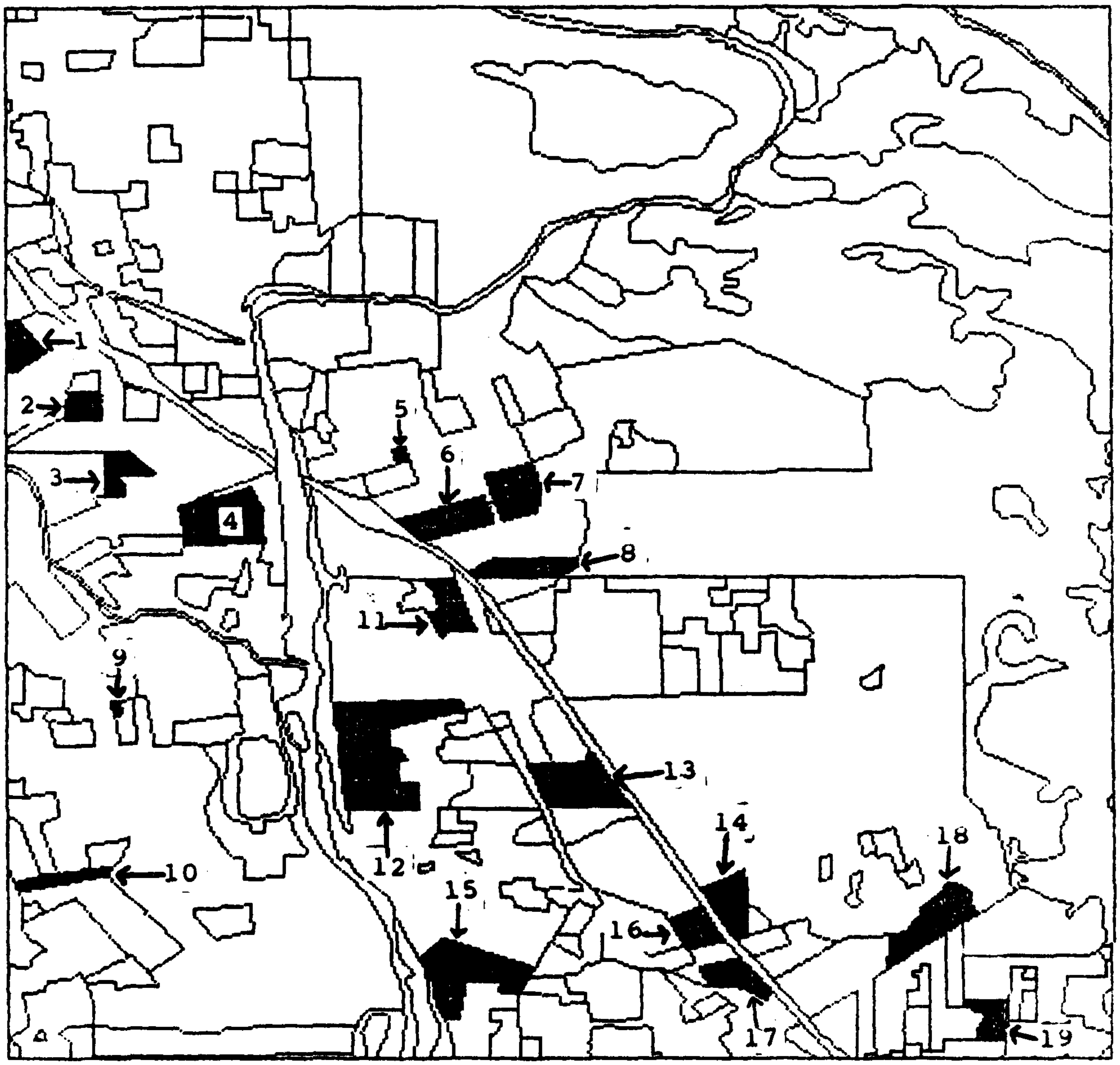


Figure 3.4. The 19 components of the land-use class ACC.

Figure 3.5. The 13 components of the land-use class ACP.

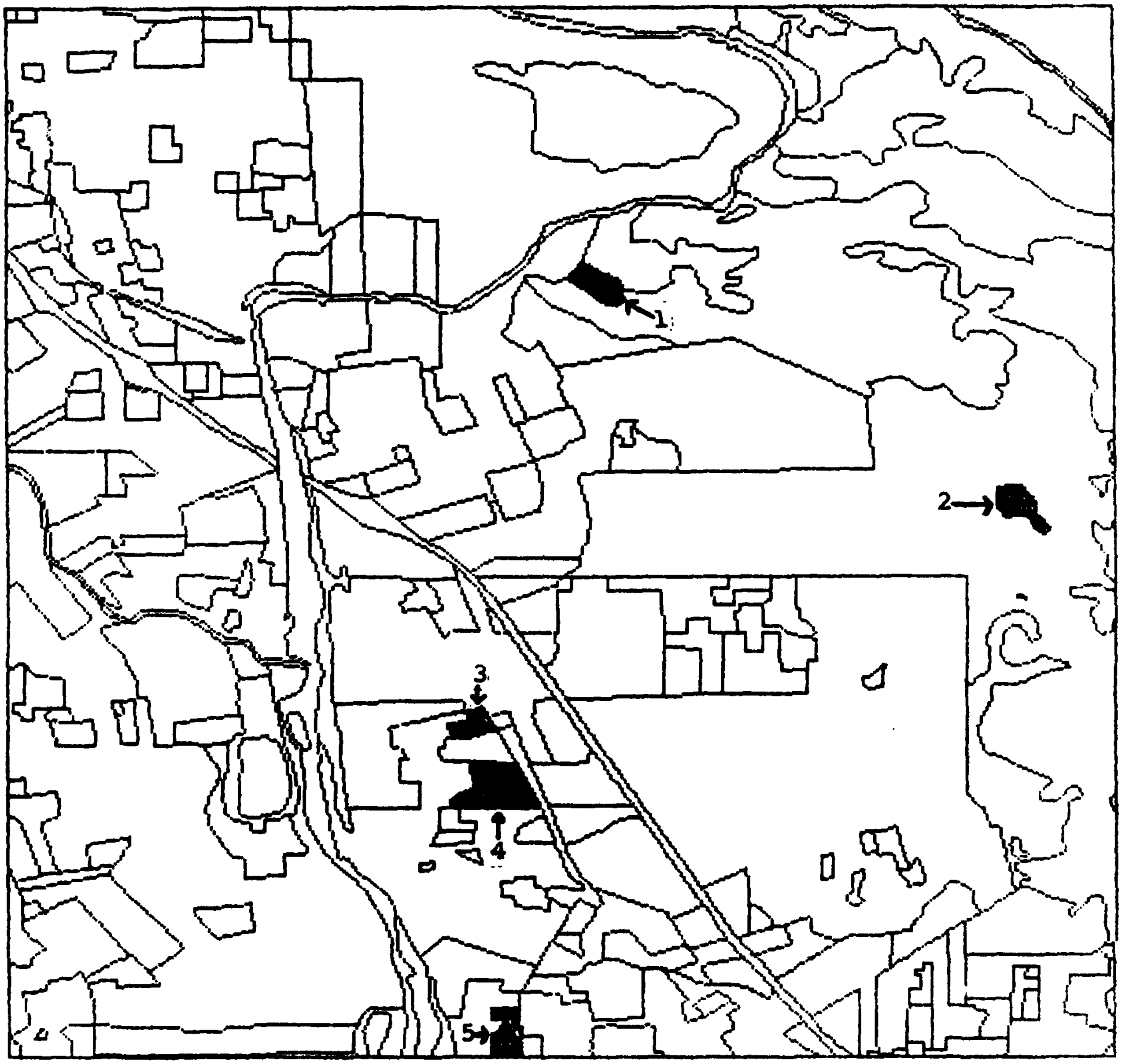


Figure 3.6. The 5 components of the land-use class AR.

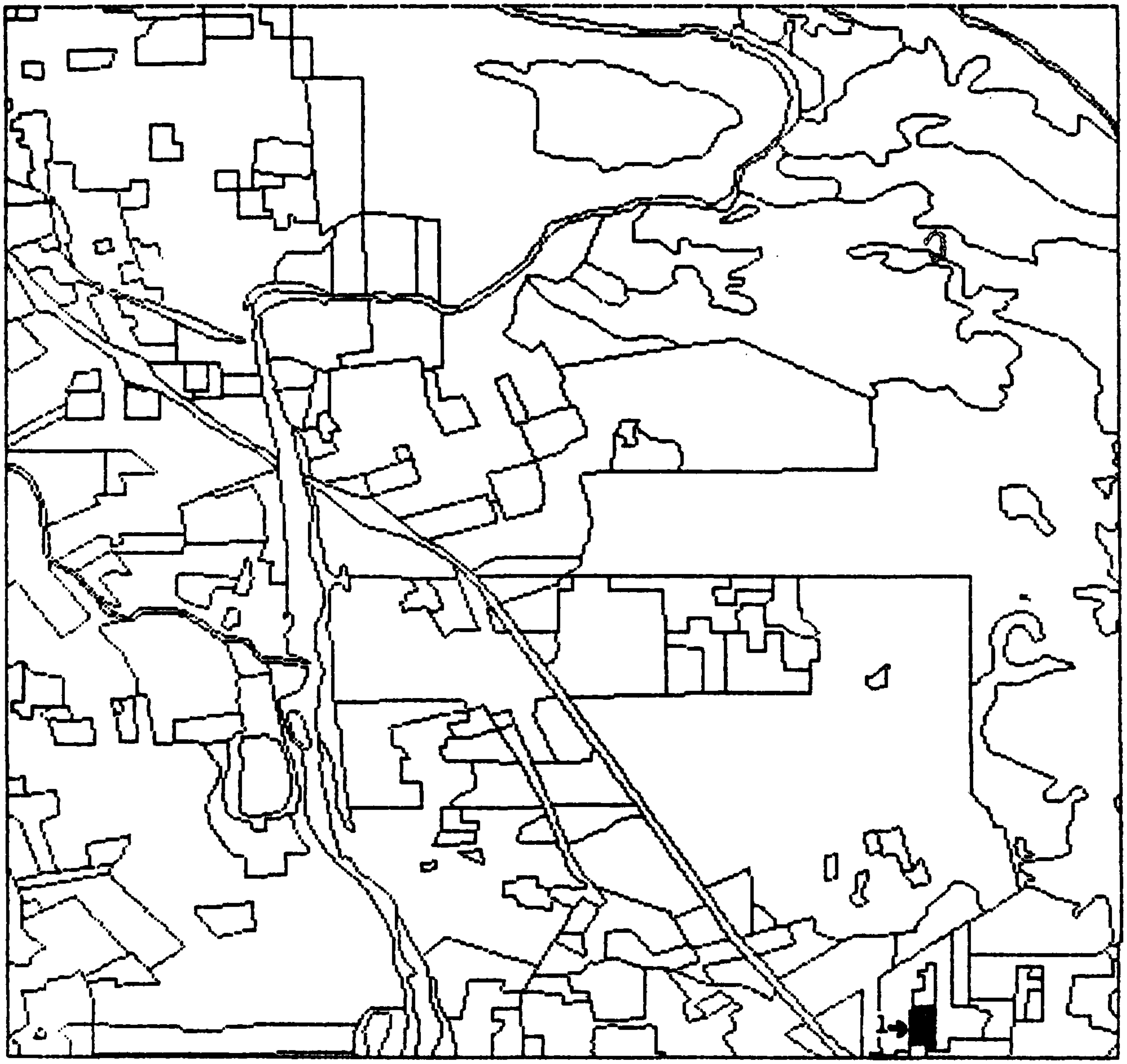


Figure 3.7. The 1 component of the land-use class ARE.

Figure 3.6. The 31 components of the land-use class AVF.

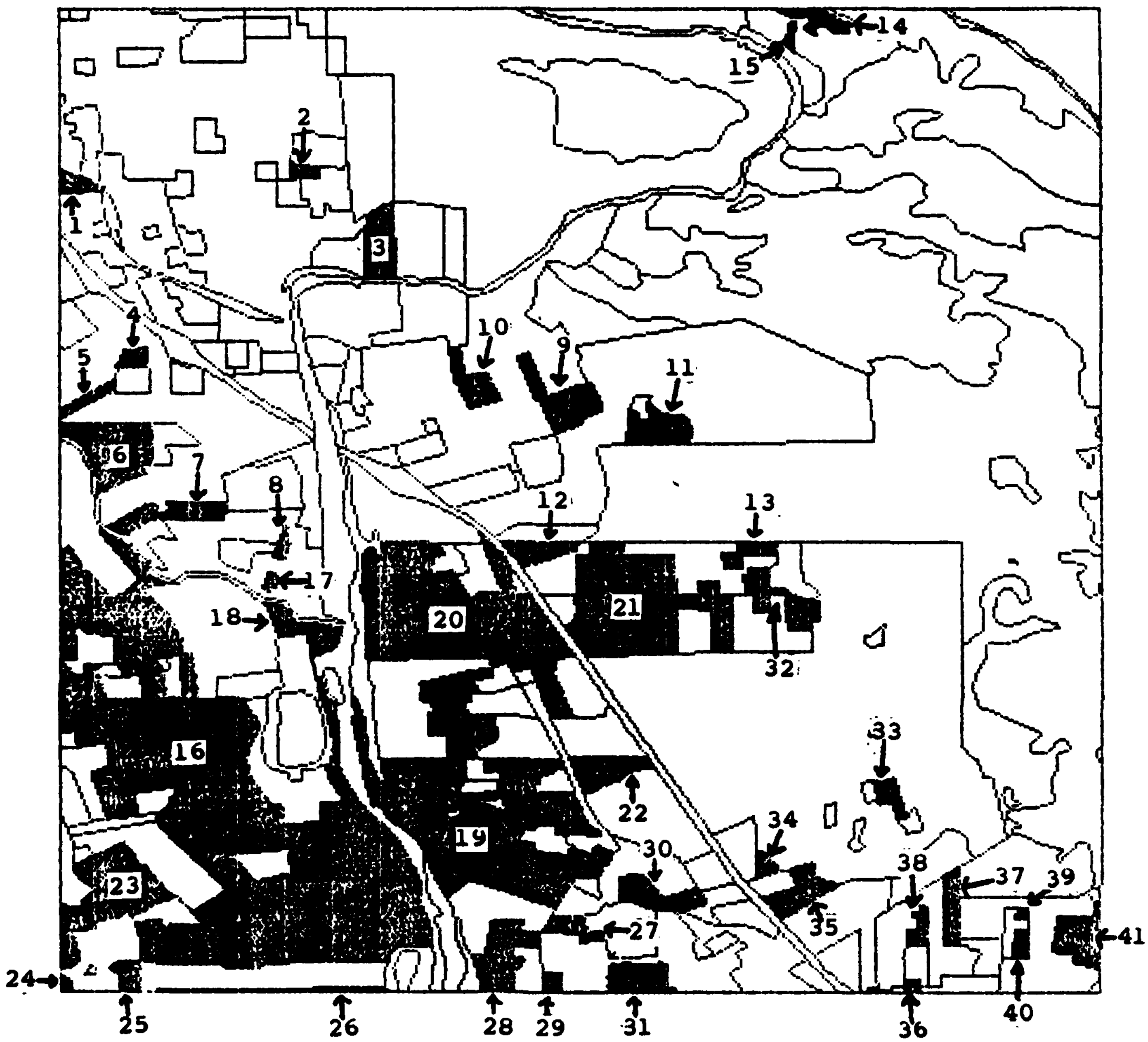


Figure 3.9. The 41 components of the land-use class AVV.

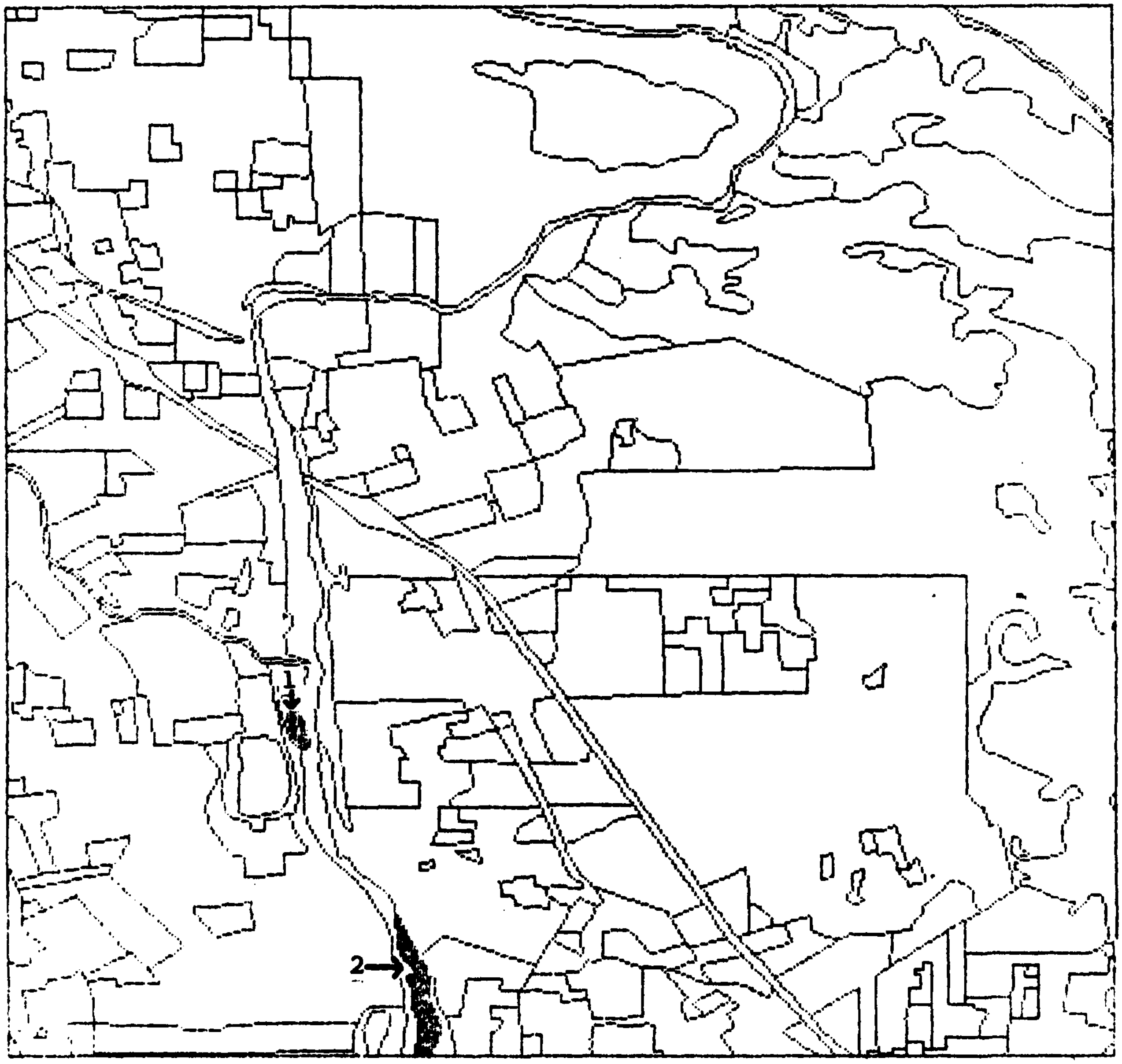


Figure 3.10. The 2 components of the land-use class BBR.

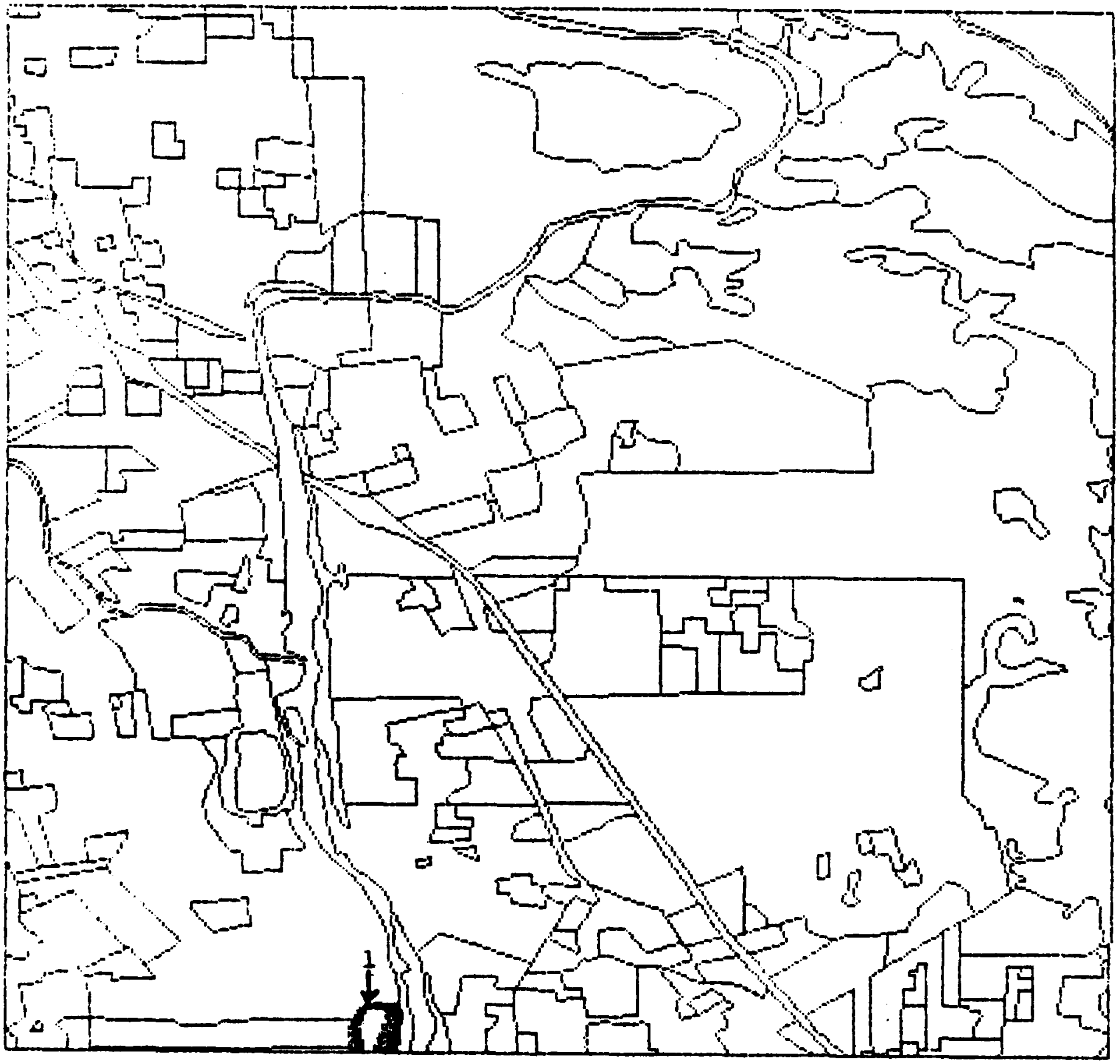


Figure 3.11. The 1 component of the land-use class BLQ.

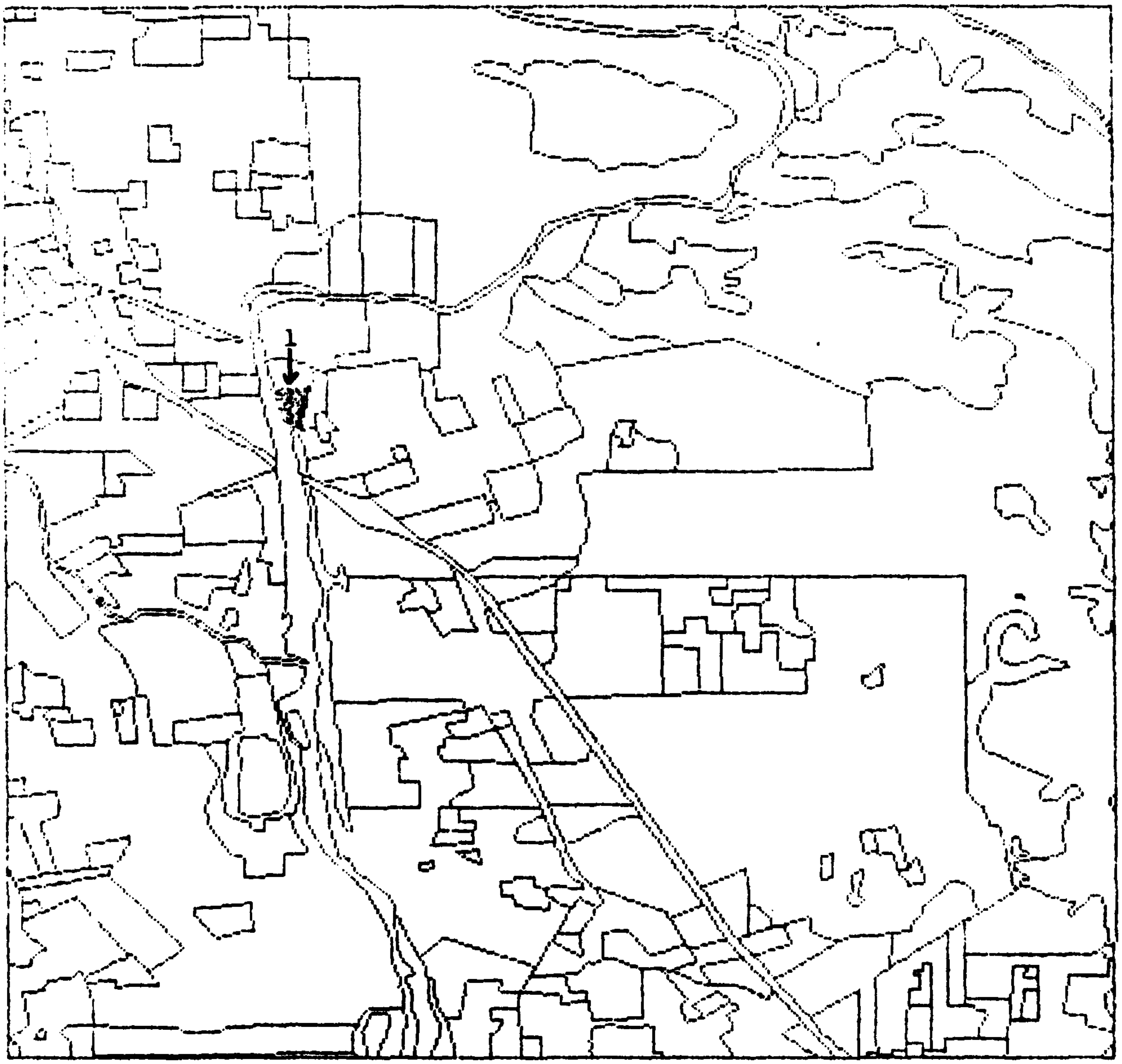


Figure 3.12. The 1 component of the land-use class BES.

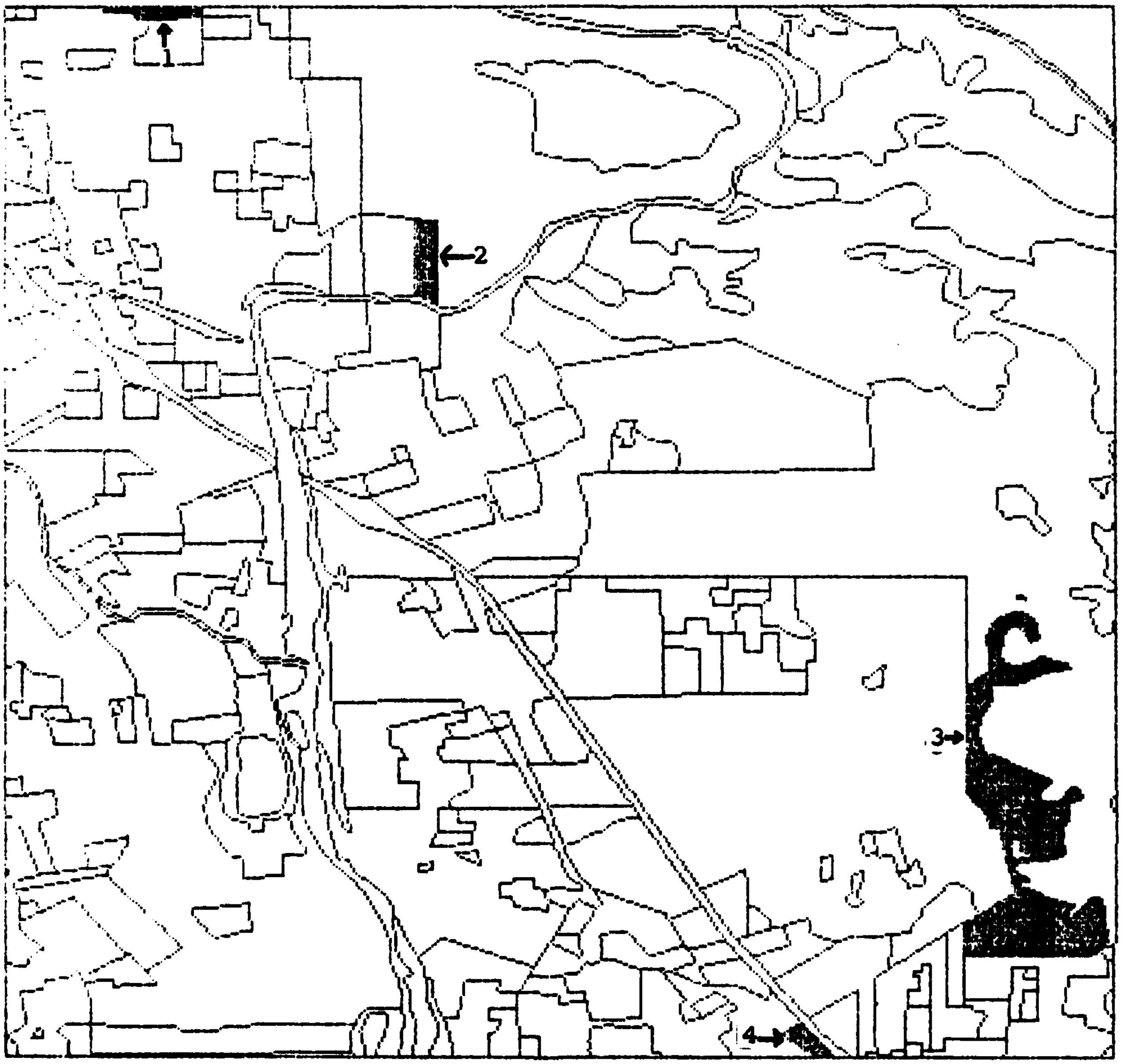


Figure 3.13. The 4 components of the land-use class BT.

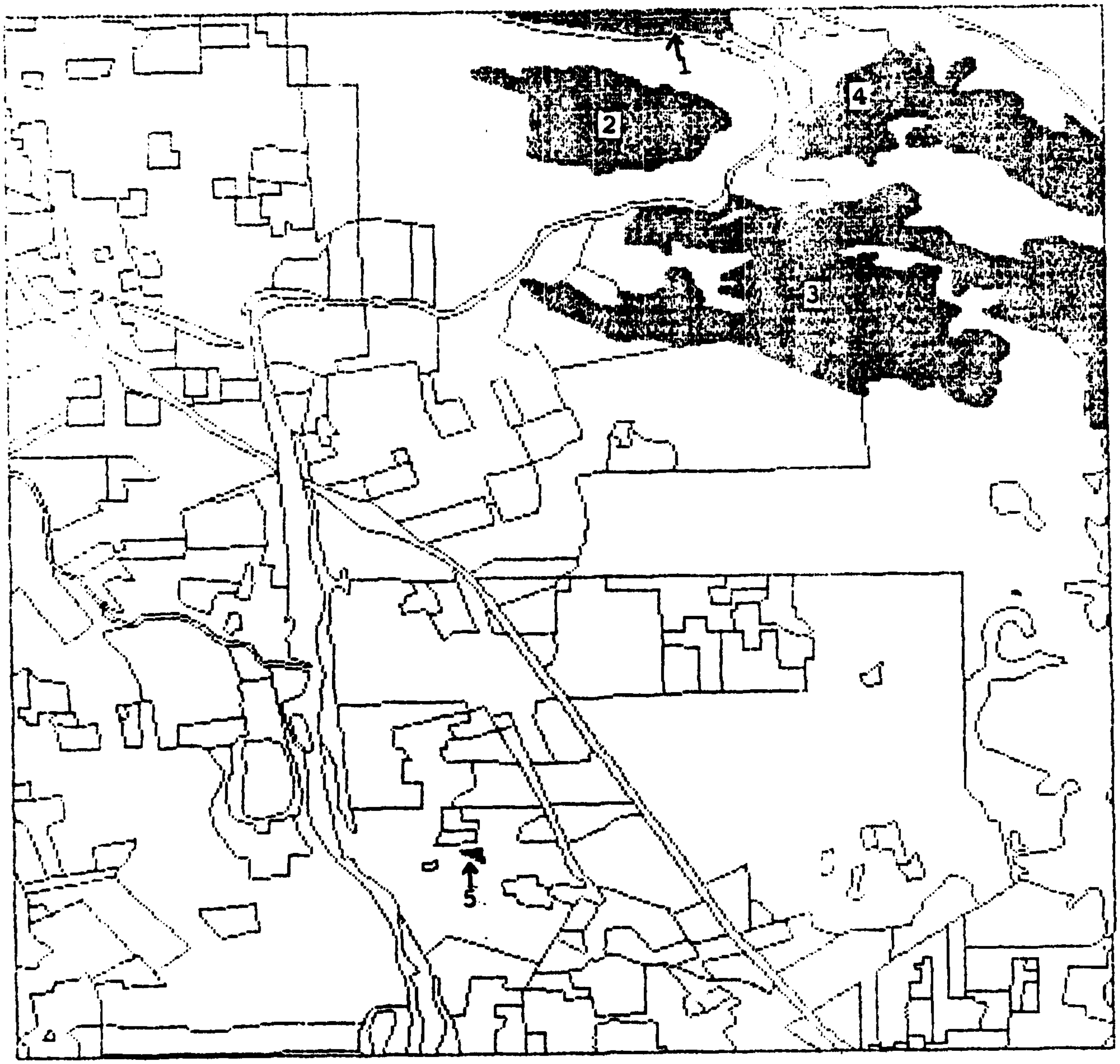


Figure 3.14. The 5 components of the land-use class FO.

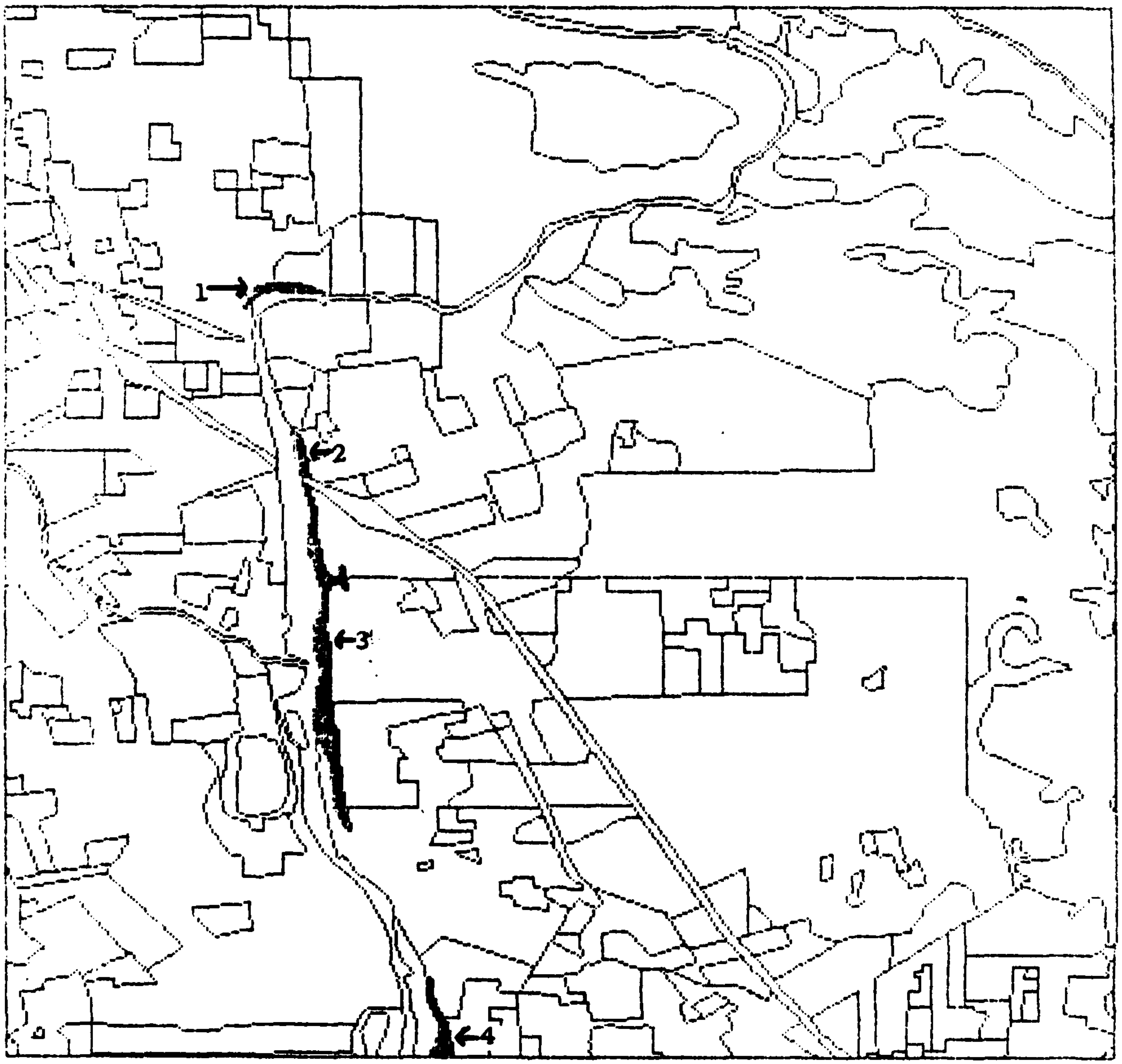


Figure 3.15. The 4 components of the land-use class LR.

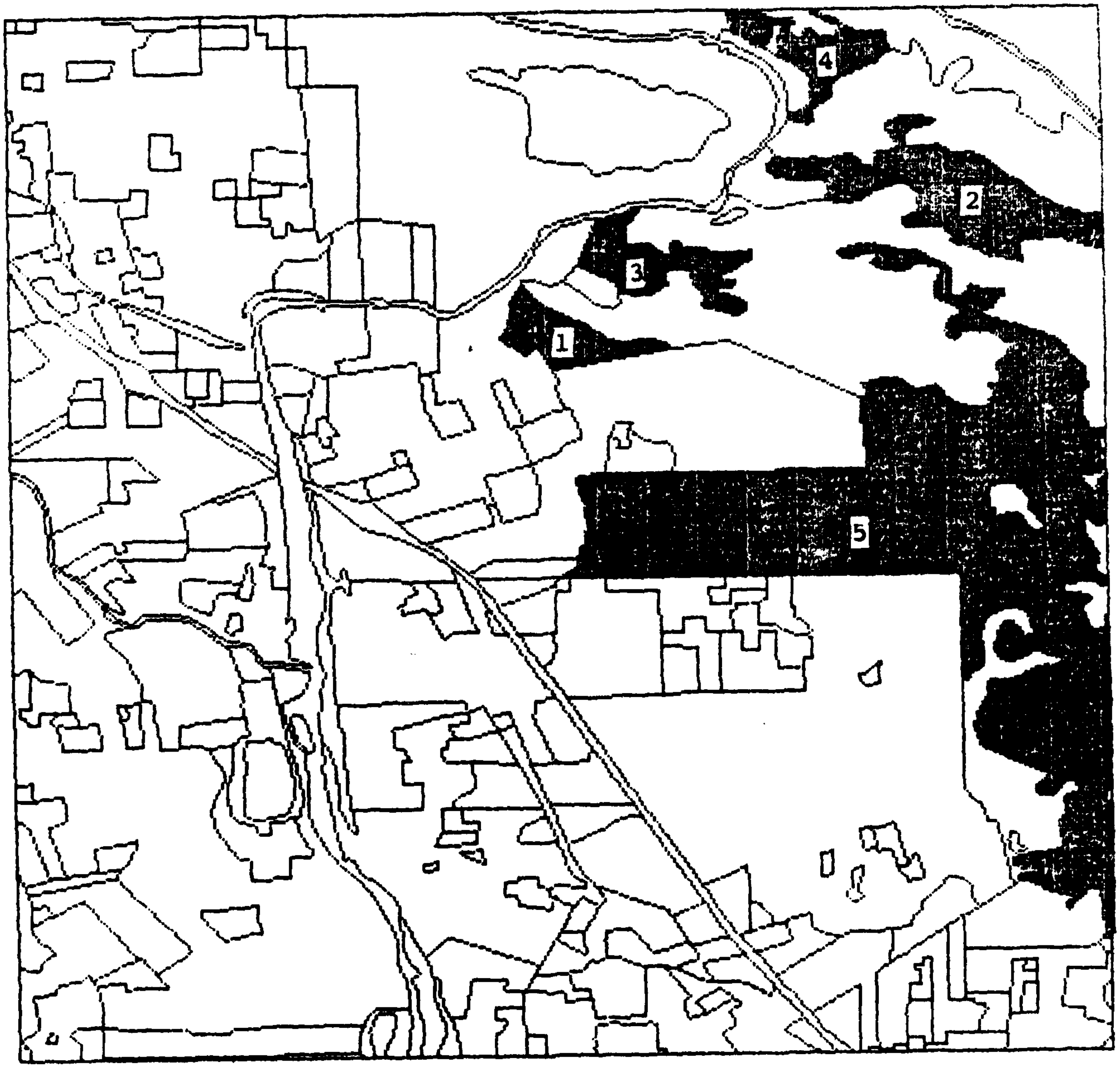


Figure 3.10. The 5 components of the land-use class K.

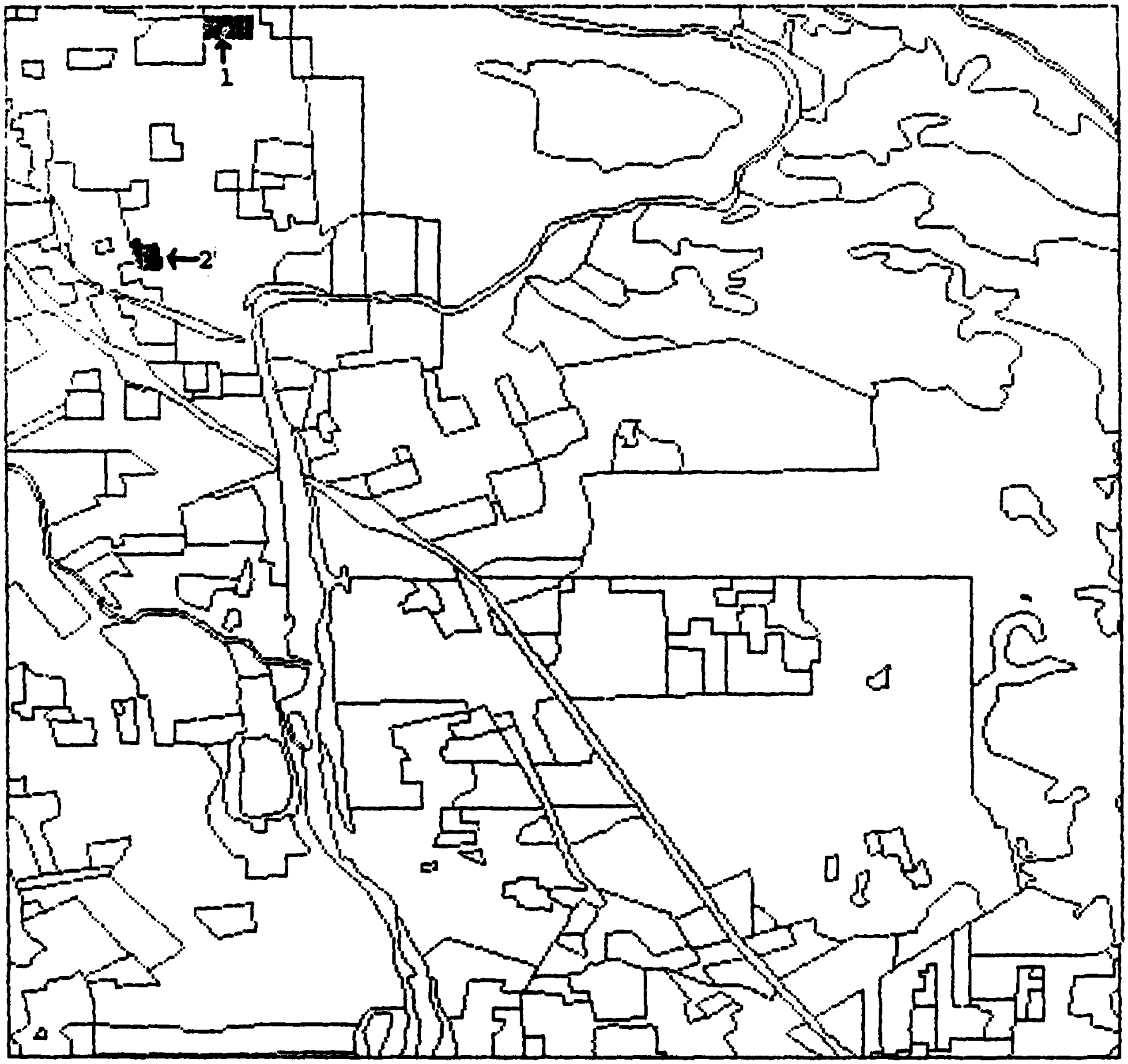


Figure 3.17. The 2 components of the land-use class UCB.

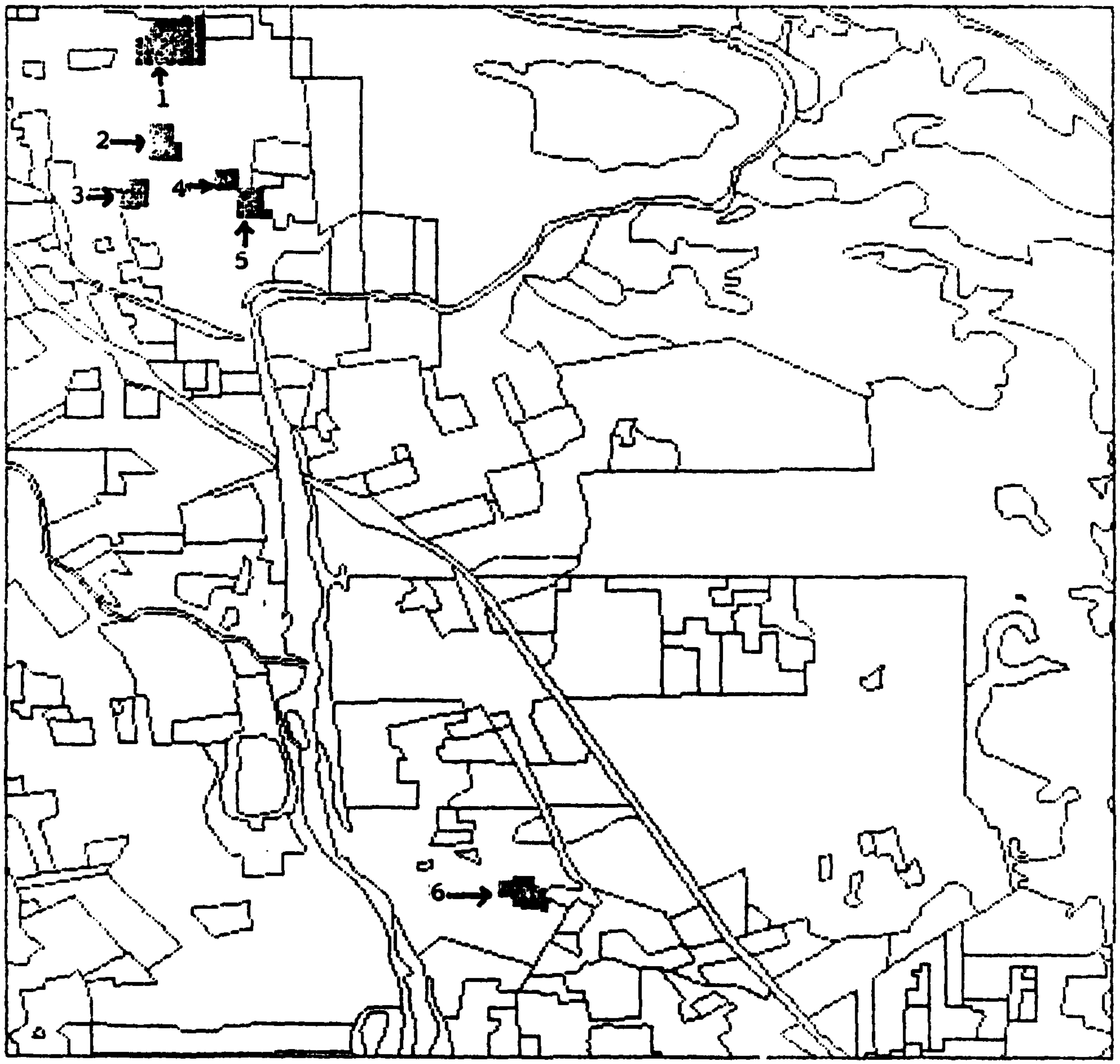


Figure 3.16. The 6 components of the land-use class UCC.

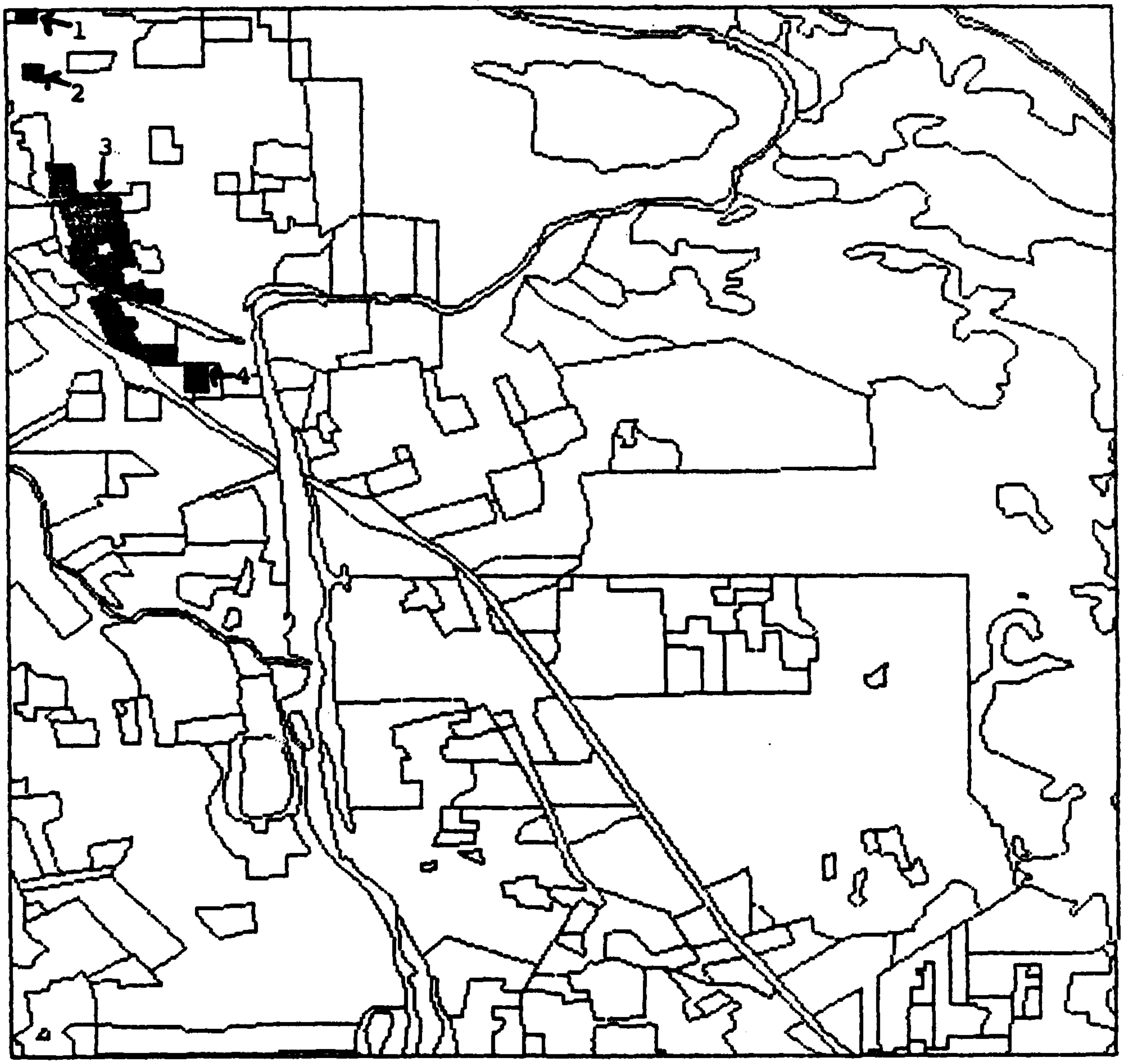


Figure 3.19. The 4 components of the land-use class UCR.

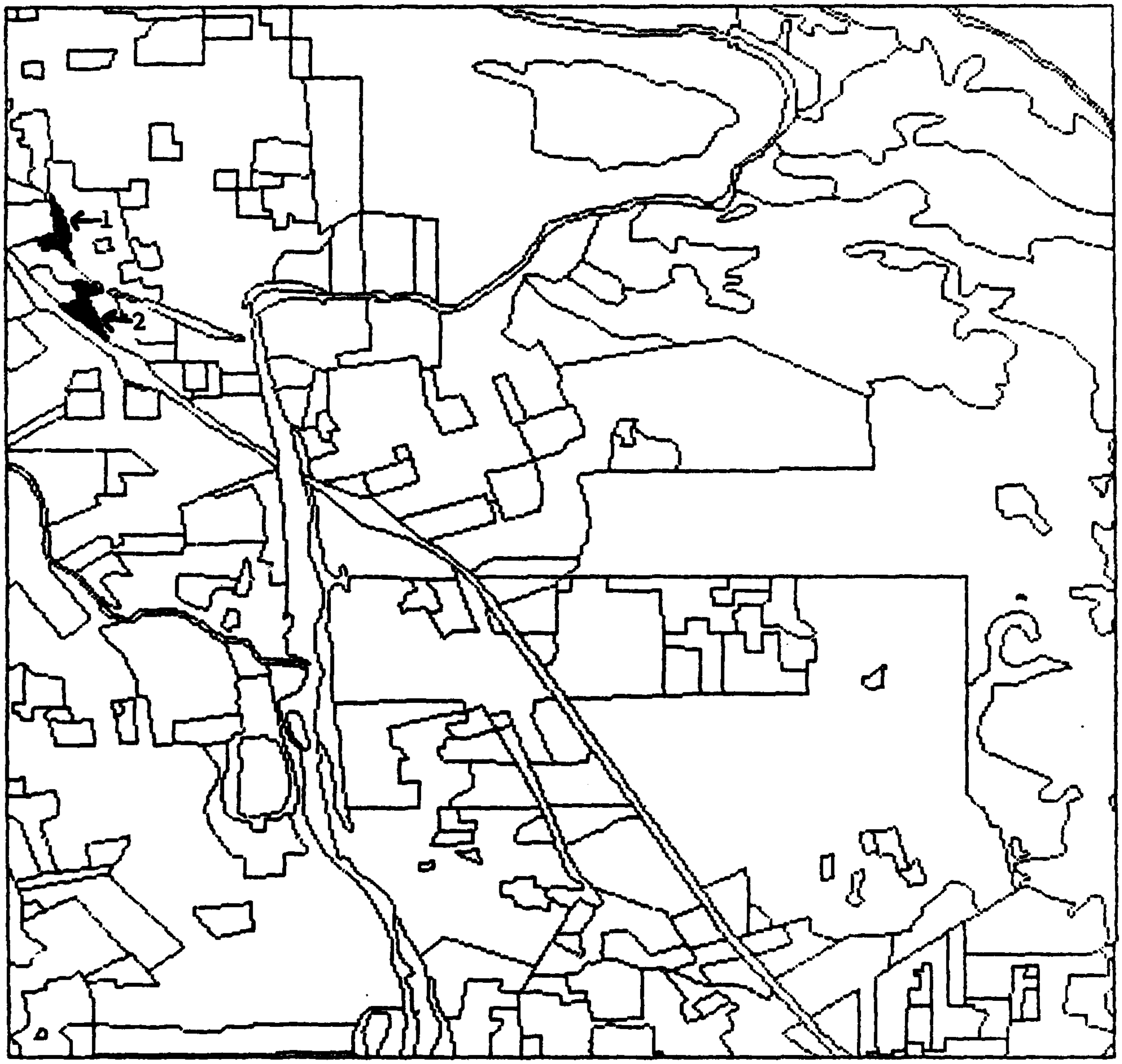


Figure 3.20. The 2 components of the land-use class UCW.

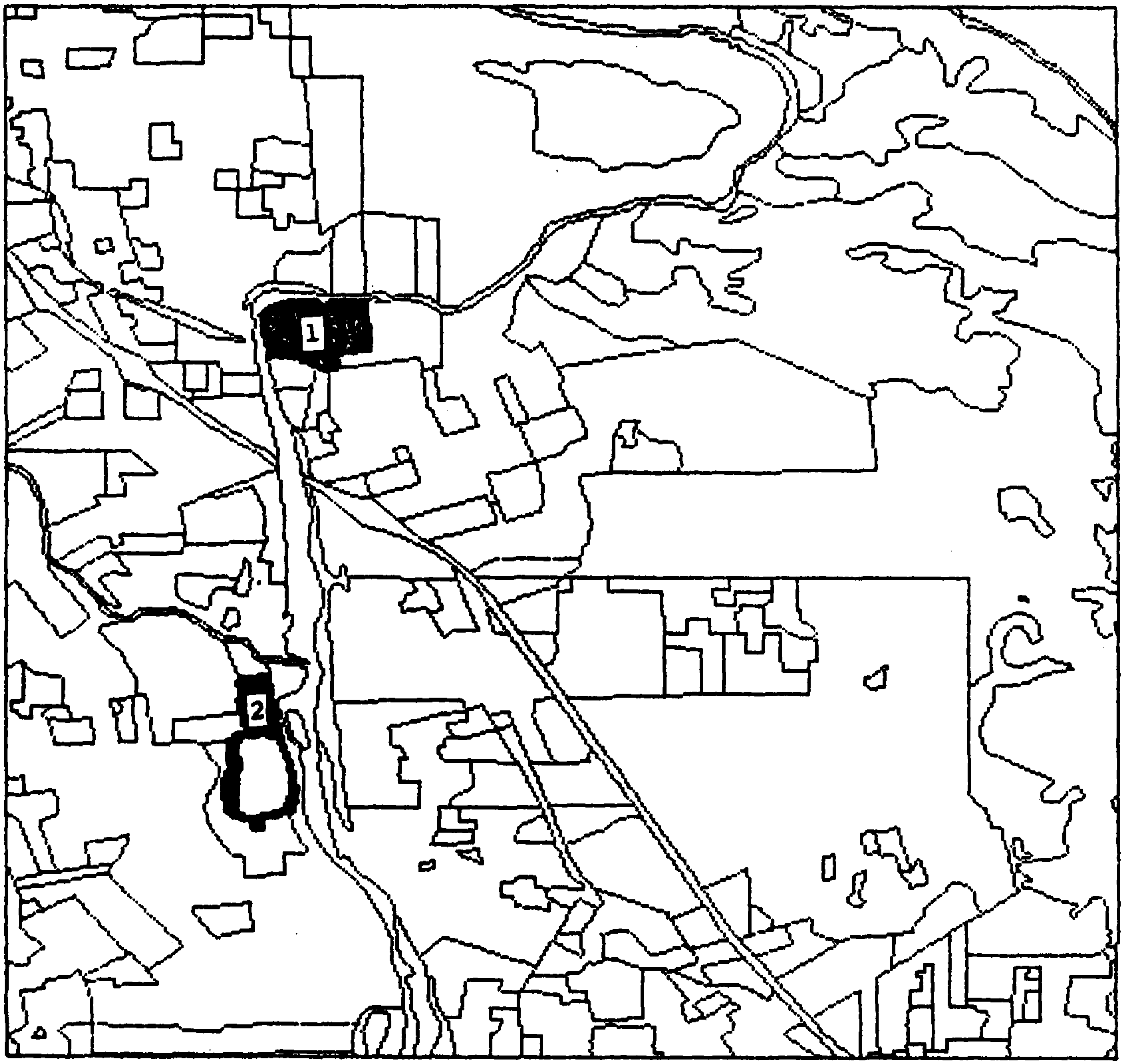


Figure 3.21 The 2 components of the land-use class ULS.

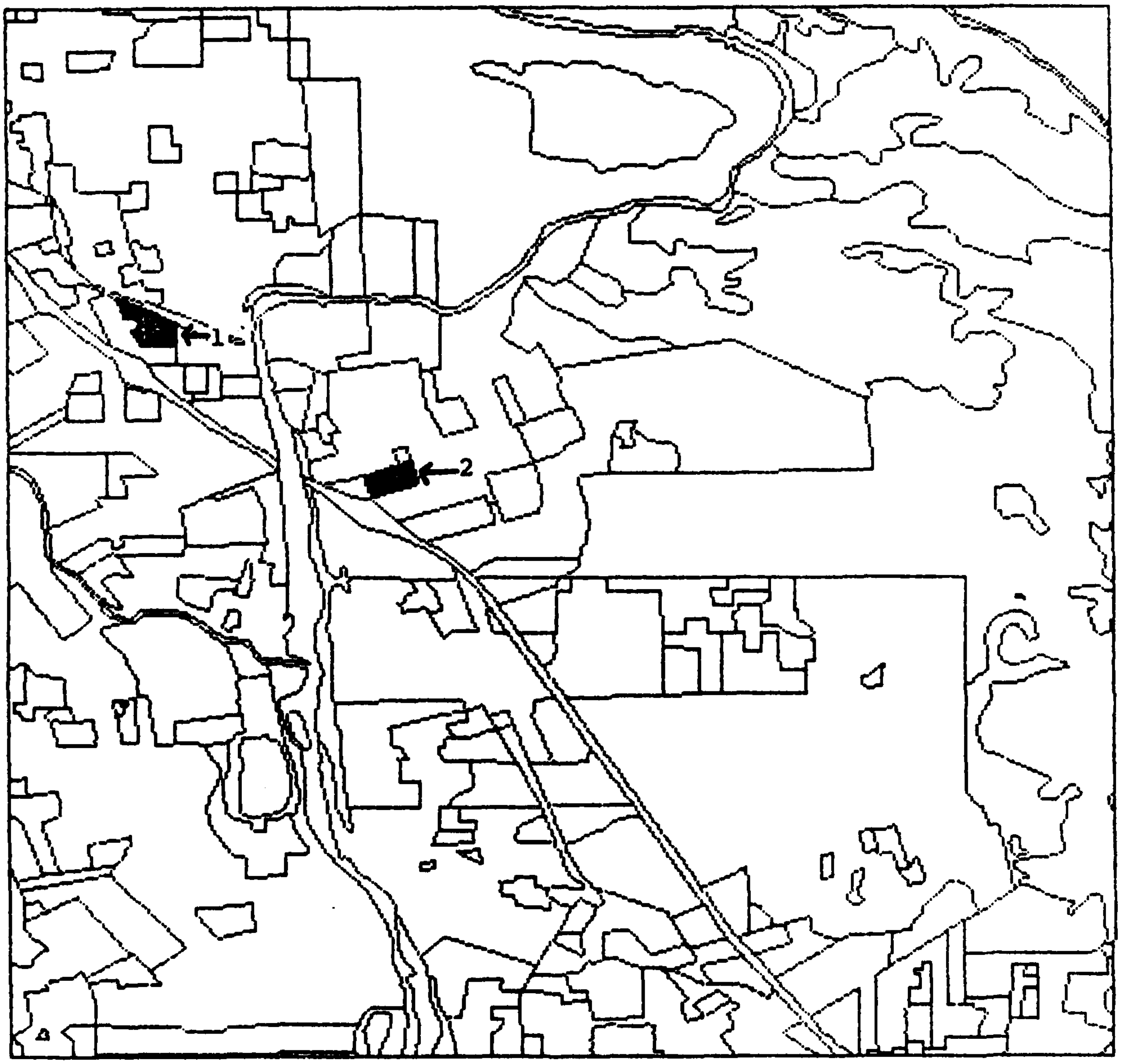


Figure 3.22. The 2 components of the land-use class UIL.

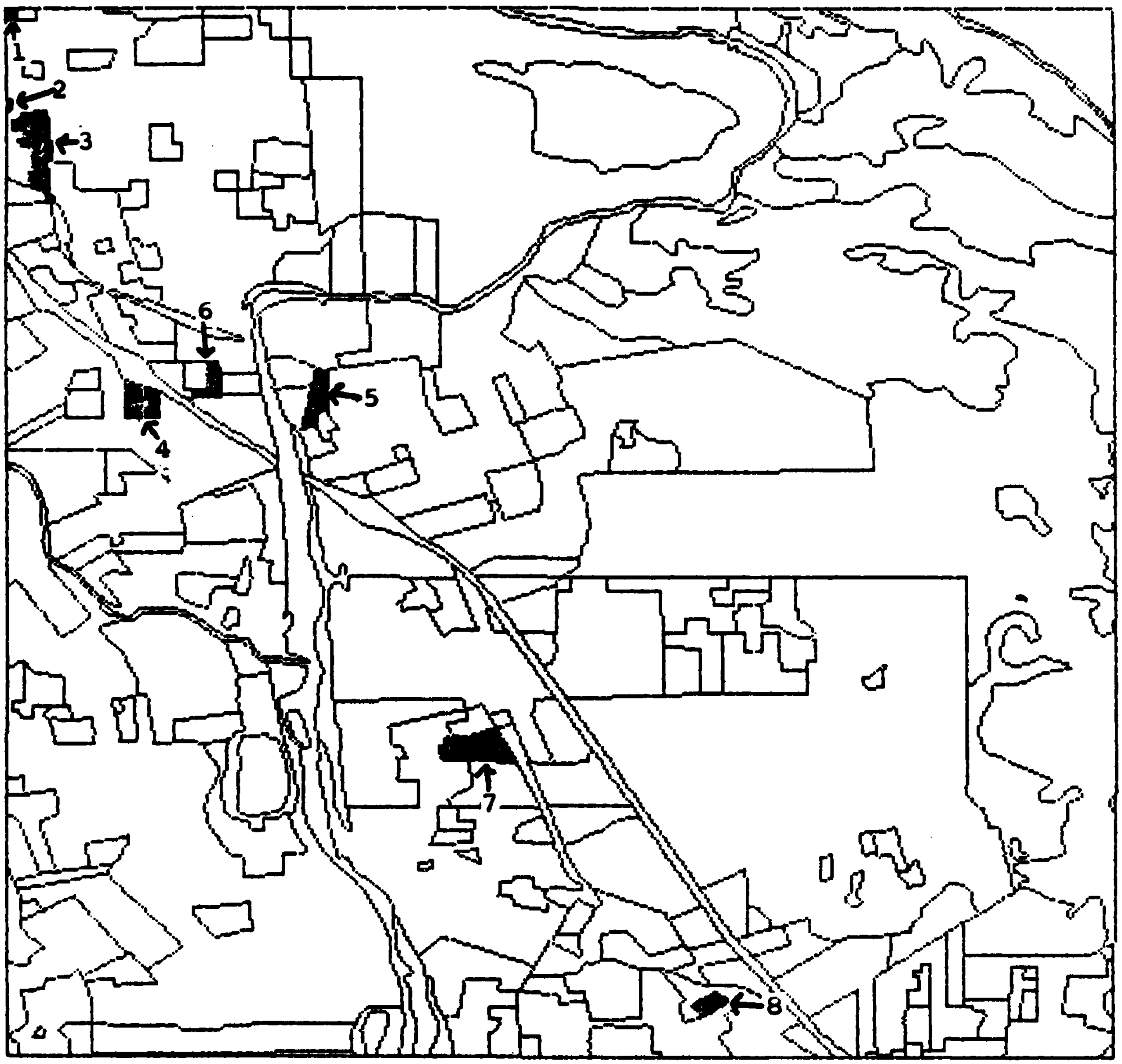


Figure 3.23. The 8 components of the land-use class UIS.

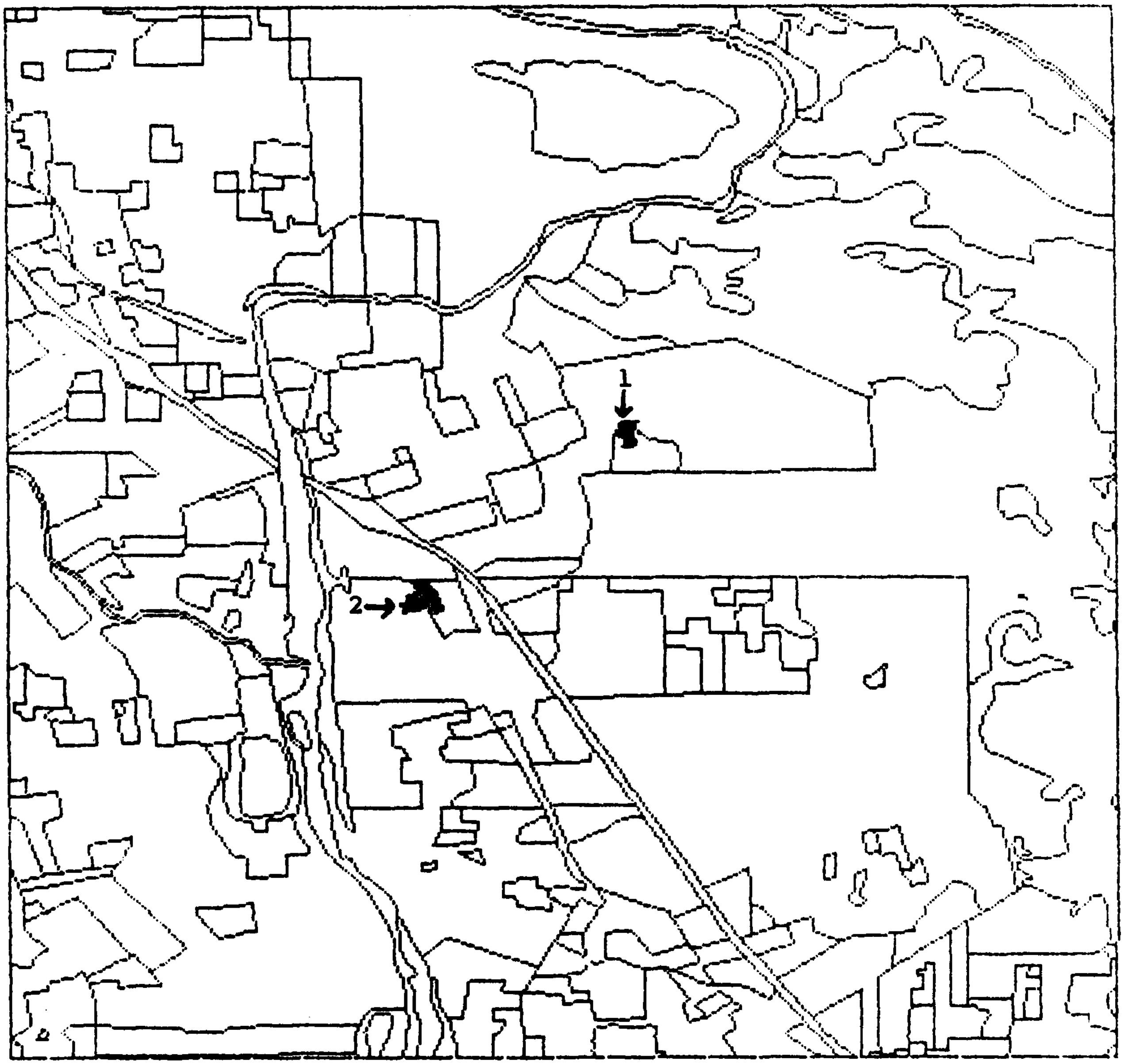


Figure 3.24. The 2 components of the land-use class U1..

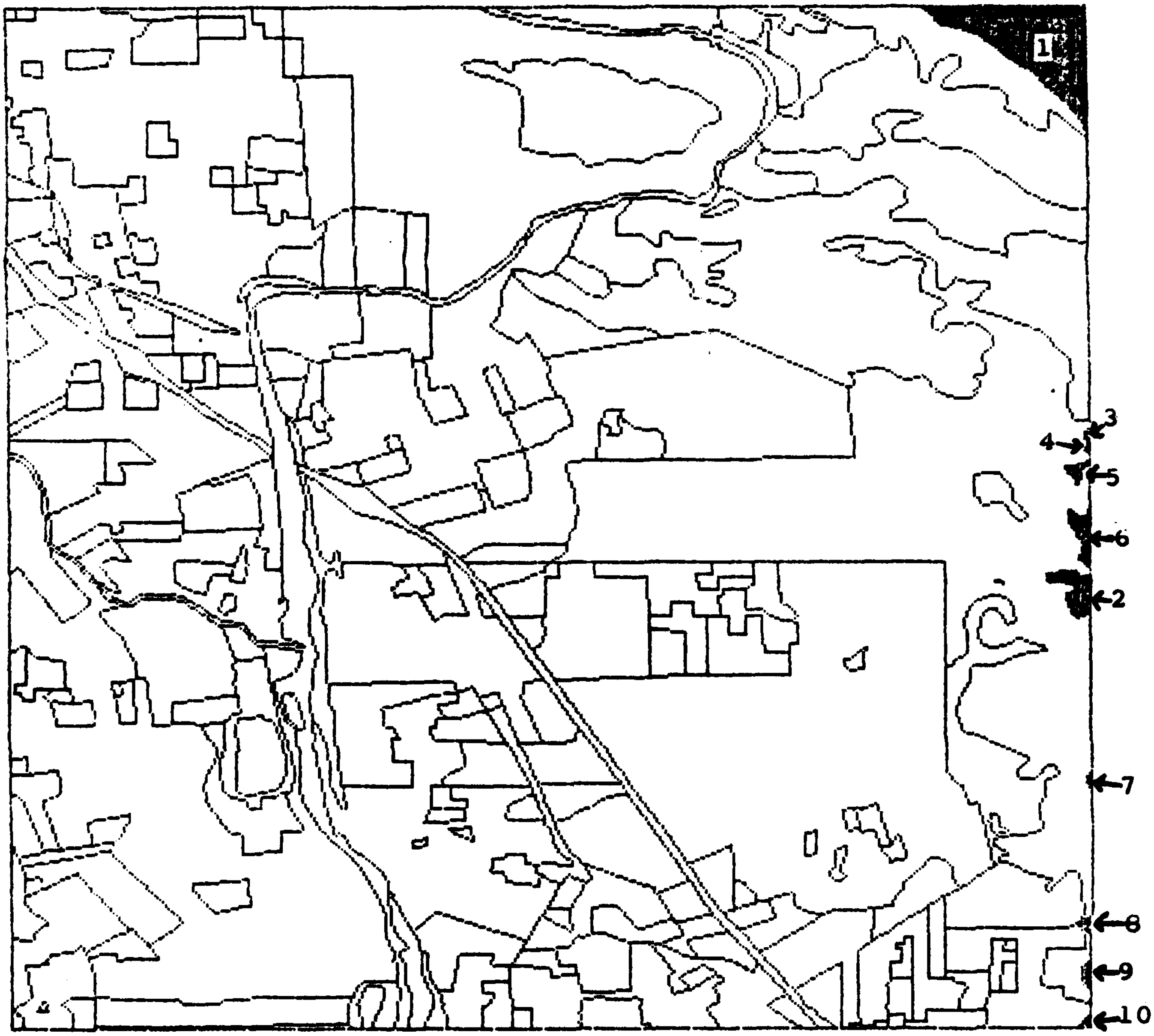


Figure 3.25. The 10 components of the land-use class UNK.

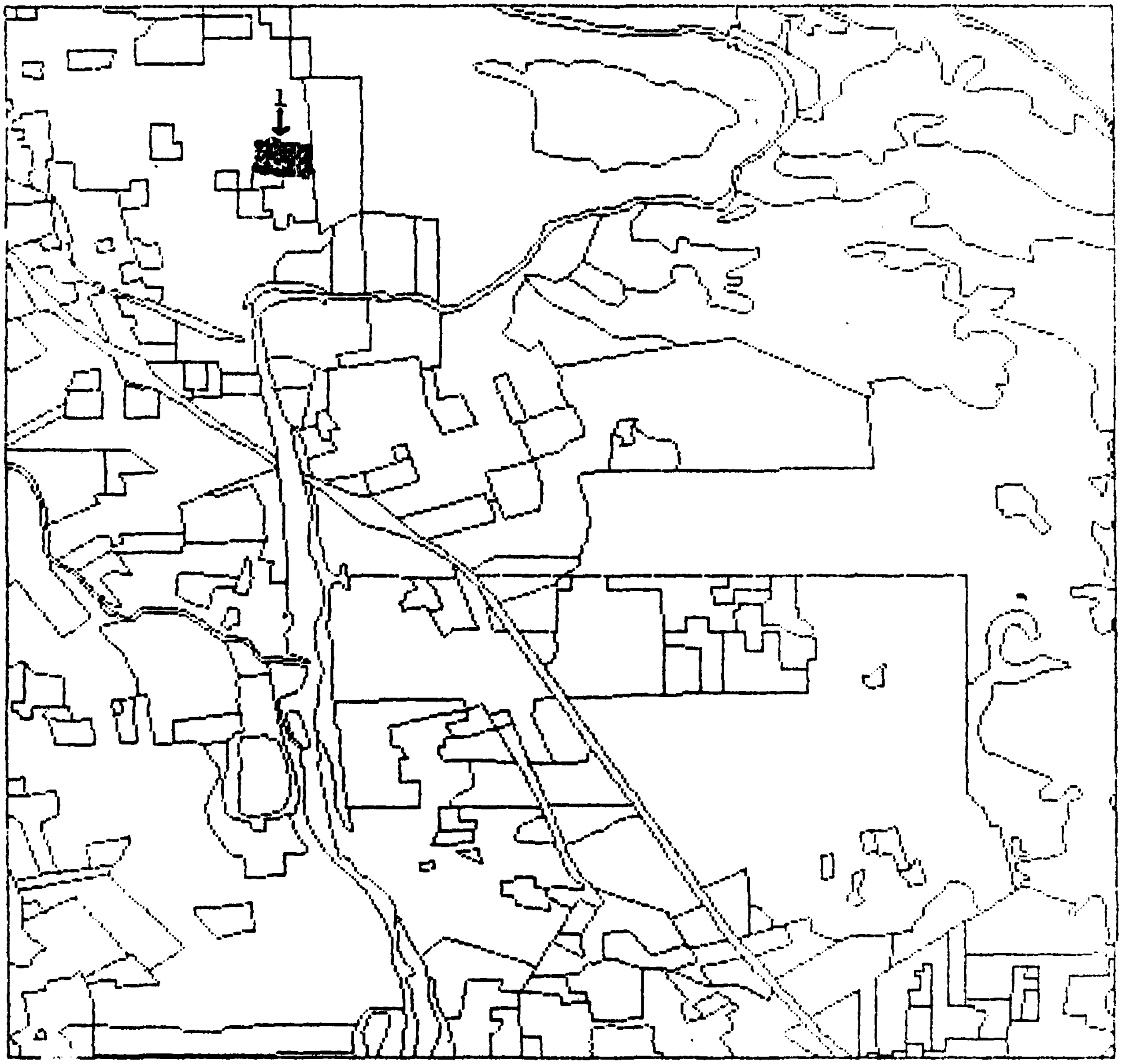


Figure 3.26. The 1 component of the land-use class UUC.

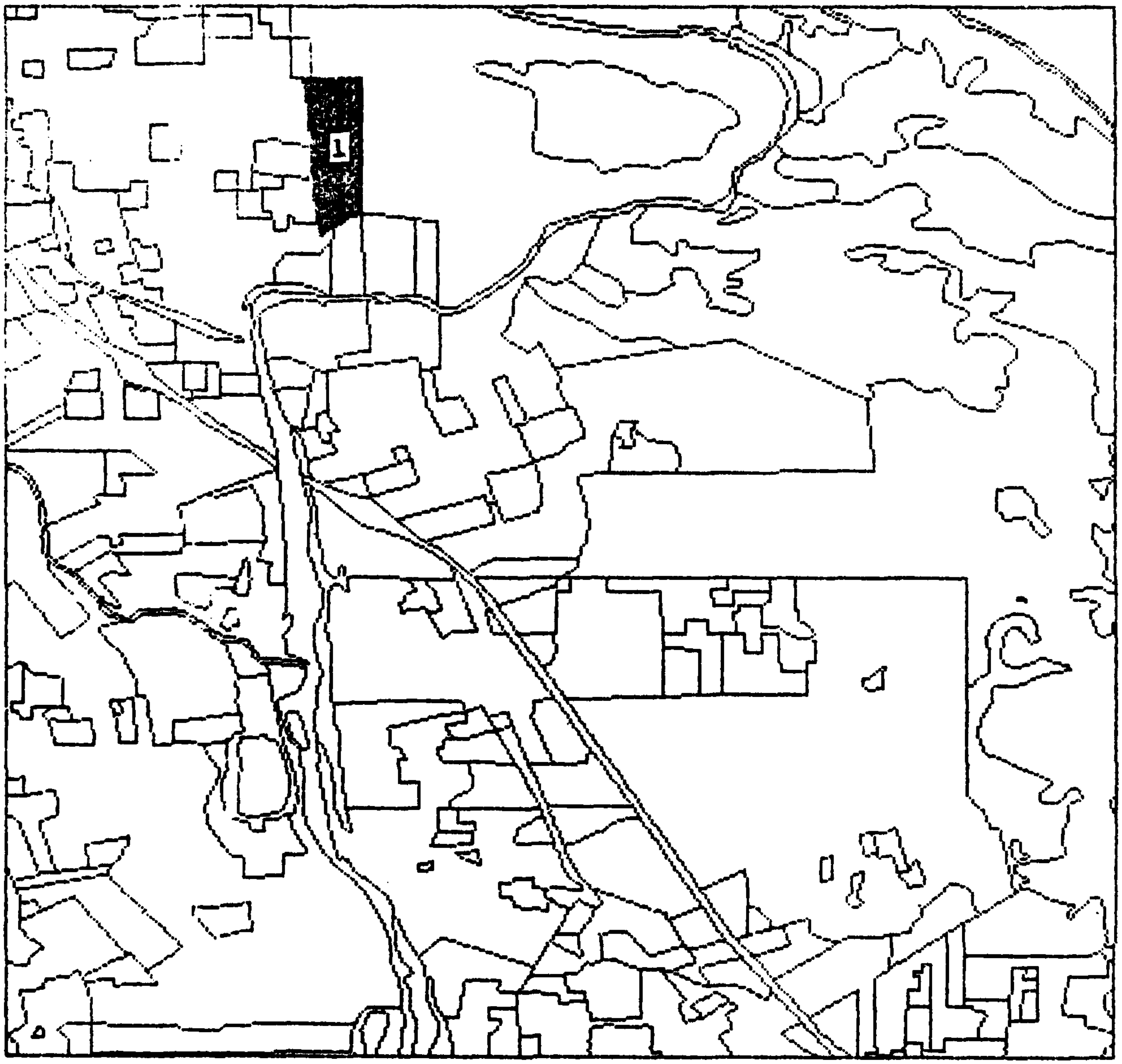


Figure 3.27. The 1 component of the land-use class UUG.

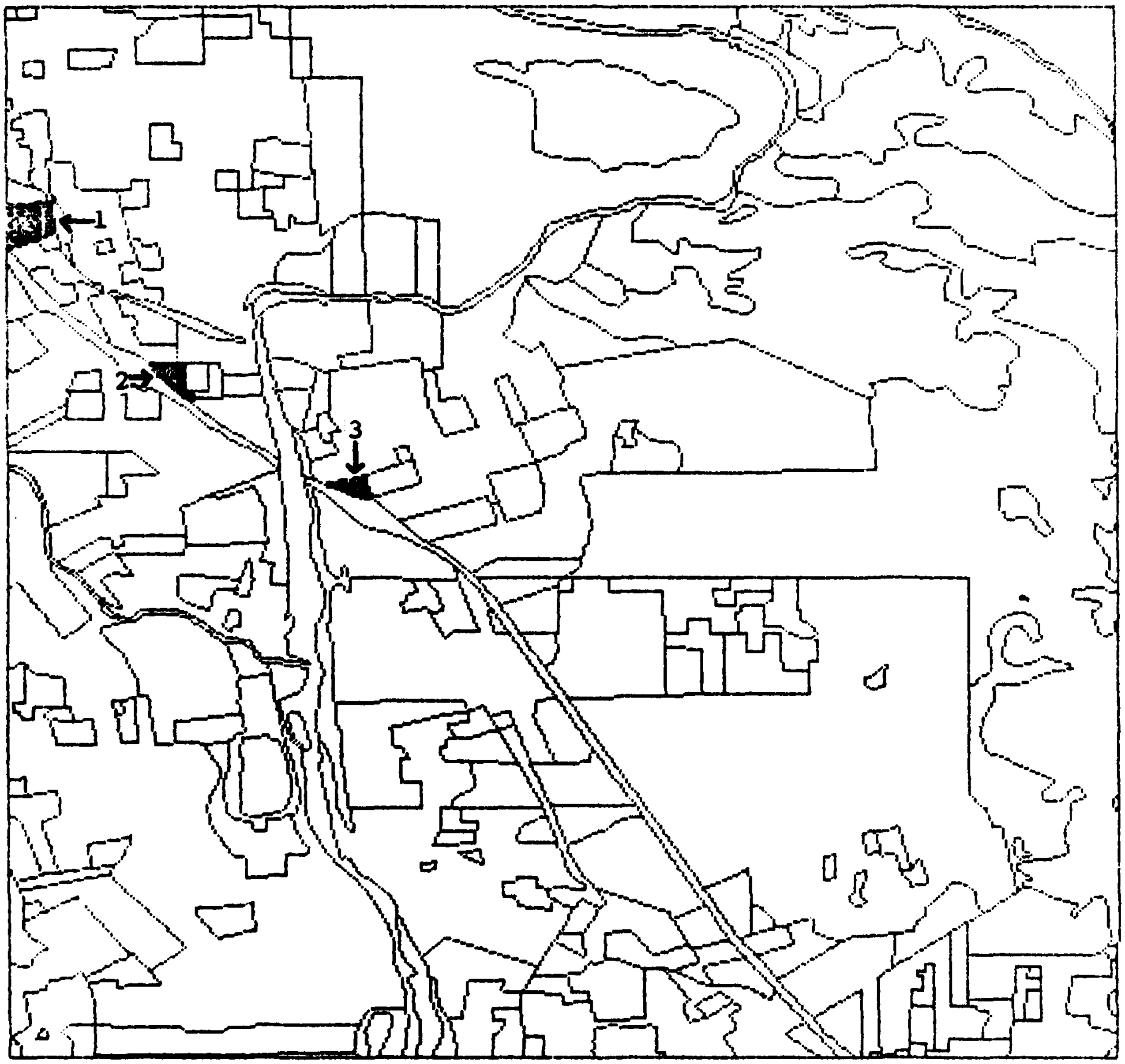


Figure 3.26. The 3 components of the land-use class '000.

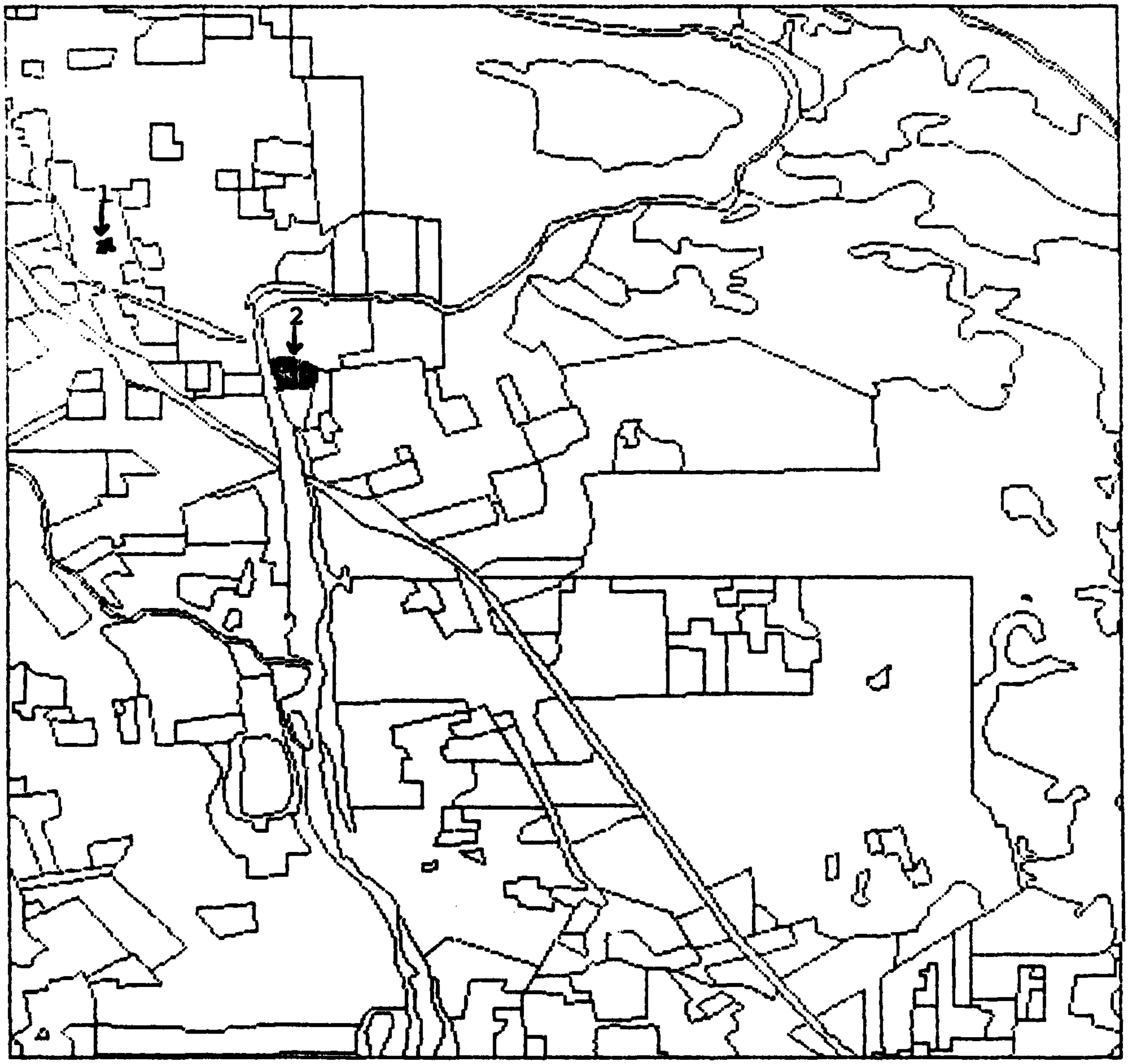


Figure 3.29. The 2 components of the land-use class UOP.

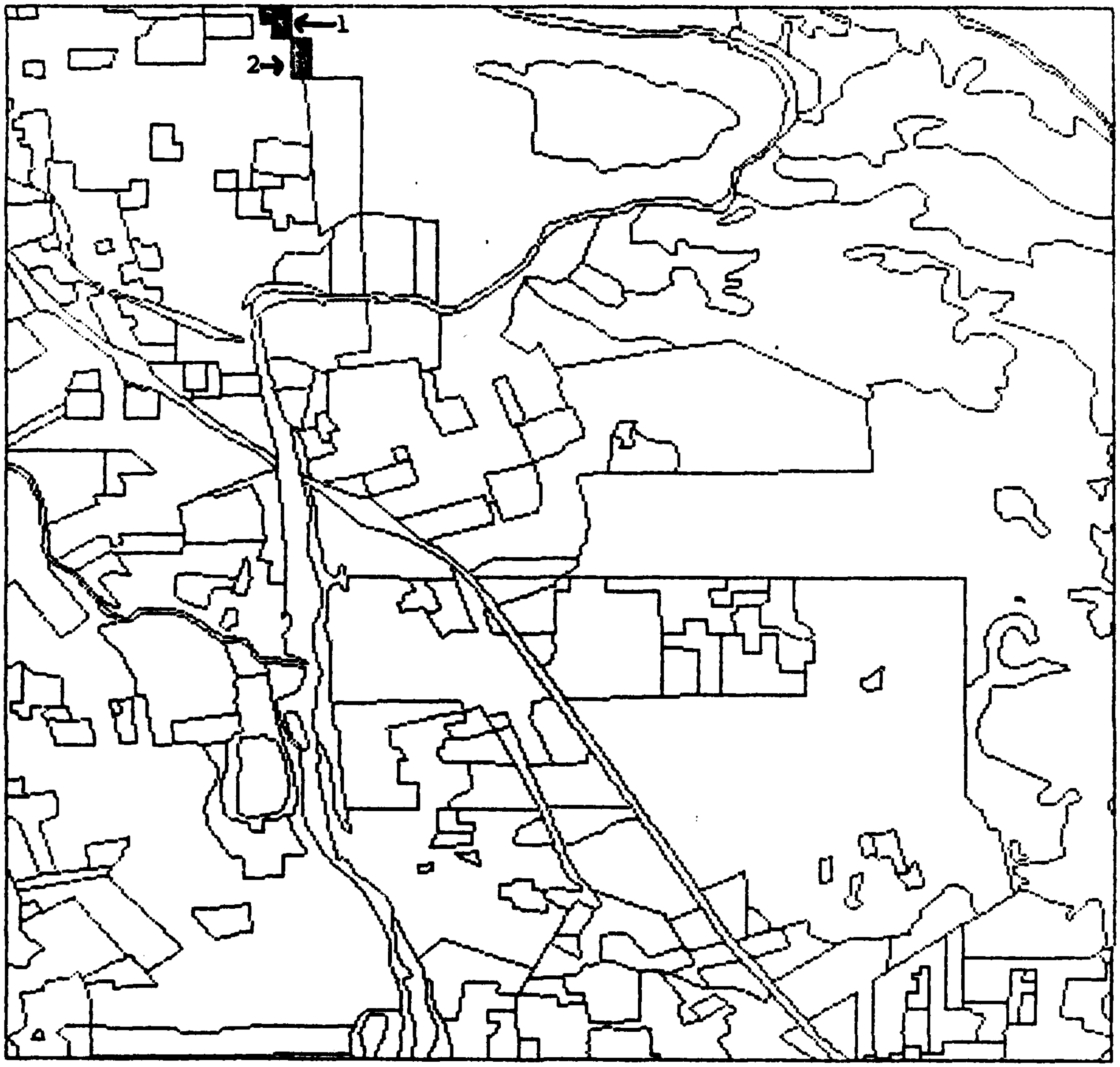


Figure 3.30. The 2 components of the land-use class UUV.

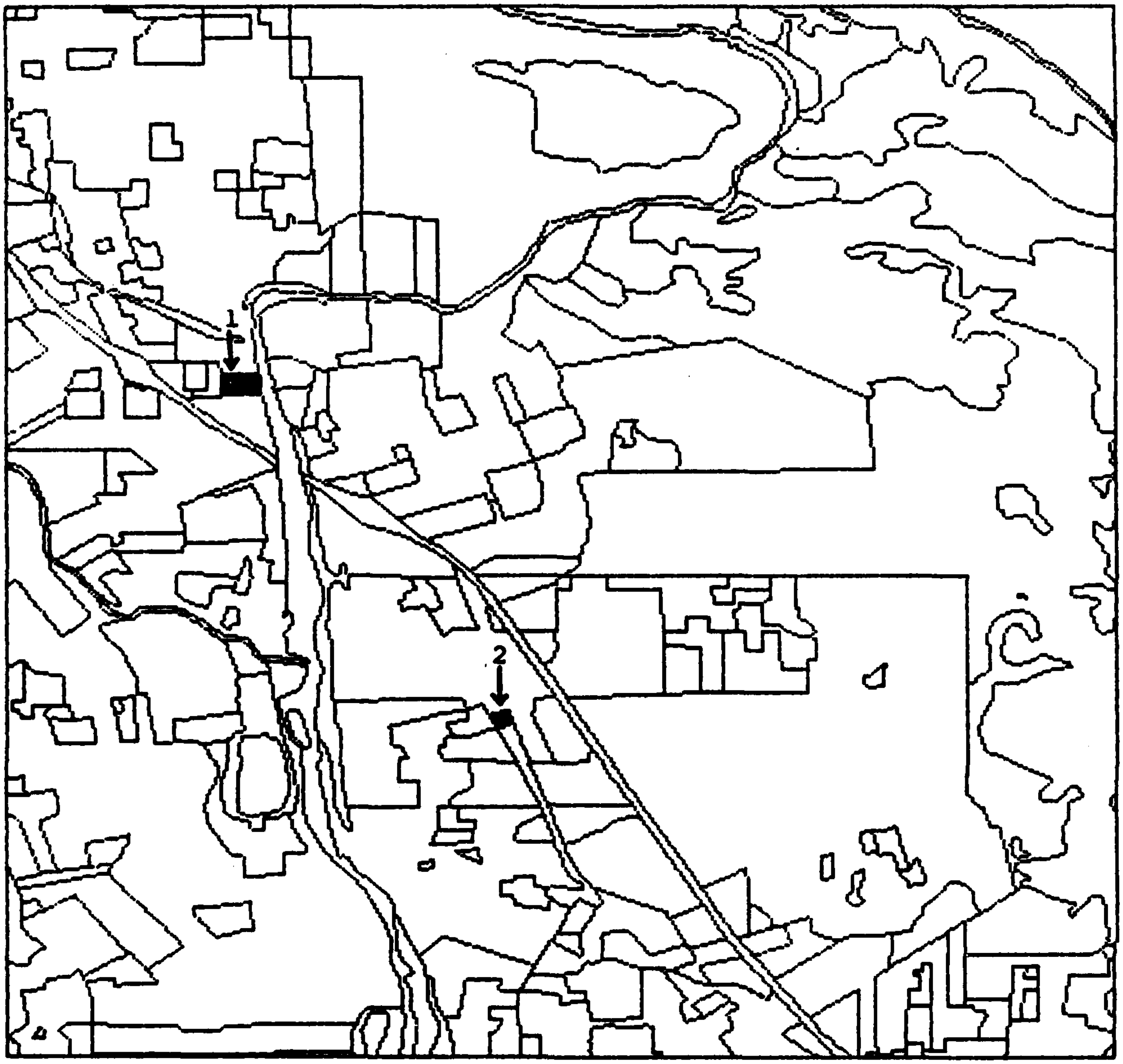


Figure 3.31. The 2 components of the land-use class URH.

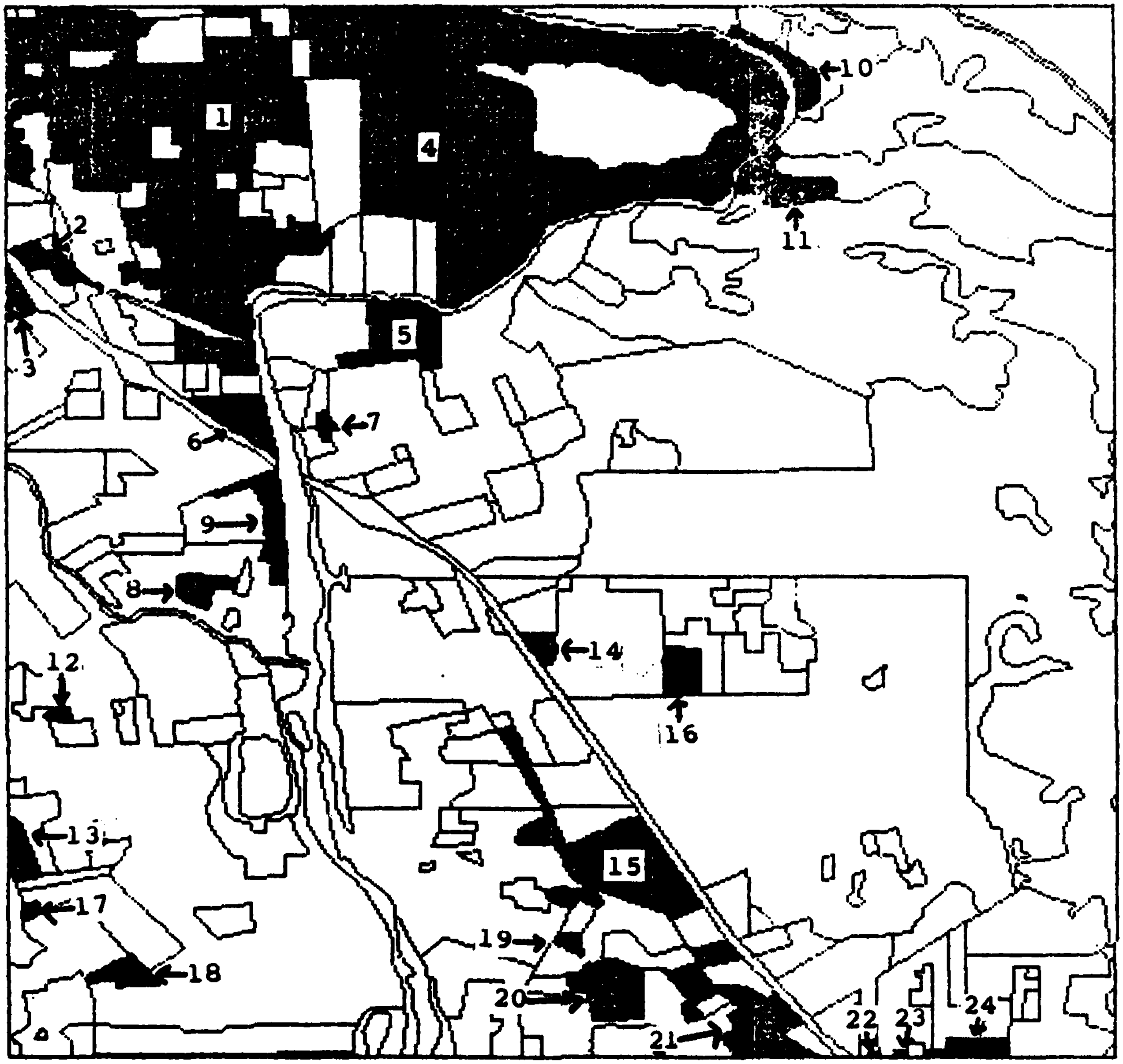


Figure 3.32. The 24 components of the land-use class URS.

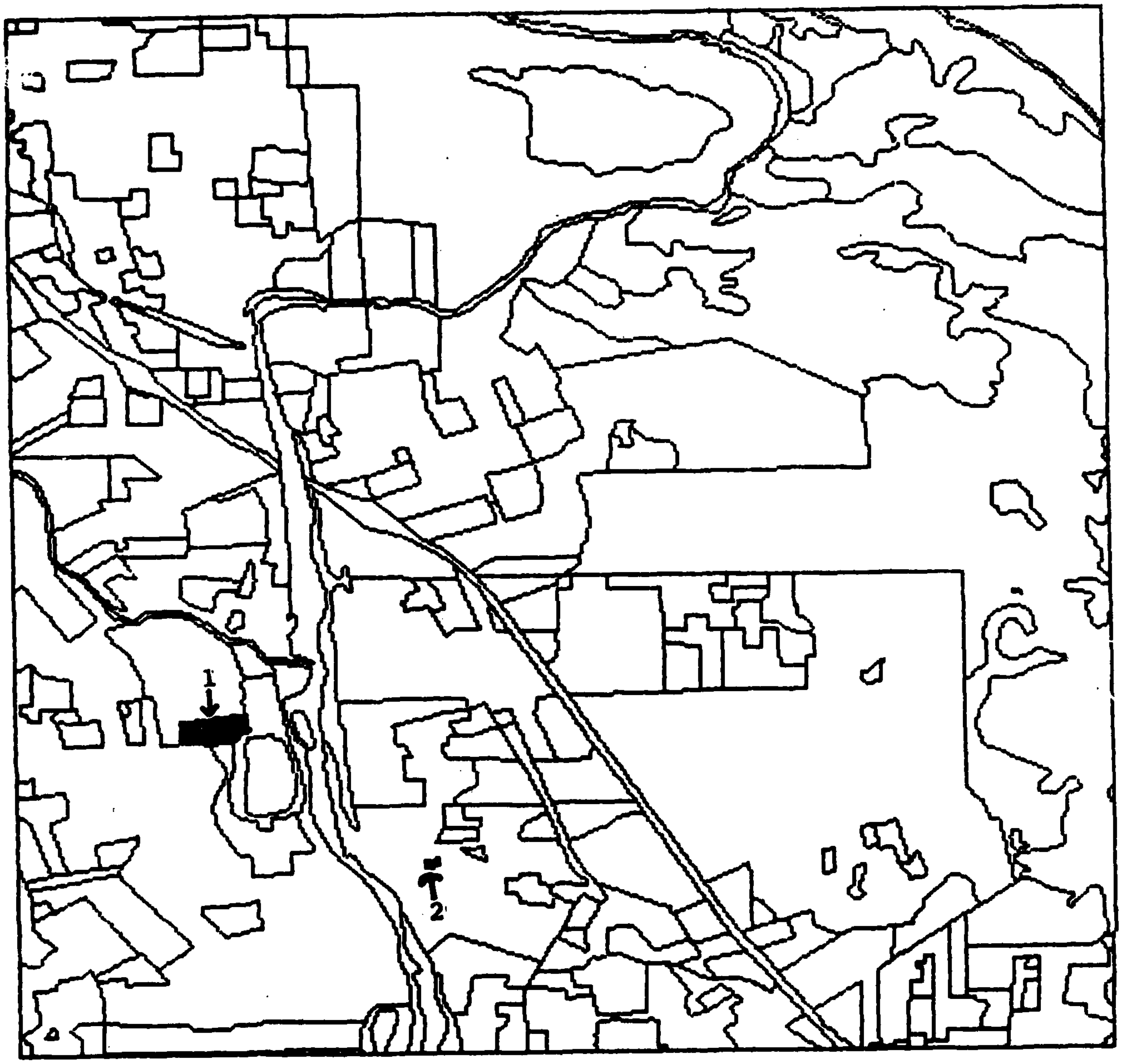


Figure 3.33. The 2 components of the land-use class UUS.

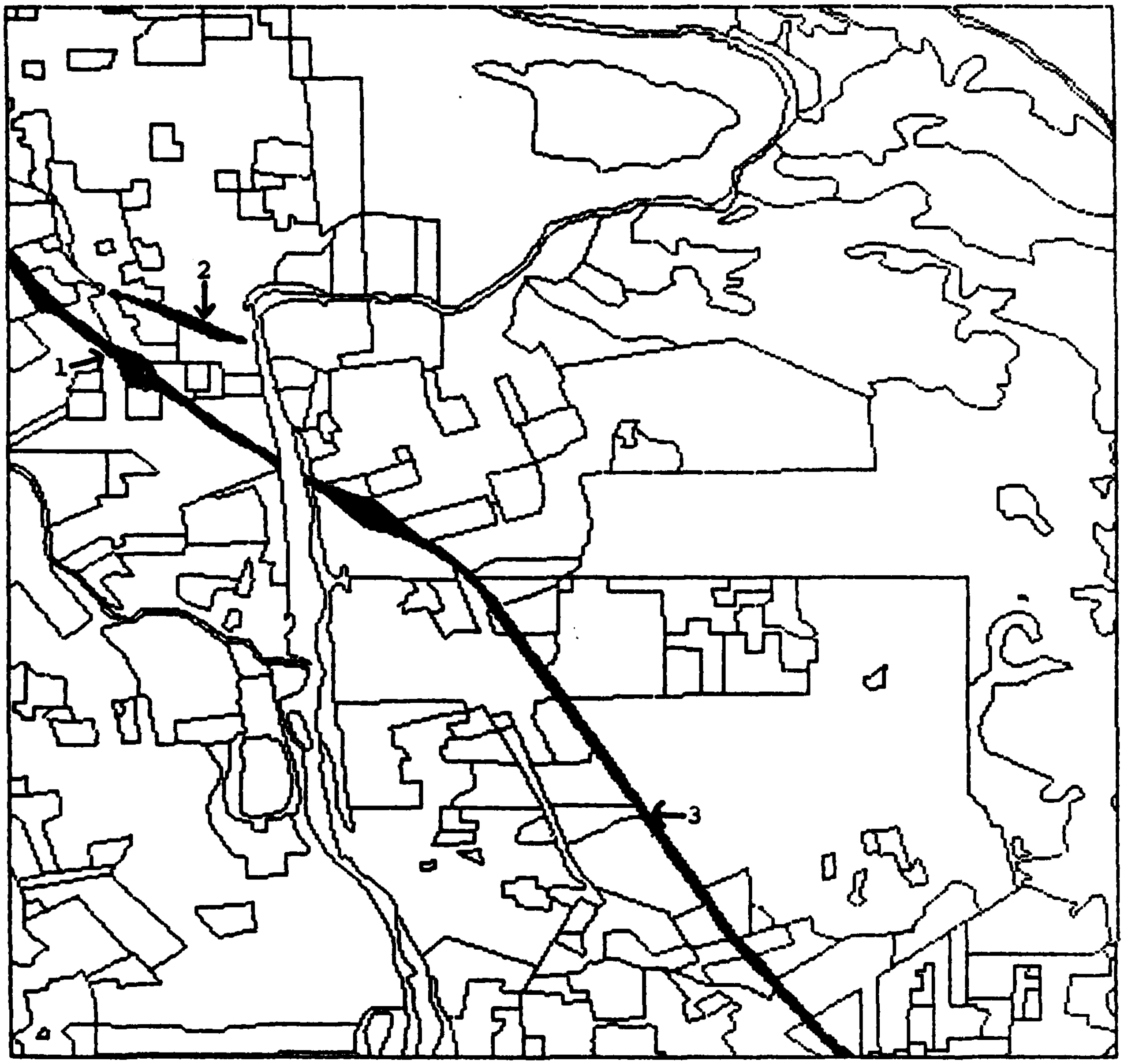


Figure 3.34. The 3 components of the land-use class UUT.

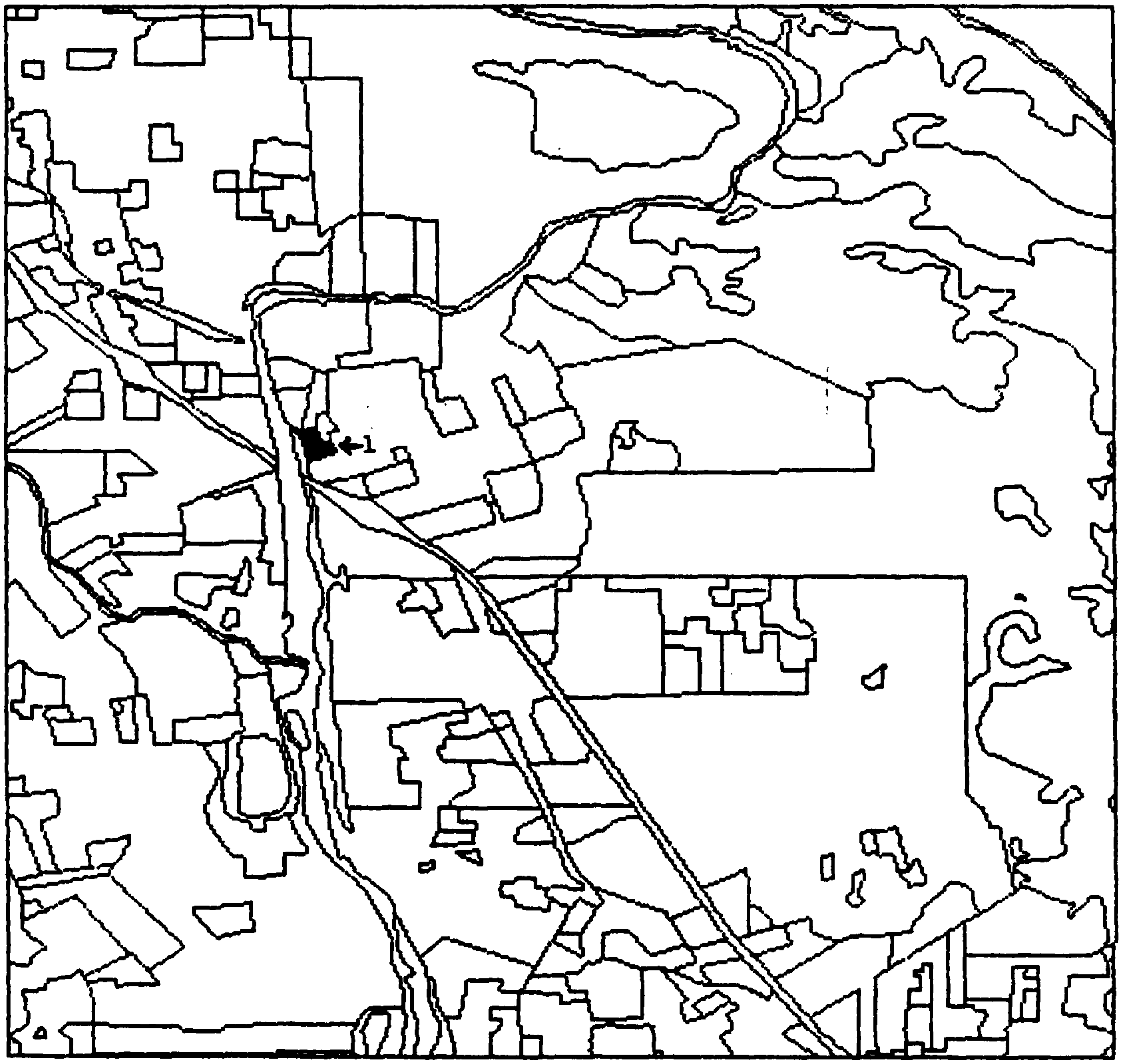


Figure 3.35. The 1 component of the land-use class W.

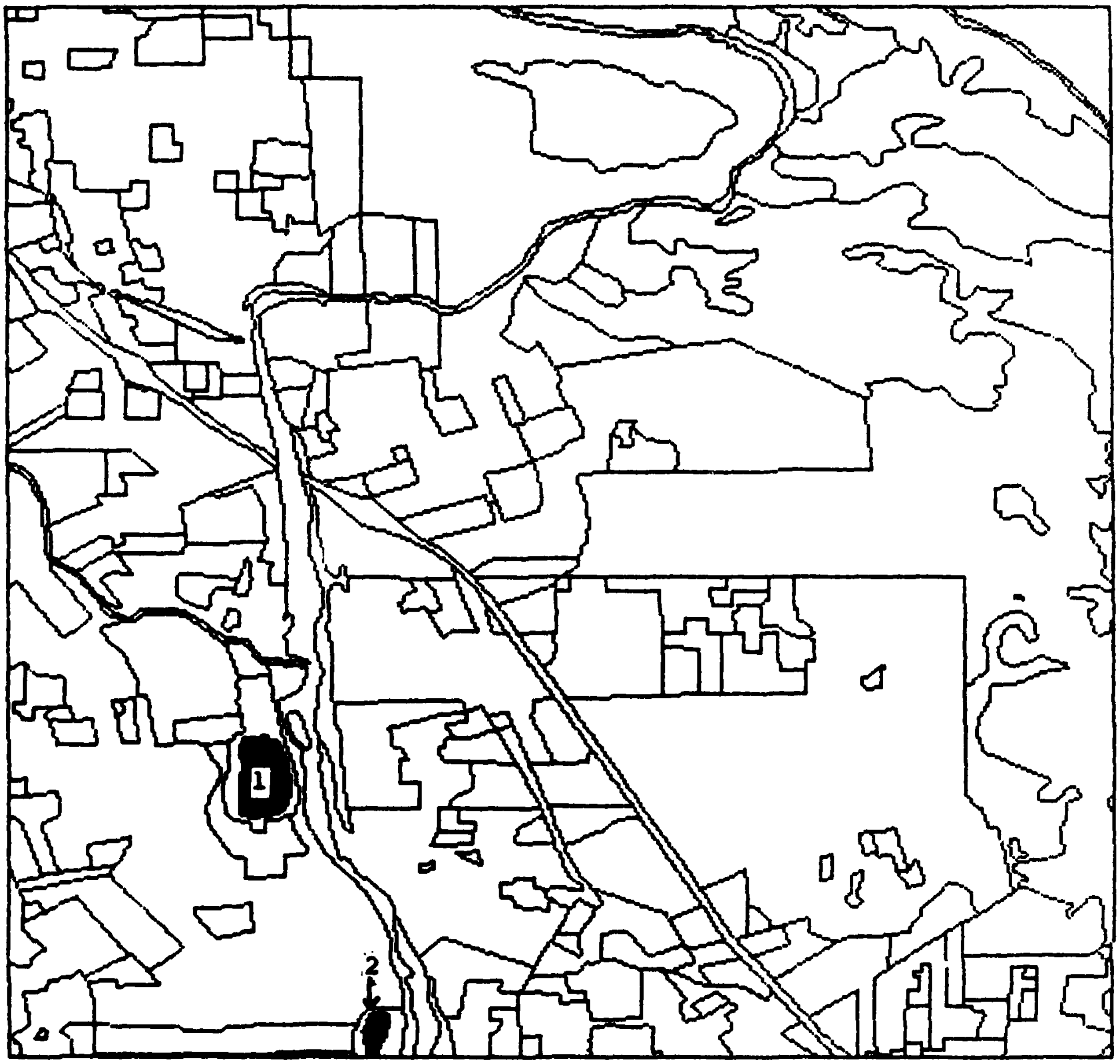


Figure 3.36. The 2 components of the land-use class W0.

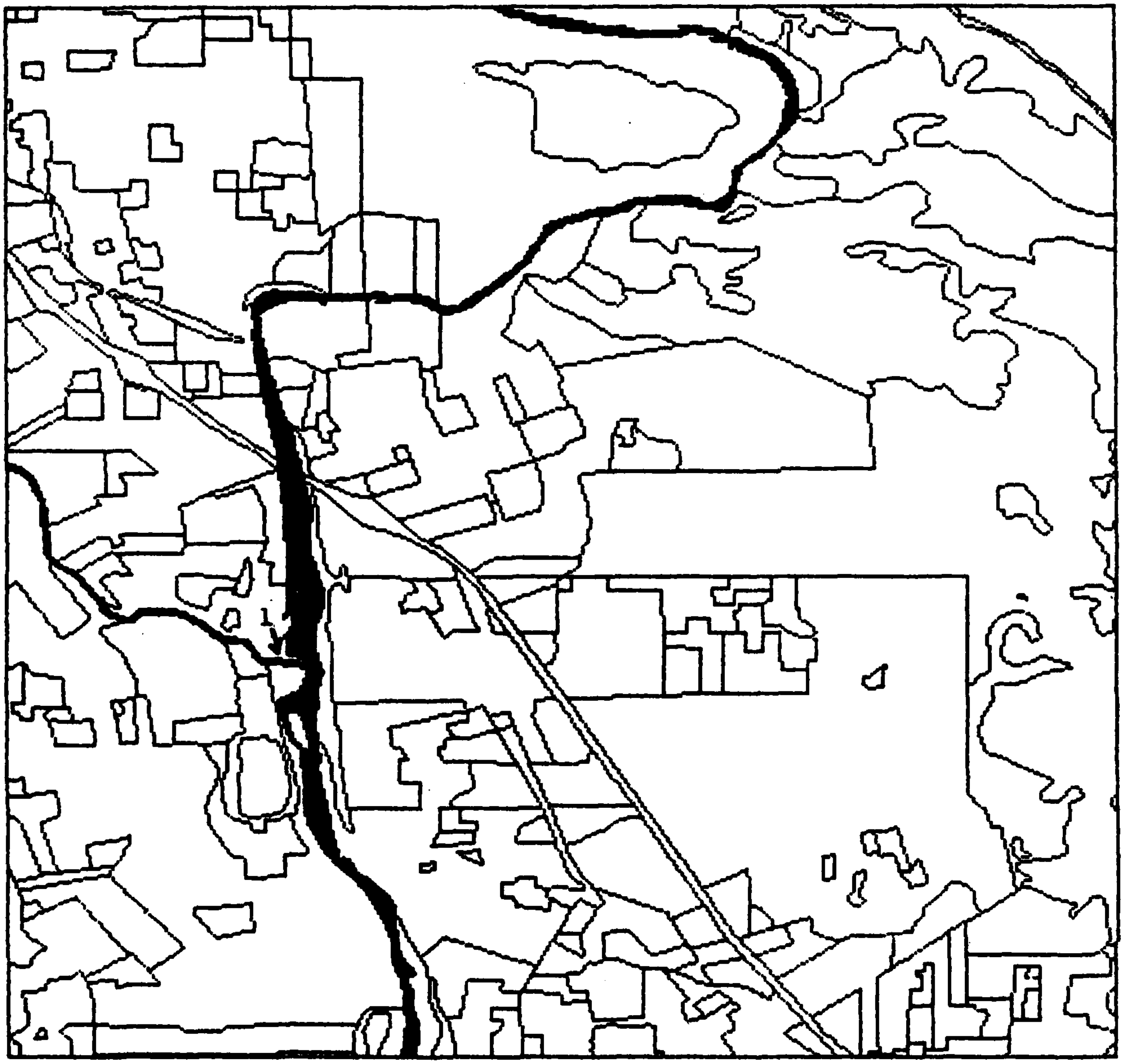


Figure 3.37. The 1 component of the land-use class WS.

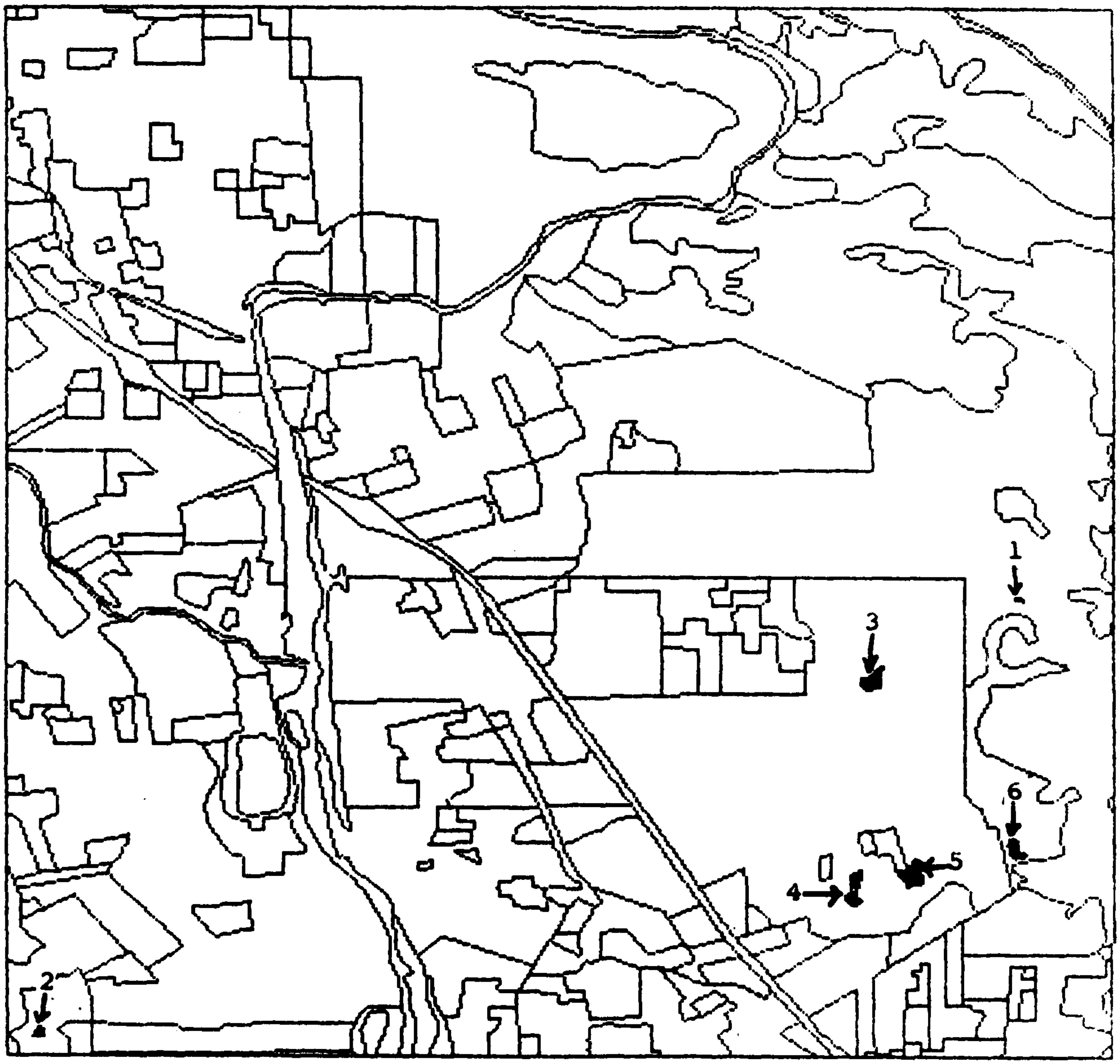


Figure 3.36. The 6 components of the land-use class WWP.

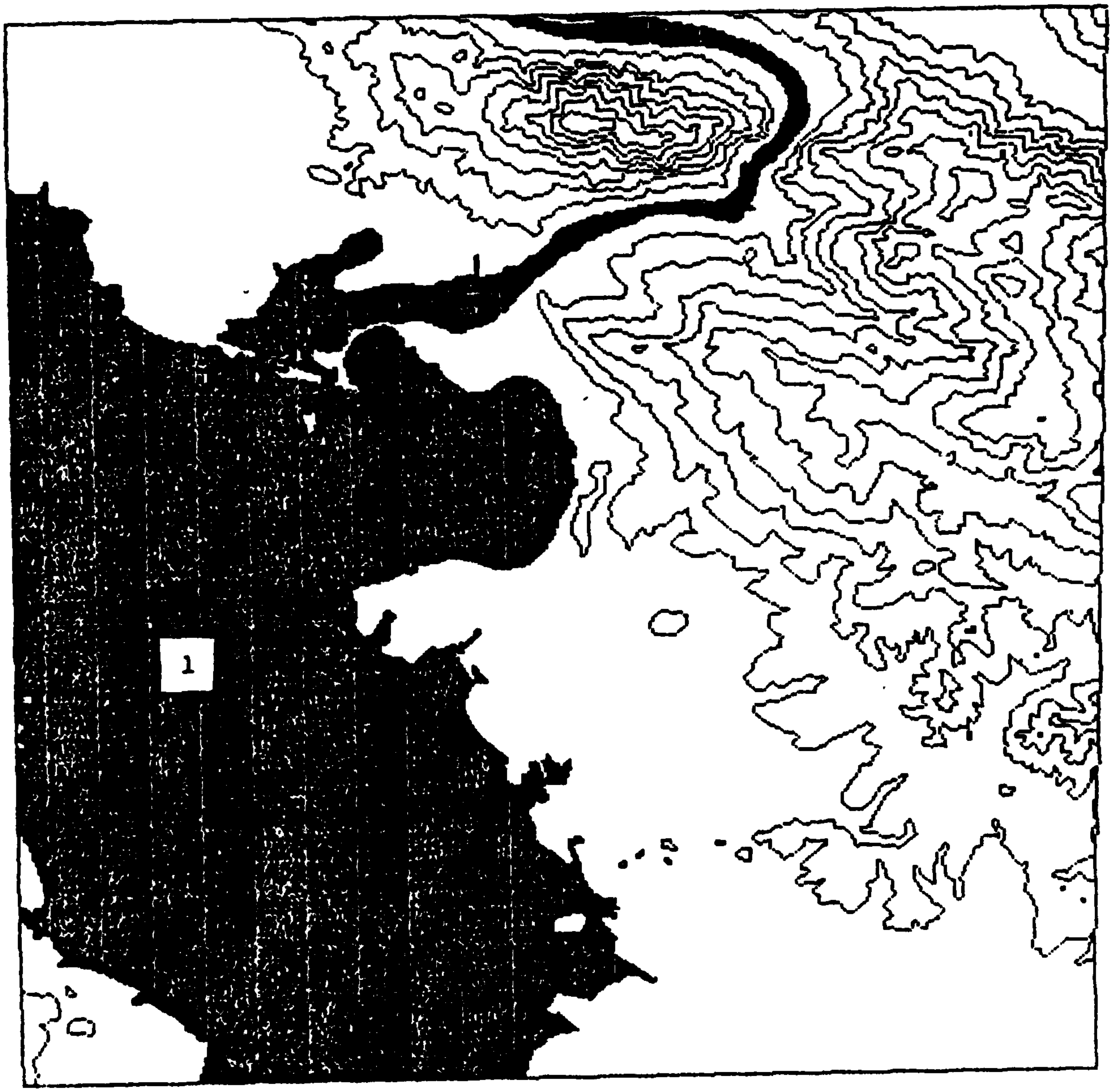


Figure 3.39. The 1 component of the 1st elevation level.
(0 - 100 ft.)

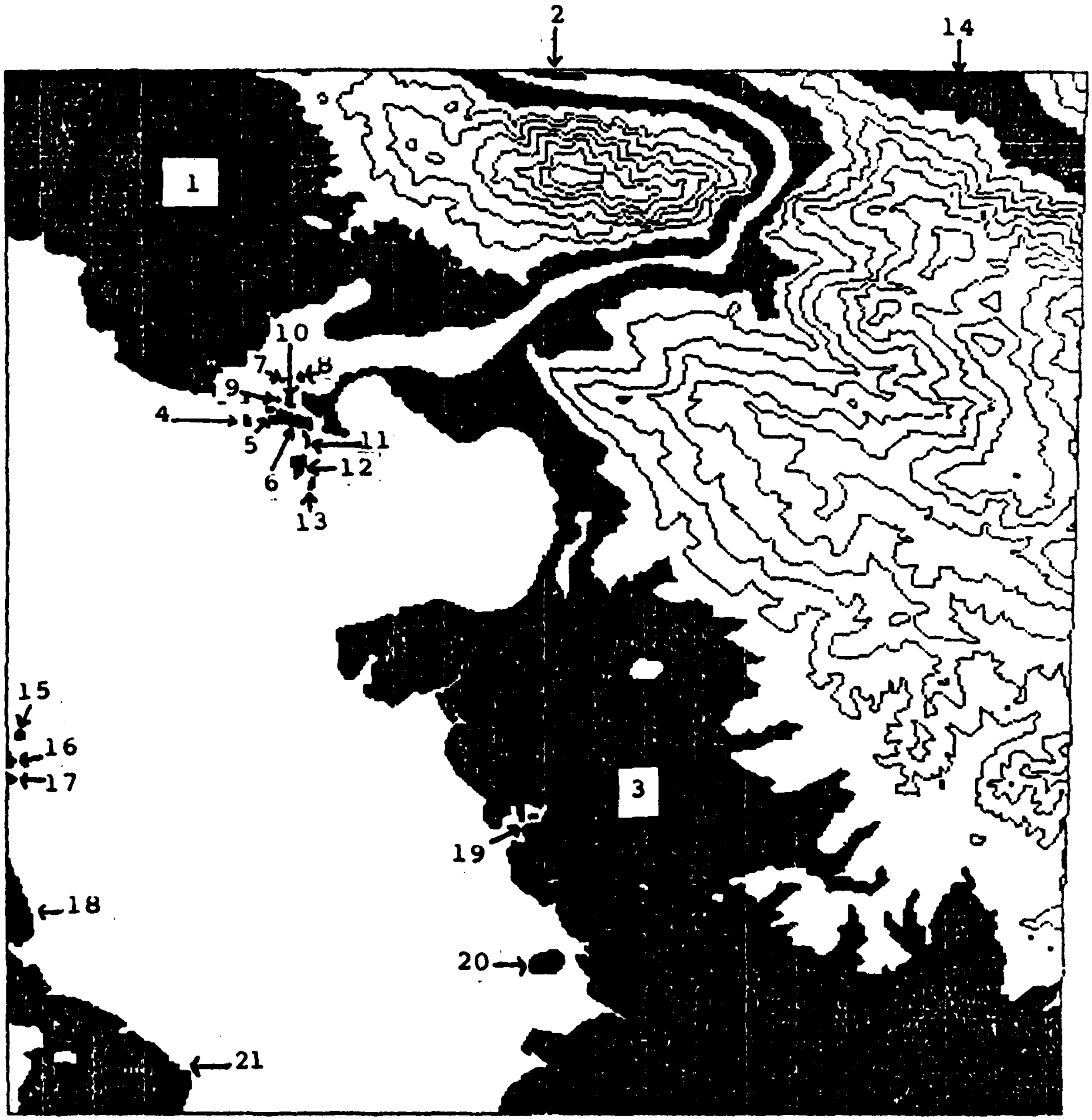


Figure 3.40. The 21 components of the 2nd elevation level.

(100 - 200 ft.)

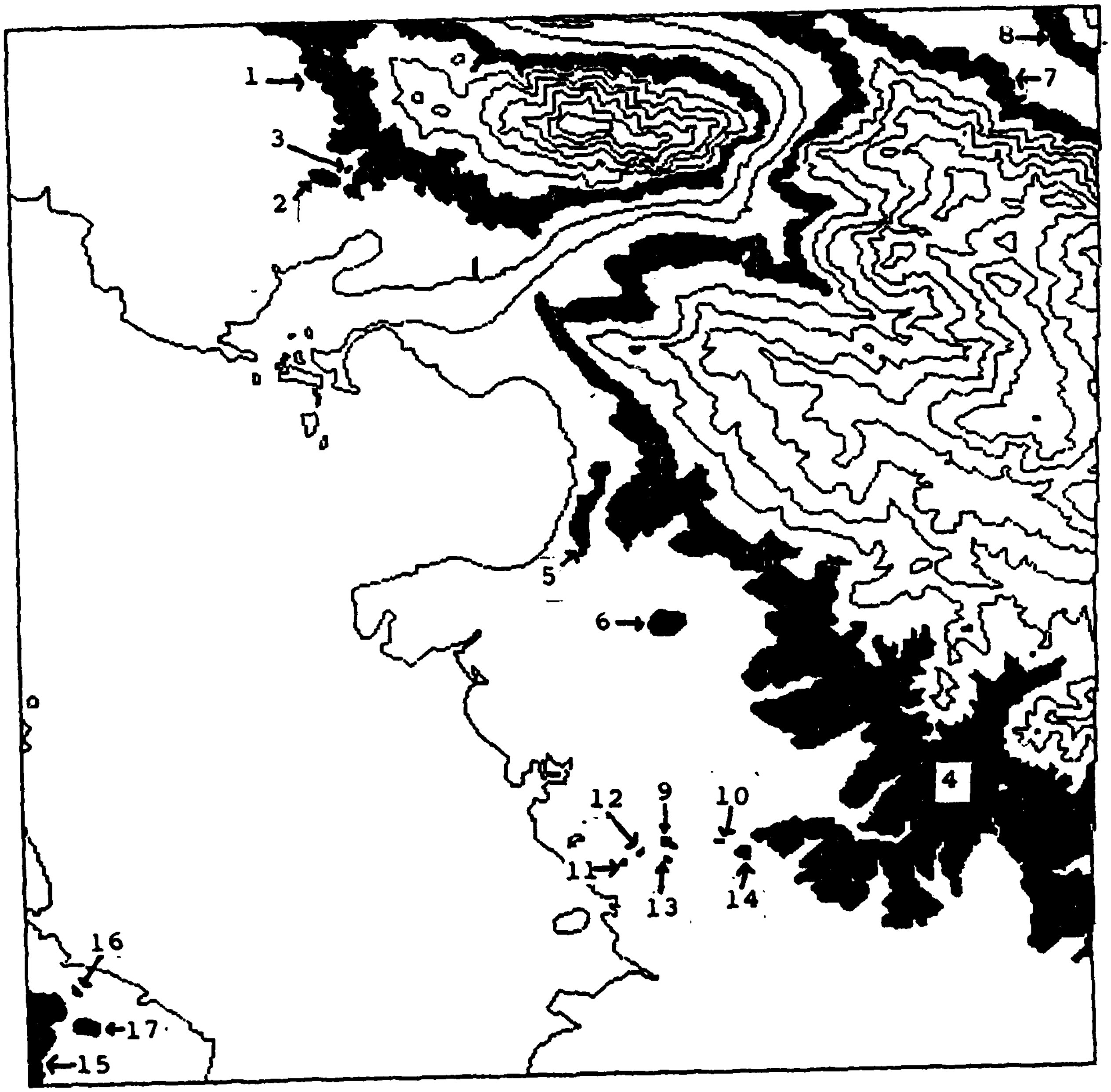


Figure 3.41. The 17 components of the 3rd elevation level.
(200 - 300 ft.)

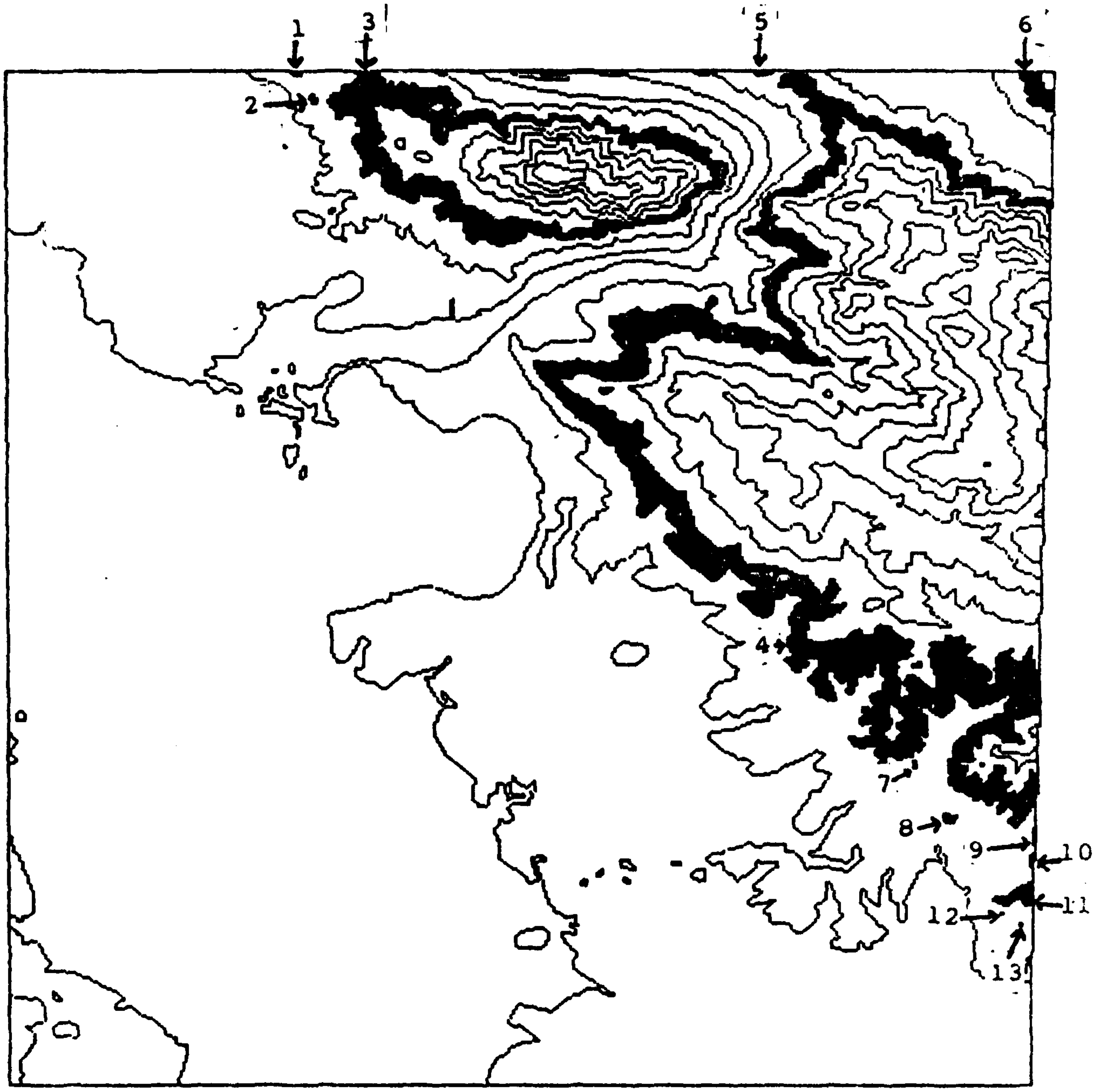


Figure 3.42. The 13 components of the 4th elevation level.

(300 - 400 ft.)

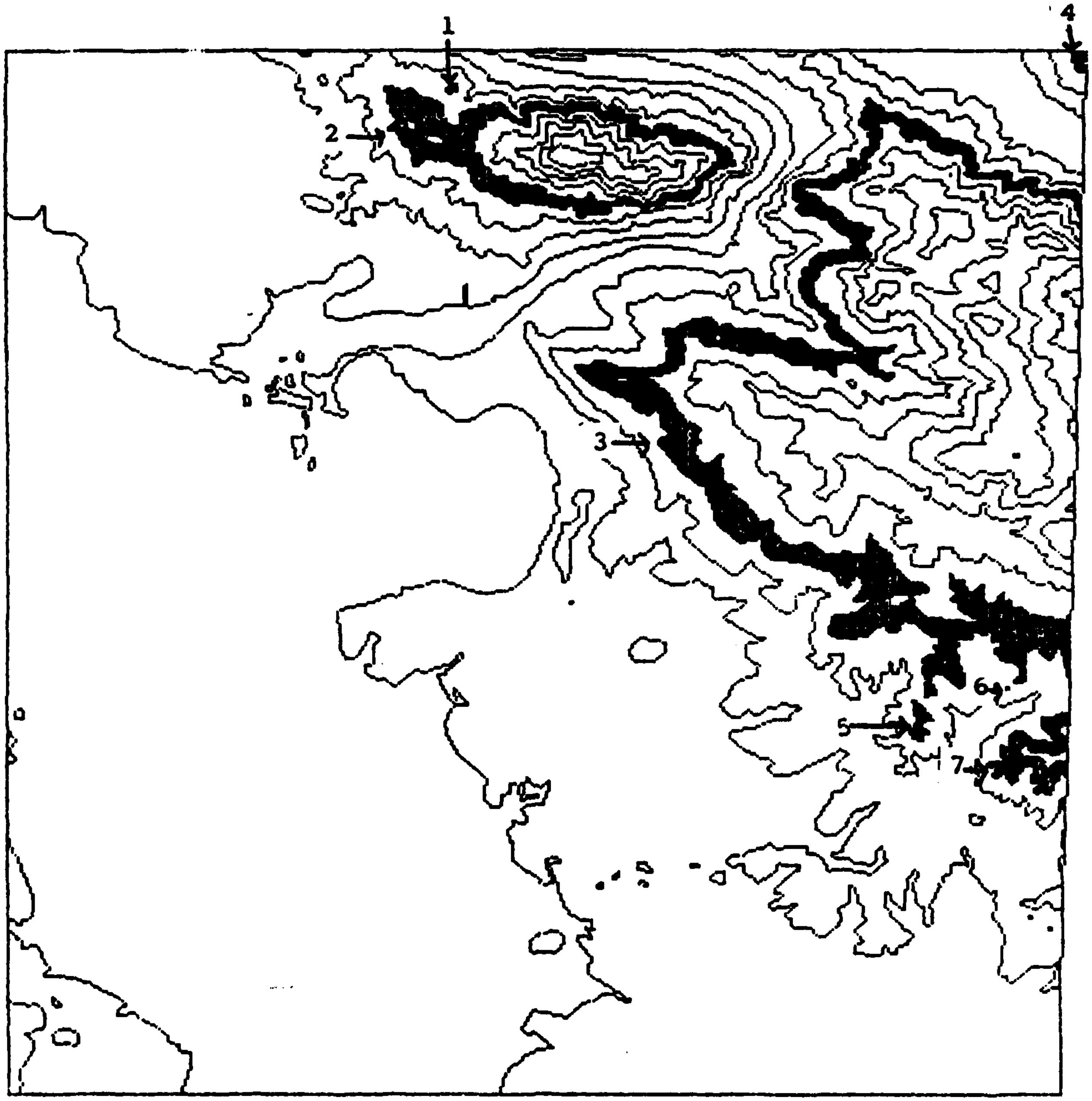


Figure 3.43. The 7 components of the 5th elevation level.
(400 - 500 ft.)

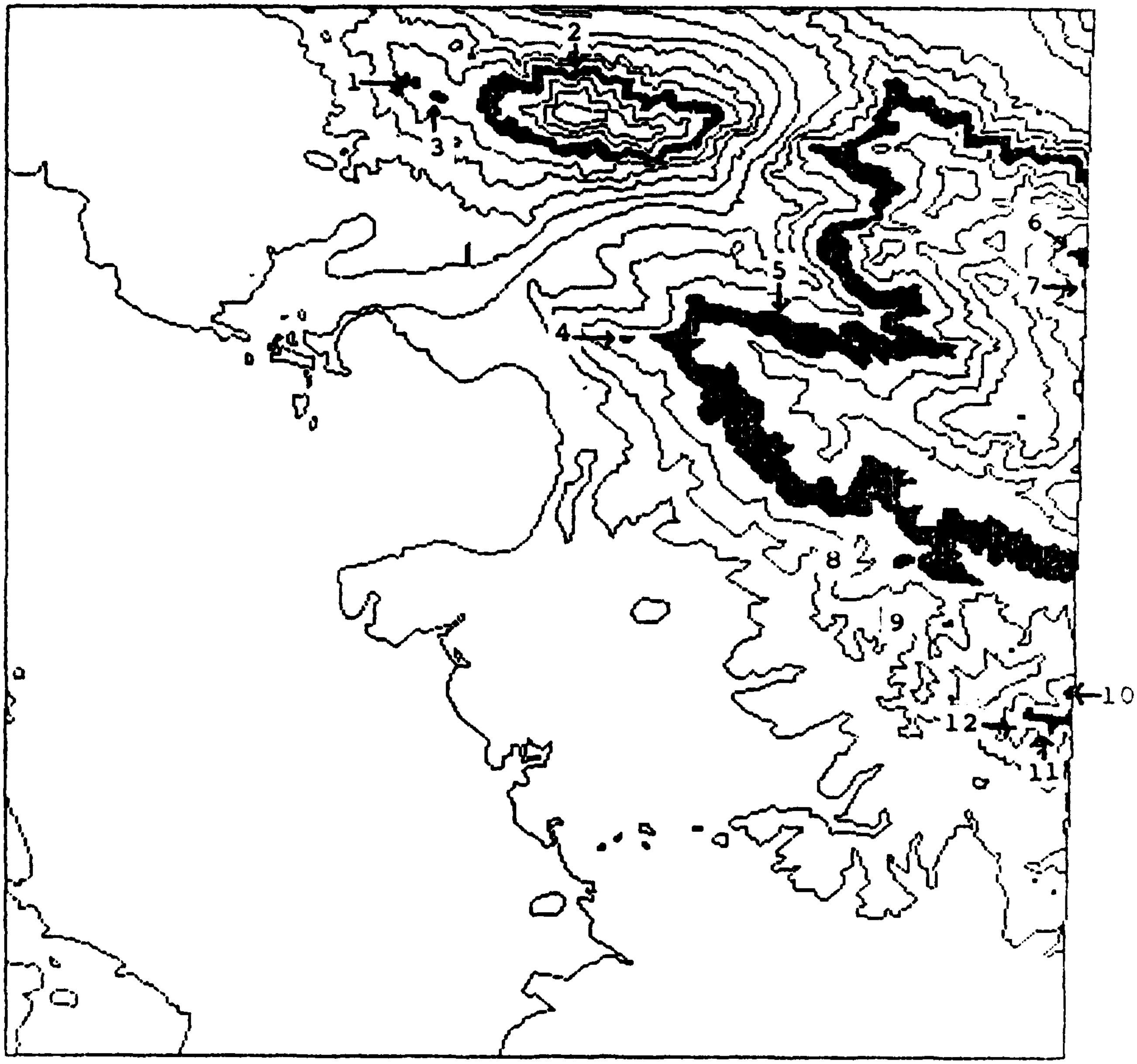


Figure 3.44. The 12 components of the 6th elevation level.

(500 - 600 ft.)

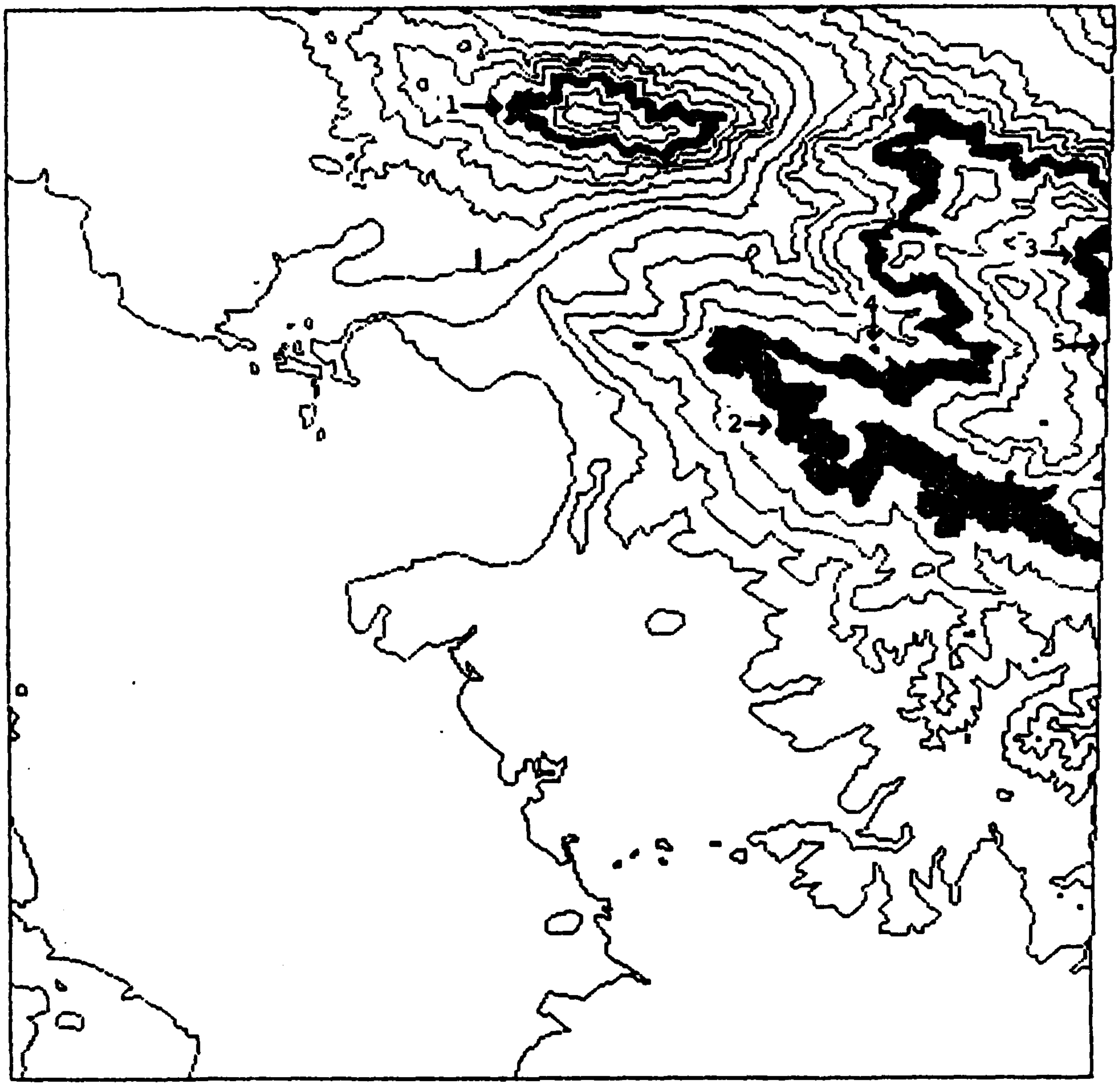


Figure 3.45. The 5 components of the 7th elevation level.

(600 - 700 ft.)

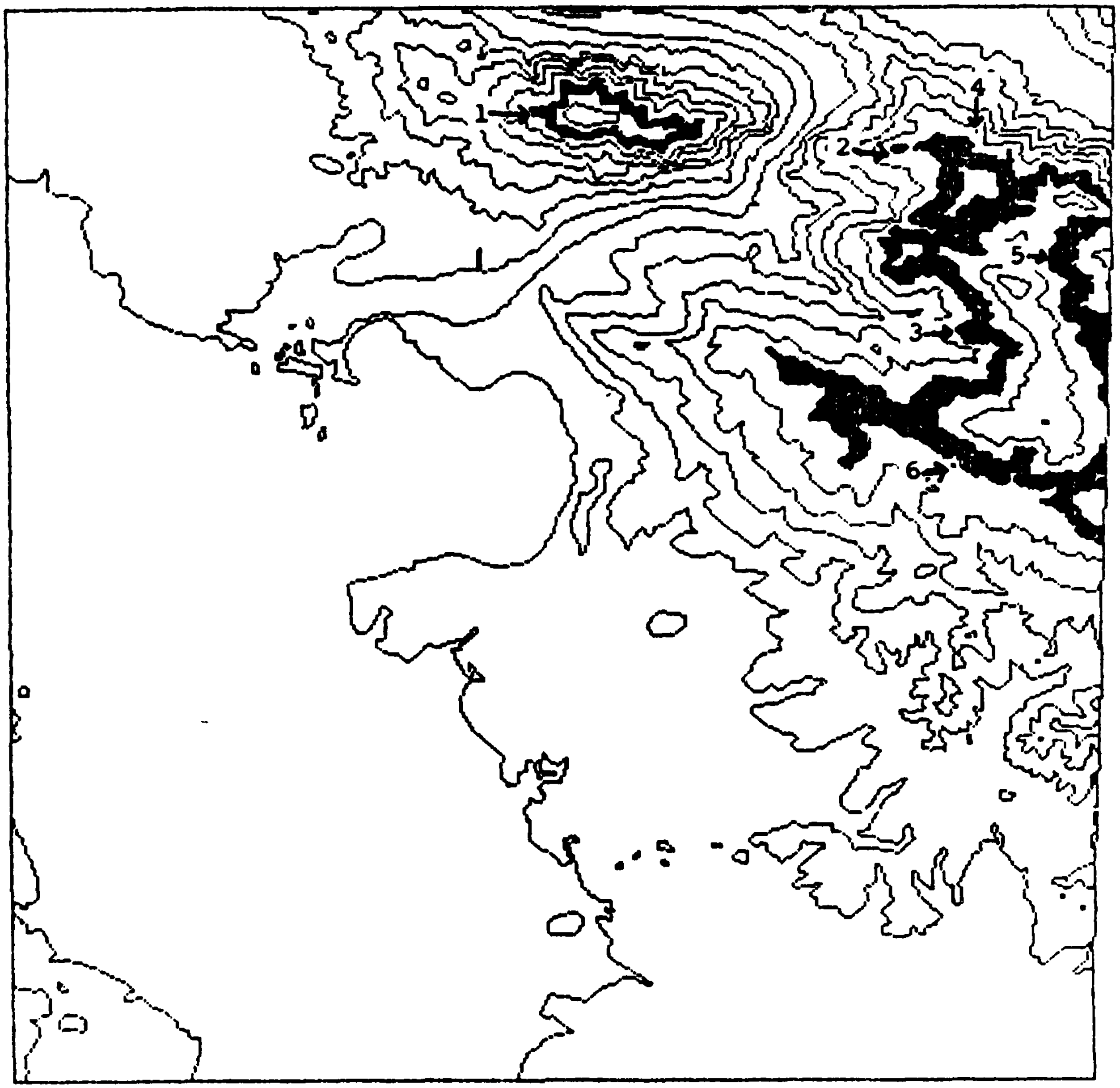


Figure 3.40. The 6 components of the 8th elevation level.

(700 - 800 ft.)

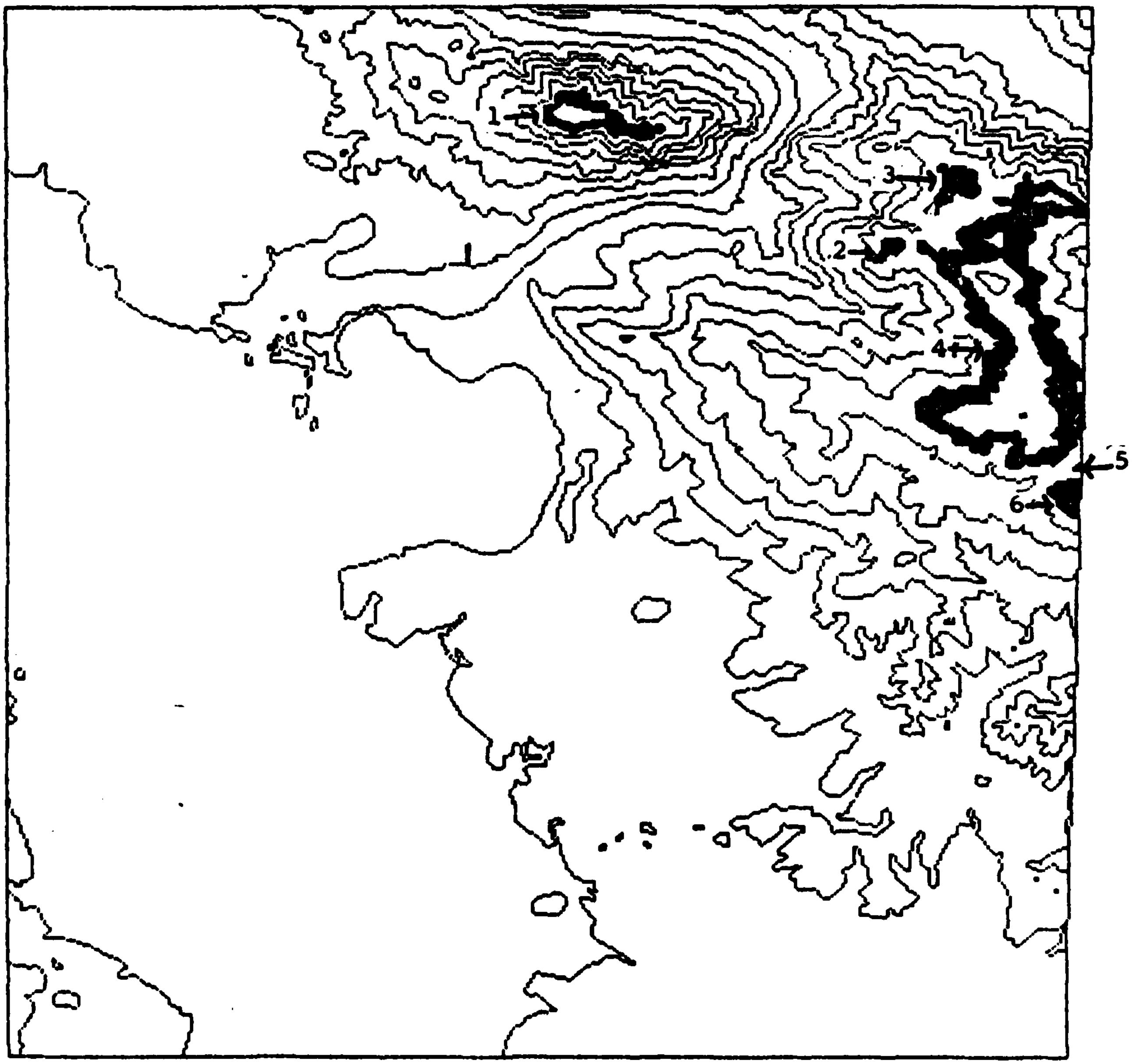


Figure 3.47. The 6 components of the 9th elevation level.
(800 - 900 ft.)

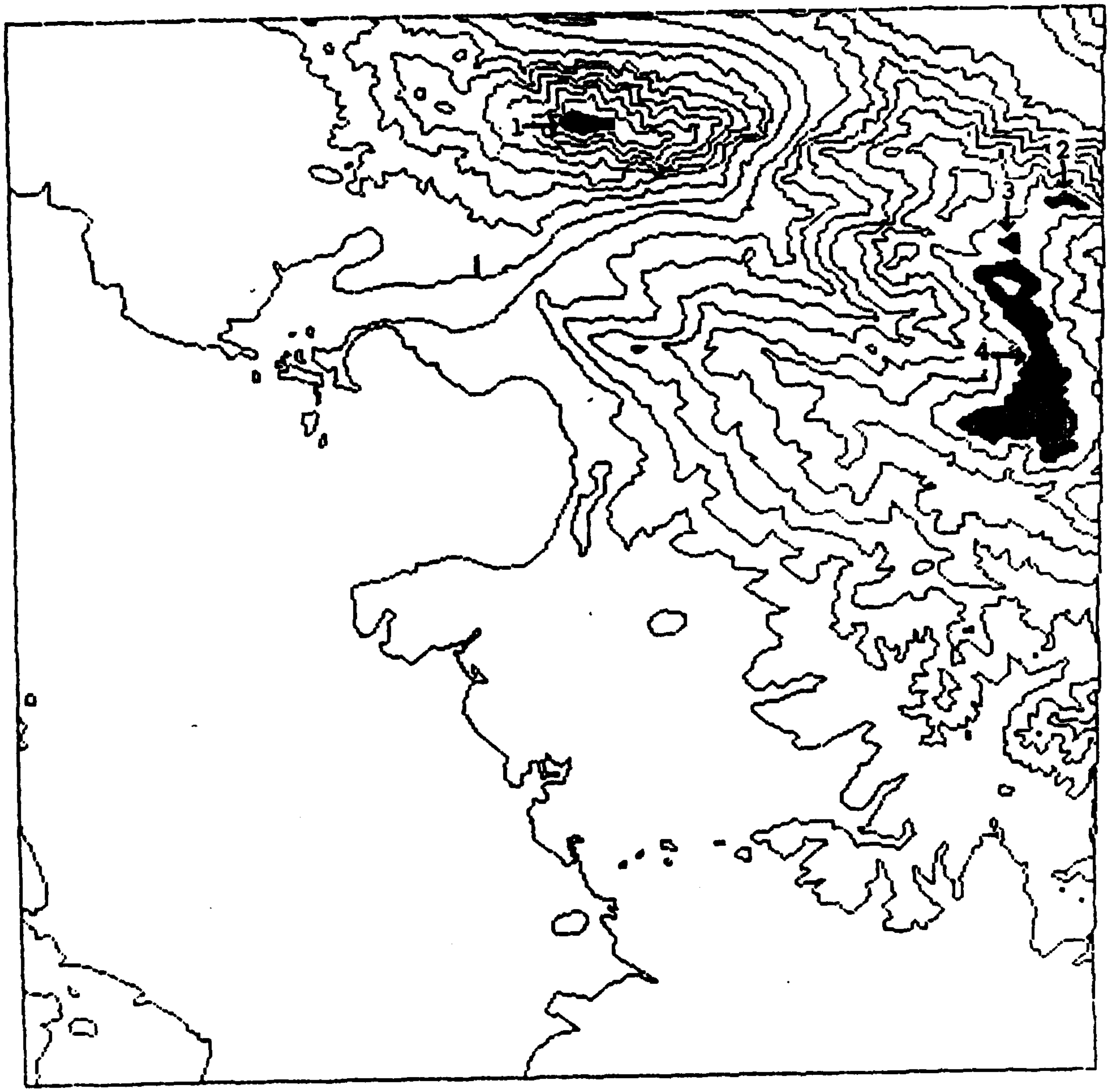


Figure 3.48. The 4 components of the 10th elevation level.

(900 - 1000 ft.)

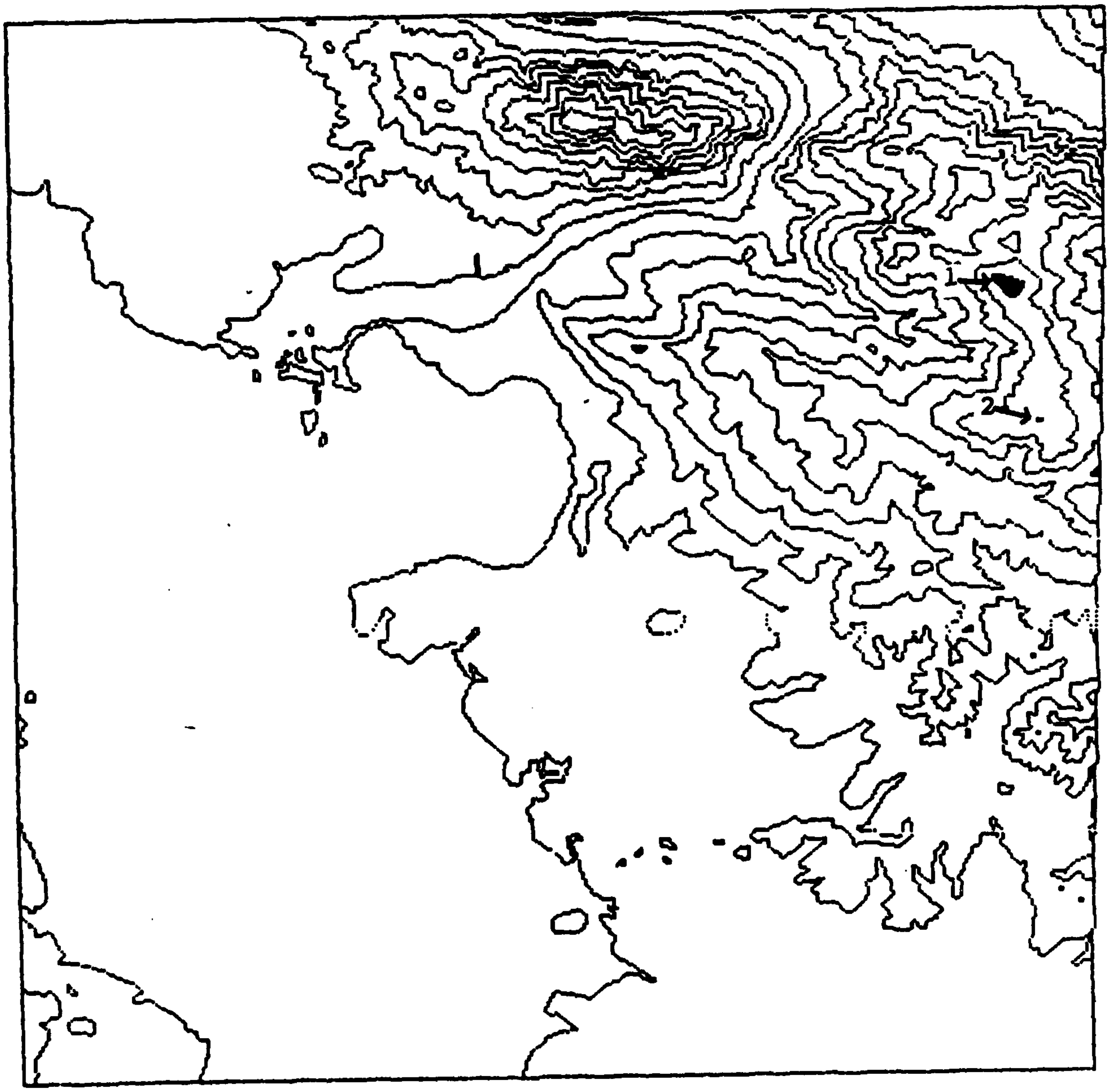


Figure 3.49. The 2 components of the 11th elevation level.
(1000 - 1100 ft.)

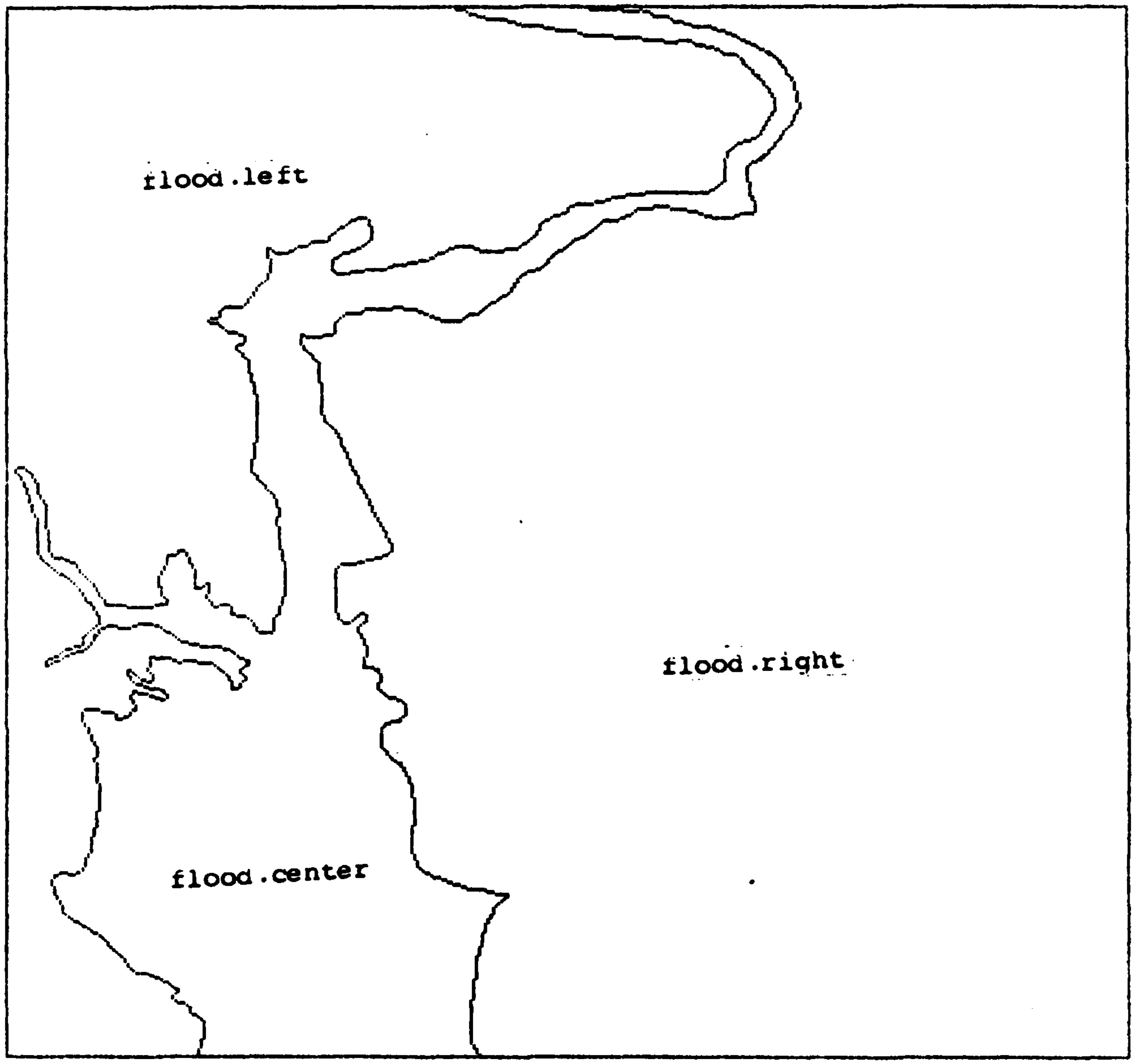


Figure 3.50. The 3 components of the flood-plain map.

3.2. Data editing functions

Below we describe the algorithms that were implemented to edit the maps prior to storing them as quadtrees. These algorithms were implemented in the programming language C to run on the PDP-11/45, VAX 11/780, and GRINNELL configuration, which is described in the Appendix to this report. The maps provided were initially hand-digitized and stored in a chain code representation, as described in Section 3.1. Although quadtrees could have been built directly from the chain code representation, it was considered useful instead to use picture files as an intermediate representation between the initial chain codes and the final quadtrees. This allowed access to many standard routines that are part of our software library. Below, we describe the nonstandard routines that were used on this project.

The algorithm descriptions proceed in the following manner. First we describe the program MKBIN, which converts the chain codes to picture file format. Then three routines for manipulating the picture files are described. These are FIXPIX (changes the value of pixels referenced by their coordinates), RELABEL (translates one list of pixel values into another), and LINERM (erases lines from maps).

The function MKBIN (make binary array) takes as input a file that describes a chain code segmentation of the original maps and creates a picture file. For the following descriptions, it is simplest to view a picture file as a binary array that has been laid out on a disk in a row by row manner. The file that results from MKBIN will describe the map as a white map broken up by a series of black lines. Since the black lines are described by the chain codes, MKBIN simply traces the chain code on a binary array, marking each pixel that lies on the chain code as black.

Having created the picture file, two minor utility routines were found useful. One of these is FIXPIX, which changes the value of a pixel when given the coordinates of the pixel and the new value. This is the equivalent of assigning to an entry in a 2-d array. FIXPIX is used to fix problems that result from errors in the entry of the chain code digitization and also errors that result from labeling the regions of the map. The other utility is RELABEL, which produces a copy of the input picture file where all the pixel values are changed according to a given translation table. Both of these utilities are used for changing pixel values, but FIXPIX does this based on specified coordinates, whereas, RELABEL changes all pixels that have a given value.

The land-use classes (as well as contours, etc.) are labeled by using a standard connected component program and then using RELABEL to merge the labels of components of the same class. There still remains the problem of which labels to assign the pixels that lie on the boundaries of the regions. It is this assignment plus the original hand encoding of the map that accounts for any errors in the calculation of statistics by the quadtree algorithms. The assignment decision is rather arbitrary (but applied consistently to each pixel alike) and implemented by the function LINERM. The decision of which region to assign a given boundary pixel is based on the direction of easiest movement through a picture file (which is from right to left and from top to bottom). Thus a BLACK pixel (a pixel having the color of the boundary line) is given the value of its neighbor on the right if that neighbor is not BLACK and the value of the neighbor above otherwise. Note that at any point during LINERM, the algorithm stores two rows of the picture in core. When working with the first row of the picture, the second neighbor used is the neighbor below since there is no neighbor from above. This processing is repeated over the entire picture file as many times as the maximum of the line's thickness (measured in pixels). A line of thickness greater than one can be created by MKBIN when many lines touch.

Algorithm 3.1. FIXPIX

```
/* Take a picture and a data file "input" which contains records
   declared: "x-coord y-coord newval". The pixel at coord
   <x-coord, y-coord> will be changed to newval. The records must
   be in ascending order of y-coords as only one pass is made
   through the picture (getting new rows as necessary).
   The procedure getrow reads the picture file filling the buffer
   with the next row.   */

fixpix(inpic, input, numcols, numrows)
 INTEGER numcols, numrows;
 DATA FILE inpic, input;
{
 INTEGER ARRAY rowbuff[numcols];
 INTEGER rownum = 0;
 INTEGER x, y, val;

 getrow(inpic,rowbuff);
 WHILE(NOT end of file "input")
  {
   getrecord(input,x,y,val);
   if(y > rownum)
   FOR(rownum=rownum TO y)
     getrow(inpic,rowbuff);
   rowbuff[x] = val;
  }
}
```

Algorithm 3.2. LINERM

```
/* Remove the black pixels from a multi-color picture.  For every
   pixel in the picture, do the following: If the pixel is black,
   then if the right neighbor is not black, give the pixel the
   value of its right neighbor.  If the right neighbor is also
   black, then give the pixel the value of the neighbor above it.
   Some pixels may remain black (they have both neighbors black),
   if so the algorithm should be repeated. */

linerm(inpic,numcols,numrows)
 INTEGER numcols,numrows;
 DATA FILE inpic;
{
 INTEGER ARRAY inbuff[2][numcols+1];
 INTEGER ARRAY outbuff[numcols];
 INTEGER POINTER currpnt, otherpnt;
 INTEGER i,j;

 currpnt = 0;
 otherpnt = 1;
 getrow(inpic,inbuff[0]);
 getrow(inpic,inbuff[0]);
 reset inpic file to beginning;
/* As a special case, the top row actually uses the neighbor below
   it rather than the neighbor above it, and the right hand col
   uses the neighbor to the left.  This is done by putting an
   imaginary row above the first, and an extra col to the right
   of the last. */
 inbuff[0][numcols] = inbuff[0][numcols-2];
 for(i=1 TO numrows)
   {
    currpnt = (currpnt == 0);
    otherpnt = (otherpnt == 0); /* flip these two pointers */
/* Currpnt always points to the current row.  Otherpnt points at
   the row above.  */
    getrow(inpic,inbuff[currpnt]);
    inbuff[currpnt][numcols] = inbuff[currpnt][numcols-2];
    FOR(j=0 TO numcols-1)
      {
       IF(inbuff[currpnt][j] == BLACK)
         IF(inbuff[currpnt][j+1] <> BLACK)
           outbuff[j] = inbuff[currpnt][j+1];
         ELSE
           outbuff[j] = inbuff[otherpnt][j];
       ELSE
         outbuff[j] = inbuff[currpnt][j];
      }
    output(outbuff);
   }
}
```

Algorithm 3.3. MKBIN

```
/* Make a binary array from a set of chaincodes. The chaincode
   used is as follows:  "<coord-part><directional-part>#"
   <coord-part> is simply a six digit number with the 3 digit
   x-coord and the 3-digit y-coord.
   <directional-part> is one or more occurrences of:
        <direction-character> or "[<number><direction-character>]".
   <number> is a 2-digit number which means that the
   <direction-character> occurs number times.
   <direction-character> is one of of:
                   i o p
                    \|/
                  k -*- ;
                    /|\
                   , . /

   If the character is "k" this would indicate one step west,
   "," would indicate one step southwest, etc.  These symbols
   were chosen because of their location on the keyboard. */

makebin(width,height)
 INTEGER width, height;
/* Create a binary array of size width X height. */
/* The function getchar returns the next character that is not a
   line-feed from file "input". */
{
 BINARY ARRAY arr[width][height];
 INTEGER xcoord, ycoord, numb, i;
 CHARACTER ch;

 WHILE(getcoords(xcoord,ycoord))
  {  /* For each chaincode in the file... */
   arr[xcoord][ycoord] = 1;
   ch = getchar(input);
   WHILE(ch <> '#')
     {
      IF(ch == '[')
        run(numb,ch);
      ELSE
        numb = 1;
      FOR(i=1 TO numb)
        CASE OF ch
          {
           'i': xcoord = xcoord - 1; ycoord = ycoord - 1;
           'o': ycoord = ycoord - 1;
           'p': xcoord = xcoord + 1; ycoord = ycoord - 1;
           'k': xcoord = xcoord - 1;
           ';': xcoord = xcoord + 1;
           ',': xcoord = xcoord - 1; ycoord = ycoord + 1;
           '.': ycoord = ycoord + 1;
           '/': xcoord = xcoord + 1; ycoord = ycoord + 1;
          }
      arr[xcoord][ycoord] = 1;
      ch = getchar(input);
     }
  }
}
```

```
BOOLEAN FUNCTION getcoords(x,y)
 INTEGER x, y;
/* If the input file is empty return FALSE. Otherwise read the
   coords from the input file (into x,y) and return TRUE. */

PROCEDURE myget(numb,ch)
 INTEGER numb;
 CHARACTER ch;
/* Read the 2-digit number and the following character from the
   input file, then skip the character "]", returning the number
   in numb and the character in ch. */
```

Algorithm 3.4. RELABEL

```
/* Change the value of the pixels in a picture as determined by the
   labels given in file "labels". This file has as its first value
   an integer which is the largest value occurring in the original
   picture, followed by records of the form "old-val new-val". */

relabel(inpic,labels,numcols,numrows)
  INTEGER numrows, numcols;
  INTEGER ARRAY inpic[numcols][numrows];
  DATA FILE labels;
{
 INTEGER val,new,old,i,j;
 INTEGER POINTER table;

 val = getnum(labels);
 table = create-storage((val+1) * sizeof val);
/* Create-storage is a system function which dynamically reserves
   the number of words given by the parameter. The sizeof operator
   returns the number of words used by the variable following. */
/* Initialize table so that the new label will be the same as the
   old label, unless a change is indicated in the file "labels". */
 FOR(i = 0 TO val)
   table[i] = i;
 WHILE(not at end of "labels")
   {
    old = getnum(labels);
    new = getnum(labels);
    table[old] = new;
   }
/* Change picture. */
 FOR(i=0 TO numcols)
    FOR(j=0 TO numrows)
      inpic[i][j] = table[inpic[i][j]];
}
```

4. Quadtree encoding

4.1. Introduction

This section describes the quadtree encoding algorithms as well as various primitive functions used in conjunction with quadtree data structures. Display of quadtree-encoded data was particularly facilitated by the ability of the GRINNELL to accept specifications of rectangles to be output. It should be noted that all of the following algorithms work on the digitized version of the maps described in Section 3 and that no new errors are introduced by these algorithms' manipulations of the quadtrees, since the representation of the digital data remains exact. No deviation from pure quadtree representation has been introduced.

The algorithm descriptions proceed in the following manner. First we present a set of primitive functions that form the building blocks for later algorithms. Then we discuss two algorithms that were instrumental in building the quadtree database from the digitized maps. The first of these two algorithms builds a quadtree from a map by scanning the map in a row by row fashion (referred to as raster scanning). The second algorithm labels the connected components of a map.

4.2. Primitive functions

The functions SON, FATHER, SONTYPE, NODETYPE, BLACK, WHITE, and GRAY can be thought of as defining the quadtree as an abstract data type. Although their implementation is trivial, their usage gives the other quadtree algorithms a certain independence from the chosen representation of the quadtree data structure. Since it is our intent to experiment with other quadtree representations, this will save future programming effort. Currently each node of the quadtree is represented by a record consisting of five pointers and an integer. The pointers are used to link to other nodes; one pointer links to the node's father and the remaining four pointers link to the node's four sons and are indexed by the quadrant in which the son lies. A value of NIL is stored to indicate the absence of a son in a given direction. An integer value is used to uniquely identify the polygon, land-use class, or contour to which the region represented by the node belongs. If this value is not unique for the region, then the value is considered gray (this term comes from the usage of gray nodes in black and white binary-valued quadtrees).

Using such a quadtree representation, the above defining functions work as follows. The function SON takes a node and a quadrant as parameters and returns the node that is the son of the given node in the given quadrant by dereferencing the appropriate pointer. Similarly, the function

FATHER takes a node as parameter and returns the father of the given node by simply dereferencing the appropriate pointer. The function SONTYPE takes a node as parameter and returns the quadrant that expresses the direction from the father of the given node to the given node by comparing the address of the given node to the address of each of its father's sons. This function returns a special value NIL to indicate that the given node is the root of a quadtree and hence has no father. The function NODETYPE takes a node as its parameter and returns the integer data item that is stored at that node which generally indicates a region color, class type, or elevation. The predicates BLACK, WHITE, and GRAY each take a node as parameter and return true if the value of NODETYPE is to be interpreted as having the value indicated by the function's name. This allows multicolor quadtrees to be easily interpreted as binary-colored quadtrees when it is convenient to do so.

The functions OPSIDE, CCSIDE, ADJ, REFLECT, QUAD, and OPQUAD provide a simple set of operations to manipulate directions. There are two important classes of directions used by quadtree algorithms. The first is the four basic directions denoted N, E, S, and W that are used to indicate the side of the square that lies in that direction from the square's center. The second is the four compound directions denoted NW, NE, SE, and SW that are used to indicate the quadrant of the square that lies in that direction from the square's center. The functions OPSIDE and CCSIDE each take a side as parameter and return respectively the side in the opposite direction and the side in the direction 90 degrees counterclockwise from the square's center. The predicate ADJ takes a side and a quadrant as parameters and returns true iff the given quadrant is adjacent to the given side. For example, the NE quadrant is adjacent to both the N and E sides but not to the S or W sides. The function REFLECT takes a side and a quadrant as parameters and returns the quadrant that is the reflection of the given quadrant with respect to a line through the center of the square that is parallel to the given side. For example, the SW quadrant is the reflection of the NW quadrant with respect to a line through the square's center that is parallel to either the N or S sides. The function QUAD takes as parameters two sides and returns the quadrant that is adjacent to both sides if this condition uniquely determines one quadrant. If it does not (i.e., the two sides are either opposite or the same), then the value NEG is returned. The function OPQUAD takes a quadrant as parameter and returns the quadrant that lies in the opposite direction from the center of the square (i.e., 180 degrees). In our particular implementation, each of the two classes of directions is represented by the integers 0 thru 3 inclusive; so the above functions are implemented by modular arithmetic where convenient and otherwise by enumeration of the possible values (i.e., table lookup via a case statement).

Central to the approach to quadtrees that we have adopted is the ability to find a node's neighbor without storing explicit links to each node's neighbors as is done in some other implementations. This ability is encoded in the function FIND_NEIGHBOR, which takes as parameters a node and a side and returns a node that abuts the indicated side of the given node and is either a leaf or is of the same depth as the given node. The manner in which this is done is described in the tutorial section of this report. The function MAKE_NEIGHBOR behaves in the same manner as FIND_NEIGHBOR except that if it fails to find a common ancestor or runs into a leaf before it has finished the mirrored path, then it modifies the tree by inserting the sought-after node and continues on.

The remaining functions GETNODE, CREATENODE, and RETURNTOAVAIL are used for storage management. Unused nodes are kept on an AVAIL list. The function GETNODE returns a used node by first looking on the AVAIL list and if the AVAIL list is empty then requesting more storage from the operating system. An error message results if no storage is available and the program terminates. The function CREATENODE takes a node, a quadrant, and an integer nodetype as parameters and uses GETNODE to create a new node of the given nodetype which has the given node as father, lies in the given quadrant of the given node, and itself has no sons. The function RETURNTOAVAIL takes a node as parameter and inserts it into the AVAIL list.

4.3. Database building

Prior to constructing the quadtree database, the maps were stored in picture files, which can be viewed as 2-d arrays laid out on a disk in row-by-row order. Thus the first task to be performed to convert each picture file into a quadtree file, which is a preorder listing of the nodes in a quadtree. This is accomplished by the R2Q (raster to quadtree) function. This function reads a picture file one row at a time (raster scan order) and builds the corresponding quadtree using the MAKE_NEIGHBOR primitive. As the quadtree is being built, identical leaf brothers are merged as indicated in the discussion of the WINDOW function. An additional efficiency results from realizing that it is only necessary to check for these mergers on even-numbered rows; any leaf on an odd-numbered row, still has two brothers that have yet to be read in.

The original picture files had each pixel labeled according to the land-use class (contour, etc.) to which it belonged. Thus, these labels were the only distinctions that could be carried over in the construction of the quadtrees by R2Q. However, the database design called for unique labels on each connected component of each class. Hence, it was necessary to perform a connected component analysis in order to label the quadtrees in the desired format. This was done by the function QCONCOM (quadtree connected component finder). This analysis was performed on the quadtree data structure directly (instead of being done on the picture files prior to quadtree construction) because the number of nodes to be processed in the quadtrees was substantially smaller than the number of pixels to be processed in the original picture thereby allowing the analysis to be performed faster. The function QCONCOM works in the following manner (processing only one class at a time). The first step assigns an initial tentative labeling to the quadtree. This labeling is based on a preorder traversal of the quadtree that starts in the northwest corner of the image and moves in the south and east directions. If a BLACK node is met that is unlabeled (with respect to the component within which it is contained), then a new label is created for it. When processing a BLACK node, FIND_NEIGHBOR is used to examine its southern and eastern neighbors to determine if they are also BLACK, but have no component label. In such a case, they are assigned the component label of the BLACK node being processed. If they already had a component label, then both labels are placed (as an ordered pair) on an equivalence list. Once all the nodes have been tentatively labeled, one merges the equivalence classes and then updates the component labels so that each connected component has just one label.

Algorithm 4.1. PRIMITIVES

```
/* The following is a description of the primitive functions
   used in the quadtree algorithms. */

node FUNCTION son(p,i)
/* Given node p and quadrant i, return the node which is the
   son representing quadrant i of node p. */

node FUNCTION father(p)
/* Given node p, return the node which is the father of p. */

INTEGER FUNCTION sontype(p)
/* Given node p, return q where son(father(p),q) = p.  If p is
   the root, then return NIL. */

INTEGER FUNCTION nodetype(p)
/* Return the value of node p.  This can be considered as GRAY,
   WHITE, or BLACK for a binary tree; and GRAY, WHITE or a class
   type or elevation level value for a multi-colored tree. */

BOOLEAN FUNCTION black(p)
/* TRUE when nodetype(p) is BLACK if the tree is binary, or
   when nodetype(p) is a value specified as BLACK by the user
   if the tree is multi-colored. */

BOOLEAN FUNCTION white(p)
/* TRUE when nodetype(p) is WHITE if the tree is binary, or
   when nodetype(p) is a value specified as WHITE by the user
   if the tree is multi-colored. */

BOOLEAN FUNCTION gray(p)
/* TRUE iff nodetype(p) is GRAY. */

INTEGER FUNCTION opside(b)
/* Given a side b, return the opposite side (e.g.,
    opside(E) = W). */

INTEGER FUNCTION ccside(b)
/* Returns the side adjacent to side b in the clockwise
   direction (e.g., ccside(E) = N). */

BOOLEAN FUNCTION adj(b,i)
/* TRUE iff quadrant i is adjacent to boundary b of the node's
   block (e.g., adj(N,NW) = TRUE; adj(N,SW) = FALSE). */

INTEGER FUNCTION reflect(b,i)
/* Returns the quadrant which is adjacent to quadrant i along
   boundary b (e.g., reflect(N,NW) = SW). */

INTEGER FUNCTION quad(b,c)
/* Returns the quadrant bounded by b and c if it exists and the
    value NEG if it does not exist. */

INTEGER FUNCTION opquad(q)
/* Returns the quadrant opposite (non-adjacent) to q
   (e.g., opquad(NW) = SW). */
```

```
node FUNCTION find_neighbor(q,s)
/* Return the node which is adjacent to side "s" of node "q"
   and is either a leaf or is at the same depth as node "q".
   This is done by following the father links until the common
   ancestor is reached and then following the reflected path
   downward, stopping short only if a leaf is met. */
node POINTER q;
INTEGER s;
{
 node POINTER p;
 INTEGER i,stypeq;

/* First find a common ancestor. */
 IF(NULL(sontype(q)))  /* Common ancestor does not exist. */
    RETURN(NIL);
 ELSE IF(adj(s,sontype(q))) /* Neighbor is not a sibling -
                               go up to next level. */
      p = find_neighbor(father(q),s);
 ELSE  /* Neighbor is a sibling.  Father is a common ancestor. */
    p = father(q);

/* After finding the common ancestor, reflect about side "s"
   back to the level of the original request. */
 IF(NULL(p) OR NULL(son(p,reflect(s,sontype(q)))))
      /* Either there was no common ancestor or p is a leaf
         and in either case p is what we want to know;
         so, don't change it. */
      RETURN(p);
 ELSE /* Return the calculated son. */
      RETURN(son(p,reflect(s,sontype(q))));
  }
}

node FUNCTION make_neighbor(q,s)
/* Return the node which is adjacent to side "s" of node "q"
   and at the same depth as the node "q".  This is done by
   following the path through the tree that would lead us
   to said neighbor if it existed and creating, along the way,
   any nodes that are necessary. Whenever such nodes are created,
   all created sons are set to WHITE.  They are later reset to
   GRAY or BLACK as appropriate, c.f., find_neighbor */
node POINTER q;
INTEGER s;
{
 node POINTER p;
 INTEGER i,stypeq;

/* First find the nearest common ancestor. */
 IF(NULL(sontype(q)))  /* Common ancestor does not exist. */
   {
    /* Create a common ancestor and initialize its
       pointers. */
    p = createnode(NULL,NULL,GRAY);
    stypeq = quad(ccside(s),opside(s));
    p->sons[stypeq] = q;
```

```
    q->fathr = p;
    /* Create the other three sons of p. */
    createnode(p,opquad(stypeq),WHITE);
    createnode(p,opquad(reflect(s,stypeq)),WHITE);
    createnode(p,reflect(s,stypeq),WHITE);
   }
 ELSE IF(adj(s,sontype(q))) /* Neighbor is not a sibling -
                               go up to the next level. */
     p = make_neighbor(father(q),s);
 ELSE /* Neighbor is a sibling.  Father is common ancestor. */
    p = father(q);

/* After finding the nearest common ancestor, reflect about
   side "s" back to the level of the original request. */
IF(NULL(son(p,reflect(s,sontype(q)))))
  { /* If the node does not have children to descend a level,
       change the node to gray and give it children. */
   p->nodetype = GRAY;
   FOR(i = NW,NE,SE,SW)
     createnode(p,i,WHITE);
   return(son(p,reflect(s,sontype(q))));
  }
}

node FUNCTION getnode()
/* Reserves storage for a quadtree node and returns a pointer
   to this unit of storage  */

node FUNCTION createnode(root,s,t)
/* Create a node p with nodetype t which corresponds to son s
   of node root and return p. */
node POINTER root;
INTEGER s,t;
{
 node POINTER p;
 p = getnode();
 if(root != NIL)
   root->sons[s] = p;
 p->fathr = root;
 p->ntype = t;
 for(i = NW,NE,SE,SW)
   p->sons[i] = NIL;
 return(p);
}

PROCEDURE returntoavail(p)
/* Return node p to the available storage pool. */
```

```
/* Run a connected components algorithm on a binary quadtree -
   i.e.,assign every connected component a unique label.  This is
   achieved in three steps.  Step 1 assigns labels to each BLACK
   node. This is done by traversing the tree and for every BLACK
   node, examining its eastern and southern neighbors. If these are
   unlabeled, a new label is generated for the current node.  If
   either of them are labeled, and the current node is unlabeled,
   then assign current node the label of its neighbor.  If the
   neighbor is labeled and the current node is labeled, then these
   labels are equivalent and so the pair of labels is added to a
   list of equivalence classes.  The second step is to put the list
   of equivelance classes into a hierarchal order so that all of
   the nodes in the class can be given one label.  No algorithm is
   given for this step - for an example of a typical algorithm of
   this kind see Knuth, Vol 1.  The third step simply traverses
   the tree again, relabeling each node to the value of the
   representative for its equivalence class.    */

component(quadtree)
/* "Quadtree" is a pointer to the input tree.  At the end of this
   algorithm, "quadtree" will point to the labeled tree. */
 node POINTER quadtree;
{
 pairlist POINTER merges; /* Pointer to the list of pairs of
                               equivelances. */
 merges = NIL;
 label(quadtree); /* step 1 */
 Process the equivalences in the list merges; /* step 2 */
 update(quadtree);  /* step 3 */
}


label(p)
/* Perform step 1.  Assigns labels to node p and its sons. */
 node POINTER p;
{
 node POINTER q;
 INTEGER i;

 IF(gray(p))
   FOR(i = NW,NE,SE,SW)
     label(son(p,i));
 ELSE IF(black(p))
   {
    q = find_neighbor(p,E);
    if(NOT(NULL(q)))
      label_adjacent(q,NW,SW,p);
    q = find_neighbor(p,S);
    if(NOT(NULL(q)))
      label_adjacent(q,NW,NE,P);
    if(NOT(labeled(p)))
      p->nodetype = getnewregion();
   }
}
```

```
label_adjacent(r,q1,q2,p)
/* Find all descendants of node r adjacent to node p - i.e.,
   in quadrants q1 and q2. */
 node POINTER r,p;
 INTEGER q1,q2;
{
 IF(gray(r))
   {
    label_adjacent(son(r,q1),q1,q2,p);
    label_adjacent(son(r,q2),q1,q2,p);
   }
 ELSE IF(black(r))
    assign_label(p,r);
}


assign_label(p,q)
/* Assign a label to nodes p and q if they do not already have one.
   If both have different labels, then enter them in "merges". */
node POINTER p,q;
{
 IF(labeled(p) AND labeled(q))
   {
    IF(nodetype(p) <> nodetype(q))
       add <nodetype(p),nodetype(q)> to merges;
   }
 ELSE IF(labeled(p))
   q->nodetype = nodetype(p);
 ELSE IF(labeled(q))
   p->nodetype = nodetype(q);
 ELSE
   p->nodetype = q->nodetype = getnewregion();
}


update(p);
/* Perform step 3. */
 node POINTER p;
{
 INTEGER i;

 IF(gray(p))
   FOR(i = NW,NE,SE,SW)
      update(son(p,i));
 ELSE IF(black(p))
   p->nodetype = equivalence of value nodetype(p);
}
```

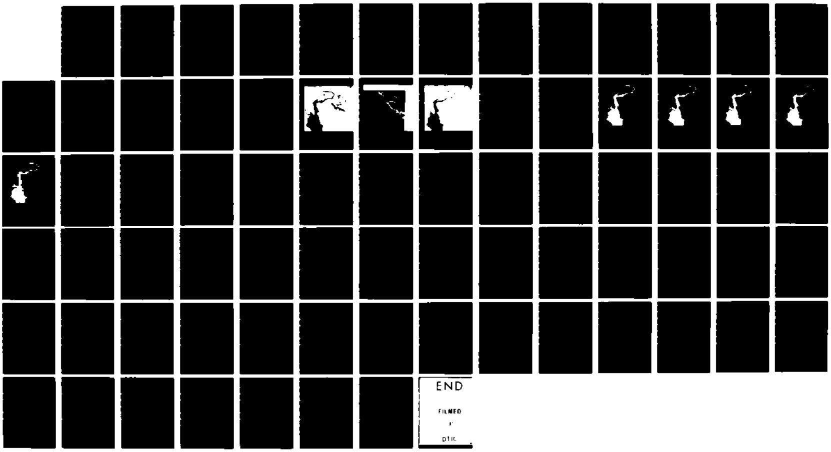
```
/* Convert an input picture in the form of a binary array (or
   raster) into a binary quadtree.  Basically, the algorithm works
   by doing a raster scan of the input picture, and as each pixel
   is read, the quadtree is modified so that it would be a valid
   quadtree representing the input picture if all unprocessed pixels
   were WHITE.  This is in contrast to an algorithm which first
   builds a complete quadtree with one node per pixel and then
   attempts to merge nodes (replace GRAY nodes that have all sons
   the same color with a node of that color).  The input picture is
   read one row at a time by a special function getrow which
   returns the next row of the picture.  The function color returns
   the color of the pixel given as an argument.  The boolean
   function lastrow is true iff the current row is the last row of
   the picture.  Whenever all the children of a node have been
   processed, an attempt is made to merge them together.  Because
   of this there is a distinction between odd rows and even rows
   (no pixels in an odd row can ever complete the processing of all
   the children of a gray node; hence there is never an attempt to
   merge after processing any of these pixels).  The picture is
   assumed to be an 2**N by 2**N picture - if not, WHITE pixels are
   assumed to fill it out.  */

node POINTER quadtree(p,width);
/* Given a picture p (viewed as a list of rows) and its width,
   return a quadtree. */
LIST p;
INTEGER width;
{
 BOOLEAN ARRAY q[1:width];  /* Holds a row of the picture. */
 node POINTER first;
 INTEGER i;

 q = getrow(p);
 first = createnode(NIL,NULL,q[1]); /* First pixel. */
 oddrow(q,first,width);
 i = 2;
 p = NEXT(p);
 first = evenrow(getrow(p),make_neighbor(first,S),i,width);
 WHILE (NOT lastrow()) DO
   {  /* Process the rest of the rows. */
    p = NEXT(p);
    oddrow(getrow(p),first,width);
    p = NEXT(p);
    i = i+2;
    first = evenrow(getrow(p),make_neighbor(first,S),i,width);
   }
 while(NOT NULL(father(first)))
   first = father(first);  /* Set first to root of the tree. */
 return(first);
}


oddrow(row,nd,width)
/* Add the odd-numbered row of width "width" represented by array
   "row" to a quadtree whose node "nd" corresponds to the first
   pixel in the row. */
```

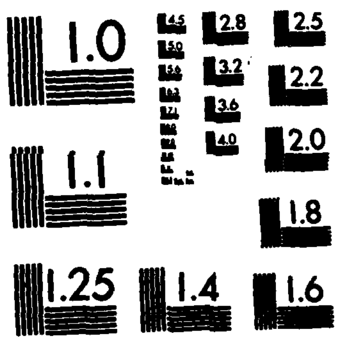
```
INTEGER width;
INTEGER ARRAY row[1:width];
node POINTER nd;
{
 nd->nodetype = color(row[1]);
 FOR(i=2 UNTIL width)
   {
    nd = make_neighbor(nd,E);
    nd->nodetype = color(row[i]);
   }
}


node FUNCTION evenrow(row,first,i,width)
/* Add even numbered row "i" of width "width" represented by array
   "row" to a quadtree whose node "first" corresponds to the first
   pixel in the row.  During this process, merges of nodes having
   four sons of the same color are performed.  */
INTEGER ARRAY row;
node POINTER first;
INTEGER i,width;
{
 node POINTER p,r;
 INTEGER j;

 p = first;
 IF(NOT lastrow())
    /* Remember the first node of the next row. */
    first = make_neighbor(p,S);
 FOR(j=1 UNTIL width-1)
   {
    r = make_neighbor(p,E);
    p->nodetype = color(row[j]);
    IF(EVEN(j))
       merge(i,j,father(p));
    p = r;
   }
 p->nodetype = color(row[width]); /* Don't invoke make_neighbor for
                                      the last pixel in a row.   */
 IF(EVEN(width))
    merge(i,width,father(p));
 RETURN(first);   /* Return the first node of the next row. */
}


node FUNCTION merge(i,j,p)
/* Attempt to merge a node having four sons of the same color
   starting with node "p" at row "i" column "j". */
 node POINTER p;
 INTEGER i,j;
{
 INTEGER k;

 WHILE(EVEN(i) AND EVEN(j) and
          (nodetype(son(p,NW)) = nodetype(son(p,NE)) =
           nodetype(son(p,SE)) = nodetype(son(p,SW)))
```

```
    {
     i = i/2;
     j = j/2;
     p->nodetype = nodetype(son(p,NW));
     FOR(k = NW,NE,SE,SW)
        {
         returntoavail(son(p,k));
         p->sons[k] = NIL;
        }
     p = father(p);
    }
 return(p);
}
```

AD-A124 196 APPLICATION OF HIERARCHICAL DATA STRUCTURES TO GEOGRAPHICAL INFORMATION SYSTEMS(U) MARYLAND UNIV COLLEGE PARK COMPUTER VISION LAB H SAMET ET AL. JUL 82 2/2

UNCLASSIFIED TR-1197 ETL-0301 DAAK70-81-C-0059 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

4.4. Tabulation of results

Below we describe the tables of data collected about the quadtree-represented regions. The times are measured in seconds by a special routine on the VAX 11/780, which allows us to factor out the disk I/O time. Thus the times reported reflect the notion of CPU time as implemented on that machine. These times were measured while the machine was not loaded with any other jobs, because the timing routine does vary in its results as the system load changes. For many algorithms, a better idea of the cost in time can be determined from such machine independent concepts as number of nodes visited. These are also included in many of the tables or are easily deducible from the descriptions of the algorithms in the algorithm overview.

The tables are discussed in the order that they appear appended to this section. These tables treat the data base as a collection of possibly unconnected regions on the maps that are logically connected by the sharing of some property, e.g., having the same land-use class.

The first group of tables (4.1-3) are the QUADTREE BUILDING STATISTICS. These reflect a process by which the R2Q algorithm was used to build a separate quadtree for each logically connected class of polygons is a picture file. No execution time is indicated because the times were dominated by the cost of reading an entire picture file, one line at a time. Each quadtree took approximately 3 minutes to build. If the quadtree had been constructed by building a complete 4-ary tree and then merging where possible, it would be necessary for the memory to be large enough to contain 262,144 (512x512) nodes. By merging during the building process,as R2Q does, a much smaller maximum memory is required in practice. The size required is recorded in the column 'nodes created'. Note that never are more than 15,000 nodes needed. The following column indicates the percentage of this maximum that was actually used when the tree was finished. The remaining columns give a breakdown of the number of nodes of each type in the resulting binary quadtree.

The CONNECTED COMPONENT RESULTS (Tables 4.4-6) record the data collected on three variations of the connected component algorithm, QCONCOM. In each variation, the algorithm uses two of the neighbors of a node, as described in the algorithms overview, to assign a tentative label to a node. The 'number of neighbors sought' is the number of times this process of finding a neighbor must be performed, thus yielding an indication of its importance to the algorithm's analysis. The three variations are three different methods of finding the required neighbors. For purposes of comparison, the average cost for a single finding of a neighbor is calculated to show clearly the variance within each technique as well as the relative costs among techniques. The average cost is measured in number of nodes accessed per neighbor found. Since the ammount of work performed by the algorithm is proportional to the number of nodes accessed, this average cost measure gives an accurate view of the relative tradeoffs among the various methods. The portion of a second required to find a neighbor (an alternative measure) could not be calculated due to the inaccuracy of the system timing algorithm. Also, measurement in seconds of algorithm efficency can be misleading because the algorithms were coded in a highlevel language and some of the timing differences could reflect the relative efficencies of the compiler's optimizer rather than that of the quadtree algorithm.

The first method is FINDNBR, which is the FIND_NEIGHBOR primitive mentioned in the algorithm overview and described in the tutorial section. The time in the final column is for this method. The second method, ROPES, is due to Hunter's quadtree work referred to in the tutorial. It consists of placing a link directly between each neighbor of the same size. This results in a reduction in execution time at a major cost in storage (due to storing the extra links). The third method was discovered during work on this project and consists of causing the traversal algorithm to pass as parameters the neighbors of each subtree's root. This requires more time than ropes, but does not require as much memory. The added memory cost with respect to the FIND_NEIGHBOR technique results from the additional stack size needed due to the larger parameter list of the recursive routines. The average value is based on equating the cost of passing a parameter to a subroutine with the cost of dereferencing a link. This equivalence is clearly compiler-dependent as well as machine-dependent. The final column of the table indicates overall execution time of the connected component algorithm, QCONCOM, which uses the FIND_NEIGHBOR.

TABLE 4.1. QUADTREE BUILDING STATISTICS FOR LANDUSE MAP

CLASS	NODES IN TREE	NODES CREATED	% USED IN TREE	GRAY NODES	WHITE NODES	BLACK NODES
acc	4337	5925	73.2	1084	1847	1406
acp	7725	8981	86.0	1931	3048	2746
ar	1145	2697	42.5	286	499	360
are	129	1725	7.5	32	71	26
avi	11937	13341	89.5	2984	4776	4177
avv	13193	14445	91.3	3298	5359	4536
bbr	537	2109	25.5	134	250	153
beq	353	1873	18.8	88	168	97
bes	193	1825	10.6	48	94	51
bt	2293	3841	59.7	573	951	769
fo	5485	7109	77.2	1371	2121	1993
lr	1481	3045	48.6	370	670	441
r	7001	8609	81.3	1750	2792	2459
ucb	249	1881	13.2	62	118	69
ucc	817	2433	33.6	204	381	232
ucr	1069	2701	39.6	267	457	345
ucw	449	2081	21.6	112	197	140
ues	1113	2737	40.7	278	506	329
uil	345	1977	17.5	86	158	101
uis	1037	2649	39.1	259	453	325
uiw	293	1917	15.3	73	139	81
unk	1121	2681	41.8	280	540	301
uoc	173	1805	9.6	43	79	51
uog	377	2009	18.8	94	149	134
uoo	429	2061	20.8	107	201	121
uop	269	1901	14.2	67	136	66
uov	229	1861	12.3	57	99	73
urn	237	1861	12.7	59	125	53
urs	9921	11313	87.7	2480	3993	3448
uus	297	1921	15.5	74	142	81
uut	3069	4621	66.4	767	1379	923
vv	153	1785	8.6	38	76	39
wo	485	2029	23.9	121	225	139
ws	4677	6245	74.9	1169	2025	1483
wwp	457	2049	22.3	114	101	242

TABLE 4.2. QUADTREE BUILDING STATISTICS FOR TOPOGRAPHY MAP

ELEVATION	NODES IN TREE	NODES CREATED	% USED IN TREE	GRAY NODES	WHITE NODES	BLACK NODES
0 - 100	6809	8161	83.4	1702	2577	2530
100 - 200	13853	14913	92.9	3463	5295	5095
200 - 300	11813	13381	88.3	2953	4713	4147
300 - 400	8845	10469	84.5	2211	3596	3038
400 - 500	7121	8745	81.4	1780	2917	2424
500 - 600	6005	7629	78.7	1501	2534	1970
600 - 700	5341	6973	76.6	1335	2140	1866
700 - 800	4725	6357	74.3	1181	1955	1589
800 - 900	3121	4753	65.7	780	1292	1049
900 - 1000	1277	2909	43.9	319	516	442
1000 - 1100	161	1793	9.0	40	88	33

TABLE 4.3. QUADTREE BUILDING STATISTICS FOR FLOODPLAIN MAP

AREA	NODES IN TREE	NODES CREATED	% USED IN TREE	GRAY NODES	WHITE NODES	BLACK NODES
left bank	4021	5473	73.5	1005	1491	1525
floodplain	6257	7645	81.8	1564	2485	2208
right bank	2885	4009	72.0	721	1133	1031

TABLE 4.4. LANDUSE CONNECTED COMPONENT RESULTS

CLASS	NUMBER OF NEIGHBORS SOUGHT	FINDNBR AVG COST	ROPES AVG COST	ARGS AVG COST	TIME IN SECS
acc	2812	3.55	1.40	3.08	1.8
acp	5492	3.58	1.40	2.81	3.6
ar	720	3.48	1.33	3.18	0.4
are	52	5.63	1.98	4.96	0.0
avf	8354	3.53	1.40	2.86	5.4
avv	9072	3.55	1.39	2.91	5.7
bbr	306	3.59	1.35	3.51	0.2
beq	194	3.82	1.31	3.64	0.1
bes	102	3.30	1.38	2.80	0.1
bt	1538	3.53	1.35	2.98	1.0
fo	3986	3.54	1.45	2.75	2.6
lr	882	3.71	1.25	3.36	0.5
r	4918	3.63	1.46	2.85	3.2
ucb	138	3.31	1.20	3.61	0.1
ucc	464	3.64	1.37	3.52	0.3
ucr	690	3.60	1.42	3.10	0.4
ucw	280	3.58	1.36	3.21	0.2
ues	658	3.81	1.38	3.38	0.4
uil	202	3.75	1.55	3.42	0.1
uis	650	3.53	1.39	3.19	0.4
uiw	162	3.84	1.35	3.62	0.1
unk	602	3.45	1.35	3.72	0.4
uoc	102	3.59	1.49	3.39	0.1
uog	268	3.66	1.40	2.81	0.2
uoo	242	3.22	1.35	3.55	0.1
uop	132	3.64	1.42	4.08	0.1
uov	146	3.42	1.29	3.14	0.1
urn	106	3.69	1.28	4.47	0.1
urs	6896	3.54	1.40	2.88	4.5
uus	162	3.55	1.35	3.67	0.1
uut	1846	3.62	1.38	3.33	1.0
vv	78	3.33	1.28	3.92	0.1
wo	278	3.97	1.38	3.49	0.2
ws	2966	3.70	1.28	3.15	1.9
wwp	202	3.62	1.38	4.52	0.2

TABLE 4.5. TOPOGRAPHY CONNECTED COMPONENT RESULTS

ELEVATION	NUMBER OF NEIGHBORS SOUGHT	FINDNBR AVG COST	ROPES AVG COST	ARGS AVG COST	TIME IN SECS
0 - 100	5060	3.48	1.41	2.69	3.7
100 - 200	10190	3.51	1.41	2.72	7.8
200 - 300	8294	3.53	1.41	2.85	5.8
300 - 400	6076	3.57	1.36	2.91	4.0
400 - 500	4848	3.62	1.36	2.94	3.0
500 - 600	3940	3.64	1.36	3.05	2.5
600 - 700	3732	3.62	1.36	2.86	2.4
700 - 800	3178	3.69	1.38	2.97	2.1
800 - 900	2098	3.57	1.37	2.98	1.3
900 - 1000	884	3.54	1.41	2.89	0.6
1000 - 1100	66	3.56	1.41	4.88	0.1

TABLE 4.6. FLOODPLAIN CONNECTED COMPONENT RESULTS

REGION	NUMBER OF NEIGHBORS SOUGHT	FINDNBR AVG COST	ROPES AVG COST	ARGS AVG COST	TIME IN SECS
left bank	3050	3.25	1.35	2.64	1.9
floodplain	4416	3.50	1.46	2.83	1.5
right bank	2062	3.62	1.66	2.80	3.1

5. Region analysis and manipulation

5.1. Region analysis

The functions described in this section are used to gather basic statistics about the regions encoded by the quadtree data structure. The simplest of these is NDCOUNT, which sets three global variables to indicate the number of gray nodes, white nodes, and black nodes in the given quadtree. This is achieved by performing a preorder traversal of the quadtree -- i.e., first it calculates its statistic for the current node (in this case incrementing a global counter) and then it recursively processes the node's four subtrees (if they exist). It should be noted that the functions described in this and in subsequent sections are currently implemented as stand alone programs that are invoked by executing a file under the UNIX operating system. This entails a fairly large amount of housekeeping details, such as decoding the arguments with which the file is invoked, opening various files, and initializing various devices. None of this will be discussed herein, nor will it appear in the algorithm descriptions. Among the other details that are thus swept under the rug, so to speak, would be the initializing of the variables queried by the functions BLACK, WHITE, and GRAY in order to determine (when processing a multicolored quadtree) which nodes should be considered of the indicated colors. Although we speak herein of functions computing values, we actually have programs that generate files and output listings containing function values. For instance, the implementation of NDCOUNT terminates by outputting the counts for the three types of nodes.

Closely related to NDCOUNT is a function called AREA. AREA takes as parameters a node and an indicated width for that node. AREA calculates the area and centroid of the region encoded by black nodes relative to this indicated size for the entire quadtree. This is achieved by a preorder traversal that works as follows. If the current node is a leaf, then its size is added to the global count in accordance to whether or not it is black. If the current node is not a leaf, then AREA processes each of its four subtrees using a width value adjusted to half of the value associated with the current node.

Next in order of complexity is HANDW, which takes as its parameters: a node, the x and y coordinates of its upper left corner, and its width. Its value is the coordinates of the upper left corner, the height, and the width of the smallest rectilinear rectangle (i.e., the smallest rectangle with sides parallel to the x and y axis) that encloses all the regions that are considered black. This is done by comparing the coordinates of each black node to the most extreme values found so far. Again we are dealing with a preorder traversal of the quadtree with the x, y, and width values being updated as one descends from a gray node to its children.

The last of the statistics gathering functions is PERIMETER. It also updates a global variable and calculates its desired value via a preorder traversal. Like AREA, it takes a node and its width as parameters. Unlike AREA, it uses the width value to calculate the length of a node's side instead of the node's area. The function PERIMETER returns as its value the sum of the lengths of the perimeters of all the black regions. Like AREA, it does this from the point of view, so to speak, of the black nodes. The side of a black node is part of the perimeter (and hence its length is to be counted) only if the neighbor on that side of the black node is a white node. The neighbor is located using FIND_NEIGHBOR.

Algorithm 5.1. NDCOUNT


```
/* A simple tree traversal to count the number of GRAY, WHITE and
   BLACK nodes. */

INTEGER numgray = 0;
INTEGER numwhite = 0;
INTEGER numblack = 0;

PROCEDURE ndcount(rt)
 node POINTER rt;
{
 INTEGER i;

 IF(gray(rt->ntype))
   {
    numgray = numgray + 1;
    FOR(i = NW,NE,SE,SW)
       ndcount(rt->sons[i]);
   }
 ELSE
   IF(black(rt->ntype))
     numblack = numblack + 1;
   ELSE
     numwhite = numwhite + 1;
}
```

```
/* Given a quadtree, compute the area and centroid of the black
   region of that tree. */

INTEGER area, xsum, ysum;


main(root, width)
/* To compute the area, simply sum up the number of pixels in each
   black node.
   To compute the centroid, for x-coord sum up all of the x-coord
   of black pixels and divide by area; for y-coord sum y-coords
   and divide by area. */
 node POINTER root;
 INTEGER width;
{
 area = xsum = ysum = 0;
 doarea(root,width,0,0);  /* compute area
                             (stored in global variable area) */
 xcent = xsum/area;      /* xcoord of centroid */
 ycent = ysum/area;      /* ycoord of centroid */
}

doarea(root, width, fx, fy)
/* This function does the work of computing the area and the
   centroid. For each black node, add the number of pixels to the
   global variable area; sum up the x-coordinates and add to the
   global variable xsum, and sum up the y-coordinates and add this
   to global variable ysum. */
node POINTER root;
INTEGER width, temp;
INTEGER fx, fy; /* Coords of the upper left pixel of the node. */
{
 if(gray(root))
   {     /* for each child, compute area */
    doarea(width/2,son(root,NW),fx,fy);
    doarea(width/2,son(root,NE),fx+width/2,fy);
    doarea(width/2,son(root,SE),fx+width/2,fy+width/2);
    doarea(width/2,son(root,SW),fx,fy+width/2);
   }
 else
   if(black(root))
     {
      /* Incorporate area of current node into
         running totals. */
      temp = width * width;
      xsum = xsum + (fx + width/2 - .5) * temp;
      ysum = ysum + (fy + width/2 - .5) * temp;
      area = area + temp;
     }
}
```

```
' Find the smallest enclosing rectangle for the black area of a
  quadtree.  Specify this rectangle by its upper left coordinates
  and its height and width. */

ITEGER leastx,leasty,highx,highy; /* The highest and lowest
                                     values yet found. */

iin(root,width)
' Given a quadtree, call handw to get the highest and lowest
  values of x and y coords.  The upper left corner is <least-x,
  least-y> and the width and height is the difference between the
  x's and y's respectively.  */
iode POINTER root;
INTEGER height,width;

leastx = leasty = width + 1;
highx = highy = 0;
handw(root,0,0,width);
height = highy - leasty;
width = highx - leastx;


indw(root,x,y,width)
' X and y are the coordinates of the upper left corner of the
  node width is the width of the node.  If the node is black,
  check if the extreme corners of the node are within the least
  and high bounds - if not, then change the bounds.  If the node
  is gray, check the sons. */
INTEGER x,y,width;
iode POINTER root;


if(gray(root))
  {  /* For each son, do handw. */
   handw(son(root,NW),x,y,width/2);
   handw(son(root,NE),x+width/2,y,width/2);
   handw(son(root,SE),x+width/2,y+width/2,width/2);
   handw(son(root,SW),x,y+width/2,width/2);
  }
else
  if(black(root))
   {
    if(x < leastx) leastx = x;
    if((x+width-1) > highx) highx = x + width - 1;
    if(y < leasty) leasty = y;
    if((y+width-1) > highy) highy = y + width - 1;
   }
```

```
/* Given a quadtree and its width, compute the length of the
   perimeter of the black areas.  This is done by traversing the
   tree and calling addperim for each black node.  Addperim looks
   at each of the neighbors of the black node. If that neighbor is
   white, then the length of the edge is added to the perimeter
   total.  If it is gray, then sons along the edge of the the black
   node are run with addperim. */

INTEGER perimlength = 0;

perim(root,width)
/* Traverse the tree calling addperim for black nodes. */
 node POINTER root;
 INTEGER width;

{
 IF(gray(root))
   FOR(i = NW to SW)
      perim(width/2,son(root,i));
 ELSE
   IF(black(root))
     {
      addperim(find_neighbor(root,N),width,SE,SW);
      addperim(find_neighbor(root,E),width,NW,SW);
      addperim(find_neighbor(root,S),width,NW,NE);
      addperim(find_neighbor(root,W),width,NE,SE);
     }
}

addperim(root,width,q1,q2)
/* Root is a neighbor of a black node.  If root is white, add width
   to the perimeter length, if it is gray; then perform addperim on
   the children which are adjacent to the original black node
   (quadrants q1 and q2). */
 node POINTER root;
 INTEGER width,q1,q2;

{
 IF(nil(root))   /* The black node was on the edge of the tree - no
                    neighbor exists. */
   perimlength = perimlength + width;
 ELSE
   IF(white(root))
     perimlength = perimlength + width;
 ELSE IF(gray(root))
     {
      addperim(son(root,q1),width,q1,q2);
      addperim(son(root,q2),width,q1,q2);
     }
}
```

5.2. Region manipulation

Of the basic operations described herein, the only one that does not produce a new quadtree is the PT2POLY function, which given a quadtree, a coordinate structure (x and y coordinates of the upper left corner and the width for the entire tree), and a (u,v) coordinate pair, returns the value (color) of the leaf node that represents the region that contains the coordinate pair. Unlike the statistics gathering programs that had to traverse the entire tree, the PT2POLY function only visits those nodes that lie on a direct path between the quadtree's root and the sought after leaf. This is done by determining the quadrant of the current node within which the coordinate pair lies and then recursing down into that subtree while updating the coordinate structure to reflect the new location.

One of the basic operations that take one quadtree as a parameter and return a new quadtree as the result is the WINDOW function. Besides its quadtree parameter, the WINDOW function also takes a specification of where the window should be placed-- i.e., the current width of the quadtree, the coordinates of the upper left corner of the window, and the width of the window. For the present, the window must be a square whose width is a power of 2. The new quadtree is constructed by recursively performing the following steps. Find the smallest subtree of the given quadtree that contains the window. If this subtree coincides with the window then return the subtree. If this subtree is a leaf then return a leaf of the same color. Otherwise, split the current window into quadrants and process each of these subwindows with respect to the current subtree. Upon returning from each recursive call, it is necessary to check if four leafs are brothers of the same color, and when this happens, replace the father by one of the four leafs. This process results in a quadtree that represents the windowed portion of the map encoded by the given quadtree.

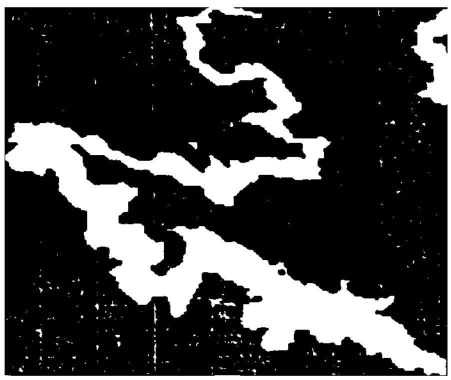
The two basic operations that take two quadtrees as parameters are the set-theoretic operations of INTERSECTION and UNION. Both of these functions work on binary quadtrees, taking two quadtrees as parameters and creating a resulting quadtree. In both cases, we assume that the input quadtrees are of the same width and have the same upper left coordinates. If this were not the case, the user could perform the WINDOW function to align the two quadtrees. Like the statistics gathering functions, these two operations perform preorder traversals of the quadtree parameters. However, now the traversals are performed in parallel; so that at any time during the processing, the algorithms keep track of the two nodes (one in each quadtree) which correspond to the same areas in the two encoded maps. The logic of these two operations is summarized below.

if current nodes are	function is INTERSECTION	function is UNION
both black	return black	return black
both white	return white	return white
both gray	recurse	recurse
one black	return other subtree	return black
one white	return white	return other subtree

In the above table, 'recurse' indicates that one needs to traverse each of the remaining subtrees, and 'return other subtree' indicates that the value of the function is a copy of the other subtree. Just as with windowing, when the recursion unwinds, one has to check to see if four brothers are identical leaves and merge them as indicated in the discussion of the WINDOW function. Examples of performing these basic set-theoretic operations are shown in Figures 5.1-5.3. Table 5.1 shows the area of the landuse polygons in Figure 5.3, using the naming conventions discussed in Section 5.3.

The set-theoretic operation of *complement* can be performed using the more general function QMASK. QMASK takes a quadtree and a range as parameters and returns a quadtree that has all the nodes with values within the range set to BLACK and all the nodes with values outside the range set to WHITE. Thus the tree that results from QMASK is always a binary-colored quadtree. The QMASK algorithm is implemented as a preorder traversal of the input quadtree that simultaneously constructs the output quadtree. In constructing the output quadtree, the QMASK algorithm merges nodes when necessary as indicated in the discussion of the WINDOW function.

The final quadtree manipulation function is QDISPLAY, which does double duty both as a quadtree manipulator (it truncates quadtrees) and as an output routine. The parameters of QDISPLAY are a quadtree, specifications of how the quadtree should be displayed (location, width, coloring algorithm, etc.), and the depth at which the quadtree should be truncated. The coloring algorithm is determined by two flags, COLOR and BLOCK. If COLOR is true, then the quadtree is displayed as is, each node's value being interpreted as a color. If COLOR is true and BLOCK is false, then the quadtree is output as a binary-colored quadtree, where the colors mapped to BLACK are defined by setting the range used by the primitive function BLACK. If BLOCK is true, then a special table of colors is used and the color of a node is determined by its depth and its binary-colored value. If COLOR is false, then one has the option of setting the maximum depth of a node that will be displayed. When a gray (internal non-leaf) node is to be displayed, the function examines the gray node's descendants and considers the node to be BLACK if the black nodes (when weighted according to their depth in the tree) exceed the white nodes (when similarly weighted). Thus a node is output as BLACK in the truncated tree, if it is BLACK or it is a gray node at the maximum depth and the average color of the region it subtends is more black than white. The algorithm is implemented as a preorder traversal of the input quadtree that outputs the nodes in the order they are visited. Figures 5.4-5.8 show the output of QDISPLAY when COLOR and BLOCKS are false and the polygon flood.center is considered BLACK. These figures give an idea of the initial gentle degradation of the image as the quadtree is truncated. Table 5.12, discussed in Section 5.3, shows that Figure 5.5 uses only two-thirds the number of nodes as Figure 5.4, with virtually no loss in the basic image shape. This shows that quadtree truncation is a useful image approximation technique.



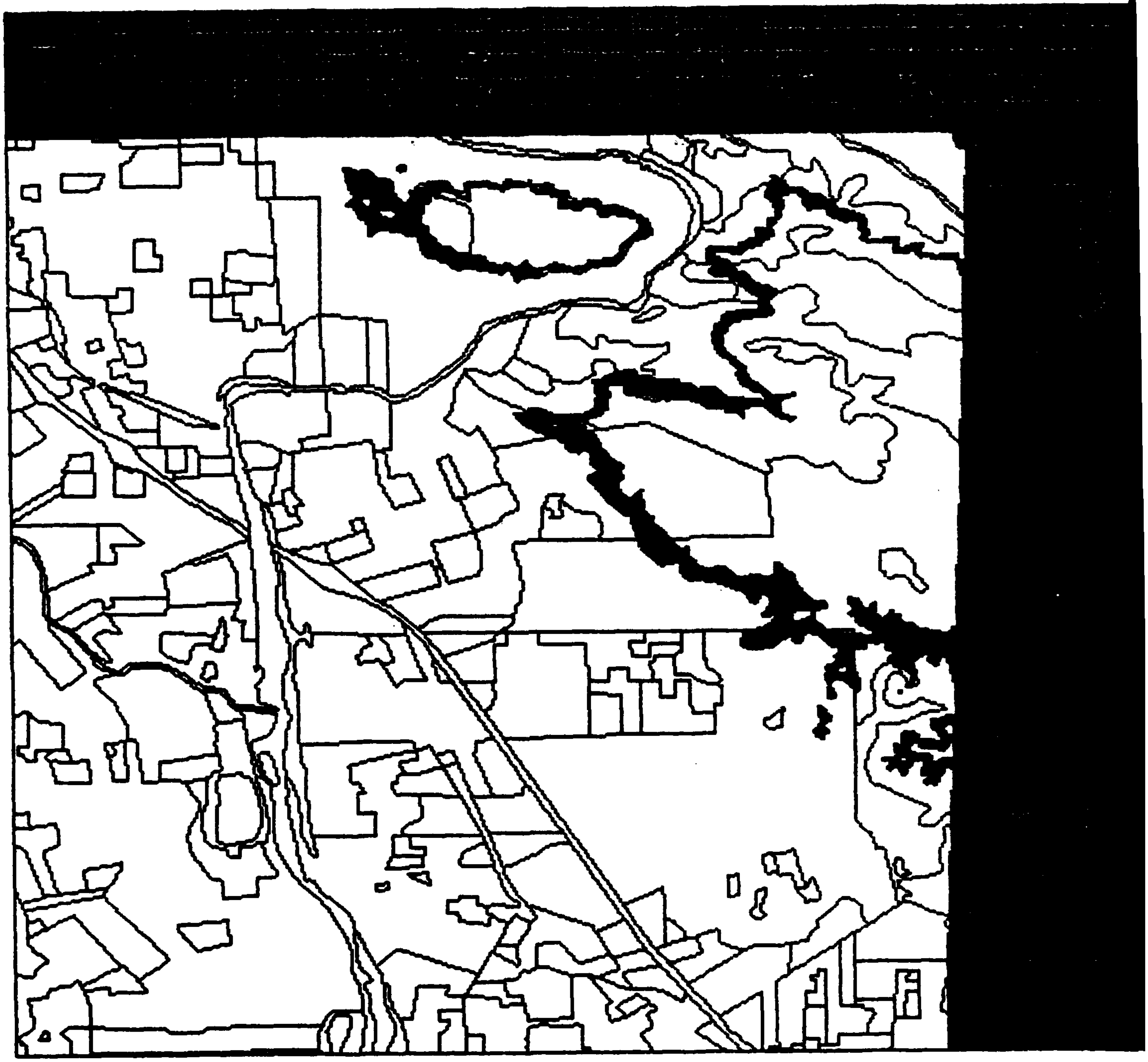


Figure 5.2. Result of executing INTERSECTION on the entire land-use map and the complement of the 5th elevation level (400-500 ft. elevation) of the topography map.

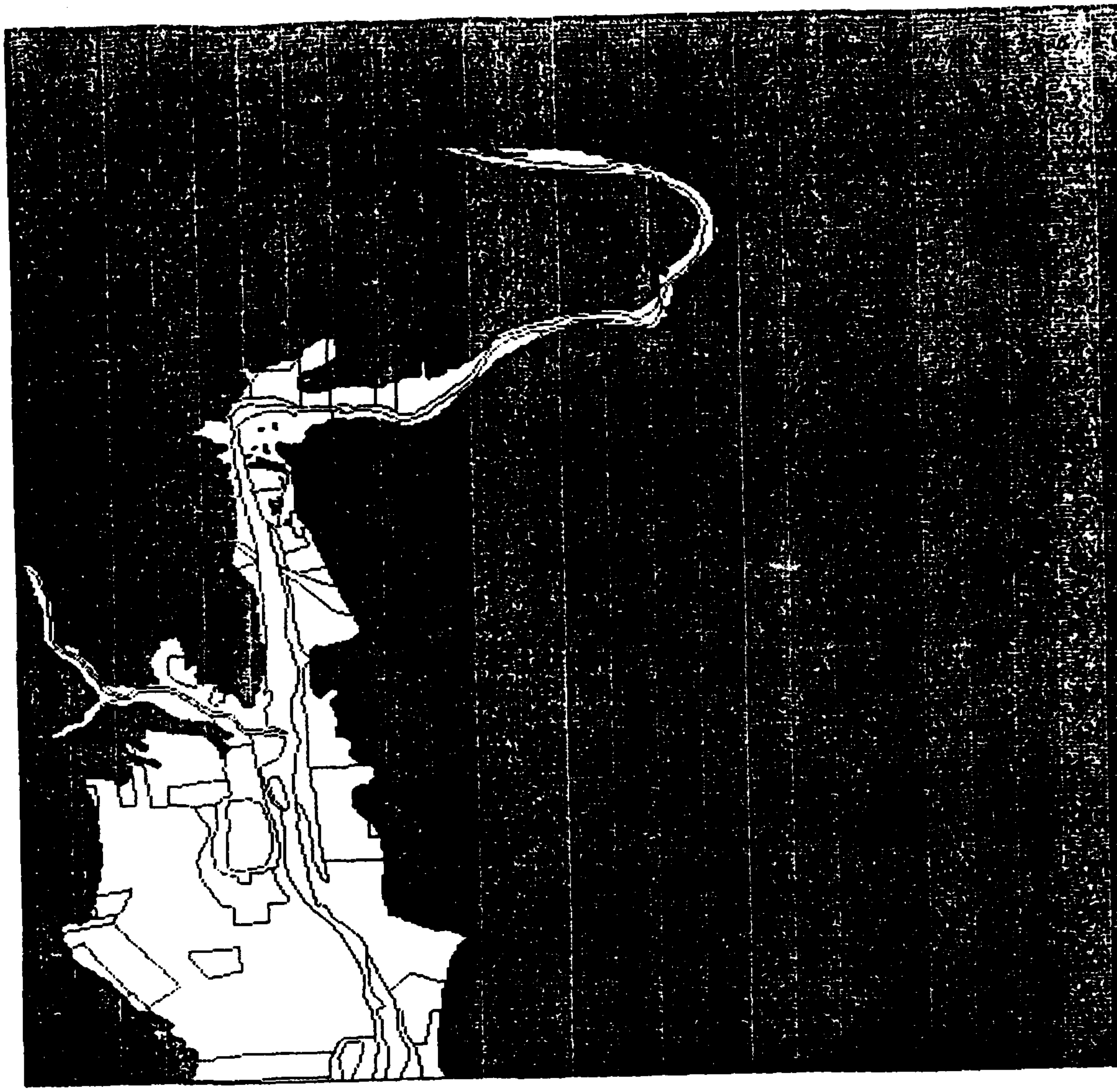


Figure 5.3. Result of executing INTERSECTION on the 1st elevation level (0-100 ft. elevation) of the topography map, the flood.center region of the floodplain map, and the entire land-use map.

Algorithm 5.9. QPASK

TABLE 5.1.
AREA RESULTS FOR LANDUSE
POLYGONS IN FIGURE 5.3
(1 of 2)

POLY-GON	AREA IN PIXELS	AREA IN ACRES
acc.9	1	0.14
acc.10	71	10.08
acc.12	948	134.62
acc.15	452	64.18
acp.1	28	3.98
acp.4	124	17.61
avf.4	209	29.68
avf.5	279	39.62
avt.6	423	60.07
avf.8	55	7.81
avf.9	81	11.50
avf.10	611	86.76
avf.11	538	76.40
avf.15	1033	146.69
avf.17	15	2.13
avf.18	72	10.22
avf.19	713	101.25
avf.21	750	106.50
avf.24	214	30.39
avf.25	659	93.58
avv.3	214	30.39
avv.6	7	0.99
avv.7	23	3.27
avv.16	7936	1126.91
avv.17	34	4.83
avv.18	255	36.21
avv.19	1880	266.96
avv.20	582	82.64
avv.23	129	18.32
avv.26	58	8.24
avv.28	217	30.81
bbr.1	71	10.08
bbr.2	361	51.26
beq.1	229	32.52
bes.1	114	16.19
bt.2	88	12.50
fo.1	293	41.61
fo.3	29	4.12
fo.4	8	1.14
lr.1	91	12.92
lr.2	63	8.95
lr.3	620	88.04
lr.4	129	18.32

AREA RESULTS FOR LANDUSE
POLYGONS IN FIGURE 5.3
(2 of 2)

POLYGON	AREA IN PIXELS	AREA IN ACRES
r.2	23	3.27
r.4	1	0.14
ucc.6	3	0.43
ues.1	548	77.82
ues.2	658	93.44
uis.5	101	14.34
uoo.3	75	10.65
uop.2	148	21.02
urh.1	20	2.84
urs.1	245	34.79
urs.4	722	102.52
urs.5	73	10.37
urs.6	107	15.19
urs.7	50	7.10
urs.8	123	17.47
urs.9	92	13.06
urs.10	34	4.83
urs.11	48	6.82
urs.18	32	4.54
uus.1	246	34.93
uus.2	3	0.43
uut.1	28	3.98
uut.3	196	27.83
vv.1	108	15.34
wo.1	535	75.97
wo.2	126	17.89
ws.1	3394	481.95

Figure 5.4. Result of executing QDISPLAY on flood.center of the flood-plain map using 10 levels.

Figure 5.5. Result of executing QDISPLAY on flood.center of the flood-plain map using 9 levels.

Figure 5.6. Result of executing QDISPLAY on flood.center of the flood-plain map using 6 levels.

Figure 5.7. Result of executing QDISPLAY on flood.center of the flood-plain map using 7 levels.

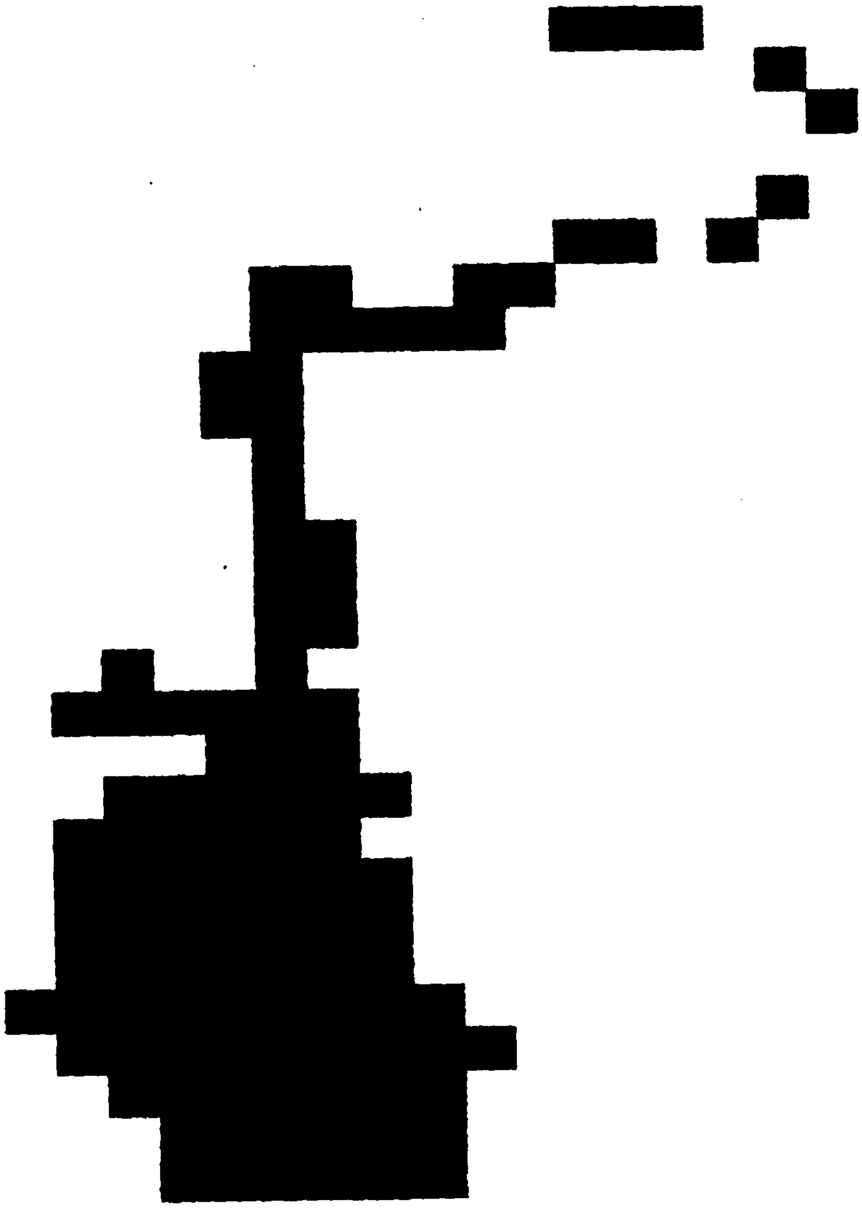


Figure 5.8. Result of executing QDISPLAY on flood.center of the flood-plain map using 6 levels.

```
/* Given a tree and the coordinates of a point, return the value of
   the leaf node containing that point.  It is assumed that in a
   database system the tree used for this operation would have a
   different node value for each polygon.  This would allow a
   simple lookup to determine the corresponding polygon once the
   node value in the tree has been found by this algorithm. */

INTEGER FUNCTION pt2poly(fx, fy, width, xcoord, ycoord, rt)
 node POINTER rt; /* The root of the tree. */
 INTEGER fx, fy;  /* This is the upper left coord of the tree. */
 INTEGER width;   /* The width of the tree. */
 INTEGER xcoord, ycoord; /* The given coord being searched for */
{
 IF(! gray(nodetype(rt)))
   RETURN(nodetype(rt));
 ELSE /* Gray tree - find which quadrant contains the coord. */
   IF(ycoord < (fy + width/2)) /* North half */
     IF(xcoord < (fx + width/2)) /* NW */
       RETURN(pt2poly(fx,fy,width/2,xcoord,ycoord,son(rt,NW));
     ELSE    /* NE */
       RETURN(pt2poly(fx+width/2,fy,width/2,xcoord,ycoord,
                              son(rt,NE));
   ELSE    /* South half */
     IF(xcoord >= (fx + width/2)) /* SE */
       RETURN(pt2poly(fx+width/2,fy+width/2,width/2,xcoord,ycoord,
                              son(rt,SE));
     ELSE  /* SW */
       RETURN(pt2poly(fx,fy+width/2,width/2,xcoord,ycoord,
}                             son(rt,SW));
```

```
/* Given a quadtree and its width, along with the specifications of
   a window, create a tree which is the section of the input tree
   specified by the window. */

main(root,width,wcol,wrow,wwidth)
 node POINTER root; /* Root is the input tree. */
 INTEGER width,wcol,wrow,wwidth;
/* Width is the width of the input tree, <wcol,wrow> is the coord
   of the upper left corner of the window, wwidth is the size of
   the window (must be a power of 2).      */
{
 node rnode;  /* Dummy node to start the answer tree. */
 node POINTER rptr;  /* Eventual root of the answer tree. */

 rptr = rnode;
 rnode.fathr = NIL;
 dowindow(root,width,0,0,rptr,NW,wwidth,wcol,wrow);
 rptr = son(rptr,NW);
 /* Rptr is now the root of the answer tree. */
}


dowindow(inrt,inwidth,incol,inrow,outfthr,whichson,outwidth,
          outcol,outrow)
/* From the input tree create a tree described by the given window
   which has father outfthr and is son whichson. */
 node POINTER inrt,outfthr; /* Inrt is the root of the current
    input tree. Outfthr is the father of the output tree. */
 INTEGER inwidth,incol,inrow,whichson,outwidth,outcol,outrow;
/* Inwidth, incol and inrow are the window described by the input
   tree; outwidth, outcol, and outrow are the window to be
   described by the tree being built and whichson indicates the
   sontype of the tree being built in relation to the whole
   output tree. */
{
 node POINTER nd;
 INTEGER i, goodquad;

/* First, cut the input tree down to the smallest subtree which
   contains the window desired - i.e., if the window is completely
   contained in one of the children of the input tree, make the
   the input tree that child (and continue for its children...) */
 goodquad = 1;
 while((goodquad <> NEG) AND (inrt == GRAY))
   {
    goodquad = NEG;
  /* For each quadrant, check if window in quadrant. */
    if(inrect(incol,inrow,inwidth/2,outcol,outrow,outwidth))
      goodquad = NW;
    if(inrect(incol+inwidth/2,inrow,inwidth/2,outcol,outrow,
                                   outwidth))
      goodquad = NE;
    if(inrect(incol+inwidth/2,inrow+inwidth/2,inwidth/2,outcol,
                                                outrow,outwidth))
      goodquad = SE;
    if(inrect(incol,inrow+inwidth/2,inwidth/2,outcol,outrow,
```

```
                                            outwidth))
      goodquad = SW;
    if(goodquad <> NEG)
      { /* Window in input quadrant -
           make input tree that quadrant */
       inrt = son(inrt,goodquad);
       inwidth = inwidth/2;
       if(goodquad == NE OR SE)
         incol = incol + width;
       if(goodquad == SE OR SW)
         inrow = inrow + width;
      }
   }
 if((incol == outcol) AND (inrow == outrow) AND
    (inwidth == outwidth))
    /* If the windows are the same, then the output tree is the
       same as the input tree, so copy the input tree. */
   copysub(inrt,outfthr,whichson);
 else
   if(nodetype(inrt) <> GRAY)
       /* If the input tree is a leaf node, then the window is
          also a leaf of the same color. */
     createnode(outfthr,whichson,nodetype(inrt));
 else
   { /* No single child of the input tree contains the window, and
        the input tree is not a leaf node.  Therefore, repeat
        dowindow on each quadrant of the window desired. To do
        this, first install a GRAY node in the output tree and
        then call dowindow for each quadrant. */
    nd = createnode(outfthr,whichson,inrt->ntype);
    dowindow(inrt,inwidth,incol,inrow,
             nd,NW,outwidth/2,outcol,outrow);
    dowindow(inrt,inwidth,incol,inrow,
             nd,NE,outwidth/2,outcol+outwidth/2,outrow);
    dowindow(inrt,inwidth,incol,inrow,
             nd,SE,outwidth/2,outcol+outwidth/2,outrow+outwidth/2);
    dowindow(inrt,inwidth,incol,inrow,
             nd,SW,outwidth/2,outcol,outrow+outwidth/2);
    if(All children of nd have the same nodetype)
      {
       nd->ntype = nodetype(son(nd,NW));
       Return all children of nd to avail list;
      }
   }
}

BOOLEAN FUNCTION inrect(bigcol,bigrow,bigwidth,litcol,litrow,
                                litwidth);
/* Return TRUE if and only if the second window is contained
   in the first window. */
 INTEGER bigcol,bigrow,bigwidth,litcol,litrow,litwidth;
{
 if((bigcol <= litcol) AND (bigrow <= litrow) AND
    (bigcol+bigwidth >= litcol+litwidth) AND
    (bigrow+bigwidth >= litrow+litwidth))
   return(TRUE);
```

```
 else
   return(FALSE);
}

copysub(root,fthr,whichson)
/* Create a copy of the subtree with root "root".  The created
   tree will be son "whichson" of the node "fthr" (part of the
   global answer tree). */
 node POINTER root,fthr;
 INTEGER whichson;
{
 node POINTER root;
 INTEGER i;

 nd = createnode(fthr,whichson,nodetype(root));
 for(i = NW,NE,SE,SW)
    copysub(son(root,i),nd,i);
}
```

```
/* Given two (possibly multi-colored) quadtrees, return a binary
   quadtree which is the intersection of the input trees. */

node FUNCTION inter(rtl, rt2, fthr, whichson)
/* Return the intersection of the trees whose roots are pointed to
   rtl and rt2. This is done by simultaneously traversing the
   input trees and performing inter on each quadrant. If either
   tree is WHITE, the intersection is WHITE. If either tree is
   BLACK, the intersection is the other tree.
   Fthr is the father of the resulting tree. This allows the
   current subtree to be inserted into the complete output tree.
   Whichson tells which son of the output tree the current tree
   is. The function is initially called with these values equal
   to NIL. */
 node POINTER rtl, rt2, fthr;
 int whichson;
{
 node POINTER nd;
 INTEGER i;

 IF (white(nodetype(rtl)) OR white(nodetype(rt2)))
   RETURN(createnode(fthr,whichson,WHITE);
 IF (black(nodetype(rtl))
   RETURN(copysub(rt2,fthr,whichson));
 IF (black(nodetype(rt2))
   RETURN(copysub(rtl,fthr,whichson));
 /* Both trees are GRAY - call inter for each quadrant. */
 nd = createnode(fthr,whichson,GRAY);
 FOR(i = NW,NE,SE,SW)
   inter(rtl->sons[i],rt2->sons[i],nd,i);
 IF(all children of nd are WHITE)
   {    /* Condense tree. */
    nd->ntype = WHITE;
    FOR(i = NW,NE,SE,SW)
      returntoavail(nd->sons[i]);
   }
 RETURN(nd);
}
```

```
/* Given two (possibly multi-colored) quadtrees, return a binary
   quadtree which is the union of the input trees. */

node FUNCTION union(rt1, rt2, fthr, whichson)
/* Return the union of the trees whose roots are pointed at by
   rt1 and rt2.  This is done by simultaneously traversing the
   input trees and performing union on each quadrant. If either
   tree is black, the union is BLACK.  If either tree is WHITE,
   the union is the other tree.  Fthr is the father of the
   resulting tree.  This allows the current subtree to be inserted
   into the complete output tree. Whichson indicates the sontype
   of the current tree relative to the output tree. Union is
   initially called with these values equal to NIL. */
 node POINTER rt1, rt2, fthr;
 int whichson;
{
 node POINTER nd;
 INTEGER i;

 IF (black(nodetype(rt1)) OR black(nodetype(rt2)))
   RETURN(createnode(fthr,whichson,BLACK));
 IF (white(nodetype(rt1))                     /* If tree1 is WHITE, */
   RETURN(copysub(rt2,fthr,whichson)); /* copy tree2. */
 IF (white(nodetype(rt2))                     /* If tree2 is WHITE, */
   RETURN(copysub(rt1,fthr,whichson)); /* copy tree1. */
 /* Both trees are GRAY and union must be applied to
    each quadrant. */
 nd = createnode(fthr,whichson,GRAY);
 FOR(i = NW,NE,SE,SW)
   union(rt1->sons[i],rt2->sons[i],nd,i);
 IF(All children of nd are BLACK)
   {    /* Condense tree. */
    nd->ntype = BLACK;
    FOR(i = NW,NE,SE,SW)
      returntoavail(nd->sons[i]);
   }
 RETURN(nd);
}
```

```
/* For a quadtree with root 'root', for each leaf node, if its
   value is between the parameters high and low (inclusive), then
   set node value to BLACK, else set to WHITE. When necessary,
   merge children of a gray node together.   */

INTEGER low, high;

qmask(rt)
 node POINTER rt;
{
 INTEGER i;

 IF(GRAY(rt))
   {
    FOR(i = NW,NE,SE,SW)
      qmask(rt->sons[i]);
    IF(all children of rt are the same leaf color)
      { /* condense tree */
       rt->ntype = nodetype(rt->sons[NW]);
       FOR(i = NW,NE,SE,SW)
          returntoavail(rt->sons[i]);
      }
   }
 ELSE
   IF((rt->ntype >= low) AND (rt->ntype <= high))
     rt->ntype = BLACK;
   ELSE
     rt->ntype = WHITE;
}


main(root,low,high)
 node POINTER root;        /* input tree */
 INTEGER low, high;     /* input parameters - values in this range
                           are BLACK. */
{
 qmask(root);
 RETURN(root);
}
```

```
/* Display a quadtree on the grinnell. This is done by traversing
   the tree. While the traversal is being done, the program keeps
   track of the size and position of the current node, and displays
   that node on the grinnell. There are two boolean options -
   color and block. If color is TRUE, then nodes will be displayed
   by their color, if FALSE then all non-WHITE nodes are displayed
   as BLACK. If block is true, then the displayed color depends
   on the depth of the node in the tree - not its value. At most
   one of these options may be true. Additionally, if color is
   FALSE, the user may wish to display nodes only down to a certain
   depth by adjusting maxdepth. In this case, the smallest node
   size displayed can be changed. For example, with a 512 X 512
   picture, there are 10 levels and the smallest node is one pixel
   wide. If the user sets maxdepth as 9, then the smallest node is
   2 X 2 pixels. For any gray nodes at level 9, the function DK
   adds up the number of black pixels of its children and if more
   than half are black, the gray node is displayed as black, other
   wise it is displayed as white.
   'next' is a function which returns the value of the next node
   of the preorder traversal of the input tree stored in tree. */


 DATA FILE tree;  /* preorder traversal of the input tree. */
 INTEGER color, block;
 INTEGER maxlevel;
 INTEGER larr[10]; /* This array holds grinnell color values to be
                        used with option block.  */



 colorof(val,currlevel)
/* Determine the actual color value to be displayed on the grinnell.
   for the node with value val. This is determined by the options,
   the value of the node and possibly the level of the node in the
   tree. */
 INTEGER val, currlevel /* currlevel is the level in the tree of
                           the current node. */
{
 INTEGER v;

 IF(block)
   {    /* color determined by level */
    v = larr[currlevel - 1];
    IF(black(val))
      v = v*256;
/* white nodes will be displayed as some tint of red, black nodes
   will be displayed as some tint of blue. Multiplying by 256
   shifts a red value to a blue one when displayed. */
    return(v);
   }
 ELSE
   IF(black(val))
     IF(color)
       RETURN(val);
     ELSE
       RETURN(BLACK);
```

```
   ELSE
     RETURN(WHITE);
}


bk(width)
  /* Return the number of pixels of the current node and its
     children which are black. */
INTEGER width;
{
 INTEGER val;

 val = next(); /* causing this function to have a side effect */
 IF(black(val))
   RETURN(width * width);
 IF(white(val))
   RETURN(0);
/* gray node - calculate black pixels in children */
 RETURN(bk(width/2) + bk(width/2) + bk(width/2) + bk(width/2));
}


display(fcol,frow,width,currlevel)
 /* Display the next node of the preorder traversal on the grinnell.
    This node has its upper left corner at fcol, frow and has width
    width. It is at level currlevel in the tree. */
 INTEGER fcol, frow, width, currlevel;
{
 INTEGER col, total, val;

 val = next();
 IF(NOT gray(val))
   {
    col = colorof(val,currlevel);
    <write to grinnell a square at fcol,frow of size width with
        color col.>  /* this is a command and not a comment */
   }
 ELSE
   IF(currlevel == maxlevel)
     IF(block)
       {
        col = colorof(val,currlevel);
        <write to grinnell a square at fcol,frow of size width and
                color col.>
       }
     ELSE
       {
        total = bk(width/2)+bk(width/2)+bk(width/2)+bk(width/2);
        IF((2 * total)) > (width * width))
          val = 1;
        ELSE
          val = 0;
        col = colorof(val,currlevel);
        <write to grinnell a square at fcol,frow of size width and
                color col.>
       }
```

```
  ELSE
    {  /* Display children of a gray node */
     display(fcol,(frow+width/2),(width/2),(currlevel+1));
     display((fcol+width/2),(frow+width/2),(width/2),(currlevel+1));
     display((fcol+width/2),frow,(width/2),(currlevel+1));
     display((fcol,frow,(width/2),(currlevel+1));
    }
}


main(fcol,frow,color,block,maxlevel,width);
 INTEGER fcol,frow,width;
{
 /* Angle brackets enclose commands written in English and are not
     just comments.   */
 <Fill larr with grinnell-dependent values used with option block. >
 display(fcol,frow,width,1);
}
```

5.3. Tabulation of results

In the tables presented in this section, the basic unit of manipulation is the connected component. The names of these basic units are created by suffixing a digit to the land-use class (or contour) to which the unit belongs. These digits can be dereferenced by referring to the figures in Section 3. For example, in Table 5.2, the first polygon name we encounter is acc.1. Looking at Figure 3.4, one sees the 19 components of the class ACC. The component labeled 1 in that figure is the polygon refered to by the name acc.1. The units of the flood-plain map are so few that they are given names of their own, i.e., left and right bank instead of bank.1 and bank.2. Note that there are no tables showing the execution times for the PT2POLY function. This is because all times were less than a tenth of a second and hence were beyond the range of the system timing algorithm.

The first group of tables (Tables 5.2-4) are the AREA RESULTS tables. They are organized according to which map the polygons (i.e., simply-connected components) belong. They summarize the results of two programs: NDCOUNT (which counts the number of black nodes, i.e., those belonging to the polygon) and AREA (which calculates the area in pixels and centroid (first moment) of the polygon). The execution times immediately follow the results of the same algorithm. The times for NDCOUNT indicate the cost of visiting every node in the tree exactly once. Hence the time is relatively constant for each map because so little processing is done at each node. This value also gives an indication of the reliability of the system timing routine used. Substantially more calculation is performed by AREA with more variation with respect to the amount of time spent at a black node vs. a gray or white node. The conversion from area in pixels to area in acres was calculated based on .142 acres per pixel. The value is given in hundredths of an acre, although the pixel size is about one seventh of an acre. The coordinates used for the centroid are based on the upper left-hand corner being (0,0) and the number of pixels in both directions range from 0 to 512. The same coordinate system is used in the other tables.

The REGION PROPERTY RESULTS (Tables 5.5-7) show the cost of two statistics gathering programs: PERIMETER and HANDW. The perimeter is measured in pixel widths. The enclosing rectangle calculated by HANDW is given by the coordinates of its upper left-hand corner and its width and height. HANDW is another algorithm that treats each node equally and hence produces little variation in its timings within a given map. This is quite different from PERIMETER, which performs four FIND_NEIGHBOR operations for each black node; hence the variations in the cost of PERIMETER.

The data for the WINDOW program is presented in the

WINDOW RESULTS (Tables 5.8-10). The window used is the smallest square whose width is a power of two, that encloses the smallest bounding rectangle of the polygon (calculated by HANDW), and sharing the same upper left-hand corner with that rectangle. The relation between the times and the input is complicated, as it is effected by both the size of the window and the greatest common denominator of the tree size and the two coordinates of the upper left-hand corner. The smaller the greatest common denominator of these three numbers, the greater the possible fracturing of large nodes in the input tree.

The next table, INTERSECTION STATISTICS (Table 5.11), is the only table showing a binary relation, that of INTERSECTION. Three large regions (the center of the flood plain and the two lowest contours) are chosen to be intersected with the land-use classes because they are most likely to yield interesting results. Since the center of the flood plain is not equivalent to a contour class, it is also intersected with each of the contour classes. Note that the cost of INTERSECTION can be less than the cost of doing a NDCOUNT on both trees because a large white node in one tree can make it unnecessary to process a large subtree in the other tree. As well as the cost of performing the INTERSECTION operation (measured in seconds), the table also gives the area and number of nodes in the result. Note that a UNION table is not shown because UNION behaves in the same manner as INTERSECTION on the logical complement of the inputs (i.e., switch the black and white node colors). Note that the INTERSECTION algorithm is greatly simplified by the digitization process's alignment of the maps so that the pixel at (0,0) corresponds to the same ground truth in each map.

The final table, QUADTREE TRUNCATION STATISTICS (Table 5.12), shows the amount of compression one can obtain by truncating the quadtree maps. The usability of the truncated quadtrees is discussed at the end of Section 5.2 and shown in Figures 5.4-5.8. Under each map's name there are two columns. The first column shows the number of nodes in the quadtree that is formed by truncating the full (depth 10) quadtree to the depth indicated in the far left column. The second column shows the percentage of nodes in the full (depth 10) quadtree that would not be needed for the truncated quadtree.

TABLE 5.2. LANDUSE AREA RESULTS
(1 OF 5)

POLY-GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
acc.1	45	1.3	201	28.54	4.6	143.7	2.7
acc.2	25	1.3	154	21.87	27.0	169.5	2.3
acc.3	61	1.3	202	28.68	40.6	197.6	2.2
acc.4	107	1.3	593	84.21	76.6	218.5	2.3
acc.5	13	1.3	28	3.98	139.7	190.4	2.5
acc.6	92	1.3	356	50.55	157.0	218.4	2.1
acc.7	72	1.3	345	48.99	180.6	206.1	2.1
acc.8	33	1.3	270	38.34	183.6	239.5	2.1
acc.9	12	1.3	18	2.56	38.2	299.1	2.1
acc.10	59	1.3	146	20.73	18.6	372.2	2.1
acc.11	85	1.3	256	36.35	157.0	256.5	2.1
acc.12	211	1.3	1174	166.71	131.7	317.7	2.2
acc.13	76	1.3	528	74.98	202.6	332.2	2.1
acc.14	85	1.3	292	41.46	256/5	382.7	2.1
acc.15	124	1.3	682	96.84	162.9	412.9	2.1
acc.16	58	1.4	214	30.39	244.3	393.9	2.1
acc.17	70	1.3	229	32.52	258.2	414.5	2.1
acc.18	120	1.3	465	66.03	329.0	390.6	2.1
acc.19	56	1.4	188	26.70	349.3	432.2	2.1
acp.1	73	1.5	187	26.55	33.4	242.6	2.1
acp.2	575	1.5	6035	857.97	238.2	182.3	2.2
acp.3	99	1.5	339	48.14	9.5	349.3	2.1
acp.4	48	1.5	132	18.74	34.3	361.5	2.1
acp.5	58	1.6	244	34.65	3.3	394.9	2.1
acp.7	877	1.5	14806	2102.45	288.9	321.0	2.2
acp.8	42	1.5	93	13.21	157.4	348.0	2.1
acp.9	120	1.5	666	94.57	15.8	432.0	2.1
acp.10	296	1.6	1412	200.50	219.0	434.6	2.1
acp.11	66	1.5	285	40.47	295.3	434.7	2.1
acp.12	179	1.5	977	138.73	323.7	425.6	2.1
acp.13	231	1.3	1356	192.55	366.9	426.2	2.1
ar.1	69	1.4	204	29.97	209.7	117.6	2.1
ar.2	60	1.3	198	28.12	359.7	213.0	2.1
ar.3	57	1.4	135	19.17	164.4	306.4	2.3
ar.4	114	1.3	453	64.33	172.9	333.9	2.4
ar.5	60	1.3	207	29.39	176.1	439.4	2.2
are.1	26	1.3	152	21.58	323.1	436.0	2.1
avf.1	28	1.3	121	17.18	29.1	21.3	2.1
avf.2	44	1.3	134	19.03	16.4	118.3	2.1
avf.3	77	1.4	326	46.29	100.6	82.3	2.1
avf.4	103	1.4	628	89.18	135.1	104.9	2.1
avf.5	90	1.3	285	40.47	105.6	111.4	2.1
avf.6	687	1.4	4914	697.79	157.5	166.1	2.2
avf.7	28	1.3	46	6.53	259.1	87.6	2.1
avf.8	565	1.3	3823	542.87	39.5	185.5	2.2
avf.9	151	1.3	901	127.94	9.8	235.6	2.1

LANDUSE AREA RESULTS
(2 OF 5)

POLY-GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
avf.10	365	1.3	1715	243.53	71.1	250.0	2.2
avf.11	154	1.3	961	136.46	129.0	231.1	2.1
avf.12	79	1.3	325	46.15	187.2	257.3	2.1
avf.13	116	1.3	704	99.97	242.3	253.5	2.1
avf.14	315	1.3	1590	225.78	347.3	22.3	2.1
avf.15	176	1.3	1619	229.90	59.6	283.4	2.1
avf.16	40	1.3	235	33.37	11.1	293.5	2.1
avf.17	38	1.3	152	21.58	22.9	310.1	2.1
avf.18	46	1.3	106	15.05	41.9	305.6	2.2
avf.19	167	1.3	713	101.25	85.5	351.9	2.1
avf.20	13	1.3	46	6.53	2.7	333.9	2.1
avf.21	182	1.3	851	120.84	36.6	388.8	2.1
avf.22	20	1.3	35	4.97	172.6	262.9	2.1
avf.23	40	1.4	76	10.79	157.2	355.2	2.1
avf.24	55	1.3	214	30.39	76.9	390.3	2.1
avf.25	136	1.3	1156	164.15	74.8	442.6	2.2
avf.26	177	1.3	771	109.48	213.6	397.0	2.2
avf.27	26	1.3	40	5.68	290.2	367.7	2.1
avf.28	17	1.3	44	6.25	305.3	357.2	2.2
avf.29	45	1.3	138	19.60	268.6	399.1	2.1
avf.30	144	1.3	984	139.73	295.9	416.5	2.2
avf.31	45	1.3	123	17.47	359.0	420.9	2.1
avv.1	29	1.3	110	15.62	5.3	78.9	2.1
avv.2	29	1.3	68	9.66	91.3	73.4	2.2
avv.3	100	1.3	373	53.97	120.5	106.5	2.2
avv.4	24	1.3	87	12.35	27.3	159.4	2.3
avv.5	54	1.3	105	14.91	9.8	177.9	2.1
avv.6	109	1.3	703	99.83	19.9	201.5	2.1
avv.7	100	1.4	328	46.58	40.6	231.2	2.1
avv.8	29	1.3	56	7.95	83.1	244.9	2.1
avv.9	107	1.3	449	63.76	189.3	177.7	2.1
avv.10	66	1.4	243	34.51	155.4	170.5	2.1
avv.11	96	1.4	339	48.14	225.6	191.2	2.2
avv.12	47	1.3	200	28.40	179.8	247.4	2.2
avv.13	35	1.3	140	19.88	259.6	248.4	2.2
avv.14	55	1.3	163	23.15	286.9	4.5	2.2
avv.15	19	1.3	40	5.68	275.9	11.8	2.3
avv.16	1076	1.4	11390	1617.38	62.8	362.2	2.3
avv.17	20	1.4	38	5.40	78.7	261.4	2.1
avv.18	85	1.4	295	41.89	91.5	283.6	2.1
avv.19	740	1.3	4580	650.36	157.6	363.5	2.2
avv.20	402	1.3	3294	467.75	149.2	277.7	2.5
avv.21	229	1.4	2158	306.44	218.0	272.0	2.5
avv.22	69	1.5	309	43.88	205.4	347.2	2.2
avv.23	175	1.5	1036	147.11	17.6	405.0	2.4
avv.24	4	1.4	28	3.98	1.4	445.9	2.1
avv.25	25	1.3	127	18.03	24.8	442.0	2.3

LANDUSE AREA RESULTS
(3 OF 5)

POLY-GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
avv.26	16	1.3	58	8.24	108.0	448.5	2.3
avv.27	52	1.3	151	21.44	192.8	419.7	2.5
avv.28	122	1.3	500	71.00	166.4	432.5	2.3
avv.29	24	1.3	63	8.95	184.9	444.9	2.1
avv.30	81	1.3	318	45.16	224.9	404.6	2.1
avv.31	68	1.3	302	42.88	217.6	442.9	2.1
avv.32	73	1.4	358	50.84	271.9	270.1	2.1
avv.33	49	1.3	124	17.61	313.4	359.0	2.1
avv.34	33	1.3	72	10.22	265.7	390.5	2.1
avv.36	15	1.3	36	5.11	321.3	446.3	2.2
avv.37	30	1.3	207	29.39	336.6	410.2	2.2
avv.38	23	1.3	101	14.34	324.3	420.0	2.1
avv.39	11	1.3	23	3.27	363.0	413.0	2.2
avv.40	22	1.3	73	10.37	362.6	427.2	2.2
avv.41	97	1.4	331	47.00	384.3	424.2	2.2
bbr.1	32	1.3	71	10.08	102.4	308.5	2.2
bbr.2	121	1.3	361	51.26	145.0	422.9	2.2
beq.1	97	1.3	229	32.52	131.8	439.3	2.1
bes.1	51	1.3	147	20.87	101.4	169.4	2.1
bt.1	30	1.3	132	18.74	54.1	1.5	2.1
bt.2	100	1.3	268	38.06	148.4	106.8	2.1
bt.3	589	1.3	2881	409.10	360.2	351.0	2.2
bt.4	50	1.3	122	17.32	283.5	442.2	2.1
fo.1	75	1.3	585	83.07	226.8	3.6	2.1
fo.2	292	1.3	2605	369.91	215.8	44.9	2.1
fo.3	1145	1.3	10010	1421.42	304.5	123.0	2.3
fo.4	465	1.4	3729	529.52	338.3	50.7	2.2
fo.5	16	1.4	25	3.55	164.0	362.8	2.1
lr.1	56	1.4	107	15.19	96.3	120.4	2.2
lr.2	33	1.4	63	8.95	103.9	190.6	2.1
lr.3	289	1.4	649	92.16	112.7	281.3	2.1
lr.4	63	1.4	129	18.32	152.7	437.0	2.1
r.1	181	1.4	889	126.24	198.7	138.6	2.2
r.2	227	1.4	1178	167.28	233.0	108.1	2.2
r.3	1428	1.4	17277	2453.33	330.2	232.2	2.3
r.4	232	1.3	1003	142.43	290.2	17.1	2.2
r.5	391	1.4	2800	397.60	338.6	79.2	2.2
ucb.1	39	1.4	153	21.73	77.9	7.9	2.1
ucb.2	30	1.3	96	13.63	49.3	105.4	2.1
ucc.1	52	1.4	430	61.06	58.4	14.3	2.1
ucc.2	48	1.3	141	20.02	55.2	57.5	2.1
ucc.3	23	1.3	101	14.34	44.8	79.9	2.2
ucc.4	13	1.4	64	9.09	77.5	73.5	2.3
ucc.5	35	1.3	119	16.90	86.4	83.7	2.3
ucc.6	61	1.3	163	23.15	183.3	379.1	2.2

LANDUSE AREA RESULTS
(4 OF 5)

POLY-GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
ucr.1	14	1.3	35	4.97	6.8	1.8	2.2
ucr.2	21	1.3	45	6.39	8.9	25.5	2.2
ucr.3	277	1.3	1342	190.56	34.2	106.5	2.2
ucr.4	33	1.3	96	13.63	66.4	157.4	2.1
ucw.1	67	1.3	139	19.74	17.6	95.5	2.1
ucw.2	73	1.3	166	23.57	27.5	127.9	2.1
ues.1	117	1.4	960	136.32	109.1	137.8	2.1
ues.2	212	1.3	668	94.86	87.4	316.1	2.1
uil.1	50	1.3	239	33.94	50.0	134.8	2.1
uil.2	51	1.3	183	26.99	135.9	200.8	2.2
uis.1	5	1.3	20	2.84	1.4	1.9	2.2
uis.2	5	1.3	8	1.14	0.3	40.3	2.1
uis.3	96	1.3	285	40.47	10.2	58.7	2.1
uis.4	25	1.4	157	22.29	47.3	168.9	2.1
uis.5	47	1.4	146	20.73	109.4	167.1	2.1
uis.6	34	1.3	88	12.50	72.0	160.0	2.1
uis.7	85	1.3	268	38.06	166.2	317.1	2.2
uis.8	28	1.4	70	9.94	248.9	426.1	2.3
uiw.1	29	1.3	56	7.95	219.2	181.2	2.2
uiw.2	52	1.4	130	18.46	146.4	252.1	2.2
unk.1	128	1.3	1343	190.71	374.5	15.3	2.3
unk.2	50	1.3	176	24.99	387.8	255.9	2.2
unk.3	2	1.3	8	1.14	392.5	187.5	2.2
unk.4	5	1.4	5	0.71	392.1	193.3	2.2
unk.5	21	1.3	33	4.69	388.7	203.0	2.1
unk.6	47	1.4	113	16.05	390.5	230.6	2.1
unk.7	4	1.3	10	1.42	392.5	340.0	2.1
unk.8	15	1.3	21	2.98	390.3	403.8	2.1
unk.9	19	1.3	28	3.98	390.9	425.3	2.1
unk.10	10	1.3	16	2.27	390.6	447.1	2.1
uoc.1	51	1.3	288	40.90	97.2	64.0	2.1
uog.1	134	1.3	1115	158.33	116.2	59.1	2.1
uoo.1	39	1.3	273	38.77	7.3	91.4	2.2
uoo.2	47	1.3	125	17.75	58.2	157.5	2.3
uoo.3	35	1.3	92	13.06	121.7	204.3	2.3
uop.1	11	1.3	29	4.12	33.9	101.4	2.2
uop.2	55	1.3	184	26.13	100.0	156.0	2.1
uov.1	35	1.3	119	16.90	95.6	5.2	2.1
uov.2	38	1.4	119	16.90	103.9	20.9	2.1
urh.1	33	1.3	126	17.89	82.4	160.9	2.1
urn.2	20	1.4	41	5.82	174.5	304.6	2.2
urs.1	873	1.3	9018	1280.56	57.8	62.8	2.3
urs.2	75	1.3	246	34.93	15.1	108.5	2.2
urs.3	37	1.3	148	21.02	3.3	123.6	2.1
urs.4	776	1.3	10427	1480.63	179.6	49.6	2.2
urs.5	130	1.3	730	103.66	139.0	139.5	2.2

LANDUSE AREA RESULTS
(5 OF 5)

POLY- GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
urs.6	85	1.3	397	56.37	82.8	175.1	2.2
urs.7	32	1.4	62	8.80	112.4	178.6	2.1
urs.8	53	1.3	194	27.55	68.1	248.2	2.2
urs.9	145	1.3	403	57.23	92.1	219.6	2.1
urs.10	105	1.3	279	39.62	277.4	25.4	2.1
urs.11	119	1.3	485	68.87	274.6	74.3	2.1
urs.12	19	1.3	49	6.96	17.6	302.3	2.2
urs.13	37	1.3	211	29.96	3.9	361.8	2.1
urs.14	37	1.4	112	15.90	169.0	272.6	2.2
urs.15	327	1.3	1650	234.30	211.9	362.8	2.3
urs.16	66	1.3	261	37.06	238.8	283.9	2.2
urs.17	25	1.3	43	6.11	7.6	366.0	2.4
urs.18	67	1.3	199	28.26	40.3	413.3	2.4
urs.19	33	1.3	63	8.95	198.5	400.0	2.3
urs.20	91	1.3	538	76.40	212.9	420.2	2.4
urs.21	246	1.3	1014	143.99	259.1	428.4	2.4
urs.22	7	1.3	10	1.42	306.4	447.9	2.3
urs.23	9	1.4	15	2.13	315.8	447.8	2.2
urs.24	54	1.3	198	28.12	341.5	445.7	2.1
uus.1	69	1.3	246	34.93	70.6	308.9	2.1
uus.2	12	1.3	15	2.13	147.6	367.2	2.1
uut.1	246	1.4	600	85.20	40.0	148.0	2.1
uut.2	91	1.3	166	23.57	61.4	132.4	2.1
uut.3	566	1.3	1162	165.00	198.3	308.2	2.1
vv.1	39	1.3	108	15.34	108.8	185.9	2.1
wo.1	94	1.3	535	75.97	89.8	329.3	2.1
wo.2	45	1.3	126	17.89	129.5	438.7	2.1
ws.1	1483	1.3	3409	484.08	131.8	213.2	2.2
wwp.1	8	1.3	8	1.14	360.0	251.3	2.1
wwp.2	8	1.3	11	1.56	10.8	439.3	2.1
wwp.3	24	1.3	45	6.39	307.5	287.0	2.1
wwp.4	24	1.3	51	7.24	300.6	377.2	2.1
wwp.5	22	1.3	61	8.66	321.5	370.7	2.1
wwp.6	15	1.3	30	4.26	358.3	359.7	2.1

TABLE 5.3. TOPOGRAPHY AREA RESULTS (1 OF 2)

POLY-GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
1.1	2530	1.1	58318	8281.16	96.4	270.1	2.2
2.1	1268	1.1	16202	2300.68	86.9	61.4	2.0
2.2	18	1.2	51	7.24	196.6	1.5	1.8
2.3	3146	1.2	35351	5019.84	242.0	312.6	2.2
2.4	8	1.2	8	1.14	85.0	150.0	1.8
2.5	5	1.2	5	0.71	93.3	144.9	1.8
2.6	35	1.2	68	9.66	101.1	149.9	1.8
2.7	2	1.2	5	0.71	98.7	132.3	1.8
2.8	4	1.2	7	0.99	104.4	131.1	1.8
2.9	4	1.1	4	0.57	95.5	141.0	1.8
2.10	5	1.1	11	1.56	100.7	142.0	1.8
2.11	7	1.1	7	0.99	106.2	157.8	1.8
2.12	21	1.1	33	4.69	103.8	169.4	1.8
2.13	3	1.1	6	0.85	108.3	177.8	1.8
2.14	179	1.1	1718	243.96	355.7	18.4	1.9
2.15	6	1.1	12	1.70	3.8	285.3	1.9
2.16	6	1.1	12	1.70	0.6	296.8	2.0
2.17	7	1.1	10	1.42	0.5	304.2	1.9
2.18	49	1.1	214	30.39	3.1	361.7	1.9
2.19	3	1.1	6	0.85	188.8	320.3	1.8
2.20	35	1.1	89	12.64	192.4	384.0	1.8
2.21	284	1.2	2453	348.33	28.3	424.9	1.8
3.1	907	1.1	3400	482.80	166.7	42.5	2.0
3.2	20	1.2	53	7.53	111.1	64.7	1.8
3.3	3	1.1	3	0.43	118.8	60.8	1.9
3.4	2677	1.1	13111	1861.76	294.3	263.7	2.2
3.5	68	1.1	182	25.84	201.7	207.4	1.8
3.6	39	1.1	102	14.48	228.7	257.4	1.9
3.7	241	1.1	841	119.42	342.6	27.3	2.0
3.8	66	1.1	243	34.51	376.7	11.7	1.8
3.9	11	1.1	14	1.99	227.5	351.1	1.8
3.10	5	1.1	8	1.14	246.3	350.3	1.8
3.11	4	1.1	4	0.57	211.0	359.0	1.8
3.12	5	1.1	5	0.71	217.3	354.3	1.8
3.13	4	1.1	4	0.57	227.0	357.5	1.8
3.14	14	1.1	20	2.84	254.1	355.5	1.8
3.15	55	1.1	313	44.45	4.1	425.5	1.8
3.16	7	1.1	7	0.99	16.5	409.5	1.8
3.17	21	1.1	51	7.24	19.9	424.5	1.8
4.1	5	1.1	8	1.14	106.8	0.1	1.9
4.2	6	1.1	9	1.28	112.8	11.2	1.8
4.3	736	1.1	2308	327.74	175.4	36.9	2.0
4.4	2165	1.1	7466	1060.17	300.1	189.3	2.2
4.5	6	1.1	12	1.70	279.8	0.3	1.9
4.6	44	1.2	134	19.03	382.2	7.7	1.9
4.7	8	1.2	8	1.14	335.0	306.0	1.8
4.8	9	1.2	15	2.13	347.4	330.2	1.8

TOPOGRAPHY AREA RESULTS
(2 OF 2)

POLY-GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
4.9	8	1.2	8	1.14	378.5	338.0	1.8
4.10	5	1.4	11	1.56	378.3	349.1	1.9
4.11	36	1.2	83	11.79	372.8	364.9	1.8
4.12	4	1.1	4	0.57	367.0	372.0	1.9
4.13	4	1.1	4	0.57	374.0	377.0	1.9
5.1	9	1.1	12	1.70	159.3	14.9	1.8
5.2	556	1.1	1669	237.00	184.9	40.4	1.9
5.3	1634	1.1	5222	741.52	299.0	168.2	2.0
5.4	14	1.1	35	4.97	386.1	3.0	1.8
5.5	36	1.1	72	10.22	327.3	288.1	1.9
5.6	4	1.1	4	0.57	359.0	273.0	1.9
5.7	171	1.1	453	64.33	368.9	302.6	1.9
6.1	8	1.1	11	1.57	144.6	31.0	1.8
6.2	395	1.1	803	114.03	205.2	43.8	1.9
6.3	9	1.1	18	2.56	153.4	37.1	1.9
6.4	6	1.1	6	0.85	220.0	140.8	1.9
6.5	1458	1.1	5031	714.40	305.4	155.2	2.2
6.6	24	1.2	33	4.69	383.8	104.5	1.8
6.7	3	1.1	3	0.43	384.5	115.5	1.9
6.8	13	1.1	28	3.98	319.6	235.6	1.8
6.9	4	1.2	7	0.99	335.9	262.1	1.8
6.10	12	1.2	12	1.70	378.8	291.4	1.9
6.11	37	1.1	64	9.09	372.6	302.7	1.9
6.12	1	1.3	4	0.57	360.5	306.5	1.8
7.1	334	1.1	703	99.83	210.4	44.4	1.9
7.2	1435	1.1	5059	718.38	314.7	152.5	1.9
7.3	90	1.1	297	42.17	380.7	109.4	1.9
7.4	6	1.1	6	0.85	302.8	142.2	1.9
7.5	1	1.1	1	0.14	383.5	139.5	1.8
8.1	231	1.1	459	65.18	213.4	45.4	1.9
8.2	8	1.1	14	1.99	312.0	58.6	1.9
8.3	1140	1.2	3849	546.56	330.5	135.8	2.0
8.4	2	1.2	2	0.28	339.5	51.0	1.9
8.5	204	1.1	645	91.59	375.7	119.1	1.9
8.6	4	1.1	4	0.57	330.0	191.0	1.8
9.1	127	1.1	310	44.02	210.8	46.0	1.9
9.2	32	1.1	62	8.80	314.8	102.5	1.9
9.3	74	1.1	200	28.40	338.4	75.5	1.8
9.4	775	1.1	2473	351.17	358.1	135.2	1.9
9.5	5	1.1	8	1.14	381.9	195.9	1.9
9.6	36	1.1	135	19.17	378.5	208.3	1.9
10.1	35	1.1	116	16.47	203.1	44.7	1.9
10.2	35	1.1	59	8.38	373.8	80.9	1.8
10.3	23	1.1	35	4.97	353.8	98.0	1.8
10.4	349	1.1	1420	201.64	356.7	154.4	1.9
11.1	27	1.1	57	8.09	352.2	116.2	1.8
11.2	6	1.1	6	0.85	362.5	173.0	1.9

TABLE 5.4. FLOODPLAIN AREA RESULTS

POLY-GON	NUMB OF NODE	TIME IN SECS	AREA IN PIXELS	AREA IN ACRES	CENTROID X	CENTROID Y	TIME IN SECS
right	1031	0.2	104270	14806.34	277.1	241.8	0.5
left	1525	0.2	46003	6532.43	82.1	141.6	0.6
center	2208	0.2	29727	4221.24	108.9	292.2	0.7

TABLE 5.5. LANDUSE REGION PROPERTY RESULTS
(1 OF 5)

POLY-GON	PER-IM-ETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
acc.1	72	3.9	0	133	14	22	2.1
acc.2	50	3.7	21	164	13	12	2.1
acc.3	78	4.2	35	190	18	19	2.1
acc.4	114	4.0	61	205	31	25	2.3
acc.5	24	3.8	137	188	6	6	2.3
acc.6	106	3.8	139	209	34	19	2.2
acc.7	88	4.4	170	195	20	24	2.2
acc.8	94	3.7	164	236	39	8	2.2
acc.9	20	3.7	37	297	4	6	2.2
acc.10	88	3.8	3	368	34	10	2.2
acc.11	92	3.8	149	245	18	24	2.3
acc.12	212	3.8	116	297	47	46	2.3
acc.13	120	3.7	185	319	37	23	2.1
acc.14	92	3.7	246	370	17	29	2.1
acc.15	162	3.7	146	399	41	34	2.1
acc.16	76	3.7	235	385	20	18	2.1
acc.17	86	3.9	246	407	25	18	2.1
acc.18	136	3.7	313	375	33	35	2.1
acc.19	72	3.7	340	425	16	17	2.1
acp.1	102	3.8	24	233	19	24	2.1
acp.2	616	4.2	149	142	159	99	2.1
acp.3	118	3.7	0	336	18	35	2.1
acp.4	56	3.9	28	356	15	13	2.1
acp.5	104	3.7	0	372	13	39	2.1
acp.6	98	3.7	186	296	21	28	2.1
acp.7	976	3.9	199	244	159	156	2.1
acp.8	46	3.8	153	343	13	10	2.1
acp.9	160	3.7	0	413	30	37	2.1
acp.10	324	3.7	181	412	78	38	2.1
acp.11	90	3.7	285	422	17	28	2.1
acp.12	264	3.7	308	397	40	53	2.1
acp.13	282	3.7	340	404	53	46	2.1
ar.1	78	3.7	199	109	21	18	2.1
ar.2	82	3.8	352	205	19	21	2.1
ar.3	62	3.8	156	299	17	14	2.1
ar.4	106	3.9	157	323	32	20	2.1
ar.5	72	3.7	171	429	11	21	2.1
are.1	56	3.8	319	428	9	19	2.1
avf.1	50	3.8	22	18	17	8	2.1
avf.2	66	3.7	8	111	17	16	2.1
avf.3	94	3.7	91	71	19	24	2.1
avf.4	112	3.7	126	88	19	36	2.1
avf.5	84	3.8	95	101	20	22	2.1
avf.6	786	4.1	107	89	104	130	2.1
avf.7	40	4.0	253	85	13	7	2.1
avf.8	688	4.0	0	129	95	108	2.1
avf.9	204	3.9	0	198	29	72	2.1

LANDUSE REGION PROPERTY RESULTS
(2 OF 5)

POLY-GON	PER-IM-ETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
avf.10	432	4.0	33	229	72	52	2.1
avf.11	190	3.9	108	206	51	39	2.1
avf.12	100	3.8	174	244	26	24	2.1
avf.13	202	3.7	213	244	58	24	2.1
avf.14	386	3.8	293	0	101	57	2.1
avf.15	214	3.8	33	260	49	56	2.1
avf.16	80	3.7	2	283	18	22	2.1
avf.17	56	3.7	15	305	16	11	2.1
avf.18	56	3.7	38	296	8	19	2.1
avf.19	216	3.8	70	315	36	58	2.1
avf.20	30	3.8	0	330	7	8	2.1
avf.21	208	3.7	4	372	58	40	2.1
avf.22	34	3.7	170	258	7	10	2.1
avf.23	40	3.7	151	353	14	6	2.1
avf.24	72	3.7	66	384	21	15	2.1
avf.25	220	3.7	30	436	93	14	2.1
avf.26	222	3.7	186	378	54	37	2.1
avf.27	30	3.7	289	363	4	11	2.1
avf.28	32	3.7	303	353	6	10	2.1
avf.29	60	3.7	260	394	18	12	2.1
avf.30	212	3.7	272	397	42	53	2.1
avf.31	68	3.7	356	411	10	23	2.1
avv.1	50	3.7	0	73	14	11	2.1
avv.2	38	3.7	87	70	11	8	2.1
avv.3	94	3.7	115	88	12	35	2.1
avv.4	42	3.7	21	155	12	9	2.1
avv.5	76	3.7	0	170	21	17	2.1
avv.6	134	3.7	0	190	35	31	2.1
avv.7	132	3.8	18	226	45	15	2.1
avv.8	44	3.8	80	237	7	15	2.1
avv.9	148	3.7	173	157	32	37	2.1
avv.10	98	3.7	147	155	19	28	2.1
avv.11	100	3.8	214	180	25	19	2.1
avv.12	80	4.0	167	244	29	11	2.1
avv.13	66	3.7	250	244	21	12	2.1
avv.14	90	3.7	272	0	28	12	2.1
avv.15	38	3.7	273	6	6	13	2.1
avv.16	1356	3.9	0	238	140	201	2.1
avv.17	30	3.7	76	258	6	9	2.1
avv.18	112	3.7	77	272	29	26	2.1
avv.19	784	3.8	110	302	99	109	2.1
avv.20	444	3.8	113	240	81	84	2.1
avv.21	264	3.7	191	244	63	52	2.1
avv.22	96	3.7	192	342	32	16	2.1
avv.23	226	3.7	0	382	42	58	2.1
avv.24	24	4.3	0	442	4	8	2.1

LANDUSE REGION PROPERTY RESULTS
(3 OF 5)

POLY-GON	PER-IM-ETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
avv.25	52	3.9	18	436	12	14	2.1
avv.26	62	4.2	94	448	29	2	2.1
avv.27	70	4.2	182	415	23	12	2.1
avv.28	128	4.2	155	417	26	33	2.1
avv.29	32	4.1	162	441	7	9	2.1
avv.30	108	4.1	211	396	33	17	2.1
avv.31	84	3.9	206	436	26	14	2.1
avv.32	130	3.8	258	256	29	28	2.1
avv.33	64	3.7	307	351	13	19	2.1
avv.34	40	3.8	263	385	8	12	2.1
avv.35	114	3.9	265	390	29	26	2.1
avv.36	24	3.7	319	444	6	6	2.1
avv.37	84	3.6	334	392	6	36	2.1
avv.38	56	3.8	320	410	8	18	2.1
avv.39	20	3.9	361	411	5	5	2.1
avv.40	38	3.7	360	421	6	13	2.1
avv.41	104	3.7	375	408	18	31	2.1
bbr.1	48	3.7	99	302	8	16	2.1
bbr.2	160	3.7	137	390	16	60	2.1
beq.1	130	3.7	122	429	19	21	2.1
bes.1	64	3.8	95	162	13	19	2.1
bt.1	80	3.9	35	0	35	5	2.1
bt.2	88	3.8	145	90	8	36	2.1
bt.3	670	4.1	340	259	51	147	2.1
bt.4	66	4.0	275	435	18	15	2.1
fo.1	190	3.9	160	0	81	12	2.1
fo.2	302	3.8	166	23	94	47	2.1
fo.3	1306	3.9	184	75	210	107	2.1
fo.4	540	3.8	273	13	121	76	2.1
fo.5	28	3.7	160	361	8	6	2.1
lr.1	84	3.7	84	118	29	11	2.1
lr.2	54	3.8	101	181	7	20	2.1
lr.3	378	3.9	105	204	17	149	2.1
lr.4	92	3.8	149	416	9	34	2.1
r.1	204	3.9	176	117	61	37	2.1
r.2	298	3.9	205	84	62	47	2.1
r.3	1724	4.1	198	101	196	301	2.1
r.4	306	3.9	260	0	59	49	2.1
r.5	458	4.0	273	46	121	63	2.1
ucb.1	52	3.9	70	4	17	9	2.1
ucb.2	50	3.8	44	99	11	13	2.1
ucc.1	88	3.8	46	4	24	20	2.1
ucc.2	52	4.2	51	50	11	15	2.1
ucc.3	46	4.2	39	74	11	12	2.1
ucc.4	32	3.9	74	70	8	8	2.1
ucc.5	50	3.7	82	77	12	13	2.1
ucc.6	64	3.8	175	373	18	14	2.1

LANDUSE REGION PROPERTY RESULTS
(4 OF 5)

POLYGON	PERIMETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
ucr.1	24	3.7	4	0	7	5	2.1
ucr.2	28	3.7	6	23	7	7	2.1
ucr.3	376	3.7	14	65	46	87	2.1
ucr.4	40	3.7	63	152	8	12	2.1
ucw.1	92	3.7	11	81	13	28	2.1
ucw.2	96	3.7	19	117	17	26	2.1
ues.1	148	3.7	89	125	40	31	2.1
ues.2	304	3.7	76	286	27	67	2.1
uil.1	86	3.7	39	124	21	20	2.1
uil.2	68	3.7	127	194	19	15	2.1
uis.1	18	3.7	0	0	4	5	2.1
uis.2	12	3.7	0	39	2	4	2.1
uis.3	116	3.7	2	44	14	38	2.1
uis.4	54	3.7	42	162	12	14	2.1
uis.5	68	3.7	105	155	9	25	2.1
uis.6	50	3.7	66	152	10	15	2.1
uis.7	82	3.7	153	310	27	14	2.1
uis.8	46	3.7	243	422	13	10	2.1
uiw.1	38	3.7	216	177	7	11	2.1
uiw.2	64	3.7	139	245	15	14	2.1
unk.1	226	3.7	335	0	59	54	2.1
unk.2	80	3.7	378	248	16	21	2.1
unk.3	12	3.7	392	186	2	4	2.1
unk.4	10	3.7	392	193	2	3	2.1
unk.5	34	3.7	385	200	9	8	2.1
unk.6	76	3.7	386	220	8	26	2.1
unk.7	16	3.7	392	338	2	6	2.1
unk.8	22	3.7	388	402	5	6	2.1
unk.9	28	3.7	390	421	3	11	2.1
unk.10	18	3.7	389	445	4	5	2.1
uoc.1	76	3.7	87	56	21	17	2.1
uog.1	174	3.7	105	30	21	66	2.1
uoo.1	74	3.7	0	83	17	20	2.1
uoo.2	64	3.7	50	152	16	16	2.1
uoo.3	54	3.7	112	200	17	10	2.1
uop.1	26	3.7	31	99	7	6	2.1
uop.2	66	3.7	92	150	18	14	2.1
uov.1	48	3.7	90	0	11	13	2.1
uov.2	48	3.7	101	13	7	17	2.1
urh.1	46	3.7	76	157	14	9	2.1
urh.2	30	3.7	171	302	8	7	2.1
urs.1	1120	3.8	0	0	116	157	2.1
urs.2	116	3.7	0	100	34	21	2.1
urs.3	68	3.7	0	111	11	23	2.1
urs.4	924	3.8	101	0	177	128	2.1
urs.5	148	3.7	117	124	37	31	2.1

LANDUSE REGION PROPERTY RESULTS
(5 OF 5)

POLY-GON	PER-IM-ETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
urs.6	116	3.7	65	166	31	27	2.1
urs.7	40	3.7	110	173	7	13	2.1
urs.8	74	3.7	60	243	22	15	2.1
urs.9	170	3.7	72	198	27	49	2.1
urs.10	136	3.7	257	8	32	36	2.1
urs.11	132	3.7	259	58	36	27	2.1
urs.12	32	3.7	13	300	10	6	2.1
urs.13	78	3.7	0	347	12	27	2.1
urs.14	54	3.7	182	268	13	14	2.1
urs.15	408	3.8	173	307	73	83	2.1
urs.16	68	3.7	233	274	13	21	2.1
urs.17	32	3.7	5	383	8	8	2.1
urs.18	82	3.7	27	406	26	14	2.1
urs.19	40	3.7	193	397	11	9	2.1
urs.20	110	3.7	197	408	28	26	2.1
urs.21	312	3.7	232	400	54	50	2.1
urs.22	14	3.7	305	447	4	3	2.1
urs.23	16	3.7	314	447	5	3	2.1
urs.24	76	3.7	325	442	30	8	2.1
uus.1	76	3.7	59	303	25	13	2.1
uus.2	18	3.7	146	366	5	4	2.1
uut.1	378	3.7	0	104	96	93	2.1
uut.2	140	3.7	36	121	48	22	2.1
uut.3	886	3.8	105	201	194	249	2.1
vv.1	60	3.7	101	180	16	13	2.1
wo.1	108	3.7	82	312	18	35	2.1
wo.2	58	3.7	125	431	10	19	2.1
ws.1	2170	3.9	0	0	281	450	2.1
wwp.1	14	3.7	359	251	4	3	2.1
wwp.2	14	3.7	10	438	3	4	2.1
wwp.3	36	3.7	304	282	8	10	2.1
wwp.4	46	3.7	298	370	6	15	2.1
wwp.5	40	3.7	317	365	9	11	2.1
wwp.6	30	3.7	356	356	7	8	2.1

TABLE 5.6. TOPOGRAPHY REGION PROPERTY RESULTS (1 OF 2)

POLY-GON	PER-IM-ETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
1.1	3100	3.7	0	0	283	450	1.9
2.1	1608	3.4	0	0	276	143	1.9
2.2	50	3.2	187	0	21	4	1.8
2.3	3646	3.8	106	0	272	450	1.9
2.4	12	3.2	85	149	2	4	1.9
2.5	10	3.2	93	145	3	2	1.8
2.6	48	3.2	94	146	15	9	1.9
2.7	10	3.2	98	132	3	2	1.8
2.8	12	3.2	104	130	2	4	1.8
2.9	10	3.2	95	141	3	2	1.9
2.10	16	3.2	100	140	3	5	1.8
2.11	16	3.2	106	156	2	6	1.8
2.12	28	3.2	102	166	5	9	1.8
2.13	10	3.2	108	177	2	3	1.8
2.14	290	3.2	304	0	84	52	1.9
2.15	14	3.2	3	284	3	4	1.8
2.16	16	3.2	0	295	3	5	1.8
2.17	14	3.2	0	303	3	4	1.8
2.18	86	3.2	0	343	9	34	1.8
2.19	10	3.3	188	320	3	2	1.9
2.20	42	3.2	187	360	12	9	1.9
2.21	354	3.3	0	380	65	70	1.8
3.1	1244	3.4	89	0	181	93	1.9
3.2	36	3.2	106	62	11	7	1.8
3.3	8	3.2	119	61	2	2	1.9
3.4	3162	3.7	186	0	194	406	1.9
3.5	108	3.2	196	190	14	38	1.8
3.6	46	3.2	223	253	13	10	1.8
3.7	334	3.3	295	0	93	57	1.9
3.8	102	3.2	366	0	23	27	1.8
3.9	18	3.2	226	350	5	4	1.8
3.10	12	3.2	245	350	4	2	1.8
3.11	8	3.2	211	359	2	2	1.9
3.12	12	3.2	217	354	3	3	1.9
3.13	10	3.2	227	357	2	3	1.8
3.14	20	3.2	252	354	5	5	1.8
3.15	114	3.2	0	410	13	40	1.9
3.16	12	3.2	16	409	3	3	1.8
3.17	32	3.2	16	422	9	7	1.8
4.1	14	3.2	105	0	5	2	1.8
4.2	14	3.2	112	10	3	4	1.8
4.3	1014	3.4	119	0	148	77	1.9
4.4	2732	3.6	195	0	193	335	1.9
4.5	18	3.2	277	0	7	2	1.8
4.6	62	3.2	376	0	13	17	1.9
4.7	12	3.2	335	305	2	4	1.8
4.8	18	3.2	346	329	5	4	1.9

TOPOGRAPHY REGION PROPERTY RESULTS
(2 OF 2)

POLYGON	PERIMETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
4.9	18	3.2	379	335	1	8	1.9
4.10	16	3.2	378	347	2	6	1.9
4.11	56	3.2	364	360	15	10	1.8
4.12	8	3.2	367	372	2	2	1.8
4.13	8	3.2	374	377	2	2	1.8
5.1	16	3.2	158	14	4	4	1.8
5.2	770	3.3	136	15	126	57	1.9
5.3	2024	3.5	204	20	183	257	1.9
5.4	26	3.2	384	0	5	8	1.8
5.5	56	3.2	324	280	8	17	1.8
5.6	8	3.2	359	273	2	2	1.8
5.7	244	3.2	352	285	30	35	1.8
6.1	14	3.2	144	30	3	4	1.8
6.2	554	3.2	168	25	88	40	1.9
6.3	20	3.2	151	36	6	4	1.9
6.4	12	3.2	219	141	4	2	1.8
6.5	1780	3.4	230	29	157	217	1.9
6.6	32	3.2	380	102	7	9	1.9
6.7	8	3.2	385	115	1	3	1.8
6.8	24	3.2	317	234	7	5	1.9
6.9	12	3.2	335	261	3	3	1.8
6.10	16	3.2	378	290	3	5	1.8
6.11	60	3.2	364	299	17	9	1.8
6.12	8	3.2	360	306	2	2	1.9
7.1	470	3.3	174	27	78	36	1.9
7.2	1696	3.4	244	41	143	191	1.9
7.3	130	3.2	373	92	14	37	1.8
7.4	12	3.2	302	142	3	3	1.8
7.5	4	3.2	384	140	1	1	1.9
8.1	348	3.2	183	30	61	31	1.9
8.2	18	3.2	310	58	6	3	1.8
8.3	1400	3.4	265	53	122	172	1.8
8.4	6	3.2	340	51	1	2	1.9
8.5	266	3.3	365	88	22	79	1.8
8.6	8	3.2	330	191	2	2	1.8
9.1	198	3.3	192	34	43	22	1.8
9.2	44	3.2	309	99	12	10	1.8
9.3	92	3.2	328	67	20	22	1.8
9.4	982	3.4	325	70	62	127	1.8
9.5	12	3.2	381	195	3	3	1.8
9.6	56	3.2	372	202	12	16	1.8
10.1	58	3.2	195	40	19	10	1.8
10.2	46	3.2	367	78	15	7	1.8
10.3	30	3.2	350	95	7	8	1.8
10.4	388	3.3	334	106	42	86	1.9
11.1	40	3.2	347	113	11	9	1.8
11.2	10	3.2	362	173	3	2	1.9

TABLE 5.7. FLOODPLAIN REGION PROPERTY RESULTS

POLY-GON	PER-IM-ETER	TIME IN SECS	ENCLOSING RECTANGLE X	Y	WID	HGT	TIME IN SECS
right	1776	0.9	105	0	295	450	0.4
left	2270	0.9	0	0	274	450	0.4
center	2642	1.0	3	0	280	450	0.4

TABLE 5.8. LANDUSE WINDOW RESULTS
(1 of 5)

POLY-GON	WINDOW FX	FY	WIDTH	TIME SECS
acc.1	0	133	32	0.8
acc.2	21	164	16	0.2
acc.3	35	190	32	0.8
acc.4	61	205	32	1.1
acc.5	137	188	8	0.1
acc.6	139	209	64	5.0
acc.7	170	195	32	0.8
acc.8	164	236	64	1.0
acc.9	37	297	8	0.1
acc.10	3	368	64	3.2
acc.11	149	245	32	1.3
acc.12	116	297	64	3.6
acc.13	165	319	64	3.9
acc.14	246	370	32	0.6
acc.15	146	399	64	3.8
acc.16	235	385	32	1.4
acc.17	246	407	32	1.2
acc.18	313	375	64	4.4
acc.19	340	425	32	1.0
acp.1	24	233	32	1.2
acp.2	149	142	256	42.4
acp.3	0	336	64	0.7
acp.4	26	356	16	0.1
acp.5	0	372	64	1.1
acp.6	166	296	32	0.5
acp.7	199	244	256	34.0
acp.8	153	343	16	0.4
acp.9	0	413	64	2.8
acp.10	181	384	128	10.5
acp.11	285	422	32	1.2
acp.12	308	397	64	3.7
acp.13	340	404	64	1.0
ar.1	199	109	32	1.2
ar.2	352	205	32	0.7
ar.3	156	299	32	1.3
ar.4	157	323	32	1.3
ar.5	171	429	32	1.1
are.1	319	428	32	0.8
avf.1	22	18	32	0.3
avf.2	8	111	32	1.4
avf.3	91	71	32	1.1
avf.4	126	88	64	1.8
avf.5	95	101	32	1.2
avf.6	107	89	256	55.1
avf.7	253	85	16	0.3
avf.8	0	129	128	15.2
avf.9	0	198	128	7.8

LANDUSE WINDOW RESULTS
(2 of 5)

POLY-GON	WINDOW FX	FY	WIDTH	TIME SECS
avf.10	33	229	128	17.1
avf.11	108	206	64	2.2
avf.12	174	244	32	0.6
avf.13	213	244	64	3.1
avf.14	293	0	128	10.5
avf.15	33	260	64	3.5
avf.16	2	283	32	0.9
avf.17	15	305	16	0.4
avf.18	38	296	32	0.5
avf.19	70	315	64	4.3
avf.20	0	330	8	0.1
avf.21	4	372	64	1.3
avf.22	170	258	16	0.2
avf.23	151	353	16	0.4
avf.24	66	384	32	0.3
avf.25	30	384	128	3.8
avf.26	186	378	64	2.5
avf.27	289	363	16	0.3
avf.28	303	353	16	0.4
avf.29	260	394	32	0.7
avf.30	272	397	64	3.9
avf.31	356	411	32	1.0
avv.1	0	73	16	0.3
avv.2	87	70	16	0.3
avv.3	115	88	64	3.4
avv.4	21	155	16	0.4
avv.5	0	170	32	0.5
avv.6	0	190	64	2.0
avv.7	18	226	64	2.2
avv.8	80	237	16	0.3
avv.9	173	157	64	3.9
avv.10	147	155	32	1.1
avv.11	214	180	32	0.5
avv.12	167	244	32	1.1
avv.13	250	244	32	0.5
avv.14	272	0	32	0.4
avv.15	273	6	16	0.4
avv.16	0	238	256	25.8
avv.17	76	258	16	0.1
avv.18	77	272	32	1.2
avv.19	110	302	128	8.1
avv.20	113	240	128	13.2
avv.21	191	244	64	2.9
avv.22	192	342	32	0.4
avv.23	0	382	64	2.0
avv.24	0	442	8	0.0
avv.25	18	436	16	0.1

LANDUSE WINDOW RESULTS
(3 of 5)

POLY-GON	WINDOW FX	FY	WIDTH	TIME SECS
avv.25	18	436	16	0.1
avv.26	94	448	32	0.2
avv.27	182	415	32	0.9
avv.28	155	417	64	3.4
avv.29	182	441	16	0.3
avv.30	211	396	64	4.2
avv.31	206	436	32	0.4
avv.32	258	256	32	0.4
avv.33	307	351	32	1.1
avv.34	263	385	16	0.3
avv.35	265	390	32	1.1
avv.36	319	444	8	0.1
avv.37	334	392	64	2.2
avv.38	320	410	32	0.5
avv.39	361	411	8	0.1
avv.40	360	421	16	0.3
avv.41	375	406	32	0.9
bbr.1	99	302	16	0.3
bbr.2	137	390	64	4.0
beq.1	122	429	32	1.2
bes.1	95	162	32	1.2
bt.1	35	0	64	2.2
bt.2	145	90	64	3.8
bt.3	256	256	256	5.1
bt.4	275	435	32	1.1
fo.1	180	0	128	4.6
fo.2	166	23	128	12.4
fo.3	184	75	256	34.5
fo.4	273	13	128	14.5
fo.5	160	361	8	0.1
lr.1	84	118	32	0.6
lr.2	101	181	32	1.4
lr.3	105	204	256	47.7
lr.4	149	416	64	2.8
r.1	176	117	64	3.4
r.2	205	84	64	3.9
r.3	0	0	512	39.4
r.4	260	0	64	1.5
r.5	273	46	128	11.5
ucb.1	70	4	32	0.4
ucb.2	44	99	16	0.3
ucc.1	46	4	32	0.3
ucc.2	51	50	16	0.3
ucc.3	39	74	16	0.3
ucc.4	74	70	8	0.0
ucc.5	82	77	16	0.4
ucc.6	175	373	32	1.4

LANDUSE WINDOW RESULTS
(4 of 5)

POLY-GON	WINDOW FX	FY	WIDTH	TIME SECS
ucr.1	4	0	8	0.0
ucr.2	6	23	8	0.1
ucr.3	14	65	128	15.6
ucr.4	63	152	16	0.3
ucw.1	11	81	32	1.2
ucw.2	19	117	32	1.5
ues.1	89	125	64	4.2
ues.2	76	288	128	8.2
uil.1	39	124	32	1.1
uil.2	127	194	32	1.0
uis.1	0	0	8	0.0
uis.2	0	39	4	0.0
uis.3	2	44	64	1.6
uis.4	42	162	16	0.1
uis.5	105	155	32	1.0
uis.6	66	152	16	0.2
uis.7	153	310	32	1.0
uis.8	243	422	16	0.4
uiw.1	216	177	16	0.3
uiw.2	139	245	16	0.4
unk.1	335	0	64	2.8
unk.2	378	248	32	0.3
unk.3	392	186	4	0.0
unk.4	392	193	4	0.0
unk.5	385	200	16	0.3
unk.6	386	220	32	0.3
unk.7	392	336	8	0.0
unk.8	388	402	8	0.1
unk.9	390	421	16	0.2
unk.10	389	445	8	0.1
uoc.1	87	56	32	0.8
uog.1	105	30	128	11.2
uoo.1	0	83	32	1.0
uoo.2	50	152	16	0.2
uoo.3	112	200	32	0.3
uop.1	31	99	8	0.1
uop.2	92	150	32	0.6
uov.1	90	0	16	0.2
uov.2	101	13	32	0.9
urh.1	76	157	16	0.3
urh.2	171	302	8	0.1
urs.1	0	0	256	12.2
urs.2	0	100	64	1.6
urs.3	0	111	32	1.3
urs.4	101	0	256	43.8
urs.5	117	124	64	3.1

LANDUSE WINDOW RESULTS
(5 of 5)

POLY-GON	WINDOW FX	FY	WIDTH	TIME SECS
urs.6	65	166	32	1.0
urs.7	110	173	16	0.2
urs.8	60	243	32	1.2
urs.9	72	198	64	2.1
urs.10	257	8	64	3.5
urs.11	259	58	64	3.0
urs.12	13	300	16	0.3
urs.13	0	347	32	1.0
urs.14	182	266	16	0.2
urs.15	173	307	128	15.1
urs.16	233	274	32	0.6
urs.17	5	383	8	0.1
urs.18	27	406	32	1.0
urs.19	193	397	16	0.3
urs.20	197	408	32	0.9
urs.21	232	400	64	1.1
urs.22	305	447	4	0.1
urs.23	314	447	8	0.1
urs.24	325	442	32	0.6
uus.1	59	303	32	1.2
uus.2	146	366	8	0.1
uut.1	0	104	128	4.7
uut.2	36	121	64	4.2
uut.3	105	201	256	58.7
vv.1	101	180	16	0.4
wo.1	82	312	64	2.0
wo.2	125	431	32	1.2
ws.1	0	0	512	35.3
wwp.1	359	251	4	0.0
wwp.2	10	438	4	0.0
wwp.3	304	282	16	0.1
wwp.4	298	370	16	0.2
wwp.5	317	365	16	0.4
wwp.6	356	356	8	0.0

TABLE 5.9. TOPOGRAPHY WINDOW RESULTS
(1 of 2)

POLY-	WINDOW			TIME
GON	FX	FY	WIDTH	SECS
1.1	0	0	512	26.7
2.1	0	0	512	29.2
2.2	187	0	32	1.5
2.3	0	0	512	30.8
2.4	85	149	4	0.0
2.5	93	145	4	0.0
2.6	94	146	16	0.2
2.7	98	132	4	0.0
2.8	104	130	4	0.0
2.9	95	141	4	0.0
2.10	100	140	8	0.1
2.11	106	156	8	0.1
2.12	102	166	16	0.1
2.13	108	177	4	0.0
2.14	304	0	128	4.4
2.15	3	284	4	0.0
2.16	0	295	8	0.1
2.17	0	303	4	0.0
2.18	0	343	64	1.5
2.19	188	320	4	0.0
2.20	187	380	16	0.3
2.21	0	380	128	1.4
3.1	89	0	256	61.1
3.2	106	62	16	0.2
3.3	119	61	2	0.0
3.4	0	0	512	33.2
3.5	196	190	64	1.9
3.6	223	253	16	0.4
3.7	295	0	128	15.5
3.8	366	0	32	0.5
3.9	226	350	8	0.5
3.10	245	350	4	0.0
3.11	211	359	2	0.0
3.12	217	354	4	0.0
3.13	227	357	4	0.0
3.14	252	354	8	0.1
3.15	0	410	64	1.1
3.16	16	409	4	0.0
3.17	16	422	16	0.1
4.1	105	0	8	0.1
4.2	112	10	4	0.0
4.3	119	0	256	66.0
4.4	0	0	512	34.5
4.5	277	0	8	0.1
4.6	376	0	32	0.2
4.7	335	305	4	0.0
4.8	346	329	8	0.1
4.9	379	335	8	0.1

TOPOGRAPHY WINDOW RESULTS
(2 of 2)

POLY-GON	WINDOW FX	FY	WIDTH	TIME SECS
4.10	378	347	8	0.1
4.11	364	360	16	0.2
4.12	367	372	2	0.0
4.13	374	377	2	0.0
5.1	158	14	4	0.1
5.2	136	15	128	16.9
5.3	0	0	512	27.8
5.4	384	0	8	0.0
5.5	324	280	32	0.4
5.6	359	273	2	0.0
5.7	352	285	64	2.1
6.1	144	30	4	0.0
6.2	168	25	128	18.9
6.3	151	36	8	0.1
6.4	219	141	4	0.0
6.5	230	29	256	48.2
6.6	380	102	16	0.1
6.7	385	115	4	0.0
6.8	317	234	8	0.1
6.9	335	261	4	0.0
6.10	378	290	8	0.1
6.11	364	299	32	0.9
6.12	360	306	2	0.0
7.1	174	27	128	20.0
7.2	244	41	256	42.7
7.3	373	92	64	1.8
7.4	302	142	4	0.0
7.5	384	140	1	0.0
8.1	183	30	64	5.7
8.2	310	58	8	0.1
8.3	256	53	256	39.6
8.4	340	51	2	0.0
8.5	365	88	128	4.6
8.6	330	191	2	0.0
9.1	192	34	64	3.2
9.2	309	99	16	0.4
9.3	328	67	32	1.1
9.4	325	70	128	11.3
9.5	381	195	4	0.1
9.6	372	202	16	0.1
10.1	195	40	32	1.5
10.2	367	78	16	0.4
10.3	350	95	8	0.1
10.4	334	106	128	5.1
11.1	347	113	16	0.4
11.2	362	173	4	0.0

TABLE 5.10. FLOODPLAIN WINDOW RESULTS

POLY-GON	WINDOW FX	FY	WIDTH	TIME SECS
right	0	0	512	5.2
left	0	0	512	5.2
center	0	0	512	5.4

TABLE 5.11. INTERSECTION STATISTICS
(1 of 3)

TREE 1	TREE 2	AREA PIXELS	AREA ACRES	NUMBER OF NODES GRAY	BLACK	WHITE	TIME SECS
f.center	t.1	28446	4039.33	1672	2394	2623	5.0
f.center	t.2	1281	181.90	798	780	1615	2.6
f.center	t.3	0	0.00	0	0	1	0.1
f.center	t.4	0	0.00	0	0	1	0.0
f.center	t.5	0	0.00	0	0	1	0.0
f.center	t.6	0	0.00	0	0	1	0.0
f.center	t.7	0	0.00	0	0	1	0.0
f.center	t.8	0	0.00	0	0	1	0.0
f.center	t.9	0	0.00	0	0	1	0.0
f.center	t.10	0	0.00	0	0	1	0.0
f.center	t.11	0	0.00	0	0	1	0.0
f.center	l.acc	1472	209.02	223	278	392	0.7
f.center	l.acp	152	21.58	72	66	149	0.3
f.center	l.ar	0	0.00	0	0	1	0.0
f.center	l.are	0	0.00	0	0	1	0.0
f.center	l.avf	5869	833.40	1225	1555	2121	4.0
f.center	l.avv	11376	1615.39	1277	1653	2179	4.1
f.center	l.bbr	432	61.34	134	153	250	0.4
f.center	l.beq	229	32.52	88	97	168	0.3
f.center	l.bes	147	20.87	48	51	94	0.1
f.center	l.bt	132	18.74	46	51	88	0.1
f.center	l.fo	469	66.60	166	178	321	0.5
f.center	l.lr	905	128.51	354	416	647	1.0
f.center	l.r	46	6.53	42	34	93	0.1
f.center	l.ucb	0	0.00	0	0	1	0.0
f.center	l.ucc	3	0.43	13	3	37	0.1
f.center	l.ucr	0	0.00	0	0	1	0.0
f.center	l.ucw	0	0.00	0	0	1	0.0
f.center	l.ues	1286	182.61	275	335	491	1.0
f.center	l.uil	0	0.00	0	0	1	0.0
f.center	l.uis	107	10.65	35	38	68	0.1
f.center	l.uiw	0	0.00	0	0	1	0.0
f.center	l.unk	0	0.00	0	0	1	0.0
f.center	l.uoc	0	0.00	0	0	1	0.0
f.center	l.uog	0	0.00	0	0	1	0.0
f.center	l.uoo	75	10.65	31	33	61	0.1
f.center	l.uop	184	26.13	47	55	87	0.2
f.center	l.uov	0	0.00	0	0	1	0.0
f.center	l.urh	20	2.84	19	11	47	0.1
f.center	l.urs	2180	309.56	817	935	1517	2.5
f.center	l.uus	249	35.36	65	72	124	0.2
f.center	l.uut	224	31.81	77	83	149	0.3
f.center	l.vv	108	15.34	38	39	76	0.2
f.center	l.wo	661	93.86	121	139	225	0.3
f.center	l.ws	3401	482.94	1170	1481	2030	3.4
f.center	l.wwp	0	0.00	0	0	1	0.0

INTERSECTION STATISTICS
(2 of 3)

TREE 1	TREE 2	AREA PIXELS	AREA ACRES	NUMBER OF NODES GRAY	BLACK	WHITE	TIME SECS
t.1	1.acc	3907	554.79	656	844	1125	2.0
t.1	1.acp	1402	199.08	408	490	735	1.2
t.1	1.ar	581	82.50	164	203	290	0.5
t.1	1.are	0	0.00	0	0	1	0.0
t.1	1.avf	16358	2322.84	2107	2972	3350	6.5
t.1	1.avv	20288	2880.90	2180	2984	3557	6.6
t.1	1.bbr	432	61.34	134	153	250	0.4
t.1	1.beq	229	32.52	88	97	168	0.3
t.1	1.bes	114	15.19	52	54	103	0.2
t.1	1.bt	88	12.50	34	37	66	0.1
t.1	1.fo	381	54.10	165	168	328	0.5
t.1	1.lr	913	129.65	359	424	654	1.1
t.1	1.r	25	3.55	38	25	90	0.1
t.1	1.ucb	0	0.00	0	0	1	0.0
t.1	1.ucc	139	19.74	53	58	102	0.2
t.1	1.ucr	769	109.20	179	223	315	0.5
t.1	1.ucw	305	43.31	112	140	197	0.3
t.1	1.ues	1404	199.37	348	447	598	1.0
t.1	1.uil	371	52.68	89	101	167	0.3
t.1	1.uis	627	89.03	177	204	328	0.5
t.1	1.uiw	0	0.00	0	0	1	0.0
t.1	1.unk	0	0.00	0	0	1	0.0
t.1	1.uoc	0	0.00	0	0	1	0.0
t.1	1.uog	0	0.00	0	0	1	0.0
t.1	1.uoo	490	69.58	107	121	201	0.3
t.1	1.uop	148	21.02	50	58	93	0.2
t.1	1.uov	0	0.00	0	0	1	0.0
t.1	1.urh	126	17.89	32	33	64	0.1
t.1	1.urs	3851	546.84	1336	1508	2501	4.0
t.1	1.uus	261	37.06	74	81	142	0.2
t.1	1.uut	938	133.20	314	359	584	1.0
t.1	1.vv	108	15.34	38	39	76	0.1
t.1	1.wo	661	93.86	121	139	225	0.4
t.1	1.ws	3402	483.08	1168	1482	2023	3.4
t.1	1.wwp	0	0.00	0	0	1	0.0

INTERSECTION STATISTICS
(3 of 3)

TREE 1	TREE 2	AREA PIXELS	AREA ACRES	NUMBER OF NODES GRAY	BLACK	WHITE	TIME SECS
t.2	l.acc	2434	345.63	506	616	903	1.5
t.2	l.acp	12852	1824.98	1750	2412	2839	5.5
t.2	l.ar	340	48.28	130	130	261	0.4
t.2	l.are	152	21.38	32	26	71	0.1
t.2	l.avf	6716	953.67	1163	1469	2021	3.8
t.2	l.avv	7922	1124.92	1535	1886	2720	4.9
t.2	l.bbr	0	0.00	0	0	1	0.0
t.2	l.beq	0	0.00	0	0	1	0.0
t.2	l.bes	33	4.69	25	21	55	0.1
t.2	l.bt	941	133.62	228	257	428	0.7
t.2	l.fo	1009	143.28	345	394	642	1.0
t.2	l.lr	35	4.97	42	29	98	0.2
t.2	l.r	2209	313.67	443	571	759	1.4
t.2	l.ucb	249	35.36	62	69	118	0.2
t.2	l.ucc	879	124.82	172	183	334	0.5
t.2	l.ucr	749	106.36	179	224	314	0.6
t.2	l.ucw	0	0.00	0	0	1	0.0
t.2	l.ues	224	31.81	106	113	206	0.3
t.2	l.uil	51	7.24	37	33	79	0.1
t.2	l.uis	415	58.93	137	151	261	0.4
t.2	l.uiw	161	22.86	69	71	137	0.2
t.2	l.unk	660	93.72	99	126	172	0.3
t.2	l.uoc	282	40.04	46	57	82	0.1
t.2	l.uog	765	108.63	140	195	226	0.4
t.2	l.uoo	0	0.00	0	0	1	0.0
t.2	l.uop	65	9.23	51	32	122	0.2
t.2	l.uov	171	24.28	57	72	100	0.2
t.2	l.urh	41	5.82	28	20	65	0.1
t.2	l.urs	15811	2245.16	2079	2866	3372	6.4
t.2	l.uus	0	0.00	0	0	1	0.0
t.2	l.uut	981	139.30	491	570	904	1.5
t.2	l.vv	0	0.00	0	0	1	0.0
t.2	l.wo	0	0.00	0	0	1	0.0
t.2	l.ws	7	0.99	29	7	81	0.3
t.2	l.wwp	118	16.76	63	55	135	0.2

TABLE 5.12. QUADTREE TRUNCATION STATISTICS FOR EACH MAP

DEPTH OF TREE	LANDUSE MAP NUM OF NODES	% RE-DUCED	TOPOGRAPHY MAP NUM OF NODES	% RE-DUCED	FLOODPLAIN MAP NUM OF NODES	% RE-DUCED
10	38233	00.00	33349	00.00	6941	00.00
9	22089	44.22	18517	44.47	4473	35.55
8	9489	75.18	7473	77.59	2297	66.91
7	3341	91.26	2537	92.39	1093	84.26
6	1057	97.23	833	97.50	529	92.38
5	309	99.19	296	99.19	213	96.94
4	85	99.78	77	99.77	77	98.89
3	21	99.95	21	99.94	21	99.70
2	5	99.99	5	99.99	5	99.93
1	1	99.99	1	99.99	1	99.99

6. Bibliography on quadtrees

1. Rutovitz, D. Data structures for operations on digital images, in Pictorial Pattern Recognition, G.C. Cheng et al., Eds., Thompson Book Co., Washington, DC, 1968, 105-133.

2. Freeman, H. Computer processing of line-drawing images, ACM Computing Surveys, 1974, 6, 57-97.

3. Ballard, D.H. Strip trees: a hierarchical representation for curves, Communications of the ACM, 1981, 24, 310 - 321.

4. Blum, H. A transformation for extracting new descriptors of shape, in W. Wathen-Dunn, Ed., Models for the Perception of Speech and Visual Form, M.I.T. Press, Cambridge, MA, 1967, 362-380.

5. Pfaltz, J.L. & Rosenfeld, A. Computer representation of planar regions by their skeletons, Communications of the ACM, 1967, 10, 119-122.

6. Finkel, R.A. & Bentley, J.L. Quad trees: a data structure for retrieval on composite keys, Acta Informatica 4, 1-9.

7. Samet, H. Deletion in two-dimensional quad trees, Communications of the ACM, 1980, 23, 703-710.

8. Lee, D.T. & Wong, C.K. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees, Acta Informatica 1977, 9, 23-29.

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

10. Eastman, C.M. Representations for space planning, Communications of the ACM, 1970, 13, 242-250.

11. Warnock, J.E. A hidden surface algorithm for computer generated half tone pictures, Computer Science Department, TR 4-15, University of Utah, June 1969.

12. Sutherland, I.E., Sproull, R.F., & Schumacker, R.A. A characterization of ten hidden-surface algorithms, ACM Computing Surveys 1974, 6, 1-55.

13. Newman, W.M. & Sproull, R.F. Principles of Interactive Computer Graphics, Second Edition, Mc-Graw Hill, New York, 1971.

14. Klinger, A. Patterns and search statistics, in Optimizing Methods in Statistics, J.S. Rustagi, Ed., Academic Press, New York, 1971.

15. Klinger, A. & Dyer, C.R., Experiments in picture representation using regular decomposition, Computer Graphics and Image Processing, 1976, 5, 68-105.

16. Tanimoto, S.L. & Pavlidis, T. A hierarchical data structure for image processing, Computer Graphics and Image Processing, 1976, 4, 104-119.

17. Tanimoto, S.L. Pictorial feature distortion in a pyramid, Computer Graphics and Image Processing, 1976, 5, 333-352.

18. Riseman, E.M. & Arbib, M.A. Computational techniques in the visual segmentation of static scenes, Computer Graphics and Image Processing, 1976, 6, 221-276.

19. Klinger, A. & Rhodes, M.L. Organization and access of image data by areas, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1979, 1, 50-60.

20. Alexandridis, N. & Klinger, A. Picture decomposition, tree data-structures, and identifying directional symmetries as node combinations, Computer Graphics and Image Processing, 1978, 8, 43-77.

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

22. Hunter, G.M. & Steiglitz, K. Operations on images using quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1979, 1, 145-153.

23. Shneier, M. Calculations of geometric properties using quadtrees, Computer Graphics and Image Processing, 1981, 16, 296-302.

24. Hunter, G.M. & Steiglitz, K. Linear transformation of pictures represented by quadtrees, Computer Graphics and Image Processing, 1979, 10, 289-296.

25. Reddy, D.R. & Rubin, S. Representation of three-dimensional objects, CMU-CS- 78-113, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1978.

26. Jackins, C.L. & Tanimoto, S.L. Oct-trees and their use in representing three-dimensional objects, Computer Graphics and Image Processing, 1980, 14, 249-270.

27. Meagher, D.J.R. Octree encoding, a new technique for the representation, manipulation, and display of arbitrary 3-d objects by computer, Rensselaer Polytechnic Institute, TR 80-111, Troy, New York, 1980.

28. Srihari, S.N. & Yau, M. A hierarchical data structure for multidimensional digital images, Department of Computer Science Technical Report Number 185, State University of New York at Buffalo, Buffalo, New York, August 1981.

29. Samet, H. Region representation: quadtrees from binary arrays, Computer Graphics and Image Processing, 1980, 13, 88-93.

30. Samet, H. An algorithm for converting rasters to quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1981, 3, 93-95.

31. Samet, H. Algorithms for the conversion of quadtrees to rasters, Computer Science TR-979, University of Maryland, College Park, MD, November 1980.

32. Samet, H. Region representation: quadtrees from boundary codes, Communications of the ACM, 1980, 23, 163-170.

33. Dyer, C.R., Rosenfeld, A., & Samet, H. Region representation: boundary codes from quadtrees, Communications of the ACM, 1980, 23, 171-179.

34. Samet, H. Connected component labeling using quadtrees, Journal of the ACM, 1981, 28, 487-501.

35. Samet, H. Computing perimeters of images represented by quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1981, 3, 683-687.

36. Dyer, C.R. Computing the Euler number of an image from its quadtree, Computer Graphics and Image Processing, 1980, 13, 270-276.

37. Samet, H. Distance transform for images represented by quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1982, 4, 298-303.

38. Shneier, M. Path-length distances for quadtrees, Information Sciences, 1981, 23, 49-67.

39. Ranade, S., Rosenfeld, A., & Samet, H. Shape approximation using quadtrees, Pattern Recognition, 1982, 15, 31-40.

40. Ranade, S. Use of quadtrees for edge enhancement, IEEE Transactions on Systems, Man, and Cybernetics, 1981, 11, 370-373.

41. Ranade, S., Rosenfeld, A., & Prewitt, J.M.S. Use of quadtrees for image segmentation, Computer Science TR-878, University of Maryland, College Park, MD, February 1980.

42. Wu, A.Y., Hong, T.H., & Rosenfeld, A. Threshold selection using quadtrees, IEEE Transactions on Pattern Analysis and Machine Intelligence 1982, 4 90-94.

43. Ranade, S. & Shneier, M. Using quadtrees to smooth images, Computer Science TR-894, IEEE Transactions on Systems, Man, and Cybernetics, 1981, 11, 373-376.

44. Samet, H. Neighbor finding techniques for images represented by quadtrees, Computer Graphics and Image Processing, 1982, 18, 37-57.

45. Rosenfeld, A. & Kak, A.C. Digital Picture Processing, Academic Press, New York, 1976.

46. Samet, H. A quadtree medial axis transform, Computer Science TR-803, University of Maryland, College Park, MD, August 1979, to appear in Communications of the ACM.

47. Jackins, C. & Tanimoto, S.L. Quad-trees, oct-trees, and K-trees: a generalized approach to recursive decomposition of euclidean space,, Department of Computer Science Technical Report 82-02-02, University of Washington, Seattle, 1982.

48. Dyer, C.R. Space efficiency of region representation by quadtrees, KSL 46, Department of Information Engineering, University of Illinois at Chicago Circle, Chicago, IL, March 1980.

49. Grosky, W.I. & Jain, R. Optimal quadtrees for image segments, Intelligent Systems Laboratory, CSC-81-010, Computer Science Department, Wayne State University, Detroit, MI, December 1980.

50. Li, M., Grosky, W.I., & Jain, R. Normalized quadtrees with respect to translations, Proceedings of PRIP 81, Dallas, Texas, August 1981, 60-62.

51. Jones, L. & Iyengar, S.S. Representation of regions as a forest of quadtrees, Proceedings of PRIP 81, Dallas, Texas, August 1981, 57-59.

52. Gargantini, I. An efficient way to represent quadtrees, University of Western Ontario, 1981.

53. Kawaguchi, E., Endo, T., & Matsunaga, J. DF-expression viewed from digital picture processing, Department of Information Systems, Kyushu University, Japan, 1982.

54. Gibson, L. & Lucas, D. Spatial data processing using generalized balanced ternary, Proceedings of PRIP 82, Las Vegas, Nevada, June 1982, 566-571.

55. Ahuja, N. Approaches to recusive image decomposition, Proceedings of PRIP 81, Dallas, Texas,. August, 1981, 75-80.

56. Samet, H. & Webber, R.E. On encoding boundaries with quadtrees, Computer Science TR-1162, University of Maryland, College Park, MD, February 1982.

7. Conclusions and future plans

7.1. Conclusions

This project gave a firm empirical basis to much of the theoretical analysis previously undertaken for quadtrees both as to their structure and their algorithmic efficiencies. In particular, the following conclusions should be noted:

(1) Errors in the calculations of properties encoded by quadtrees (e.g., areas and perimeters of various land use classes) are due entirely to errors introduced by the original digitization. No new errors are introduced by quadtree manipulation.

(2) Significant reductions in file size are achieved when an image is converted from a binary array representation to a quadtree representation. This is true for both the multicolored and black/white cases.

(3) The block decomposition of the image resulting from the quadtree representations yields major increases in display speed.

(4) Truncation of quadtrees can be used to generate reasonable image approximations that are consistently more compact.

(5) Quadtree algorithms are easy to implement in structured programming languages (e.g., C).

(6) Neighbor finding was found to require visiting 3.5 nodes on the average for each instantiation. This was even better than what was expected theoretically.

(7) Ropes (an alternative neighbor finding technique) were found to be not worth the added expense of extra storage.

(8) Set operations such as union and intersection are efficient and can be used to extract information from images containing different properties.

It should also be noted (in conjunction with (3) and (4) above) that quadtrees could be used effectively in image transmission, enabling the viewer to recieve a very compact approximation of the image followed by a series of modifications that render the image increasingly more precise.

7.2. Future plans

The first phase of this project has dealt with digitization of a government-furnished geographic database and its representation in quadtree form; and with development of algorithms for basic operations on quadtree-represented regions (set-theoretic operations, point-in-region determination, region property computation; submap generation). The efficiency of these algorithms was studied theoretically and experimentally.

The following tasks are planned for the second phase:

(a) Query language. Design of a high-level query language permitting easy interaction with the database by users, thus making the quadtree representation transparent to the users.

(b) Database updating. Develpment of algorithms for addition, deletion, and editing of data items in a quadtree-encoded database.

(c) Point and linear feature data. Quadtree-like data structures will also be used for the storage, retrieval, and editing of point geographic data. Algorithms will be incorporated for performing these functions and for interfacing between tree representations of point and area data. Recently, quadtree-like data structures have been developed for representing region borders and curves. The interface between these structures and the tree representations of points and regions will be investigated.

8. Appendix: Facilities used

Two computers produced by the Digital Equipment Corporation are used by this project. Program development and small-scale testing are performed on a PDP 11/45. Our PDP 11/45 has a 256k bytes of actual memory of which only 64k bytes are directly addressable, no virtual memory capabilities, a disk fetch speed of 1.2 megabits/second, and a memory cycle speed of approximately 500 microseconds. The execution times in the tables of this report refer to the execution speed on the VAX 11/780. The VAX 11/780 has 2000k bytes of actual memory, 6000k bytes of virtual memory, a disk fetch speed of approximately 0.6 megabits/second, and a memory cycle speed of approximately 1400 nanoseconds. The size of a quadtree node is 12 bytes on the PDP 11/45 and 24 bytes on the VAX 11/780. This difference is caused by the different word size on each machine. Both the PDP 11/45 and the VAX 11/780 run the UNIX operation system (versions 6 and 7 respectively).

The picture output device used by this project is a Grinnell GMR-27 Display Processor. Its memory consists of thirteen 512x512 bitplanes. Twelve of these bitplanes carry color information (4 bits for each of the colors: blue, green, and red). The thirteenth bitplane is used for a white overlay capability. The high order eight bitplanes of the twelve color bitplanes can also be displayed to create a grayscale output. The output speed of quadtrees on this device is considerably faster than a raster scan output of a picture file, because the GMR-27 can output a rectangle on the display screen directly from the rectangle's coordinates (i.e., a separate command is not necessary for each pixel in the rectangle as is done when a picture file is output in raster scan mode).

As our display device is connected to a computer with restricted memory (see Appendix), we will, in addition to the above, be investigating more compact in-core representations and the effect of user-controlled paging on algorithm efficiencies. This will be done in conjunction with the development of a quadtree editor (which requires interactive use of display device).

END