The Quadtree and Related Hierarchical Data Structures

HANAN SAMET

Computer Science Department, University of Maryland, College Park, Maryland 20742

A tutorial survey is presented of the quadtree and related hierarchical data structures. They are based on the principle of recursive decomposition. The emphasis is on the representation of data used in applications in image processing, computer graphics, geographic information systems, and robotics. There is a greater emphasis on region data (i.e., two-dimensional shapes) and to a lesser extent on point, curvilinear, and three-dimensional data. A number of operations in which such data structures find use are examined in greater detail.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*trees*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems—*cartography*; I.2.10 [**Artificial Intelligence**]: Vision and Scene Understanding—*representations, data structures, and transforms*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*display algorithms*; *viewing algorithms*; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*curve, surface, solid, and object representations*; *geometric algorithms, languages, and systems*; I.4.2 [**Image Processing**]: Compression (Coding)—*approximate methods*; *exact coding*; I.4.7 [**Image Processing**]: Feature Measurement—*moments*; *projections*; *size and shape*; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design (CAD)

General Terms: Algorithms

Additional Key Words and Phrases: Geographic information systems, hierarchical data structures, image databases, multiattribute data, multidimensional data structures, octrees, pattern recognition, point data, quadtrees, robotics

INTRODUCTION

Hierarchical data structures are becoming increasingly important representation techniques in the domains of computer graphics, image processing, computational geometry, geographic information systems, and robotics. They are based on the principle of recursive decomposition (similar to *divide and conquer* methods [Aho et al. 1974]). One such data structure is the quadtree. As we shall see, the term *quadtree* has taken on a generic meaning. In this survey it is our goal to show how a number of data structures used in different domains are related to each other and to quadtrees. This presentation concentrates on these different representations and illustrates how a number of basic operations that use them are performed.

Hierarchical data structures are useful because of their ability to focus on the interesting subsets of the data. This focusing results in an efficient representation and improved execution times and is thus particularly useful for performing set operations. Many of the operations that we describe can often be performed equally as

CONTENTS

efficiently, or more so, with other data structures. However, hierarchical data structures are attractive because of their conceptual clarity and ease of implementation.

As an example of the type of problems to which the techniques described in this survey are applicable, consider a cartographic database consisting of a number of maps and some typical queries. The database contains a contour map, say at 50-foot elevation intervals, and a land use map classifying areas according to crop growth. Our wish is to determine all regions between 400- and 600-foot elevation levels where wheat is grown. This will require an intersection operation on the two maps. Such an analysis could be rather costly, depending on the way the data are represented. For example, areas where corn is grown are of no interest, and we wish to spend a minimal amount of effort searching such regions. Yet, traditional region representations such as the boundary code [Freeman 1974] are very local in application, making it difficult to avoid examining a corn-growing area that meets the desired elevation criterion. In contrast, hierarchical methods such as the region quadtree are more global in nature and enable the elimination of larger areas from consideration. Another query might be to determine whether two roads intersect within a given area. We could check them point by point, but a more efficient method of analysis would be to represent them by a hierarchical sequence of enclosing rectangles and to discover whether in fact the rectangles do overlap. If they do not, then the search is terminated, but if an intersection is possible, then more work may have to be done, depending on which method of representation is used. A similar query can be constructed for point data—for example, to determine all cities within 50 miles of St. Louis that have a population in excess of 20,000 people. Again, we could check each city individually, but using a representation that decomposes the United States into square areas having sides of length 100 miles would mean that at most four squares need to be examined. Thus California and its adjacent states can be safely ignored. Finally, suppose that we wish to integrate our queries over a database containing many different types of data (e.g., points, lines, and areas). A typical query might be, "Find all cities with a population in excess of 5000 people in wheat-growing regions within 20 miles of the Mississippi River." In the remainder of this survey we shall present a number of different ways of representing data so that such queries and other operations can be efficiently processed.

The coverage and scope of the survey are focused on region data, and are concerned to a lesser extent with point, curvilinear, and three-dimensional data. Owing to space limitations, algorithms are presented only in a descriptive manner. Whenever possible, however, we have tried to motivate critical steps by a liberal use of examples. The concept of a pyramid is discussed only

briefly, and the reader is referred to the collection of papers edited by Rosenfeld [1983] for a more comprehensive exposition. Similarly, we discuss image compression and coding only in the context of hierarchical data structures. Results from computational geometry, although related to many of the topics covered in this survey, are only discussed briefly in the context of representations for curvilinear data. For more details on early results involving some of these and related topics, the interested reader may consult the surveys by Bentley and Friedman [1979], Edelsbrunner [1984], Nagy and Wagle [1979], Requicha [1980], Srihari [1981], Samet and Rosenfeld [1980], and Toussaint [1980]. Overmars [1983] has produced a particularly good treatment of point data. A broader view of the literature can be found in related bibliographies, for example, Edelsbrunner and van Leeuwen [1983] and Rosenfeld [1984]. Nevertheless, given the broad and rapidly expanding nature of the field, we are bound to have omitted significant concepts and references. In addition we at times devote a disproportionate amount of attention to some concepts at the expense of others. This is principally for expository purposes as we feel that it is better to understand some structures well rather than to give the reader a quick run-through of "buzz words." For these indiscretions, we beg your pardon.

1. OVERVIEW OF QUADTREES

The term *quadtree* is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases: (1) the type of data that they are used to represent, (2) the principle guiding the decomposition process, and (3) the resolution (variable or not). Currently, they are used for point data, regions, curves, surfaces, and volumes. The decomposition may be into equal parts on each level (i.e., regular polygons and termed a *regular decomposition*), or it may be governed by the input. The resolution of the decomposition (i.e., the number of times that the decomposition process is applied) may be fixed beforehand, or it may be governed by properties of the input data.

Our first example of quadtree representation of data is concerned with the representation of region data. The most studied quadtree approach to region representation, termed a *region quadtree*, is based on the successive subdivision of the image array into four equal-sized quadrants. If the array does not consist entirely of 1's or entirely of 0's (i.e., the region does not cover the entire array), it is then subdivided into quadrants, subquadrants, etc. until blocks are obtained (possibly single pixels) that consist entirely of 1's or entirely of 0's; that is, each block is entirely contained in the region or entirely disjoint from it. Thus the region quadtree can be characterized as a variable resolution data structure. For example, consider the region shown in Figure 1a, which is represented by the 2^3 by 2^3 binary array in Figure 1b. Observe that the 1's correspond to picture elements (termed *pixels*) that are in the region and the 0's correspond to picture elements that are outside the region. The resulting blocks for the array of Figure 1b are shown in Figure 1c. This process is represented by a tree of degree 4 (i.e., each nonleaf node has four sons). The root node corresponds to the entire array. Each son of a node represents a quadrant (labeled in order NW, NE, SW, SE) of the region represented by that node. The leaf nodes of the tree correspond to those blocks for which no further subdivision is necessary. A leaf node is said to be BLACK or WHITE, depending on whether its corresponding block is entirely inside or entirely outside of the represented region. All nonleaf nodes are said to be GRAY. The quadtree representation for Figure 1c is shown in Figure 1d.

At this point it is appropriate to define a few terms. We use the term *image* to refer to the original array of pixels. If its elements are either BLACK or WHITE then it is said to be *binary*. If shades of gray are possible (i.e., gray levels), then the image is said to be a *gray-scale* image. In our discussion we are primarily concerned with binary images. The *border* of the image is the outer boundary of the square corresponding

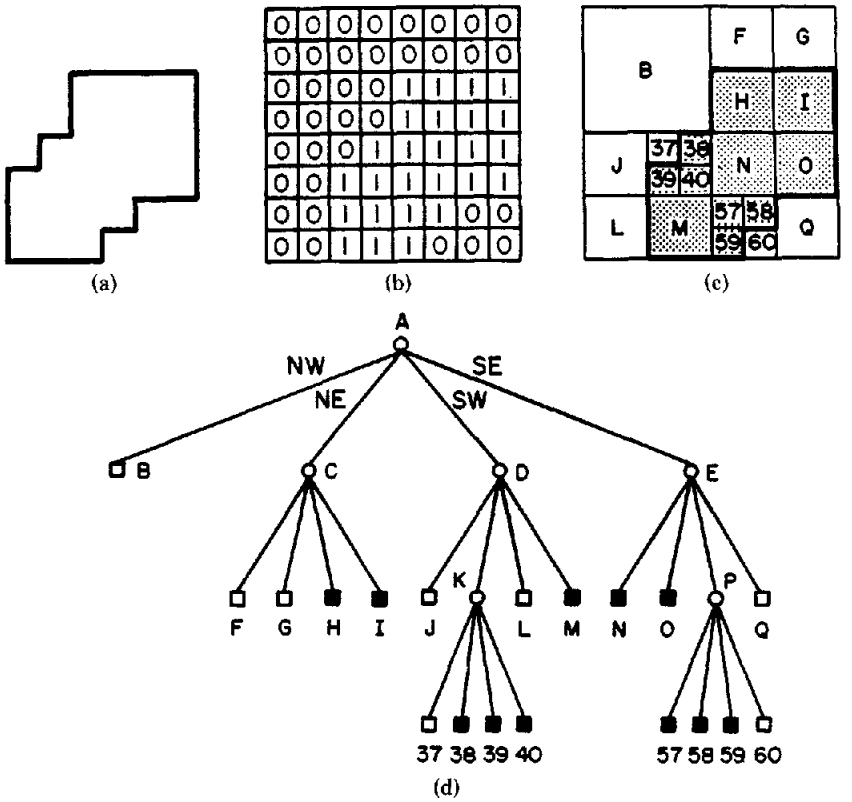


Figure 1. A region, its binary array, its maximal blocks, and the corresponding quadtree. (a) Region. (b) Binary array. (c) Block decomposition of the region in (a). Blocks in the region are shaded. (d) Quadtree representation of the blocks in (c).

to the array. Two pixels are said to be *4-adjacent* if they are adjacent to each other in the horizontal or vertical directions. If the concept of adjacency also includes adjacency at a corner (i.e., diagonal adjacencies), then the pixels are said to be *8-adjacent*. A BLACK region is a maximal *four-connected* set of BLACK pixels, that is, a set S such that for any pixels p, q, in S there exists a sequence of pixels $p = p_0, p_1, \ldots, p_n = q$ in S such that p_{i+1} is 4-adjacent to p_i, $0 \leq i < n$. A WHITE region is a maximal *eight-connected* set of WHITE pixels, which is defined analogously. A pixel is said to have four edges, each of which is of unit length. The *boundary* of a BLACK region consists of the set of edges of its constituent pixels that also serve as edges of WHITE pixels. Similar definitions can be formulated in terms of blocks. For example, two disjoint blocks, P and Q, are said to be *4-adjacent* if there exists a pixel p in P and a pixel q in Q such that p and q are 4-adjacent. Eight-adjacency for blocks is defined analogously.

Unfortunately, the term *quadtree* has taken on more than one meaning. The region quadtree, as shown above, is a partition of space into a set of squares whose sides are all a power of two long. This formulation is due to Klinger [1971; Klinger and Dyer 1976], who used the term Q-tree, whereas Hunter [1978] was the first to use the term quadtree in such a context. Actually, a more precise term would be *quadtrie*, as it is really a trie structure [Fredkin 1960] (i.e., a data item or key is treated as a sequence of characters, where each character has M possible values and a node at level i in the trie represents an M-

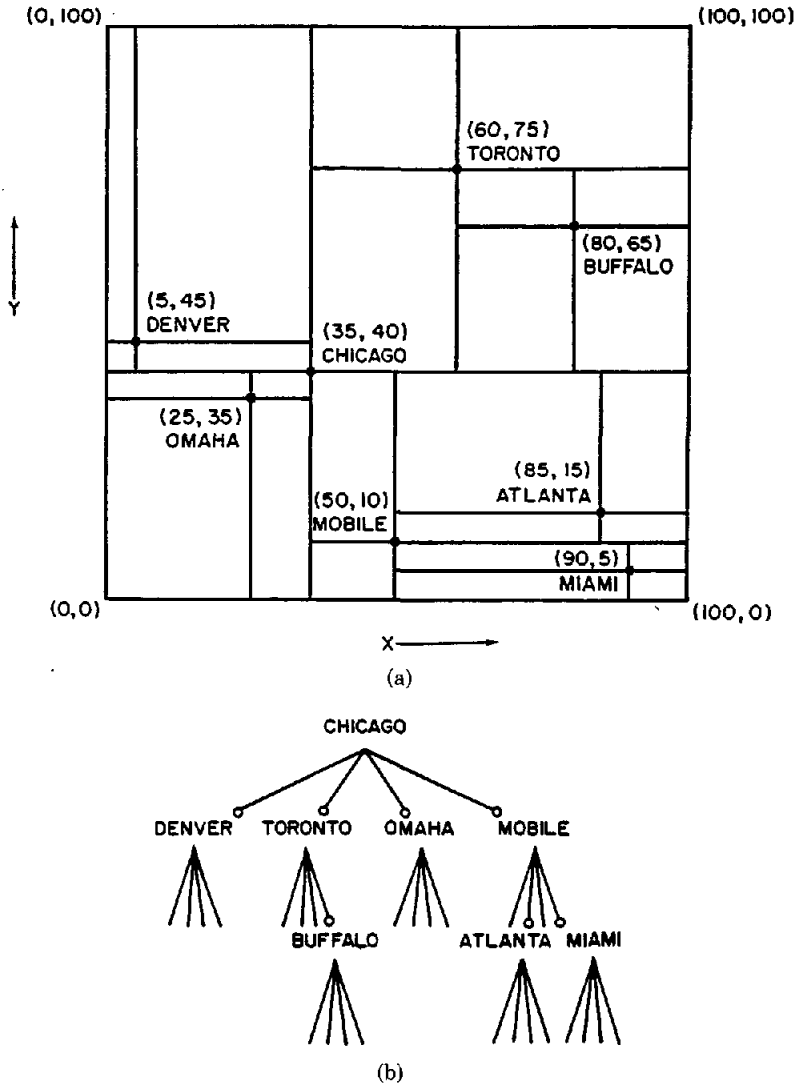


Figure 2. A point quadtree (b) and the records it represents (a).

way branch depending on the *i*th character). A similar partition of space into rectangular quadrants, also termed a quadtree, was used by Finkel and Bentley [1974]. It is an adaptation of the binary search tree [Knuth 1975] to two dimensions (which can be easily extended to an arbitrary number of dimensions). It is primarily used to represent multidimensional point data, and we shall refer to it as a *point quadtree* when confusion with a region quadtree is possible. As an example, consider the point quadtree in Figure 2, which is built for the sequence Chicago, Mobile, Toronto, Buffalo, Denver, Omaha, Atlanta, and Miami.[1] Note that its shape is highly dependent on the order in which the points are added to it.

[1] We have taken liberty in the assignment of coordinates to city names so that the same example can be used throughout the text to illustrate a variety of concepts.

The origin of the principle of recursive decomposition upon which, as we have said, all quadtrees are based is difficult to ascertain. Below, in order to give some indication of the uses of the quadtree, we briefly, and incompletely, trace some of its applications to geometric data. Most likely it was first seen as a way of aggregating blocks of zeros in sparse matrices. Indeed, Hoare [1972] attributes a one-level decomposition of a matrix into square blocks to Dijkstra. Morton [1966] used it as a means of indexing into a geographic database. Warnock [1969; Sutherland et al. 1974] implemented a hidden surface elimination algorithm by using a recursive decomposition of the picture area. The picture area is repeatedly subdivided into successively smaller rectangles while a search is made for areas sufficiently simple to be displayed. The SRI robot project [Nilsson 1969] used a three-level decomposition of space to represent a map of the robot's world. Eastman [1970] observes that recursive decomposition might be used for space planning in an architectural context. He presents a simplified version of the SRI robot representation. A quadtreelike representation in the form of production rules called depth-first (DF)-expressions is discussed by Kawaguchi and Endo [1980] and Kawaguchi et al. [1980]. Tucker [1984a] uses quadtree refinement as a control strategy for an expert vision system.

Parallel to the above development of the quadtree data structure there has been related work by researchers in the field of image understanding. Kelly [1971] introduced the concept of a plan which is a small picture whose pixels represent gray-scale averages over 8 by 8 blocks of a larger picture. Needless effort in edge detection is avoided by first determining edges in the plan and then using these edges to search selectively for edges in the larger picture. Generalizations of this idea motivated the development of multiresolution image representations, for example, the recognition cone of Uhr [1972], the preprocessing cone of Riseman and Arbib [1977], and the pyramid of Tanimoto and Pavlidis [1975]. Of these representations, the pyramid is the closest relative of the region quadtree. A pyramid is an exponentially tapering stack of arrays, each one-quarter the size of the previous array. It has been applied to the problems of feature detection and segmentation. In contrast, the region quadtree is a variable-resolution data structure.

In the remainder of this paper we discuss the use of the quadtree and other hierarchical data structures as they apply to region representation, and to a lesser extent, point data and curvilinear data. Section 2 deals with region representation. We are primarily concerned with two-dimensional binary regions and how basic operations common to computer graphics, image processing, and geographic information systems can be implemented when the underlying representation is a quadtree. Nevertheless, we do show how the quadtree can be extended to represent surfaces and volumes in three dimensions. A brief overview of pyramids and their applications is also presented. For more details, the reader is urged to consult Tanimoto and Klinger [1980] and Rosenfeld [1983]. In Section 3 we present various hierarchical representations of point data. Our attention is focused primarily on the point quadtree and its relative, the k-d tree. A more extensive discussion of point–space data structures can be found in the survey of Bentley and Friedman [1979]. In Section 4 we show how hierarchical data structures are used to handle curvilinear data. We demonstrate the way in which the region quadtree can be adapted to cope with such data and compare this adaptation with other hierarchical data structures.

2. REGION DATA

There are two major approaches to region representation: those that specify the boundaries of a region and those that organize the interior of a region. Owing to the inherent two-dimensionality of region information, our discussion focuses on the second approach.

The *region quadtree* (termed a quadtree in the rest of this section) is a member of a class of representations that are characterized as being a collection of maximal blocks that partition a given region. The simplest such representation is the run length code, where the blocks are restricted to 1 by m

rectangles [Rutovitz 1968]. A more general representation treats the region as a union of maximal square blocks (or blocks of any desired shape) that may possibly overlap. Usually, the blocks are specified by their centers and radii. This representation is called the *medial axis transformation* (MAT) [Blum 1967; Rosenfeld and Pfaltz 1966].

The quadtree is a variant on the maximal block representation. It requires that the blocks be disjoint and have standard sizes (i.e., sides of lengths that are powers of two) and standard locations. The motivation for its development was a desire to obtain a systematic way to represent homogeneous parts of an image. Thus, in order to transform the data into a quadtree, a criterion must be chosen for deciding that an image is homogeneous (i.e., uniform). One such criterion is that the standard deviation of its gray levels is below a given threshold t. By using this criterion the image array is successively subdivided into quadrants, subquadrants, etc. until homogeneous blocks are obtained. This process leads to a regular decomposition. If one associates with each leaf node the mean gray level of its block, the resulting quadtree then will completely specify a piecewise approximation to the image, where each homogeneous block is represented by its mean. The case where $t = 0$ (i.e., a block is not homogeneous unless its gray level is constant) is of particular interest, since it permits an exact reconstruction of the image from its quadtree.

Note that the blocks of the quadtree do not necessarily correspond to maximal homogeneous regions in the image. Most likely there exist unions of the blocks that are still homogeneous. To obtain a segmentation of the image into maximal homogeneous regions, we must allow merging of adjacent blocks (or unions of blocks) as long as the resulting region remains homogeneous. This is achieved by a "split and merge" algorithm [Horowitz and Pavlidis 1976]. However, the resulting partition will no longer be represented by a quadtree; instead, the final representation is in the form of an adjacency graph. Thus the quadtree is used as an initial step in the segmentation process. For example, Figure 3b, c, and d demonstrate the results of the application, in sequence, of merging, splitting, and grouping to the initial image decomposition of Figure 3a. In this case, the image is initially decomposed into 16 equal-sized square blocks. Next, the "merge" step attempts to form larger blocks by recursively merging groups of four homogeneous "brothers" (e.g., the four blocks in the NW and SE quadrants of Figure 3b). The "split" step recursively decomposes blocks which are not homogeneous (e.g., the NE and SW quadrants of Figure 3c). Finally, the "grouping" step aggregates all homogeneous 4-adjacent BLACK blocks into one region apiece; the 8-adjacent WHITE blocks are likewise aggregated into WHITE regions.

An alternative to the quadtree representation is to use a decomposition method that is not regular (i.e., rectangles of arbitrary size rather than squares). This alternative has the potential of requiring less space. However, its drawback is that the determination of optimal partition points necessitates a search. The homogeneity criterion that is ultimately chosen to guide the subdivision process depends on the type of region data that is being represented. In the remainder of this section we shall assume that our domain is a 2^n by 2^n binary image with 1 or BLACK corresponding to foreground and 0 or WHITE corresponding to background (e.g., Figure 1). It is interesting to note that Kawaguchi et al. [1983] use a sequence of m binary-valued quadtrees to encode image data of 2^m gray levels, where the various gray levels are encoded by use of Gray codes [McCluskey 1965]. This should lead to compaction (i.e., larger sized blocks), since the Gray code guarantees that adjacent gray-level values differ by only one binary digit.

In general, any planar decomposition for image representation should possess the following two properties:

(1) The partition should be an infinitely repetitive pattern so that it can be used for images of any size.
(2) The partition should be infinitely decomposable into increasingly finer patterns (i.e., higher resolution).

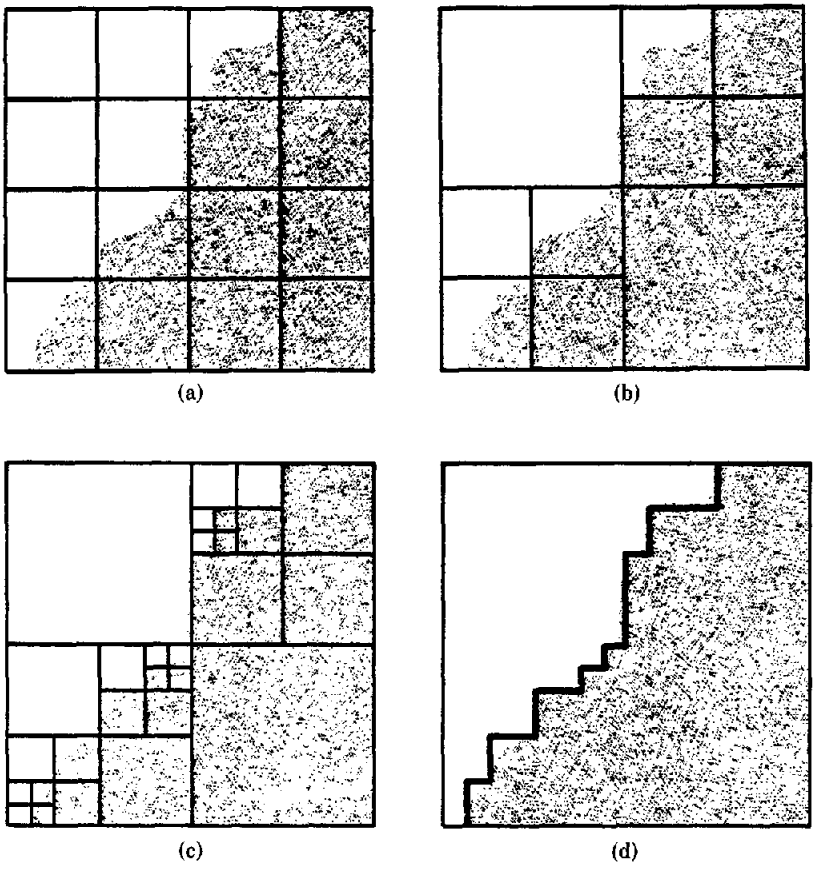


Figure 3. Example illustrating the "split and merge" segmentation procedure. (a) Start. (b) Merge. (c) Split. (d) Grouping.

Bell et al. [1983] discuss a number of tilings of the plane (i.e., tessellations) that satisfy Property (1). They also present a taxonomy of criteria to distinguish among the various tilings. Most relevant to our discussion is the distinction between limited and unlimited hierarchies of tilings. A tiling that satisfies Property (2) is said to be *unlimited*. An alternative characterization of such a tiling is that each edge of each tile lies on an infinite straight line composed entirely of edges. Four tilings satisfy this criterion; of these $[4^4]$,[2] consisting of square atomic tiles (Fig. 4a), and $[6^3]$, consisting of equilateral triangle atomic tiles (Figure 4b), are well-known regular tessellations [Ahuja 1983]. For these two tilings we consider only the molecular tiles given in Figure 5a and b. The tilings $[4^4]$ and $[6^3]$ can generate an infinite number of different molecular tiles where each molecular tile consists of n^2 atomic tiles ($n \geq 1$). The remaining nonregular triangular tilings $[4.8^2]$ (Figure 4c) and [4.6.12] (Figure 4d) are less well understood. One way of generating $[4.8^2]$ and [4.6.12] is to join the centroids of the tiles of $[4^4]$ and $[6^3]$, respectively, to both their vertices and midpoints of their edges. Each of the resulting tilings has two types of hierarchy: in the case of $[4.8^2]$ an ordinary (Figure 5c) and a rotation hierarchy (Figure

[2] The notation is based on the degree of each vertex taken in order around the "atomic" tiling polygon. For example, for $[4.8^2]$ the first vertex of a constituent triangle has degree 4, while the remaining two vertices have degree 8 apiece.

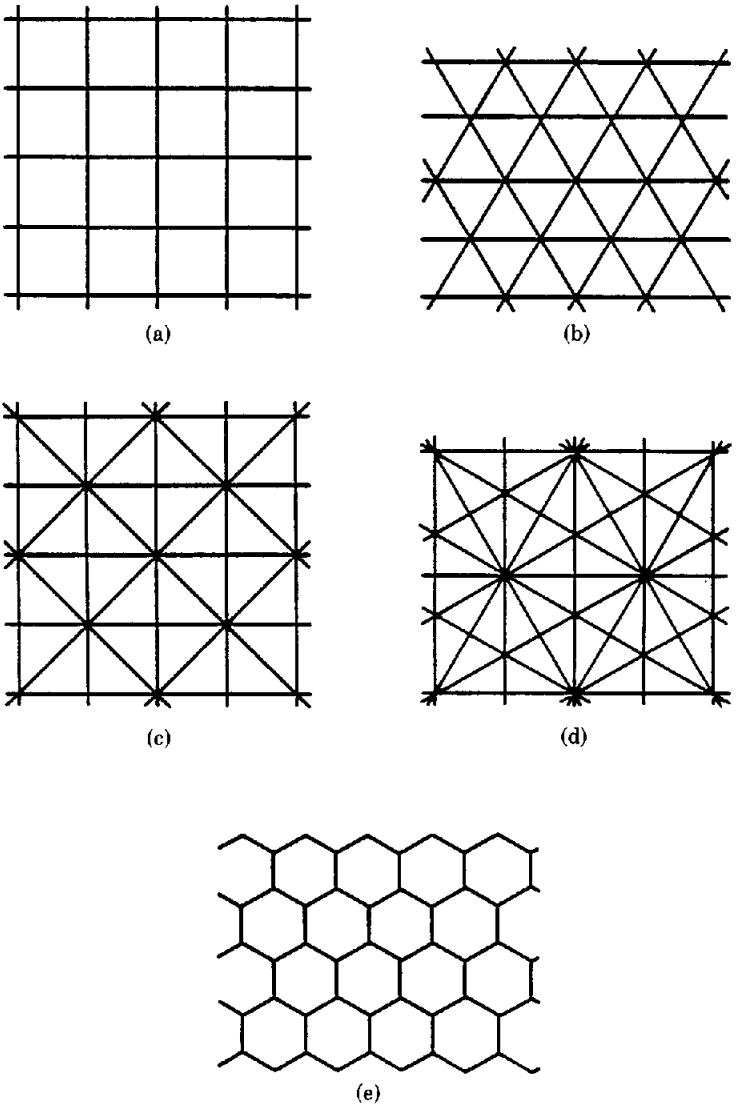


Figure 4. Sample tesselations. (a) $[4^4]$ square. (b) $[6^3]$ equilateral triangle. (c) $[4.8^2]$ isoceles triangle. (d) [4.6.12] 30–60 right triangle. (e) $[3^6]$ hexagon.

5e) and in the case of [4.6.12] an ordinary (Figure 5d) and a reflection hierarchy (Figure 5f). Of the *limited* tilings, many types of hierarchies may be generated [Bell et al. 1983]; however, they cannot, in general, be decomposed beyond the atomic tiling without changing the basic tile shape. This is a serious deficiency of the hexagonal tesselation $[3^6]$ (Figure 4e), which is, however, regular, since the atomic hexagon can only be decomposed into triangles.

Thus we see that to represent data in the Euclidean plane any of the unlimited tilings could have been chosen. For a regular decomposition, the tilings $[4.8^2]$ and [4.6.12] are ruled out. Upon comparing "square" $[4^4]$ and "triangular" $[6^3]$ quadtrees we find that they differ in terms of adjacency and

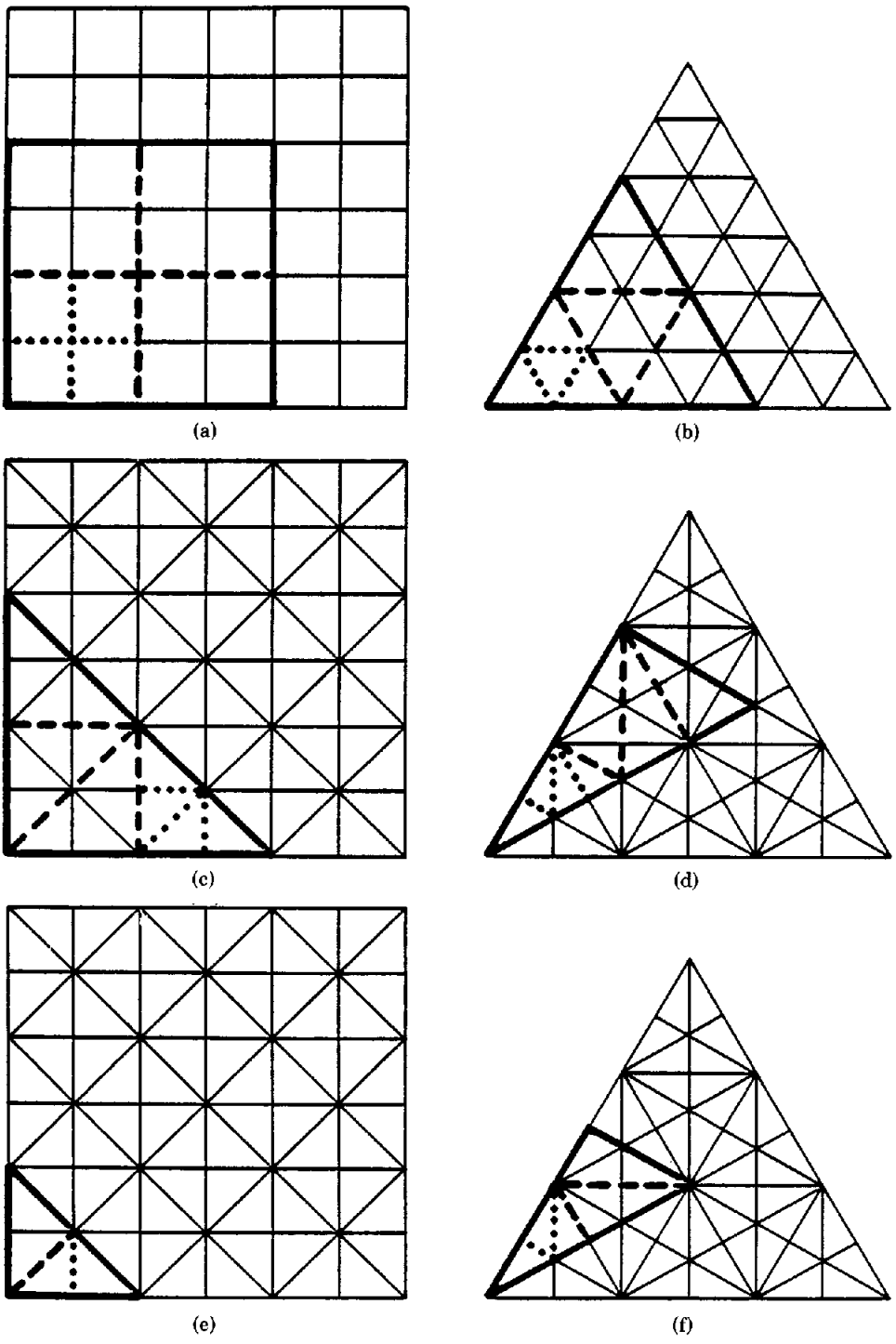


Figure 5. Examples illustrating unlimited tilings. (a) $[4^4]$ hierarchy. (b) $[6^3]$ hierarchy. (c) Ordinary $[4.8^2]$ hierarchy. (d) Ordinary [4.6.12] hierarchy. (e) Rotation $[4.8^2]$ hierarchy. (f) Reflection [4.6.12] hierarchy.

orientation. For example, let us say that two tiles are *neighbors* if they are adjacent either along an edge or at a vertex. A tiling is *uniformly adjacent* if the distances between the centroid of one tile and the centroids of all its neighbors are the same. The adjacency number of a tiling is the number of different intercentroid distances between any one tile and its neighbors. In the case of $[4^4]$, there are only two adjacency distances, whereas for $[6^3]$ there are three adjacency distances. A tiling is said to have *uniform orientation* if all tiles with the same orientation can be mapped into each other by translations of the plane that do not involve rotation or reflection. Tiling $[4^4]$ displays uniform orientation, whereas that of $[6^3]$ does not. Thus we see that $[4^4]$ is more useful than $[6^3]$. It is also very easy to implement. Nevertheless, $[6^3]$ has its uses. For example, Yamaguchi et al. [1984] use a triangular quadtree to generate an isometric view from an octree (a three-dimensional region quadtree discussed in greater detail in Section 2.14) representation of an object.

The type of quadtree used often depends on the grid formed by the image sampling process: Square quadtrees are appropriate for square grids and triangular quadtrees are appropriate for triangular grids. In the case of a hexagonal grid [Burt 1980], since a hexagon cannot be decomposed into hexagons, a *rosette*like molecule of seven hexagons (i.e., septrees) must be built. Note that these rosettes have jagged edges as they are merged to form larger units (e.g., Figure 6). The hexagonal tiling is regular, has a uniform orientation, and most importantly displays a uniform adjacency. These properties are exploited by Gibson and Lucas [1982] in the development of algorithms for septrees (called *generalized balanced ternary* or *GBT* for short) analogous to those existing for quadtrees. Although the septree can be built up to yield large septrees, the smallest resolution in the septree must be decided upon in advance, since its primitive components (i.e., hexagons) cannot be decomposed into septrees later. Thus the septree yields only a partial hierarchical decomposition in the sense that the components can always be merged into larger units, but they cannot always be broken down.

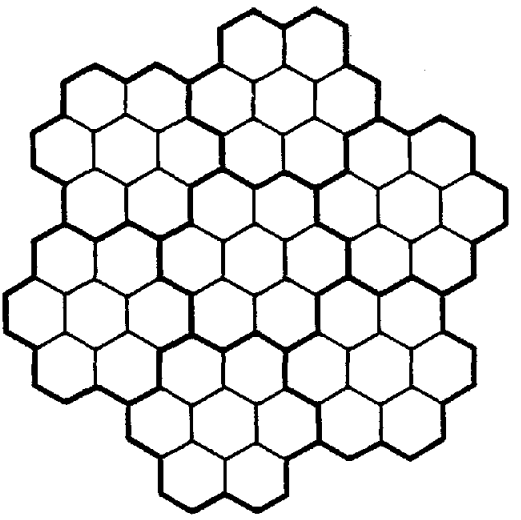


Figure 6. Example septree or "rosette" for a hexagonal grid.

2.1 Neighbor-Finding Techniques

A natural by-product of the treelike nature of the quadtree representation is that many basic operations can be implemented as tree traversals. The difference among them is in the nature of the computation that is performed at the node. Often these computations involve the examination of nodes whose corresponding blocks are "adjacent" to the block corresponding to the node being processed. We shall speak of these adjacent nodes as "neighbors." However, we must be careful to note that adjacency in space does not imply that any simple relationship exists among the nodes in the quadtree. This relationship is the subject of this section. In order to be more precise, we digress briefly and discuss the concepts of adjacency and neighbor in greater detail.

Each node of a quadtree corresponds to a block in the original image. We use the terms *block* and *node* interchangeably. The term that will be used depends on whether we are referring to decomposition into blocks (i.e., Figure 1c) or a tree (i.e., Figure 1d). Each block has four sides and four

corners. At times we speak of sides and corners collectively as directions. Let the four sides of a node's block be called its N, E, S, and W sides. The four corners of a block are labeled NW, NE, SW, and SE with the obvious meaning. Given two nodes P and Q whose corresponding blocks do not overlap, and a direction D, we define a predicate *adjacent* such that adjacent(P, Q, D) is true if there exist two pixels p and q, contained in P and Q, respectively, such that either q is adjacent to side D of p, or corner D of p is adjacent to the opposite corner of q. In such a case, nodes P and Q are considered to be neighbors. For example, nodes J and 39 in Figure 1 are neighbors, since J is to the west of 39, as are nodes 38 and H since H is to the NE of 38. Two blocks may be adjacent both along a side and along a corner (e.g., B is both to the north and NE of J; however, 39 is to the east of J but not to the SE of J). Note that the adjacent relation also holds for nonterminal (i.e, GRAY) as well as terminal (i.e., leaf) nodes.

Unfortunately, the neighbor relation is not a function in a mathematical sense. The problem is that given a node P, and a direction D, there is often more than one node, say Q, that is adjacent. For example, nodes 38, 40, K, and D are all western neighbors of node N. Similarly, nodes 40, K, and D are all NW neighbors of node 57. This means that in order to specify a neighbor more precise information is necessary about its nature (i.e., leaf or nonterminal) and location. In particular, it is necessary to be able to distinguish between neighbors that are adjacent to the entire side of a node (e.g., B is a northern neighbor of J) and those that are only adjacent to a segment of a node's side (e.g., 37 is one of the eastern neighbors of J). An alternative characterization of the difference is that in the former case we are interested in determining a node Q such that its corresponding block is the smallest block (possibly GRAY) of size greater than or equal to the block corresponding to P, whereas in the latter case we specify the neighbor in greater detail, in our case, by indicating the corner of P to which Q must be adjacent. The same distinction can also be made for corner directions. Below we define these relations more formally. In the construction of names we use the following correspondence: G for "greater than or equal," C for "corner," S for "side," and N for "neighbor."

(1) GSN(P, D) = Q. Node Q corresponds to the smallest block (it may be GRAY) adjacent to side D of node P of size greater than or equal to the block corresponding to P.
(2) CSN(P, D, C) = Q. Node Q corresponds to the smallest block that is adjacent to side D of the C corner of node P.
(3) GCN(P, C) = Q. Node Q corresponds to the smallest block (it may be GRAY) opposite the C corner of node P of size greater than or equal to the block corresponding to P.
(4) CCN(P, C) = Q. Node Q corresponds to the smallest block that is opposite to the C corner of node P.

For example, GSN(J, E) = K, GSN(J, S) = L, CSN(J, E, SE) = 39, GCN(H, NE) = G, GCN(H, SW) = K, and CCN(H, SW) = 38. From the above we see that GCN is the corner counterpart of GSN and likewise CCN for CSN. It should be noted that the block corresponding to a node returned as the value of GCN or CCN must overlap some of the region bounded by the designated corner. Thus CCN(J, NE) = B and not 37. The following observations are also in order. First, none of GSN, CSN, GCN, or CCN define a 1-to-1 correspondence (i.e., a node may be a neighbor in a given direction of several nodes, e.g., GSN(J, N) = B, GSN(37, N) = B, and GSN(38, N) = B). Second, GSN, CSN, GCN, and CCN are not necessarily symmetric. For example, GSN(H, W) = B but GSN(B, E) = C.

In the remaining discussions in this survey we focus strictly on GSN and GCN. When we use the term *neighbor*, that is, P is a neighbor of Q, we mean that P is a node of size greater than or equal to Q. For example, node 40 in Figure 1d (or equivalently block 40 in Figure 1c) has neighbors 38, N, 57, M, 39, and 37. A block that is not adjacent to a border of the image has a minimum of five neighbors. This can be seen by observing that a node cannot be adjacent to two nodes of greater size on

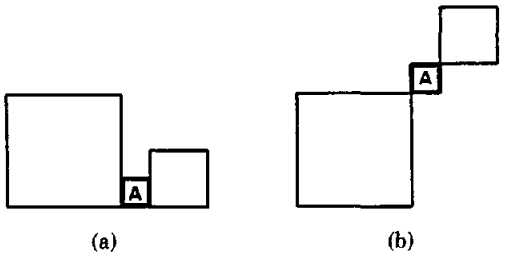


Figure 7. Impossible node configurations in a quadtree.

opposite sides (e.g., Figure 7a) or on opposite corners (e.g., Figure 7b). For further clarification, we observe that a split of a block creates four subblocks of equal size. Each subblock is 4-adjacent to two other subblocks (one horizontally adjacent neighbor and one vertically adjacent neighbor) at one of its vertices and 8-adjacent to the remaining subblock (corner adjacent neighbor) at the same vertex. As an example, given node P such that nodes Q and R are adjacent to its eastern and western sides, respectively, then at most one of nodes Q and R can be of greater size than P. Thus a node can have at most two larger sized neighbors adjacent to its nonopposite sides. One of these neighbors can overlap three neighboring directions, while the other can overlap two neighboring directions. The remaining three neighbors must be of equal size. For example, for node 37 in Figure 1, node B overlaps the NW, N, and NE neighboring directions, node J overlaps the W and SW directions, and the remaining neighbors are nodes 38, 40, and 39 in the E, SE, and S directions, respectively. A node has a maximum of eight neighbors, in which case all but one of the neighbors in the corner direction correspond to blocks of equal size. For example, for node N in Figure 1, the neighbors are nodes H, I, O, Q, P, M, K, and B. It is interesting to observe that for any BLACK node in the image, its neighbors cannot all be BLACK since otherwise merging would have taken place and the node would not be in the image. The same result holds for WHITE nodes.

As mentioned above, most operations on quadtrees can be implemented as tree traversals, with the operation being performed by examining the neighbors of selected nodes in the quadtree. In order that the operation be performed in the most general manner, we must be able to locate neighbors in a way that is independent of both position (i.e., the coordinates) and size of the node. We also do not want to maintain any additional links to adjacent nodes. In other words, we only use the structure of the tree and no pointers in excess of the four links from a node to its four sons and one link to its father for a nonroot node. This is in contrast to the methods of Klinger and Rhodes [1979], which make use of size and position information, and those of Hunter [1978] and Hunter and Steiglitz [1979a, 1979b], which locate neighbors through the use of explicit links (termed ropes and nets). Yet another approach is to hypothesize a point across the boundary in the desired direction and then search for it. This is undesirable for two reasons. First, hypothesizing a point requires that we know the size of the block whose neighbor we are seeking. Second, the search requires that we make use of coordinate information.

Locating adjacent neighbors in the horizontal or vertical directions (i.e., GSN) is relatively straightforward [Samet 1982a]. The basic idea is to ascend the quadtree until a common ancestor with the neighbor is located, and then descend back down the quadtree in search of the neighboring node. It is obvious that we can always ascend as far as the root quadtree and then start our descent. However, our goal is to find the nearest common ancestor, as this minimizes the number of nodes that must be visited. Suppose, for example, that we wish to find the western neighbor of node N in Figure 1, that is, GSN(N, W). The nearest common ancestor is the first ancestor node which is reached via its NE or SE son (i.e., the first ancestor node of which N is not a western descendant). Next, we retrace the path used to locate the nearest common ancestor, except that we make mirror image moves about an axis formed by the common boundary between the nodes. In the case of a western neighbor, the mirror images of NW and SW are NE and SE, respectively. Therefore the western neighbor of node N in Figure 1 is node K. It is located by

ascending the quadtree until the nearest common ancestor A has been located. This requires going through a NW link to reach node E, and a SE link to reach node A. Node K is subsequently located by backtracking along the previous path with the appropriate mirror image moves (i.e., by following a SW link to reach node D, and a NE link to reach node K).

Neighbors in the horizontal or vertical directions need not correspond to blocks of the same size. If the neighbor is larger, then only part of the path from the nearest common ancestor is retraced. Otherwise the neighbor corresponds to a block of equal size and a pointer to a BLACK, WHITE, or GRAY node, as is appropriate, of equal size is returned. If there is no neighbor (i.e., the node whose neighbor is being sought is adjacent to the border of the image in the specified direction), then NIL is returned.

Locating a neighbor in a corner direction (i.e., GCN) is considerably more complex [Samet 1982a]. Once again, we traverse ancestor links until a common ancestor of the two nodes is located. This is a process that requires two or three steps. First, we locate the given node's nearest ancestor, say P, which is also adjacent (horizontally or vertically) to an ancestor, say Q, of the sought neighbor (to see how this is determined, please read on!). If the node P does not exist, then we are at the true nearest common ancestor (e.g., when we are at node D when trying to find the SE neighbor of node J in Figure 1). Otherwise, the second step is one that finds Q by using the procedure for locating horizontally and vertically adjacent neighbors. The final step retraces the remainder of the path while it makes directly opposite moves (e.g., a SE move becomes a NW move). The nearest ancestor of the first step is the first ancestor node that is not reached by a link equal to the direction of the desired neighbor (e.g., to find a SE neighbor, the nearest such ancestor is the first ancestor node that is not reached via its son in the SE direction).[3] As an example of the corner neighbor-finding process, suppose that we wish to locate the SE neighbor of node 40 in Figure 1, which is 57, that is, GCN(40, SE). It is located by ascending the quadtree until we find the nearest ancestor D, which is also adjacent (horizontally in this case) to an ancestor of 57, that is, E. This requires that we go through a SE link to reach K and a NE link to reach D. Node E is now reached by applying the horizontal neighbor-finding techniques in the direction of the adjacency (i.e., east). This forces us to go through a SW link to reach node A. Backtracking results in descending a SE link to reach node E. Finally, we backtrack along the remainder of the path by making 180-degree moves; that is, we descend a SW link to reach node P and a NW link to reach node 57. Note that neighbors in the corner directions need not correspond to blocks of the same size. If the neighbor is larger, then it is handled in the same manner as outlined above for the horizontal and vertical directions (i.e., only part of the path from the nearest common ancestor is retraced). Webber [1984] discusses proofs of the correctness of the various neighbor-finding algorithms presented in this section.

Hunter [1978] and Hunter and Steiglitz [1979a, 1979b] describe a number of algorithms for operating on images represented by quadtrees by using explicit links from a node to its neighbors. These links connect adjacent nodes in the vertical and horizontal directions. A *rope* is defined as a link between two adjacent nodes of equal size where at least one of them is a leaf node. For example, there is a rope between nodes K and N in Figure 1. A *D-adjacency tree* in direction D exists whenever there is a rope between a leaf node, say X, and a GRAY node, say Y. In such a case, the D-adjacency tree of X is said to be the binary tree rooted at Y whose nodes consist of all the descendants of Y (BLACK, WHITE, or GRAY) that are adjacent to X. For example, Figure 8 contains the S-adjacency tree of node B

[3] If the ancestor node is reached by a link directly opposite to the required direction, then we are already at the nearest common ancestor of the sought neighbor. Otherwise, we obtain the neighbor in the direction that did not change (i.e., this determines whether we go in the N, E, S, or W direction for Step 2.

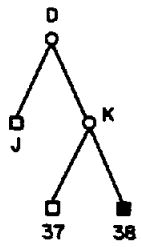


Figure 8. Adjacency tree corresponding to the rope between nodes D and B in Figure 1 (i.e., B's S-adjacency tree).

corresponding to the rope between nodes B and D that crosses the S side of node B.

The process of finding a neighbor by using a roped quadtree is quite simple. The rope is essentially a way to short-circuit the need to find a nearest common ancestor. Suppose that we want to find the neighbor of node X on side N using a rope. If a rope from X on side N exists, then it leads to the desired neighbor. Otherwise the desired neighbor is larger. Next, the tree is ascended until a node having a rope on side N, which will lead to the desired neighbor, is encountered. What we are doing is ascending the S-adjacency tree of the northern neighbor of node X. For example, to find the northern neighbor of node 38 in Figure 1, we ascend through node K to node D, which has a rope along its north side leading to node B (i.e., B's S-adjacency tree).

At times it is not even desirable to ascend nodes in the search for a rope. In such a case Hunter and Steiglitz make use of a *net*. This is a linked list whose elements are all the nodes, regardless of their relative size, that are adjacent along a given side of a node. For example, in Figure 1 there is a net for the southern side of node B consisting of nodes J, 37, and 38.

The advantage of ropes and nets is that the number of nodes that must be visited in the process of finding neighbors is reduced. However, the disadvantage is that the storage requirements are increased considerably. In contrast, our methods [Samet 1982a] only make use of the structure of the quadtree, that is, four links from a nonleaf node to its sons and a link from a nonroot node to its father. Using a suitably defined model, Samet [1982a] and Samet and Shaffer [1984] have shown that in order to locate a neighbor of greater than or equal size in the horizontal or vertical direction, on the average, less than four nodes will be visited when using the nearest common ancestor techniques, whereas less than two nodes must be visited on the average when using ropes.[4] Empirical results confirming this have been reported by Rosenfeld et al. [1982], Samet and Shaffer [1984], and Tucker [1984b]. Thus in practice it is not necessary to add the extra overhead of roping and netting of a quadtree, particularly upon considering that it requires extra storage. It should be noted that, at times, the algorithms that perform the basic operations on the image can be reformulated so that they do not require the computation of the neighbors. This is achieved by transmitting the neighbors of each node in the principal directions as actual parameters. Such techniques are termed *top down* in contrast with the *bottom-up* methods discussed earlier. One such technique is used by Jackins and Tanimoto [1983] in the computation of an n-dimensional perimeter. Their algorithm requires making n passes over the data and works only for neighbors that are adjacent along a side rather than at a corner. Independently, a similar algorithm was devised that does not require n passes but only uses one pass [Rosenfeld et al. 1982b; Samet and Webber 1982]. Another top-down algorithm that is able to compute all neighbors (i.e., adjacent along a side as well as a corner) with just one pass is reported by Samet [1985a].

2.2 Alternative Ways to Represent Quadtrees

As is shown in Section 1 the most natural way to represent a quadtree is to use a tree structure. In this case each node is represented as a record with four pointers to the records corresponding to its sons. If the node is a leaf node, it will have four pointers

[4] A similar result is reported by DeMillo et al. [1978] in the context of embedding a two-dimensional array in a binary tree.

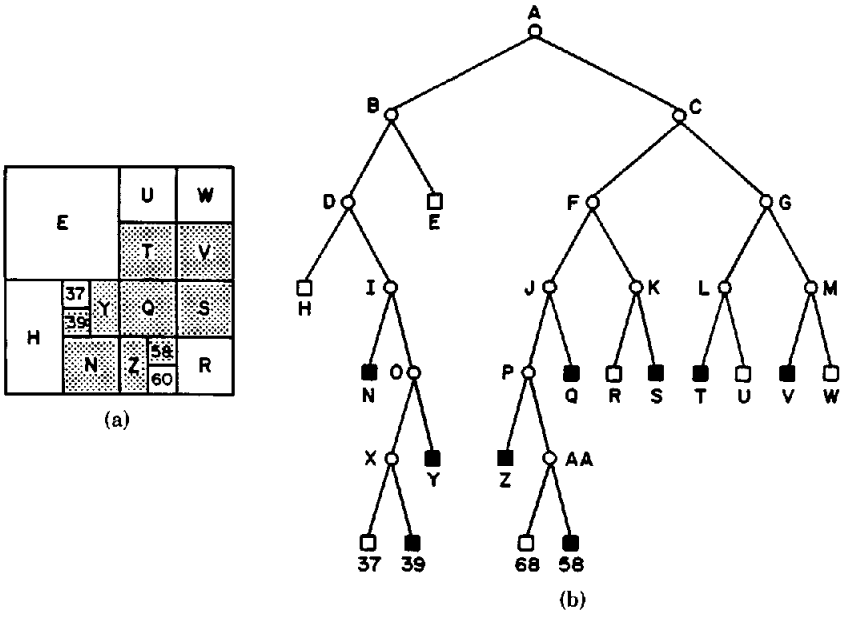


Figure 9. The bintree corresponding to Figure 1. (a) Block decomposition. (b) Bintree representation of the blocks in (a).

to the empty record. In order to facilitate certain operations an additional pointer is at times also included from a node to its father. This greatly eases the motion between arbitrary nodes in the quadtree and is exploited in a number of algorithms in order to perform basic image processing operations.

An alternative tree structure that uses an analogy to the k-d tree [Bentley 1975b] (see Section 3.1) is the *bintree* [Knowlton 1980; Samet and Tamminen 1984; Tamminen 1984a]. In essence, the space is always subdivided into two equal-sized parts alternating between the x and y axes. The advantage is that a node requires space only for pointers to its two sons instead of four sons. In addition, its use generally leads to fewer leaf nodes. Its attractiveness increases further when dealing with higher dimensional data (e.g., three dimensions) since less space is wasted on NIL pointers for terminal nodes and many algorithms are simpler to formulate. For example, Figure 9 is the bintree representation corresponding to the image of Figure 1.

The problem with the tree representation of a quadtree is that it has a considerable amount of overhead associated with it. For example, given an image that can be aggregated to yield B and W BLACK and WHITE nodes, respectively, $(B + W - 1)/3$ additional nodes are necessary for the internal (i.e., GRAY) nodes. Moreover, each node requires additional space for the pointers to its sons. This is a problem when dealing with large images that cannot fit into core memory. Consequently, there has been a considerable amount of interest in pointerless quadtree representations. They can be grouped into two categories. The first treats the image as a collection of leaf nodes. The second represents the image in the form of a traversal of the nodes of its quadtree. The following discussion briefly summarizes the type of operations that can be achieved using such representations. Some of these operations are discussed in greater detail in subsequent sections in the context of pointer-based quadtree representations.

When an image is represented as a collection of the leaf nodes comprising it, each leaf is encoded by a base 4 number termed a *locational code*, corresponding to a sequence of directional codes that locate the

leaf along a path from the root of the quadtree. It is analogous to taking the binary representation of the x and y coordinates of a designated pixel in the block (e.g., the one at the lower left corner) and interleaving them (i.e., alternating the bits for each coordinate). It is difficult to determine the origin of this technique. It was used as an index to a geographic database by Morton [1966] and is termed a *Morton matrix*. Klinger and Rhodes [1979] presented it as a means of organizing quadtrees on external storage. It has also been widely discussed in the literature in the context of multidimensional point data (see Section 3.5). A base 5 variant of it (although all arithmetic operations on the locational code are performed by using base 4), which has an additional code as a *don't care*, is used by Gargantini [1982a] and Abel and Smith [1983] (see also Burton and Kollias [1983], Cook [1978], Klinger and Dyer [1976], Oliver and Wiseman [1983a], Weber [1978], and Woodwark [1982]) to yield an encoding where each leaf in a 2^n by 2^n image is n digits long. A leaf corresponding to a 2^k by 2^k block ($k < n$) will have $n - k$ *don't care* digits. As an example, assuming that codes 0, 1, 2, and 3 correspond to quadrants NW, NE, SW, and SE, respectively, and 4 denotes a *don't care*, block H in Figure 1 is represented by the base 5 number 124. Such an encoding has the interesting property that when the codes of the leaf nodes are sorted in increasing order, the resulting sequence is the postorder (also preorder or inorder since the nonleaf nodes are excluded) traversal of the blocks of the quadtree.

Actually, in the representation described above there is no need to include the locational code of every leaf node. Gargantini [1982a] only retains the locational codes of the BLACK nodes and terms the resulting representation a *linear quadtree*. The codes for the WHITE blocks can be obtained by using the ordering imposed by the sort without having physically to construct the quadtree. Lauzon et al. [1984] propose tha the collection of the leaf nodes be represented by using a variant of the run length code [Rutovitz 1968] termed a *two-dimensional run encoding*. They make use of a Morton matrix. Once the codes of the leaf node have been sorted in increasing order, the resulting list is viewed as a set of subsequences of codes corresponding to blocks of the same color. The final step in its construction is to discard all but the first element of each subsequence of blocks of the same color. The codes of the intervening blocks can be reconstructed by knowing the codes of two successive blocks. In comparison to linear quadtrees, this representation is more compact and more efficient for superposition. However, translation and rotation by multiples of 90 degrees are easier with the linear quadtree [Gargantini 1983]. In addition, given a code for a particular BLACK node, its horizontal and vertical neighbors can be obtained by performing arithmetic operations on the locational code [Abel and Smith 1983; Gargantini 1982a]. However, this often involves search, and can be made more efficient by special-purpose hardware. Nevertheless, this result is significant in that many of the standard quadtree algorithms that rely on neighbor computation can be applied to images represented by linear quadtrees. Abel [1984] describes an organization of the postorder sequence in the form of a B^+-tree [Comer 1979].

Jones and Iyengar [1984] (see also Raman and Iyengar [1983]) introduced the concept of a forest of quadtrees that is a decomposition of a quadtree into a collection of subquadtrees, each of which corresponds to a maximal square. The maximal squares are identified by refining the concept of a nonterminal node to indicate some information about its subtrees. An internal node is said to be of type GB if at least two of its sons are BLACK or of type GB. Otherwise the node is said to be of type GW. For example, in Figure 10, nodes C, E, and F are of type GB and nodes A, B, and D are of type GW. Each BLACK node or an internal node with a label of GB is said to be a maximal square. A forest is the set of maximal squares that are not contained in other maximal squares and that span the BLACK area of the image. Thus the forest corresponding to Figure 10 is {C, E, F}. The elements of the forest are identified by base 4 locational codes. Such a

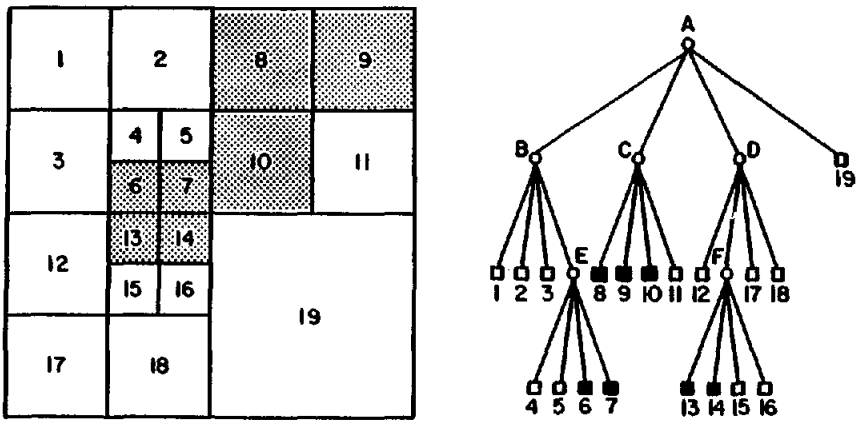


Figure 10. A sample image and its quadtree illustrating the concept of a forest.

representation can lead to a savings of space since large WHITE items are ignored by it.

The second pointerless representation is in the form of a preorder tree traversal (i.e., depth first) of the nodes of the quadtree. The result is a string consisting of the symbols "(", "B", "W" corresponding to GRAY, BLACK, and WHITE nodes, respectively. This representation is due to Kawaguchi and Endo [1980] and is called a DF-expression. For example, the image of Figure 1 has

(W(WWBB(W(WBBBWB(BB(BBBWW

as its DF-expression (assuming that sons are traversed in the order NW, NE, SW, SE). The original image can be reconstructed from the DF-expression by observing that the degree of each nonterminal (i.e., GRAY) node is always 4. DeCoulon and Johnsen [1976] use a very similar scheme termed *autoadaptive block coding*. The difference is that the alphabet consists solely of two symbols, "0" and "1". The "0" corresponds to a block composed of WHITE pixels only. Otherwise, a "1" is used and the block is subdivided into four subblocks. Therefore the "0" is analogous to "W" and the "1" is analogous to "(" and "B". In other words, there is no merging of BLACK pixels into blocks, and thus the coding scheme is asymmetric, whereas the DF-expression method is symmetric with respect to both BLACK and WHITE. The two methods are shown to yield encodings that require a comparable number of bits. A binary tree variant of the DF-expression based on the bintree is discussed by Tamminen [1984b].

Kawaguchi et al. [1983] show how a number of basic image processing operations can be performed on an image represented by a DF-expression. In particular, they demonstrate centroid computation, rotation, scaling, shifting, and set operations. Representation of an image using a preorder traversal is also reported by Oliver and Wiseman [1983a]. They show how to perform operations as mentioned above as well as merging, masking, construction of a quadtree from a polygon, and area filling. Neighbor finding is also possible when traversal-based representations are used, although it is rather cumbersome and time consuming.

In the remainder of this survey we shall be using the pointer-based quadtree representation unless specified otherwise. This should not pose a problem as we have already discussed some of the problems associated with the pointerless representations (i.e., that neighbor finding is more complicated, etc.).

2.3 Conversion

The quadtree is proposed as a representation for binary images because its hierarchical nature facilitates the performance of a large number of operations. However, most images are traditionally represented

by use of methods such as binary arrays, rasters (i.e., run lengths), chain codes (i.e., boundaries), or polygons (vectors), some of which are chosen for hardware reasons (e.g., run lengths are particularly useful for rasterlike devices such as television). Techniques are therefore needed that can efficiently switch between these various representations.

The most common image representation is probably the binary array. There are a number of ways to construct a quadtree from a binary array. The simplest approach is one that converts the array to a complete quadtree (i.e., for a 2^n by 2^n image, a tree of height n with one node per pixel). The resulting quadtree is subsequently reduced in size by repeated attempts at merging groups of four pixels or four blocks of a uniform color that are appropriately aligned. This approach is simple, but is extremely wasteful of storage, since many nodes may be needlessly created. In fact, it is not inconceivable that available memory may be exhausted when an algorithm employing this approach is used, whereas the resulting quadtree fits in the available memory.

We can avoid the needless creation of nodes by visiting the elements of the binary array in the order defined by the labels on the array in Figure 11 (which corresponds to the image of Figure 1). This order is also known as a Morton matrix [Morton 1966] (discussed in Section 2.2). By using such a method a leaf node is never created until it is known to be maximal. An equivalent statement is that the situation does not arise in which four leaves of the same color necessitate the changing of the color of their parent from GRAY to BLACK or WHITE as is appropriate. For example, we note that since pixels 25, 26, 27, and 28 are all BLACK, no quadtree nodes were created for them; that is, node H corresponds to the part of the image spanned by them. This algorithm is shown to have an execution time proportional to the number of pixels in the image [Samet 1980b].

At times the array must be scanned in a row-by-row manner as we build the quadtree (e.g., when a raster representation is used). For example, the pixels of the image

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	34	37	38	49	50	53	54
35	36	39	40	51	52	55	56
41	42	45	46	57	58	61	62
43	44	47	48	59	60	63	64

Figure 11. Binary array representation of the region in Figure 1a.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Figure 12. A labeling of the pixels of the region in Figure 1 that indicates the order of visiting them in the process of constructing a quadtree from the raster representation.

of Figure 1 would be visited in the order defined by the labels on the array of Figure 12. The amount of work that is required depends on whether an odd-numbered or even-numbered row is being processed. For an odd-numbered row, the quadtree is constructed by processing the row from left to right, adding a node to the quadtree for each pixel. As the quadtree is constructed, nonterminal nodes must also be added in such a way that at any given instant, a valid quadtree exists. Even-numbered rows require more work since merging may also take place. In particular, a check for a possible merger must be performed at every even-numbered vertical position (i.e., every even-numbered pixel in a row). Upon the creation of any merger, it must be checked to determine whether another merger is possible. In particular, for pixel position $(a \cdot 2^i, b \cdot 2^j)$ where $(a \bmod 2) = (b \bmod 2) = 1$, a maximum of $k = \min(i, j)$ mergers is possible. In this discussion, a pixel position is the coordinate of its lower right corner with respect to an origin in the upper

left corner of the image. For example, at pixel 60 of Figure 12, that is, position (4, 8), a maximum of two merges is possible. An algorithm using these techniques, which has an execution time proportional to the number of pixels in the image, is described by Samet [1981a]. Unnikrishnan and Venkatesh [1984] present an algorithm for converting rasters to linear quadtrees.

As output is usually produced on a raster device, we need a method for converting a quadtree representation into a suitable form. The most obvious method is to generate an array corresponding to the quadtree, but this method may require more memory than is available and thus is not considered here. Samet [1984] describes a number of quadtree-to-raster algorithms. All of the algorithms traverse the quadtree by rows and visit each quadtree node once for each row that intersects it. For example, a node that corresponds to a block of size 2^k by 2^k is visited 2^k times, and each visit results in the output of a sequence of 2^k 0's or 1's as is appropriate. Some of the algorithms are *top down* and others are *bottom up*. The bottom-up algorithms visit adjacent blocks by use of neighbor-finding techniques, whereas the top-down method starts at the root each time it visits a node. The bottom-up methods are superior as the image resolution gets larger (i.e., n for a 2^n by 2^n image) since the number of nodes that must be visited in locating neighbors is smaller than that necessary when the process is constantly restarted from the root. All of the algorithms have execution times that depend only on the number of blocks in the image (irrespective of their color) and not on their particular configuration. In addition, they do not require memory in excess of that necessary to store the quadtree being output. For example, the two images shown in Figure 13 require the same amount of time to be output since they both have 11 blocks of size 2 by 2 pixels and 20 blocks of 1 pixel. This is important when considerations such as refresh times, etc. must be taken into account.

The chain code representation [Freeman 1974] (also known as a boundary or border code) is very commonly used in carto-

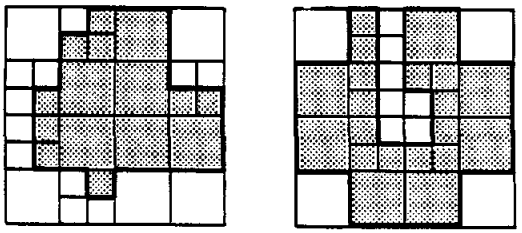


Figure 13. Two images that require the same amount of work to be converted from a quadtree to a raster representation.

graphic applications. It can be specified, relative to a given starting point, as a sequence of unit vectors (i.e., one pixel wide) in the principal directions. We can represent the directions by numbers; for example, let i, an integer quantity ranging from 0 to 3, represent a unit vector having a direction of $90 \cdot i$ degrees. For example, the chain code for the boundary of the BLACK region in Figure 1, moving clockwise starting from the midpoint of the extreme right boundary, is

$$3^2 2^2 3^1 2^1 3^1 2^3 1^3 0^1 1^1 0^1 1^2 0^4 3^2.$$

The above is a four-direction chain code. Generalized chain codes involving more than four directions can also be used. Chain codes are not only compact, but they also simplify the detection of features of a region boundary, such as sharp turns (i.e., corners) or concavities. On the other hand, chain codes do not facilitate the determination of properties such as elongatedness, and it is difficult to perform set operations such as union and intersection as well. Thus it is useful to be able to construct a quadtree from a chain code representation of a binary image. Such an algorithm described by Samet [1980a] is briefly outlined below.

The algorithm has two phases. The first phase traces the boundary in the clockwise direction and constructs a quadtree with BLACK nodes of size unit code length. All terminal nodes are said to be at level 0 and correspond to blocks that are adjacent to the boundary and are within the region whose boundary is being traced. The process begins by choosing a link in the chain code at random and creating a node for it, say P. Next, the following link in the chain

code, say NEW, is examined, and its direction is compared with that of the immediately preceding link, say OLD. At this point, three courses of action are possible. If the directions of NEW and OLD are the same, then a node, say Q, which is a neighbor of P in direction OLD, may need to be added (see Figure 14a). If NEW's direction is to the right of OLD, a new node is unnecessary (see Figure 14b); but if NEW's direction is to the left of OLD, then we may have to add two nodes. First, a node, say Q, that is a neighbor of P in direction OLD is added (if not already present). Second, a node, say R, that is a neighbor of Q in direction NEW is added (see Figure 14c). These nodes are added to the quadtree by using the neighbor-finding techniques discussed previously. As the various links in the chain code are processed, some nodes may be encountered more than once, indicating that they are adjacent to the boundary on more than one side. This information is recorded for each node. Figure 15 shows the block decomposition and partial quadtree after the application of the first phase to the boundary code representation corresponding to Figure 1. The BLACK nodes have been labeled in the order in which they have been visited, starting at the midpoint of the extreme right boundary of the image and proceeding in a clockwise manner. All uncolored nodes in Figure 15 are depicted as short lines emanating from their father.

The first phase of the algorithm leaves many nodes uncolored since it only marks nodes adjacent to the boundary as BLACK. The second phase of the algorithm performs a postorder traversal of the partial quadtree resulting from the first phase and sets all the uncolored nodes to BLACK or WHITE as is appropriate. For an uncolored node to eventually correspond to a BLACK node, it must be totally surrounded by BLACK nodes since otherwise it would have been adjacent to the boundary and could not be uncolored. The algorithm therefore sets every uncolored node to BLACK, unless any of its neighbors is WHITE, or if one of its neighbors is BLACK with a boundary along the shared side. This information is easy to ascertain

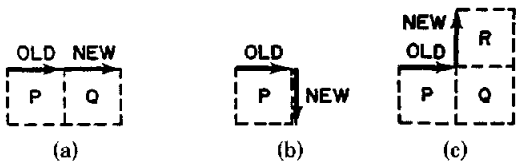


Figure 14. Examples of the actions to be taken when the chain code (a) maintains its direction, (b) turns clockwise, and (c) turns counterclockwise.

by virtue of the boundary adjacency information that is recorded for each BLACK terminal node during the first phase. Also, any GRAY node that has four BLACK sons is replaced by a BLACK node. The above algorithm has a worst-case execution time that is proportional to the product of the region's perimeter (i.e., the length of the chain node) and the log of the diameter of the image (i.e., n for a 2^n by 2^n image) [Samet 1980a]. Webber [1984] presents a variation of this algorithm that shifts the chain code to an optimal position before building the quadtree. The total cost of the shift and build operations is proportional to the region's perimeter.

It is also useful to be able to convert a quadtree representation of a region to its chain code [Dyer et al. 1980]. This is achieved by traversing the boundary in such a way that the region always lies to the right once an appropriate starting point has been determined. The boundary consists of a sequence of (BLACK, WHITE) node pairs. Assume for the sake of this discussion that P is a BLACK node, Q is a WHITE node, and that the block corresponding to node P is to the north of Q. For each BLACK–WHITE adjacency, a two-step procedure is executed. First, the chain link associated with that part of P's boundary that is adjacent to Q is output. The length of the chain is equal to the minimum of the sizes of the two blocks.

Second, the (BLACK, WHITE) node pair that defines the subsequent link in the chain as we traverse the boundary is determined. There are three possible relative positions of P and Q as outlined in Figure 16: (1) P extends past Q (Figure 16a), (2) Q extends past P (Figure 16b), or (3) P and Q meet at the same point (Figure 16c). In order to determine the next pair, the adja-

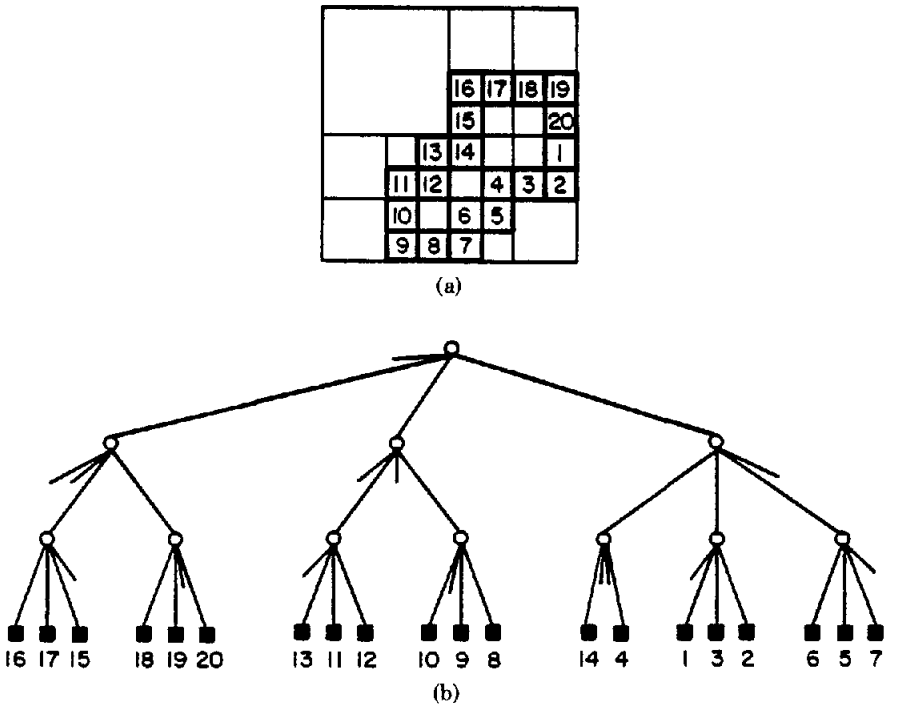


Figure 15. Block decomposition (a) and quadtree (b) of the region in Figure 1 after application of phase one of the chain code to quadtree algorithm.

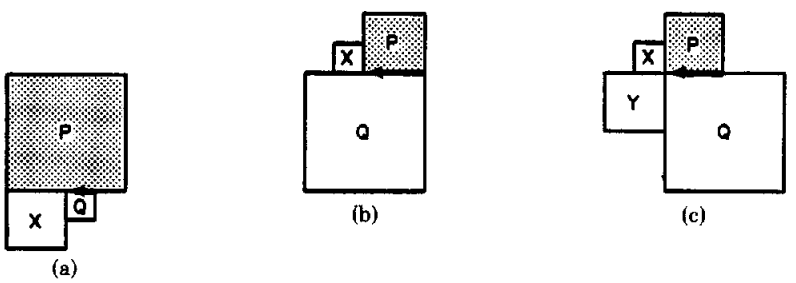


Figure 16. Possible overlap relationships between the (BLACK, WHITE) adjacent node pair (P, Q). The arrow indicates the boundary segment just output. (a) P extends past Q. (b) Q extends past P. (c) P and Q meet at the same point.

cent nodes X and Y are located by using the neighbor-finding techniques discussed previously. At this point the next pair can be determined by referring to Figure 17 and choosing the two blocks that are adjacent to the arrow in the appropriate case. Note that we assume that the region is four-connected so that blocks touching only at a corner are not adjacent. For example, the new pair in Figure 17g is (P, X); that is, the boundary turns right regardless of the type of node Y. The algorithm has an average execution time that is proportional to the region's perimeter [Dyer et al. 1980].

In the case where a region contains holes, the algorithm can be extended by systematically traversing all BLACK nodes upon completion of the first boundary-following sequence. Whenever a BLACK node is encountered with a boundary edge unmarked

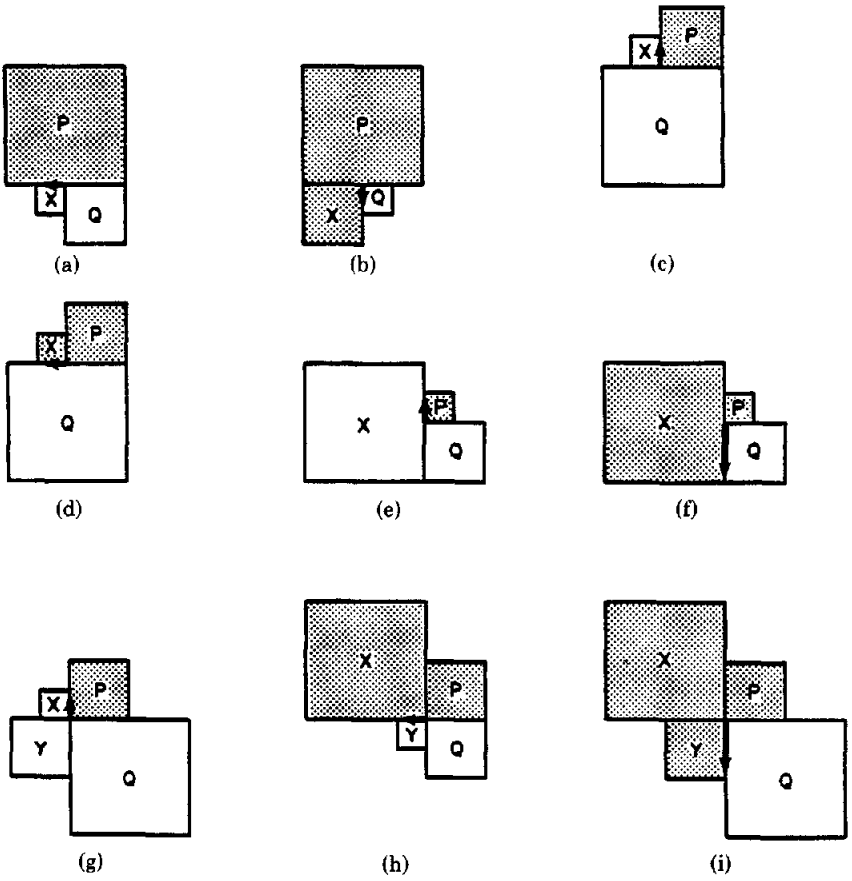


Figure 17. Possible configurations of P, Q, and their neighbor blocks in determining the next (BLACK, WHITE) pair. The arrow indicates the next boundary segment to be output.

by the boundary follower, its boundary is followed, after which the traversal of the quadtree continues.

The chain code can be used as an approximation of a polygon by unit vectors. It is also common to represent polygonal data by a set of vertices, or even a point and a sequence of vectors consisting of pairs (i.e., (magnitude, direction)). Hunter [1978] and Hunter and Steiglitz [1979a, 1979b] address the problem of representing simple polygons (i.e., polygons with nonintersecting edges and without holes) by using quadtrees. A polygon is represented by a three-color variant of the quadtree. In essence, there are three types of nodes—interior, boundary, and exterior. A node is said to be of type *boundary* if an edge of the polygon passes through it. Boundary nodes are not subject to merging (they are analogous to BLACK nodes in the matrix (MX) quadtree described in Section 3.2). *Interior* and *exterior* nodes correspond to areas respectively within and outside of the polygon and can be merged to yield larger nodes. Figure 18 illustrates a sample polygon and its quadtree corresponding to the definition of Hunter and Steiglitz [1979a]. One disadvantage of such a representation for polygonal lines is that a width is associated with them, whereas in a purely technical sense these lines have a width of zero. Also a shift in operations may result in information loss. (For more appropriate representations of polygonal lines see Section 4.)

Hunter and Steiglitz present two algorithms for building a quadtree from a polygon. The first is a top-down algorithm that

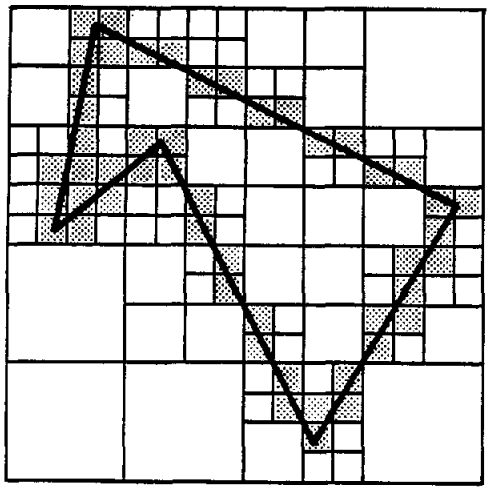


Figure 18. Hunter and Steiglitz's [1979a] quadtree representation of a polygon.

starts at the root and splits the space into four blocks, creating the necessary nodes. Each node whose block (which is not a pixel) intersects the polygonal boundary is recursively split. Given a polygon with v vertices and a perimeter p (in units of pixel width), construction of a quadtree within a 2^n by 2^n space from a polygon has an execution time of $O(v + p + n)$. Unfortunately, the quadtree from the polygon construction algorithm does not distinguish between an interior and an exterior node. A coloring algorithm is then performed that propagates the color of the boundary nodes inward by initially traversing the boundary and stacking all sides that are within the polygon for each boundary node. Coloring is propagated by examining stack entries and their adjacent leaves. For stack entry S, if the block corresponding to its adjacent leaf node, say T, is not smaller and is uncolored, then T is colored and all of its sides with the exception of S are placed on the stack. S is removed from the stack and colored. The key to the algorithm is that boundary nodes (i.e., pixels) are small and their neighbors get larger as the center of the polygon is approached. This algorithm makes use of a netted quadtree to compute neighboring nodes. It has been shown to have an execution time proportional to the number of nodes in the quadtree being colored.

The second algorithm for constructing a quadtree from a polygon is termed an *outline algorithm.* It combines a top-down decomposition of the space in which the polygon is embedded with a traversal of the boundary, resulting in a roped quadtree. During the construction process neighbors are computed as a by-product of the top-down decomposition process. The outline algorithm similarly has an execution time of $O(v + p + n)$. Combining the outline algorithm, a netting process, and the coloring algorithm leads to a quadtree for polygon algorithm with execution time of $O(v + p + n)$.

2.4 Set Operations

The quadtree is especially useful for performing set operations such as the union (i.e., overlay) and intersection of several images. This is described in greater detail by Hunter [1978], Hunter and Steiglitz [1979a], and Shneier [1981a]. For example, obtaining the quadtree corresponding to the union of S and T merely requires a traversal of the two quadtrees in parallel, an examination of corresponding nodes, and construction of the resulting quadtree, say in U. If either of the two nodes is BLACK, then the corresponding node in U is BLACK. If one node is WHITE, say in S, then the corresponding node in U is set to the other node, that is, in T. If both nodes are GRAY, then U is set to GRAY and the algorithm is applied recursively to the sons of S and T. However, once the sons have been processed, when both nodes are GRAY, a check must be made if a merger is to take place since all four sons could be BLACK. For example, consider the union of the quadtrees of Figures 19 and 20. Node B in Figure 19 and node E in Figure 20 are both GRAY. However, the union of their corresponding sons yields four BLACK nodes, which must be merged to yield a BLACK node in U, where the corresponding nodes in S and T were GRAY. Figure 21 shows the result of the union of Figures 19 and 20.

Computing the intersection of two quadtrees is just as simple. The algorithm described above for union is applied, except

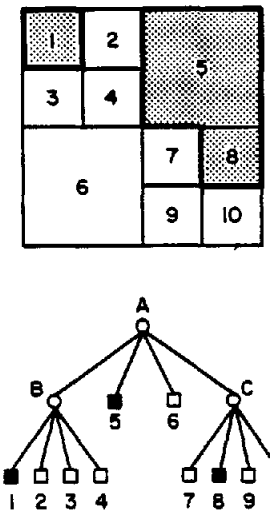


Figure 19. Sample image and its quadtree.

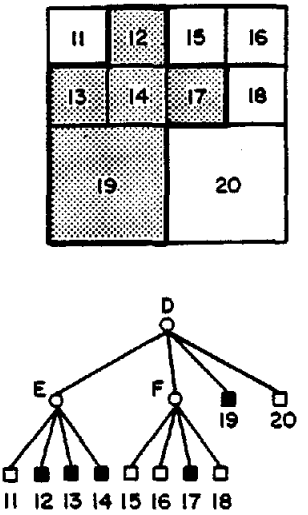


Figure 20. Sample image and its quadtree.

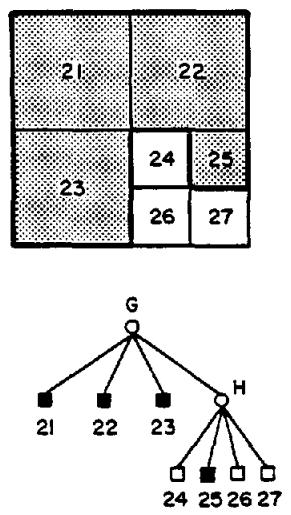


Figure 21. Union of the images in Figures 19 and 20.

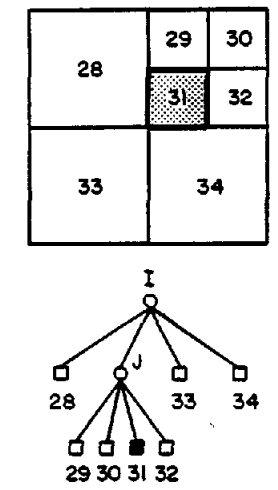


Figure 22. Intersection of the images in Figures 19 and 20.

that the roles of BLACK and WHITE are interchanged. When both nodes are GRAY, the check for a merger is performed to determine if all four sons are WHITE. Figure 22 shows the result of the intersection of Figures 19 and 20.

The time required for these algorithms is proportional to the minimum of the number of nodes at corresponding levels of the two quadtrees. In order to achieve this time bound, the resulting quadtree is composed of subtrees from the quadtrees serving as operands of the set operation. If a new quadtree is constructed, then the operations have an execution time that is proportional to the number of nodes in the quadtrees. An upper bound on this time is the size of the smaller of the two quadtrees. The ability to perform set operations quickly is one of the primary reasons for the popularity of quadtrees over alternative representations such as the chain code. The

chain code can be characterized as a local data structure, since each segment of the chain code conveys information only about the part of the image to which it is adjacent; that is, the image is to its right. Performing an overlay operation on two images represented by chain codes thus requires a considerable amount of work. In contrast, the quadtree is a hierarchical data structure that yields successive refinements at lower levels in the tree. Of course, a hierarchical chain code can be defined, but this is primarily useful in handling extreme cases (null intersection, etc.).

Hunter [1978] suggested a novel approach to solving the problem of determining whether or not two polygons intersect when polygons are represented as quadtrees. One constructs the two quadtrees from the polygons, intersects them, and then checks the result to see whether it is the empty quadtree. This process has an execution time of $O(v + p + n)$ (see Section 2.3). Of course, this time bound is a function of the accuracy required and is subject to errors resulting from limitations imposed by the digitization process. In contrast, Shamos and Hoey [1975] show that the problem can be solved in $O(v \log v)$ time. The reader is cautioned that in actuality the different nature of the representations that are involved may make it difficult to compare the two algorithms (i.e., the constants and quantities are considerably different).

2.5 Transformations

One of the primary motivations for the development of the quadtree concept is a desire to provide an efficient data structure for computer graphics. Warnock [1969] has used recursive decomposition as the basis for the hidden surface elimination algorithm. Hunter's doctoral thesis [Hunter 1978], which addressed the problem of efficiently performing animation by computer, was a significant extension of the quadtree concept from both a theoretical and practical standpoint. In order to do this, the system must have the capability of performing a number of basic transformations. Scaling by a power of two is trivial when using quadtrees since it is simply a reduction in resolution. Rotation by multiples of 90 degrees is equally simple, that is, a recursive rotation of sons at each level of the quadtree. For example, Figure 23b is the result of rotating Figure 23a by 90 degrees counterclockwise. Notice how the NW, NE, SW, and SE sons have become SW, NW, SE, and NE sons, respectively, at each level in the quadtree.

It is also useful to transform a quadtree encoding of a picture in the form of a collection of polygons and holes into another quadtree by applying a linear operator. One simple algorithm [Hunter and Steiglitz 1979b] traces all the polygons in the input quadtree to find vertices. The images of the vertices that result from the application of the linear operator determine the polygons in the output quadtree. The outline and color algorithm [Hunter and Steiglitz 1979a] (see Section 2.3) is used to construct the actual output quadtree for each polygon (as well as holes). The final step is the superposition of the polygons, which is performed by using techniques discussed in Section 2.4. The outline algorithm saves some work by ignoring the boundaries of the input polygons that will not be visible in the output. By assuming that the transformation does not change the resolution (or scale) of the input picture, it can be shown that the transformation algorithm requires time and space of $O(t + p)$ [Hunter and Steiglitz 1979b], where t is the total number of nodes in the input quadtree and p is the total perimeter of the nonbackground visible portions of the input picture.

The linear transformation algorithm and the scaling and rotation operations share a common failing. With the exception of scaling by a power of two, translations, or rotations in multiples of 90 degrees, they result in approximations. Straight lines are not necessarily transformed into straight lines. This failing is often mistakenly attributed to the quadtree representation, whereas in fact it is a direct result of the underlying digitization process. It manifests itself no matter what underlying representation is used when doing raster graphics. (For a quadtree-based representation that is free of such a problem see the

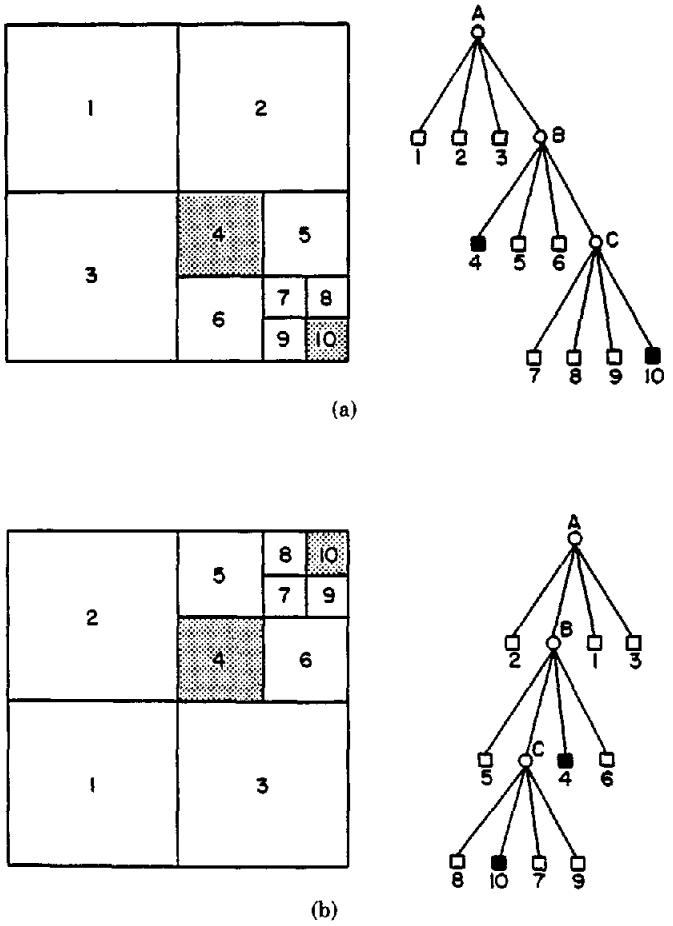


Figure 23. Rotating (a) by 90 degrees counterclockwise yields (b).

discussion of the PM quadtree [Samet and Webber 1983b] in Section 4.2).

Another operation that is useful in graphics applications is termed *windowing*. It is the process of extracting a rectangular window from an image represented by a quadtree and building a quadtree for the window. An algorithm designed to achieve this effect for a square window of size 2^k by 2^k at an arbitrary position in a 2^n by 2^n image is described by Rosenfeld et al. [1982b]. In essence, the new quadtree is constructed as the input quadtree is decomposed and relevant blocks are copied into the new quadtree. The execution time of this process depends both on the relative position of the center of the window with respect to the center of the input quadtree, and on the sizes of the blocks in the input quadtree that overlap the window. A generalization of this windowing algorithm for pointer-based quadtrees [Peters 1984] and linear quadtrees [van Lierop 1984] performs the calculation of a general linear transformation (including scaling and rotation) without extracting the polygon from the quadtree and then rebuilding the quadtree from the transformed polygon [Hunter and Steiglitz 1979b]. For rectangular windows, windowing is trivial to implement if the *squarecode* representation of Oliver and Wiseman [1983b] is used. The squarecode is a variant of the locational code (see Section 2.2), which is used to represent the image as a collection of disjoint squares of arbitrary side length and at arbitrary posi-

tions by recording the length and the address of one of the square's corners.

Quadtrees have also been used for image-processing operations that involve gray-scale images rather than binary images. Some examples include image segmentation [Ranade et al. 1980], edge enhancement [Ranade 1981], image smoothing [Ranade and Shneier 1981], and threshold selection [Wu et al. 1982].

2.6 Areas and Moments

Areas and moments for images represented by quadtrees are extremely simple to compute. To find the area it is necessary to traverse the quadtree in postorder and accumulate the sizes of the BLACK blocks. Assume that the root of a 2^n by 2^n image is at level n and the number of pixels in such an image is 2^{2n}. For a BLACK block at level k, the contribution to the area is 2^{2k}. Moments are obtained by summing the moments of the BLACK blocks. The position of each BLACK block is easy to ascertain because the path that was taken to reach the block is known when processing starts at the root of the quadtree. Knowledge of the area and the first moments permits the computation of the coordinates of the centroid, and thereupon central moments relative to the centroid can be obtained. It should be noted that all of these algorithms have an execution time proportional to the number of nodes in the quadtree [Shneier 1981a]. Chien and Aggarwal [1984] use a normalized representation of the quadtree with respect to the centroid to match noisy objects against models. This method also relies on the selection of a principal axis and scaling to a fixed resolution.

2.7 Connected Component Labeling

Connected component labeling is one of the basic operations of an image-processing system. It is analogous to finding the connected components of a graph. For example, the image of Figure 24 has two components. Given a binary array representation of an image, the traditional method of performing this operation [Rosenfeld and Pfaltz 1966] would be a "breadth-first" approach, which scans the image row by row from left to right and assigns the same label to adjacent BLACK pixels that are found to the right and in the downward direction. During this process pairs of equivalences may be generated, thus necessitating two more steps: one to merge the equivalences and the second to update the labels associated with the various pixels to reflect the merger of the equivalences. (However, Lumia [1983] and Lumia et al. [1983] have made improvements on Rosenfeld and Pfaltz's [1966] method of keeping track of equivalences.)

Using a quadtree to perform the same operation involves an analogous three-step process [Samet 1981b]. The first step is a postorder tree traversal (in order NW, NE, SW, SE), where for each BLACK node that is encountered, say A, all adjacent BLACK nodes on the southern and eastern sides of A are found, and assigned the same label as A. The adjacency exploration is done by using the neighbor-finding techniques of Samet [1982a] (see Section 2.1). At times, the adjacent node may already have been assigned a label, in which case the equivalence is noted. The second step merges all the equivalence pairs that were generated during the first step. The third step performs another traversal of the quadtree and updates the labels on the nodes to reflect the equivalences generated by the first two steps of the algorithm.

As an example, consider the image of Figure 24a, whose quadtree block decomposition is given in Figure 24b and c. All blocks are labeled with a different identifying number in their upper left corner; their lower right corner contains the label assigned by the first step of the connected component labeling process. Two items are worthy of further note. First, when block 15 is processed, neither it nor block 31, its southern neighbor, have been labeled yet and thus label B is assigned to them. When block 20 is processed, it has no label, but its southern neighbor, 31, has already been assigned B as a label, and thus block 20 is assigned label B as well. Second, Figure 24b shows the status of the image at the conclusion of the second step of the algorithm. It has three different labels (i.e., A, B, and

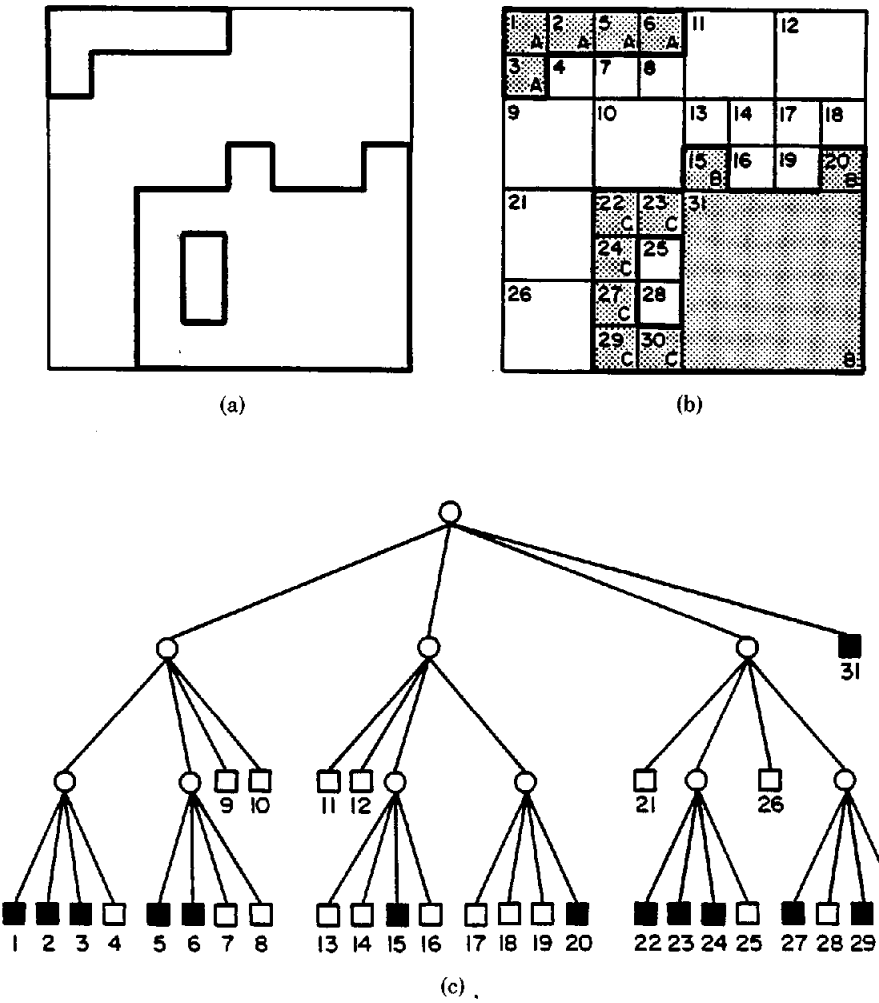


Figure 24. An image, its maximal blocks, and the corresponding quadtree. Blocks in the image are shaded; background blocks are blank. (a) Image. (b) Block decomposition of the image in (a). (c) Quadtree representation of the blocks in (b).

C) with B equivalent to C. This equivalence was generated when the eastern adjacency of block 23 was explored. In essence, block 23 was labeled with C when block 22's eastern adjacency was explored, whereas block 31 was labeled with B when block 15's southern adjacency was explored. Thus we see that the third step of the algorithm will have to be applied, thereby relabeling all C blocks with B.

The execution time of connected component labeling is obtained by examining the three steps of the algorithm. Let B be the number of BLACK nodes in the quadtree. Step 1 is a tree traversal in which neighbors are examined as well. Since the average cost of examining a neighbor is a constant [Samet 1982a; Samet and Shaffer 1984], Step 1 is $O(B)$. Step 3 is also a tree traversal and is $O(B)$ as well. Step 2, the merger of equivalence classes, can be done in $O(B \log B)$ time; the algorithm thus has an average execution time that is $O(B \log B)$. In fact, almost linear average execution time can be obtained by combining Steps 1 and 2 by using the *UNION–FIND* algorithm [Tarjan 1975]. This is a very important result because it means that the exe-

cution time of the connected component labeling process is dependent only on the number of blocks in the image and not on their size. In contrast, the analogous algorithm for the binary array [Rosenfeld and Pfaltz 1966] has an execution time that is proportional to the number of pixels and hence to the area of the blocks. Thus we see that the hierarchical structure of the quadtree data structure saves not only space but time. The cost of the neighbor computation process is avoided when the top-down algorithm of Samet [1984d] is used.

Note that the coloring algorithm of Hunter and Steiglitz [1979a] by itself does not achieve the same effect as the connected component labeling algorithm described above. The coloring algorithm starts with a polygon and traverses its boundary before propagating the boundary color inward and thereby colors the interior nodes. It does not need to merge equivalence classes since the polygon is itself one equivalence class. However, the combination of the coloring algorithm with a polygon identification step will, in effect, yield a connected component labeling algorithm. These results are similar to those achieved using a "depth-first" approach. Use of such an approach leads to algorithms having execution times that are a function of the perimeter of the individual polygons. We also observe that some of the speed of the coloring algorithm is derived from the use of nets that avoid neighbor computation; however, this speed is achieved at the expense of extra storage for the added links as well as the stack.

The algorithm that we have described makes use of a pointer-based quadtree representation. Connected component labeling can also be performed by using some of the pointerless quadtree representations described in Section 2.2. There are two ways to proceed. The first method simply mimics the breadth-first algorithm above and requires use of search to implement the neighbor-finding operation. An alternative method [Samet and Tamminen 1984, 1985] makes use of a staircaselike data structure to remember components on sides of blocks that have already been processed. This method works for perimeter computation (Section 2.8), component counting (Section 2.9), and also three-dimensional data. Both the linear quadtree [Gargantini 1982a] and the DF-expression [Kawaguchi and Endo 1980] representations can be used.

2.8 Perimeter

Computing the perimeter of an image represented by a quadtree can be done in a manner analogous to Step 1 of the connected component labeling process described in Section 2.7. The difference is that in the previous case, when connected components were labeled, the algorithm searched for adjacent BLACK nodes, whereas adjacent WHITE nodes must be searched for when computing the perimeter. In other words, a postorder tree traversal is performed, and for each BLACK node that is encountered its four adjacent sides are explored in the search for adjacent WHITE nodes. For each adjacent WHITE node that is found the length of the corresponding shared side is included in the perimeter.

Use of such an algorithm will result in a certain amount of duplication of effort because each adjacency between two BLACK blocks is explored twice, and neither of these adjacency explorations contributes to the value of the perimeter (e.g., the eastern side of node N and the western side of node O in Figure 1). An alternative algorithm performs adjacency exploration only for southern and eastern neighbors. That is, for each BLACK node a search is made for adjacent WHITE southern and eastern neighbors, and for each WHITE node, a search is made for adjacent BLACK southern and eastern neighbors. The only problem with such a method is that the northern and western boundaries of the image are never explored. This can be alleviated by embedding the image in a white region, as shown in Figure 25.

Both formulations of the algorithm have expected execution times that are proportional to the number of nodes in the quadtree [Samet 1981c]. Jackins and Tanimoto [1983] have developed an asymptotically faster alternative perimeter computation

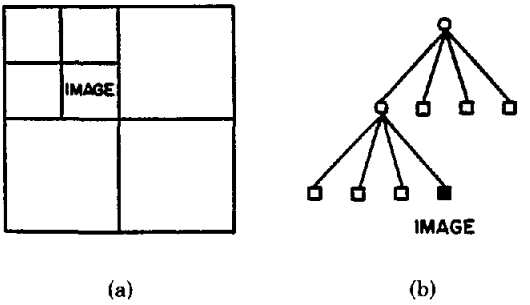


Figure 25. An image totally surrounded by background.

algorithm that works for an arbitrary number of dimensions. This method achieves its efficiency by transmitting the neighbors as parameters rather than by having to rely on neighbor exploration as happens in the approach described above. However, it requires that a separate pass be made over the data for each dimension (see also Samet [1985a]).

2.9 Component Counting

Once the connected components of the image have been labeled, it is easy to count them, since the result is the same as the number of different equivalence classes resulting from Step 2 of the connected component labeling algorithm. An alternative quantity is known as the Euler number or genus, say G, which is $V - E + F$, where V, E, and F correspond to the number of vertices, edges, and faces, respectively, in a planar graph [Harary 1969]. It is defined as the difference between the number of connected components and the number of holes. It is well known [Minsky and Papert 1969] that for a binary image represented by a binary array $G = V - E + F$, where V, E, and F are defined as follows. Let a BLACK pixel be represented by 1 and a WHITE pixel by 0. V is the number of 1's in the image, E is the number of horizontally adjacent pairs of 1's (i.e., 11) or vertically adjacent pairs of 1's, and F is the number of 2 by 2 blocks of 1's.

Dyer [1980] has obtained the same result for a quadtree representation of a binary image by redefining V, E, and F in the context of a quadtree. This is accomplished by letting V be the number of BLACK blocks, E be the number of pairs of adjacent BLACK blocks in the horizontal and vertical directions (see Figure 26), and F be the number of triples or quadruples surrounding a point, that is, 2 by 2 blocks of pixels that are contained by three or four BLACK blocks (e.g., Figure 27). The algorithm for the computation of the genus is a postorder tree traversal that is analogous to Step 1 of the connected component labeling algorithm with the additional proviso that when F is determined, the leaf nodes surrounding the southeastern corner of a BLACK node must be examined. The algorithm's expected execution time is proportional to the number of blocks in the image.

The value of Dyer's result lies not in the mechanics of the algorithm, but in the theory generated thereby. He has shown that the quadtree representation is hierarchical in the sense that the most critical measure is the number of blocks and not their size. It also demonstrates another instance of the use of an algorithm originally formulated for the binary array representation being used in an analogous manner for a quadtree representation by treating blocks (of possibly different sizes) as if they were pixels. This technique was used previously in the labeling of connected components.

2.10 Space Requirements

The prime motivation for the development of the quadtree has been the desire to reduce the amount of space necessary to store data through the use of aggregation of homogeneous blocks. As the previous discussion has demonstrated, an important byproduct of this aggregation has been to decrease the execution time of a number of operations (e.g., connected component labeling and component counting). Nevertheless, the quadtree is not always the ideal representation. The worst case for a quadtree of a given depth in terms of storage requirements occurs when the region corresponds to a checkerboard pattern, as in Figure 28. The amount of space required is

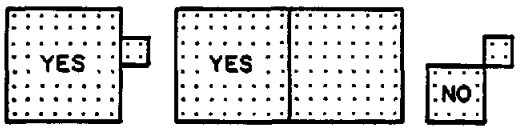


Figure 26. Examples of adjacencies for the computation of the genus of an image.

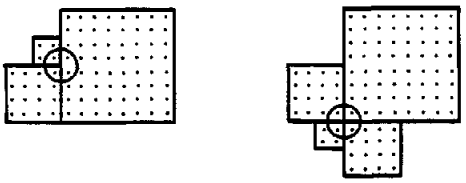


Figure 27. Example of three or four blocks meeting at a corner.

obviously a function of the resolution (i.e., the number of levels in the quadtree). Hunter [1978] presents a fundamental theorem on the space complexity of quadtrees. He has shown that for a simple polygon (i.e., nonintersecting edges) of perimeter p (measured in pixel widths) and a resolution n, the number of nodes in the quadtree is of $O(p + n)$. He goes on to show that $O(p + n)$ is an attainable upper bound. From this, Hunter obtains a corollary of great importance, showing that the quadtree grows linearly in number of nodes as the resolution is doubled, whereas a binary array representation leads to a quadrupling of the number of pixels through doubling the resolution. For experiments on cartographic data that verify the linear growth see Rosenfeld et al. [1982].

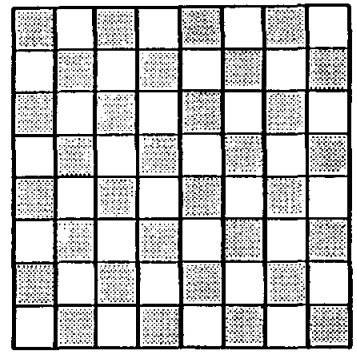
The amount of space occupied by a quadtree is extremely sensitive to its orientation. Dyer [1982] has shown that arbitrarily placing a square of size 2^m by 2^m at any position in a 2^n by 2^n image requires an average of $O(2^{m+2} + n - m)$ quadtree nodes. An alternative characterization of this result is that the amount of space necessary is $O(p + n)$, where p is the perimeter (in pixel widths) of the block. Shifting the image within the space in which it is embedded can reduce the total number of nodes. Grosky and Jain [1983] have shown that for a region such that d is the maximum of its horizontal and vertical extent (measured in pixel widths) and $2^{n-1} < d < 2^n$, the optimal grid resolution is either n or $n + 1$. In other words, embedding the region in a larger area than 2^{n+1} by 2^{n+1} and shifting it around will not result in fewer nodes. This result is used by Li et al. [1982] to obtain an algorithm that finds the configuration of the quadtree requiring a minimum number of nodes. The algorithm proceeds by using a binary array representation of the image and attempting translations of magnitude power of 2 in the vertical, horizontal, and corner directions. When d is defined as above, the algorithm requires $O(2^{2n})$ space and has an execution time that is $O(n \cdot 2^{2n})$.

In Section 2.2 we showed that a tree implementation of a quadtree has overhead in terms of the internal nodes. For an image with B and W BLACK and WHITE blocks, respectively, $(4/3)$ $(B + W)$ nodes are required. In contrast, a binary array representation of a 2^n by 2^n image requires only 2^{2n} bits; however, this quantity grows quite quickly. Furthermore, if the amount of aggregation is minimal (e.g., a checkerboard image), then the quadtree is not very efficient. The overhead for the internal nodes is avoided by using some of the pointerless representations discussed in Section 2.2 such as the linear quadtree and the DF-expression. In fact, the DF-expression requires at most two bits per node. The compression characteristics of DF-expressions compared to boundary and run length codes are discussed by Tamminen [1984b].

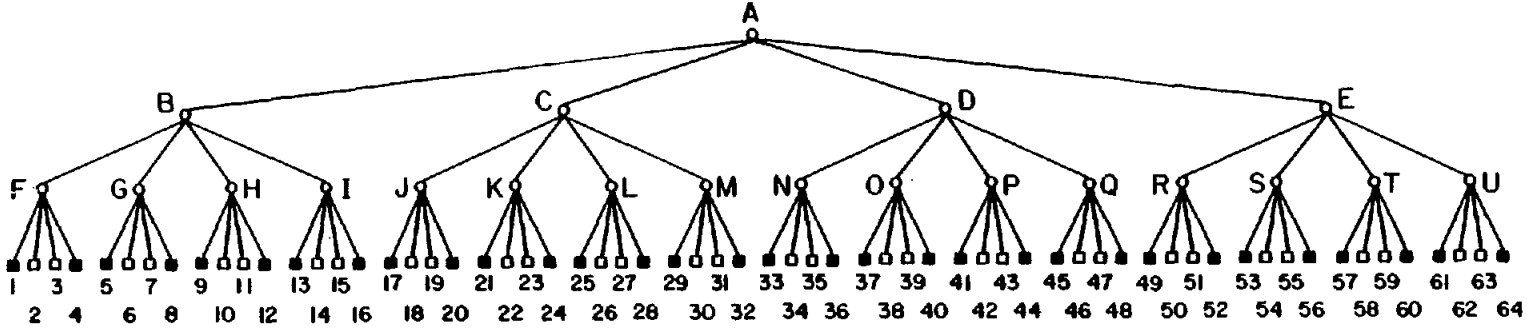
2.11 Skeletons and Medial Axis Transforms

The medial axis of a region [Blum 1967; Duda and Hart 1973; Pfaltz and Rosenfeld 1967; Rosenfeld and Kak 1982] is a subset of points, each of which has a distance from the component of the region (e.g., its boundary), using a suitably defined 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. Before proceeding any further we shall review the definition of a metric.

Let d be a function that maps pairs of points into nonnegative numbers. It is



(a)



(b)

Figure 28. A checkerboard (a) and its quadtree (b).

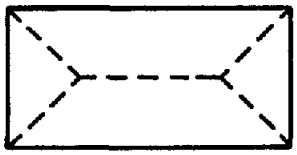


Figure 29. A rectangle and its skeleton using d_E.

called a *metric* or a *distance function* if for all points p, q, and r the following relations are satisfied:

(1) $d(p, q) \geq 0$ and $d(p, q) = 0$ if and only if $p = q$ (positive definiteness);
(2) $dp, q) = d(q, p)$ (symmetry);
(3) $d(p, r) \leq d(p, q) + d(q, r)$ (triangle inequality).

Some of the more common metrics are examined below in the context of the points $p = (p_x, p_y)$ and $q = (q_x, q_y)$. By far the most popular metric is the *Euclidean* metric

$$d_E(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Two other metrics which are used in image processing are the absolute value metric, also known as the *city block* metric (or *Manhattan* metric),

$$d_A(p, q) = |p_x - q_x| + |p_y - q_y|,$$

and the maximum value metric, also known as the *Chessboard* metric,

$$d_M(p, q) = \max\{|p_x - q_x|, |p_y - q_y|\}.$$

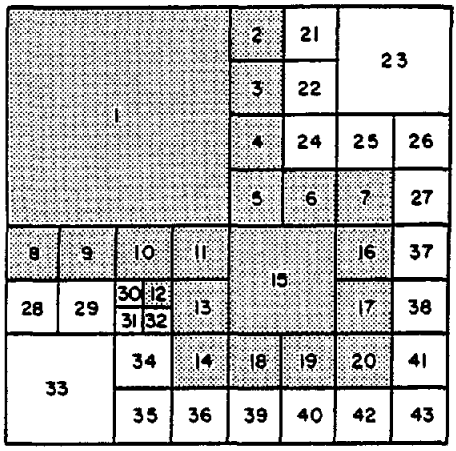
The set of points having $d_E(p, q) \leq T$ are those points contained in a circle centered at p having radius T. Similarly, $d_A(p, q) \leq T$ yields a diamond, centered at p, with side length $T \cdot \sqrt{2}$, and $d_M(p, q) \leq T$ yields a square, centered at p, with side length $2 \cdot T$. For example, by using the Euclidean metric, the skeleton of a circle is its center. The skeleton of the rectangle in Figure 29 is the set of dashed lines within it. An alternative characterization of a skeleton is achieved by drawing an analogy to a "brush fire." That is, imagine that the boundary of the object is set on fire; the remains would be the skeleton.

Skeletons and medial axis transforms are traditionally used in image processing for the purpose of obtaining an approximation of the image. We wish here to obtain an exact representation of the image. The application of the concept of metric to an image represented by a quadtree is discussed by Shneier [1981b] and Samet [1982b]. In particular, Samet [1982b] has shown that the Chessboard metric is most appropriate for an image represented by a quadtree since it has the property that the set of points $\{q\}$ such that $d_M(p, q) \leq T$ is a square. This metric is used to define the Chessboard distance transform for a quadtree as a function DIST, which yields for each BLACK block in the quadtree the Chessboard distance from the center of the block to the nearest point which is on a BLACK–WHITE boundary. In addition, DIST of a WHITE or GRAY block is said to be zero, and the border of the image is assumed to be BLACK. For example, in Figure 30, node 1 has a DIST value of 6, whereas node 12 has a DIST value of 0.5, assuming a 2^4 by 2^4 image. The process of computing the Chessboard distance transform is relatively simple. It consists of a postorder tree traversal where for each BLACK block the eight adjacent (horizontal, vertical, and corner) neighbors are examined to determine the closest WHITE block. This process is analogous to that used for connected component labeling and perimeter computation.

The quadtree skeleton is defined as follows. Given a BLACK block b, it is convenient to use $S(b)$ to refer to the set of pixels in the image spanned by a square with side width $2 \cdot \text{DIST}(b)$ centered about block b. Let the set of BLACK blocks in the image be denoted by B. The *quadtree skeleton* is the set T of BLACK blocks, denoted by t_i, satisfying the following properties:

(1) the set of pixels in $B = \bigcup_i S(t_i)$;
(2) for any t_j in T there does not exist b_k in B ($b_k \neq t_j$) such that $S(t_j) \subseteq S(b_k)$;
(3) for all b_i in B there exists t_j in T such that $S(b_i) \subseteq S(t_j)$.

For example, for the quadtree of Figure 30, the quadtree skeleton consists of nodes



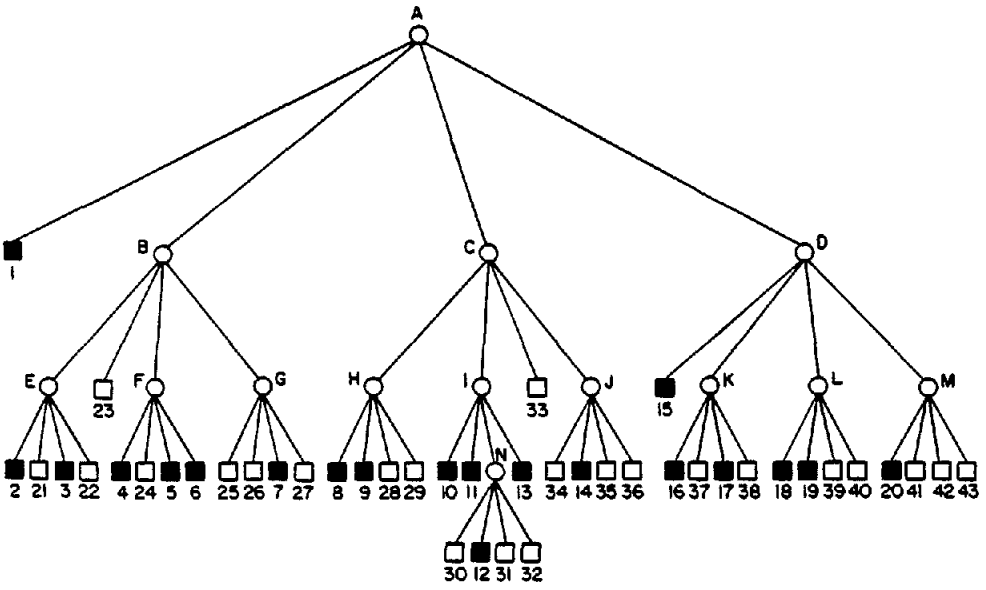


Figure 30. Sample quadtree.

1, 11, and 15 with Chessboard distance transform values of 6, 2, and 4, respectively. Property (1) ensures that the entire image is spanned by the quadtree skeleton. Property (2) is termed the *subsumption* property, wherein b_j is *subsumed* by b_k when $S(b_j) \subseteq S(b_k)$. Property (2) means that the elements of the quadtree skeleton are the blocks with the largest distance transform values. Property (3) ensures that no block in B and not in T requires more than one element of T for its subsumption. Therefore the case where one-half of the block is subsumed by one element of T and the other half is subsumed by another element of T is not permitted. Samet [1983] has shown that the quadtree skeleton of an image is unique.

The *quadtree medial axis transform* (QMAT) of an image is the quadtree whose BLACK nodes correspond to the BLACK blocks comprising the quadtree skeleton

and their associated Chessboard distance transform values. All remaining nodes in the QMAT are WHITE and GRAY with distance value zero. For example, Figure 31 contains the block and tree representations of the QMAT of Figure 30. The algorithm for the construction of a QMAT from its quadtree is straightforward [Samet 1983]. In essence, it is a modified postorder tree traversal where GRAY nodes are processed first, and for each BLACK node a check is made to see if it is subsumed by one of its eight neighbors. If it is, then the node is changed from BLACK to WHITE. This algorithm is facilitated by Property (3), which ensures that there is no need to check whether a node is subsumed partially by one neighbor and partially by another neighbor. The reverse process of reconstructing a quadtree from its QMAT is also possible [Samet 1985b]. It is potentially useful for thinning an image.

The QMAT has a number of important properties. First, it results in a partition of the image into a set of possibly nondisjoint squares having sides whose lengths are sums of powers of two rather than, as in the case of quadtrees, a set of disjoint squares having sides of lengths that are powers of two. Second, the QMAT is more compact than the quadtree, as it never contains more nodes than the quadtree and often contains considerably fewer nodes (e.g., compare Figures 30 and 31). Third, the QMAT representation is less sensitive to shift operations in the sense that a small shift of the image will not, in general, cause the QMAT to get as large as the shifted quadtree. This should be apparent when we realize that the QMAT is most economical storagewise, vis à vis the quadtree when large blocks are surrounded by smaller blocks; this is normally the situation when a shift operation takes place. For example, compare Figure 32 and the result of shifting it by one pixel to the right, as shown in Figure 33.

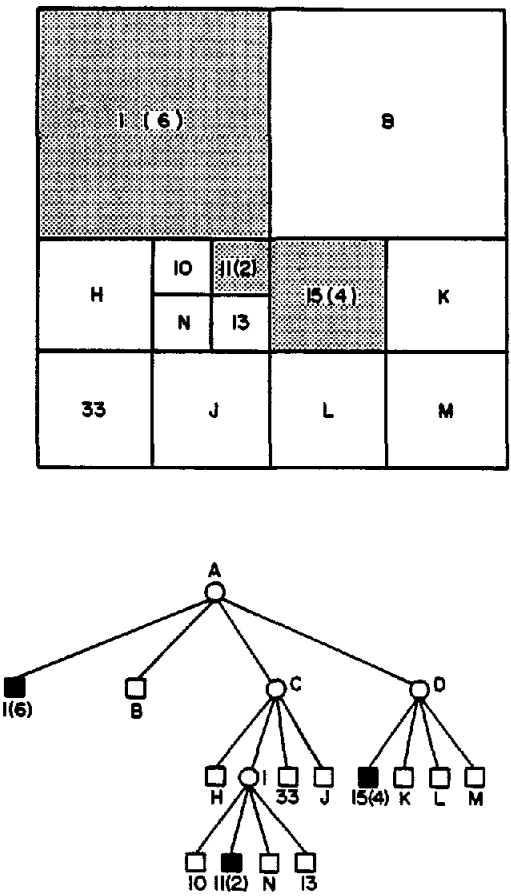
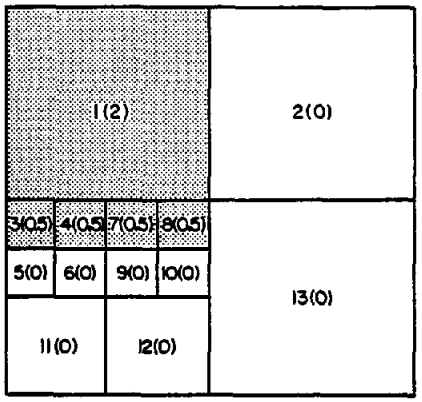


Figure 31. QMAT corresponding to the quadtree of Figure 30.

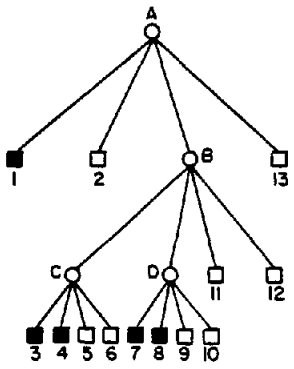
2.12 Pyramids

Given a 2^n by 2^n image array, say $A(n)$, a *pyramid* is a sequence of arrays $\{A(i)\}$ such that $A(i-1)$ is a version of $A(i)$ at half the resolution of $A(i)$, etc. $A(0)$ is a single pixel. For example, Figure 34 shows the structure of a pyramid having three levels. It should be clear that a pyramid can also be defined in a more general way by permitting finer scales of resolution than the power of two scale.

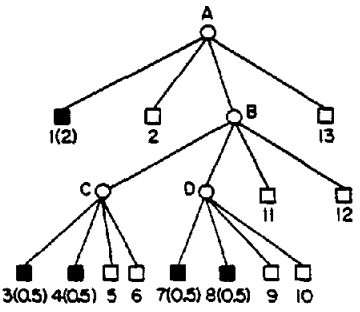
At times it is more convenient to define a pyramid in the form of a tree. Again, assuming a 2^n by 2^n image, a recursive decomposition into quadrants is performed, just as in quadtree construction, except that we keep subdividing until we reach the individual pixels. The leaf nodes of the resulting tree represent the pixels, whereas the nodes immediately above the leaf nodes correspond to the array $A(n-1)$, which is



(a)



(b)



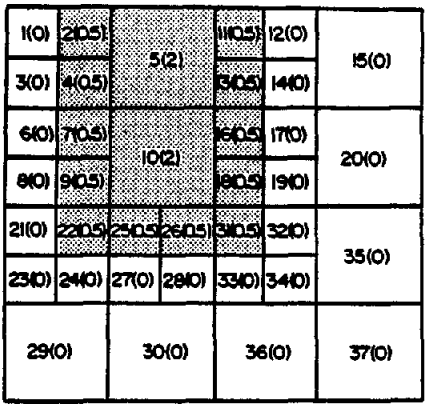
(c)

Figure 32. Sample quadtree. (a) Block decomposition. (b) Tree representation. (c) QMAT.

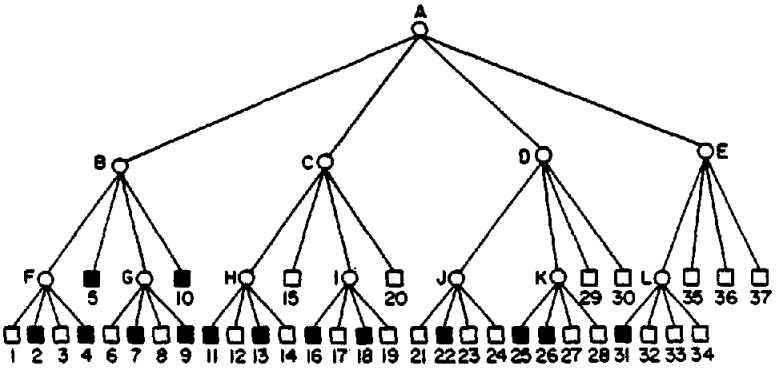
of size 2^{n-1} by 2^{n-1}. The nonterminal nodes are assigned a value that is a function of the nodes below them (i.e., their sons) such as the average gray level.

The above definition of a pyramid is based on a nonoverlapping 2 by 2 blocks of pixels. An alternative definition uses overlapping blocks of pixels. One of the simplest schemes makes use of 4 by 4 blocks that overlap by 50 percent in both the horizontal and vertical directions [Burt et al. 1981]. For example, Figure 35 is a 2^3 by 2^3 array, say $A(3)$, whose pixels are labeled 1–64. Figure 36 is $A(2)$ corresponding to Figure 35, and its elements are labeled A–P. The 4 by 4 neighborhood corresponding to element F in Figure 36 consists of pixels 10–13, 18–21, 26–29, and 34–37. This method implies that each block at a given level participates in four blocks at the immediately higher level. Thus the containment relations between blocks no longer form a tree. For example, pixel 28 participates in blocks F, G, J, and K in the next higher level (see Figure 37 where the four neighborhoods corresponding to F, G, J, and K are drawn as squares). In order to avoid treating border cases differently, each level is assumed to be cyclically closed (i.e., the top row at each level is adjacent to the bottom row and likewise for the columns at the extreme left and right of each level). Once again, we say that the value of a node is the average of the values of the nodes in its block on the immediately lower level. The overlapped pyramid may be compared with the QMAT (see Section 2.11) in the sense that both may result in nondisjoint decompositions of space.

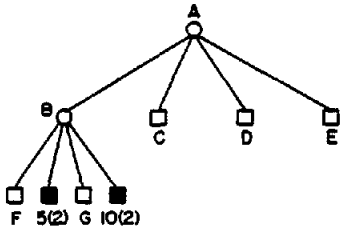
Pyramids are used for feature detection and extraction since they can be used to limit the scope of the search. Once a piece of information of interest is found at a coarse level, the finer resolution levels can be searched. This approach was used for approximate pattern matching by Davis and Roussopoulos [1980]. Pyramids can also be used for encoding information about edges, lines, and curves in an image [Shneier 1981c]. One note of caution: The reduction of resolution will affect the visual appearance of edges and small objects [Tanimoto 1976]. In particular, at a coarser level of resolution edges tend to get smeared and region separation may disappear. Pyramids have also been used as the starting point for a "split and merge" segmentation algorithm [Pietikainen et al. 1982].



(a)



(b)



(c)

Figure 33. The result of shifting the image in Figure 32 by one pixel to the right. (a) Block decomposition. (b) Tree representation. (c) QMAT.

Before leaving this section, it is important to reiterate a comment made in Section 1 that pyramids and quadtrees, although related, are different entities. A pyramid is a multiresolution representation, whereas the quadtree is a variable resolution representation. Another analogy is that the pyramid is a complete quadtree [Knuth 1975, p. 401].

2.13 Quadtree Approximation Methods

The quadtree can be used as an image approximation device. By truncating the

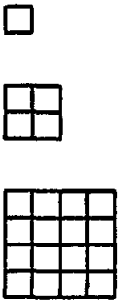


Figure 34. Structure of a pyramid having three levels.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Figure 35. Example pyramid $A(3)$.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Figure 36. $A(2)$ corresponding to Figure 35.

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	24	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Figure 37. The overlapping blocks in which pixel 28 participates.

quadtree (i.e., ignoring all nodes below a certain level), a crude approximation can be realized. Ranade et al. [1982] define a sequence of inner and outer approximations to an image and use it for shape approximation. The inner approximation consists of treating GRAY nodes as WHITE, whereas the outer approximation treats them as BLACK. Rosenfeld et al. [1982] discuss a quadtree truncation technique that treats a GRAY node as BLACK or WHITE, depending on the type of the majority of its constituent pixels. This is shown to lead to a very gentle degradation of the image in contrast to the abruptness of the inner and outer approximation methods.

Quadtree-based approximation methods have also been devised for use in transmission of binary and gray-scale images. In such a case, it is desirable for the chosen method to exhibit compression as well as be progressive. By progressive we mean that as more of the image is transmitted, the receiving device progressively constructs a better approximation. At the end of the transmission, the original image is to be reconstructed perfectly. Progressive approximation should be contrasted with facsimile techniques that transmit the image a line at a time. Thus the goal is to receive a crude picture first and the details later, thereby enabling browsing operations.

Sloan and Tanimoto [1979] (see also Tanimoto [1979]) propose a number of pyramid-based approaches to the problem of transmitting a gray-scale image. In Section 2.12 a pyramid was described as a sequence of arrays $\{A(i)\}$ such that $A(i-1)$ is a version of $A(i)$ at half the resolution; $A(0)$ is said to be a single element. Given a 2^n by 2^n image, the pyramid can be considered to be a complete quadtree with $A(n)$ corresponding to the image. The simplest transmission technique that they propose is analogous to a breadth-first traversal of the complete quadtree. The shortcoming of this approach is that redundant information must be transmitted (i.e., one-third more information), and thus no compression exists. Sloan and Tanimoto propose a number of refinements to this method. First, a level number and coordinates for each pixel are included, but are transmitted only if they

differ from the value of the pixel's predecessor. The second refinement requires that the receiver deduce one pixel's value from those of its predecessor and its three siblings. By using such a method, there is no need for redundant pixel information to be transmitted; however, there is no compression since the amount of information transmitted is equal to the number of pixels. The predecessor's value can be a sum of the values of the four sons; an even better method, in the sense that less computational overhead is involved, is simply to use one of the values of the sons.

Knowlton [1980] discusses techniques for transmission of both gray-scale and binary images. He makes use of a binary tree version of a quadtree (i.e. bintree). In essence, an image is repeatedly split into two halves alternating between vertical and horizontal splits. For gray-scale images, he describes each two-pixel group (hence the binary subdivision) by two numbers such that the first is analogous to an average, termed a *composite value*, and the second is like a differentiator that enables the computation of their corresponding intensities. These two pixel groups are recursively aggregated in groups of two to form a binary tree. Knowlton shows that all that needs to be transmitted is the composite value for the root of the tree and the successive sets of differentiators. Thus the sequence of transmission is a breadth-first traversal of the binary tree of differentiator values. The result is that an image of p pixels of 2^b gray levels can be transmitted and reformatted by using $p \cdot b$ bits. Use of the composite values leads to successively better approximations to the image until an exact reconstruction is obtained at the end of the transmission. Compression can be achieved by using Huffman codes [Huffman 1952] to encode the differentiator values.

Knowlton also presents a technique for the progressive transmission of binary images. Again, the binary tree version of the quadtree is used to represent the image. Nodes are labeled BLACK, WHITE, or GRAY. The basic unit of decomposition is a pixel and these are aggregated into 2 by 3 rectangles. At this level all nodes are described by using a seven-valued entity corresponding to the number of constituent BLACK pixels. The image is transmitted in order of a breadth-first traversal of the binary tree. Whenever a block of size greater than 2 by 3 is described as BLACK or WHITE, it ceases to participate in the remainder of the transmission process. In order to obtain an approximation, two values are transmitted for the 2 by 3 GRAY blocks. The first set of values is just a five-valued number indicating the shade of GRAY for each block. Next, the exact details of each pixel in each block are transmitted. Knowlton makes considerable use of coding methods to obtain compression factors as high as 8:1 (i.e., for a 2^n and 2^n image, instead of transmitting 2^{2n} bits, as few as 2^{2n-3} bits are necessary). It should be noted that these high compression factors do not necessarily result from the use of a bintree over a quadtree. Instead, they result from the uniformity of the image (i.e., blocks of WHITE and BLACK), and when this is not the case, then they result from coding groups of pixels. Somewhat similar compression results, although they do not exhibit progressiveness, have been obtained by using DF-expressions [Kawaguchi and Endo 1980].

The notion of a forest (see Section 2.2) has been extended by Samet [Samet 1985c] to develop a sequence of approximations to a quadtree-encoded binary image that also exhibits compression as well as progressiveness. The approximation sequence consists of using the roots of the elements of the forest of Jones and Iyengar [1984]. Each successive approximation constructs a new forest for each element of the previous approximation that is not a terminal node. The only difference is that the second approximation uses a forest of maximal squares that span the WHITE area of the components of the first approximation. This process is repeated, alternating BLACK and WHITE approximations until all elements of the approximation are terminal nodes. This method works by alternating an overestimation of the BLACK area with an underestimation, and spiraling in to the true image. Thus it avoids the one-sidedness of the inner and outer approximation methods of Ranade et al.

[1982]. The nodes comprising the elements of the approximation sequences are specified by use of locational codes.

It can be shown that use of forest-based approximation methods leads to a number of interesting properties. First, the total number of nodes in the approximation sequence does not exceed the minimum of the BLACK or WHITE nodes in the original quadtree. Using a 512 by 512 image, reductions as high as 22 percent (with respect to the minimum of the BLACK or WHITE nodes) in the number of nodes have been obtained [Samet 1985c]. As larger images are used, the compression factor becomes considerably greater. Second, the methods yield a saving of space whenever the situation arises that three out of four sons have the same type (i.e., BLACK or WHITE). The worst-case scenario from a node-counting standpoint is the checkerboard (all the BLACK nodes must be transmitted). Finally, we observe that the forest method is biased in favor of approximating objects having the shape of a "panhandle," whereas the inner and outer approximations [Ranade et al. 1982] are insensitive to them.

Ismail and Steele [1980] make use of an approximation method, termed *aplc*, that is similar in spirit to the forest method [Samet 1985c]. They treat each M by M block in the image as BLACK if at least $M^2 - 1$ of its constituent pixels are BLACK. Similarly, a M by M block is treated as WHITE if at least $M^2 - 1$ of its pixels are WHITE. Otherwise, the block is decomposed into four blocks, and the same coding process is recursively attempted. The principal difference between the two methods is that the forest method is hierarchical, whereas the aplc approximation is not in that if four brother blocks of size 2 by 2 each contain one WHITE pixel, they are treated as four 2 by 2 BLACK blocks and not as one 4 by 4 BLACK block. Also, the aplc method does not lead to an exact reconstruction of the image, whereas the forest method does.

2.14 Volume Data

Extension of the quadtree to represent three-dimensional objects by use of octrees has been proposed independently by many researchers [Hunter 1978; Jackins and Tanimoto 1980; Meagher 1982; Reddy and Rubin 1978]. The process begins with a 2^n by 2^n by 2^n object array of unit cubes or voxels [Jackins and Tanimoto 1980] (also termed obels [Meagher 1982]). The octree is an approach to object representation similar to the region quadtree, and is based on the successive subdivision of an object array into octants. If the array does not consist entirely of 1's or entirely of 0's, then it is subdivided into octants, suboctants, etc. until cubes (possibly single voxels) are obtained that consist of 1's or of 0's; that is, they are entirely contained in the region or entirely disjoint from it. This process is represented by a tree of out degree 8 in which the root node represents the entire object with octants labeled as in Figure 38, and the leaf nodes correspond to those cubes of the array for which no further subdivision is necessary. Leaf nodes are said to be BLACK or WHITE (alternatively, FULL or VOID), depending on whether their corresponding cubes are entirely within or outside of the object, respectively. All nonleaf nodes are said to be GRAY. Figure 39 contains an example object in the form of a staircase and its corresponding octree. The labels denote the octant numbers associated with each son by using the labeling convention of Figure 38.

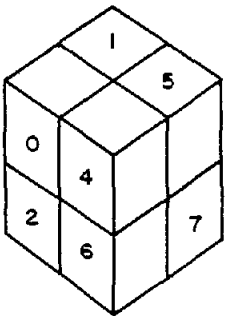


Figure 38. *Labeling of octants in an octree (octant 3 is not visible).*

Many of the algorithms obtained for the region quadtree, for example, the Boolean operations, are easily extended to the octree domain. Jackins and Tanimoto [1980] have

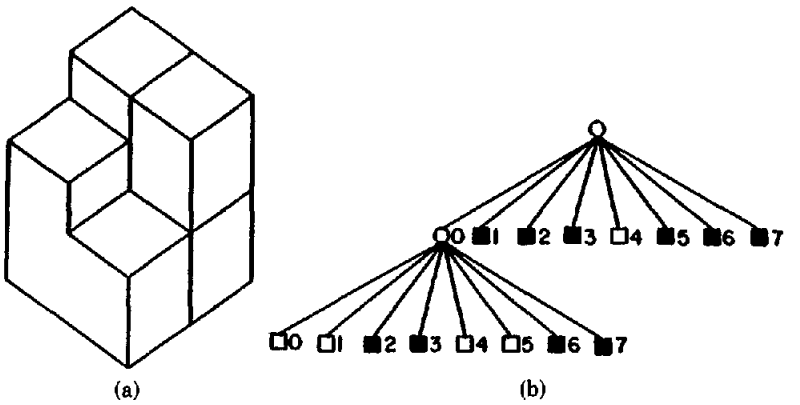


Figure 39. Example object (a) and its octree (b). ■ = BLACK = "Full"; □ = WHITE = "VOID" (empty); ○ = GRAY.

adapted Hunter and Steiglitz's [1979a] translation algorithm to the three-dimensional domain. They also discuss rotation by multiples of 90 degrees. Meagher [1982] and Ahuja and Nash [1984] discuss different approaches. Meagher [1982] also describes algorithms to perform scaling, rotation, perspective transformation, and hidden surface display. He observes that memory and processing time requirements for operations involving three-dimensional objects are proportional to the surface area. This is analogous to Hunter and Steiglitz' [1979a] observation that for quadtrees these requirements are proportional to the perimeter of the object being represented. Tamminen and Samet [1984] show how to convert a boundary representation of a solid to its corresponding octree by use of connectivity labeling. Gillespie and Davis [1981] discuss the projection of an octree onto a plane formed by the axes resulting in a quadtree. This is useful for display purposes (see also Doctor and Torborg 1981]). Yau [1984] solves the projection problem for sections orthogonal to a coordinate axis as well as the more general problem of a projection onto a plane of arbitrary position and orientation. Yau and Srihari [1983] present an algorithm for constructing an N-dimensional octreelike representation from multiple $(N-1)$-dimensional cross-sectional images for use in processing medical images. Connolly [1984] treats a similar problem by using range data. Gargantini [1982b] makes use of a pointerless representation termed a *linear octree* (analogous to the linear quadtree [Gargantini 1982a]) and shows how a number of primitive operations can be performed.

Reddy and Rubin [1978] discuss three representations for solid objects, one of which is the octree. The second is a three-dimensional generalization of the point quadtree [Finkel and Bentley 1974], that is, a decomposition into rectangular parallelepipeds (as opposed to cubes) with planes perpendicular to the x, y, and z axes. The third breaks the object into rectangular parallelepipeds that are not necessarily aligned with an axis. The parallelepipeds are of an arbitrary size and orientation. The top level of the tree has a branching factor of N. At this level of the tree, they store N transformation matrices, T_1 through T_N, where each matrix is a 4 by 4 transformation that converts the object space point into the coordinate system of its parallelepiped. Each parallelepiped is recursively subdivided into parallelepipeds in the coordinate space of the enclosing parallelepiped. Reddy and Rubin prefer the third approach for its ease of display. Situated somewhere in between the second and third approaches outlined above is the method of Brooks and Lozano-Perez [1983] (see also Lozano-Perez 1981]), who use a recursive decomposition of space into an arbitrary number of rectangular parallelepipeds, with planes perpendicular to the x, y, and z axes, to model space in solving the *findpath* or

mover's problem in robotics. This problem arises when planning the motion of a robot in an environment containing known obstacles and the desired solution is a collision-free path that is obtained by use of a search. Faverjon [1984] discusses an approach to this problem that uses an octree.

Faugeras and Ponce [1983] describe a hierarchical data structure that they term a *prism tree*. It is a ternary tree structure that is a generalization of the strip tree of Ballard [1981] (see Section 4) to hierarchically approximate surfaces by using a prism as an enclosing box. The prism tree is built from an initial triangulation of an object by using a polyhedral approximation algorithm [Faugeras et al. 1984]. Algorithms are presented for intersecting surfaces and finding neighbors in the sense of Samet [1982a].

Representing surfaces (i.e., $2\frac{1}{2}$-dimensional images) by hierarchical methods is an interesting area in which, unfortunately, only a limited amount of work has been done. DeFloriani et al. [1982] discuss a data structure for multilevel surface representation consisting of a nested triangulated irregular network [Lee and Schachter 1980] that is used for surface interpolation and also serves as a data compression mechanism. Gomez and Guzman [1979] use a data structure that is somewhat related to the point quadtree. It is a recursive subdivision of the surface into four triangles of unequal size, which uses a process that stops when a triangle matches the surface within a prespecified error. Carlson [1982] describes a quadtree-based data structure for representing surfaces to be used in the synthesis of three-dimensional objects in the domain of computer graphics. In general, the principle of recursive subdivision is of considerable importance in the processing of curved surfaces [Cohen et al. 1980]. (See also Mudur and Koparkar [1984].)

Multidimensional data in excess of three dimensions can also be represented by n-dimensional generalizations of the quadtree. Also interesting is the use of the fourth dimension to represent time [Gillespie and Davis 1981; Jackins and Tanimoto 1983; Yau and Srihari 1983]; however, this representation is somewhat difficult since the dimensional units of the extra dimension are different [i.e., units of time instead of distance). Such techniques are potentially useful in dealing with time-varying imagery.

3. POINT DATA

Multidimensional point data can be represented in a variety of ways. The representation ultimately chosen for a specific task will be heavily influenced by the type of operations to be performed on the data. Our focus is on dynamic files (i.e., the number of data can grow and shrink at will) and applications involving search. Knuth [1973] lists three typical queries: (1) a *point query*, which determines whether a given data point is in the database, and if so, the address corresponding to it; (2) a *range query* (i.e., region search), which asks for a set of data points within a given range (this category includes the partially specified query); (3) a *Boolean query*, which consists of the previous type combined with the Boolean operations AND, OR, NOT, etc. A related operation is to find the n nearest neighbors of a given point [Bentley 1975a].

Nievergelt et al. [1984] group searching techniques into two categories: those that organize the data to be stored and those that organize the embedding space from which the data are drawn. In a more formal sense, the distinction is between *trees* and *tries*, respectively. The binary search tree [Knuth 1973] is an example of the former since the boundaries of different regions in the search space are determined by the data being stored. Address computation methods such as radix searching [Knuth 1973] (also known as digital searching) are examples of the latter, since region boundaries are drawn at locations that are fixed regardless of the content of the file. The distinction can also be seen by comparing the region quadtree [Klinger 1971] with the point quadtree [Finkel and Bentley 1974]; that is, the former is based on a regular decomposition, whereas the latter is not.

In the remainder of this section we further elaborate on the point quadtree and the k-d tree [Bentley 1975b]. Then, some representations that are based on the re-

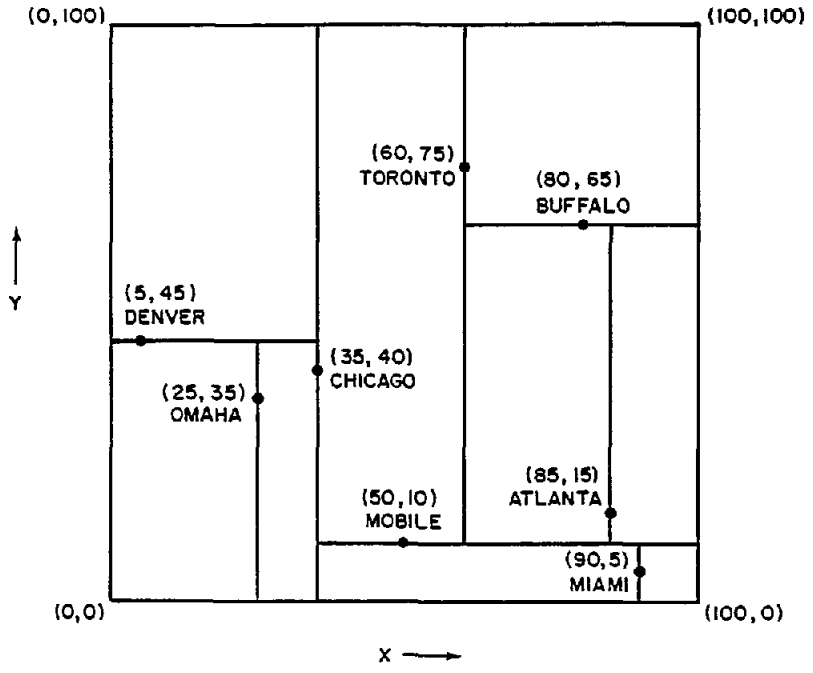
gion quadtree (i.e., on a regular decomposition) are discussed and compared with the point quadtree. We also present an application of a region-based quadtree in representing small rectangles for very large-scale integration (VLSI) applications [Kedem 1981], concluding with a brief overview of methods that replace the hierarchical structure of quadtrees by address computation. These techniques are directed, in part, toward ensuring efficient access to disk data, and are termed *bucket methods*. In the same context some tree-based methods are also discussed. All of the examples are limited to two dimensions although they can be easily generalized to an arbitrary number of dimensions. It should be borne in mind that our presentation is very brief; that is, we do not analyze the performance of these methods. Actually, the field of multidimensional data structures is a rapidly developing one, and this discussion is necessarily limited to a detailed presentation of methods that can be viewed as direct applications of a quadtreelike recursive subdivision approach.

3.1 Point Quadtrees and *k*-d Trees

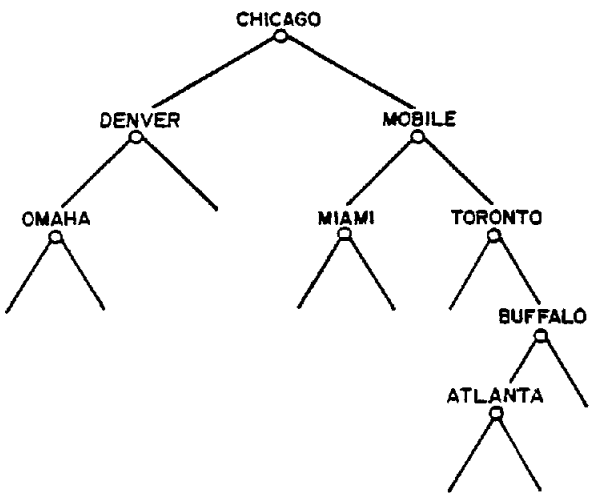
The point quadtree [Finkel and Bentley 1974] is a multidimensional generalization of a binary search tree. In two dimensions, each data point is a node in a tree having four sons, which are roots of subtrees corresponding to quadrants labeled in order NW, NE, SW, and SE. Each data point is assumed to be unique. The process of inserting into point quadtrees is analogous to that used for binary search trees. In essence, we search for the desired record on the basis of its x and y coordinates. At each node of the tree a four-way comparison operation is performed and the appropriate subtree is chosen for the next test. Reaching the bottom of the tree without finding the record means that it should be inserted at this position. The shape of the resulting tree depends on the order in which records are inserted into it. For example, the tree in Figure 2 is the point quadtree for the sequence Chicago, Mobile, Toronto, Buffalo, Denver, Omaha, Atlanta, and Miami. Deletion of a node is more complex [Samet 1980c] as is balancing [Overmars and van Leeuwen 1982].

Point quadtrees are especially attractive in applications that involve search. However, they have also been used to solve a measure problem for rectangular ranges in three-space [van Leeuwen and Wood 1981]. A typical query is one that requests the determination of all records within a specified distance of a given record, that is, all cities within 50 miles of Washington, D.C. The efficiency of the point quadtree lies in its role as a pruning device on the amount of search that is required. Thus many records will not need to be examined. For example, suppose that in the hypothetical database of Figure 2 we wish to find all cities within eight units of a data point with coordinates (83, 10). In such a case, there is no need to search the NW, NE, and SW quadrants of the root (i.e., Chicago with coordinates (35, 40)). Thus we can restrict our search to the SE quadrant of the tree rooted at Chicago. Similarly, there is no need to search the NW and SW quadrants of the tree rooted at Mobile (i.e., coordinates (50, 10)). Search operations using point quadtrees are analyzed by Bentley and Stanat [1975] and Lee and Wong [1977]. Note that the search ranges are usually orthogonally defined regions such as rectangles or boxes. Other shapes are also feasible as the above example demonstrated (i.e., a circle). In order to handle more complex search regions such as polygons, Willard [1982] defines a *polygon tree* where the x–y plane is subdivided by J lines that need not be orthogonal, although there are other restrictions on these lines. When $J = 2$, the result is a point quadtree with nonorthogonal axes.

Our examples of the use of the point quadtree have been limited to two dimensions. The problem with a large number of dimensions is that the branching factor becomes very large (i.e., 2^k for k dimensions), thereby requiring much storage for each node as well as many NIL pointers for terminal nodes. The k-d tree of Bentley [Bentley 1975b] is an improvement on the point quadtree that avoids the large branching factor. In principle, it is a binary search tree with the distinction that at each



(a)



(b)

Figure 40. A k-d tree (b) and the records it represents (a).

level of the tree a different coordinate is tested when determining the direction in which a branch is to be made. Therefore in the two-dimensional case (i.e., a 2-d tree!), we compare x coordinates at the root and at even levels (assuming the root is at level 0) and y coordinates at odd levels. Each node has two sons. Figure 40 is the k-d tree corresponding to the point quadtree of Figure 2, where the records have been inserted

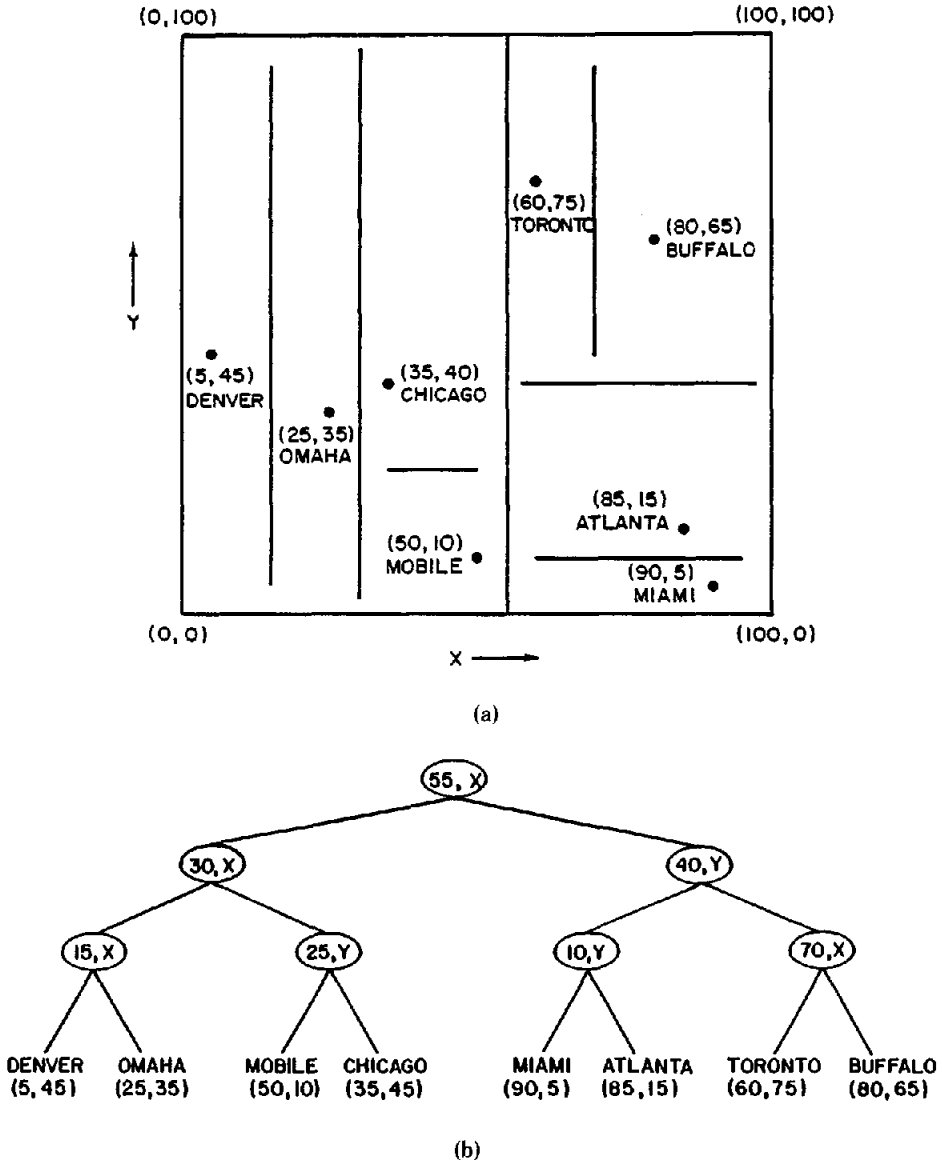


Figure 41. Adaptive k-d tree. (a) Set of points in 2-space. (b) 2-d tree.

in the same order. Friedman et al. [1977] report an improvement on the k-d tree that relaxes the requirement of alternating tests at the price of storing at each node an indication of which coordinate is being tested. Using this data structure, termed an *adaptive k-d tree*, we can construct a balanced k-d tree where records are stored only at the terminal nodes. Figure 41 is the adaptive k-d tree corresponding to the point quadtree of Figure 2. Before constructing such a tree we must know all of the constituent records. Thus its shape is independent of the order in which the records were encountered. However, adding a new record requires rebuilding the tree. Thus it is not a dynamic data structure.

In general, k-d trees are superior to point quadtrees, with one exception: The point quadtree is an inherently parallel data

structure and thus the comparison operation can be performed in parallel for the k key values, whereas this cannot be done for the k-d tree. Thus we can characterize the k-d tree as a superior serial data structure and the point quadtree as a superior parallel data structure. Linn [1973] discusses the use of point quadtrees in a multiprocessor environment.

3.2 Region-Based Quadtrees

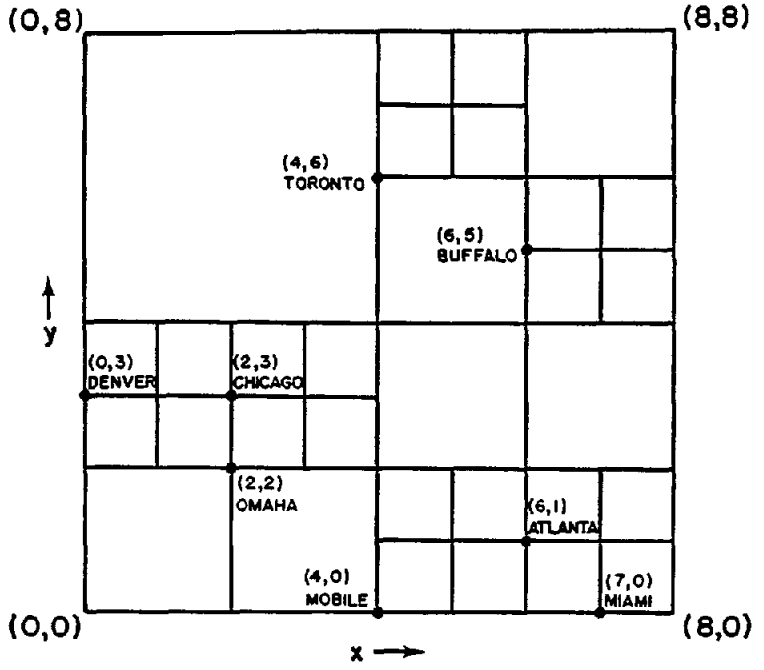
Although conceivably there are many ways of adapting the region quadtree to represent point data, our discussion is limited to two methods. The first method assumes that the domain of data points is discrete; they are treated as if they are BLACK pixels in a region quadtree. An alternative characterization is to think of the data points as nonzero elements in a square matrix. The resulting data structure is called an *MX quadtree* (MX for matrix), although the term *MX quadtrie* would probably be more appropriate. The MX quadtree is organized in a similar way to the region quadtree. The difference is that leaf nodes are BLACK or empty (i.e., WHITE) corresponding to the presence or absence, respectively, of a data point in the appropriate position in the matrix. For example, Figure 42 is the 2^3 by 2^3 MX quadtree corresponding to the data of Figure 2. It is obtained by applying the mapping f such that $f(Z) = Z$ div 12.5 to both x and y coordinates. The result of the mapping is reflected in the coordinate values in the figure.

Each data point in an MX quadtree corresponds to a 1 by 1 square. For ease of notation and operation using modulo and integer division operations, the data point is associated with the lower left corner of the square. This adheres to the general convention followed throughout this presentation that the NE and SE quadrants are closed with respect to the x coordinate and the NW and NE quadrants are closed with respect to the y coordinate. Note that nodes corresponding to data points are not merged, whereas this is not the case for empty leaf nodes. For example, the NW and NE sons of node D in Figure 42 are NIL and likewise for the NW son of node A. However, it is undesirable to merge nodes corresponding to data points as this results in a loss of the identifying information about the data points. Recall that each data point is different, whereas the empty leaf nodes have the absence of information as their common property and thus can be safely merged.

Data points are inserted into an MX quadtree by searching for them. This search is based on the location of the data point in the matrix (e.g., the discretized values of its x and y coordinate in the example of Figure 42). An unsuccessful search terminates at a leaf node. If this leaf node is NIL, the space spanned by it may have to be repeatedly subdivided until it is a 1 by 1 square. This process is termed *splitting* and for a 2^n by 2^n MX quadtree, it will have to be performed at most n times. The shape of the MX quadtree is independent of the order in which data points are inserted into it. Deletion of nodes is slightly more complex and may require collapsing of nodes—the direct counterpart of the node-splitting process outlined above.

The MX quadtree is useful in a number of applications. It serves as a basis of a quadtree matrix manipulation system [Samet and Krishnamurthy 1983]. It is used by Letelier [1983] to represent silhouettes of hand motions to aid in the telephonic transmission of sign language for the hearing impaired. DeCoulon and Johnsen [1976] describe its use in the coding of black and white facsimiles for efficient transmission.

The MX quadtree is adequate as long as the domain of the data points is discrete and finite. If this is not the case, then the data points cannot be represented since the minimum separation between the data points is unknown. This leads us to an alternative adaptation of the region quadtree to point data that associates data points (that need not be discrete) with quadrants. We call it a *PR quadtree* (P for point and R for region) although again the term *PR quadtrie* would probably be more appropriate. The PR quadtree is organized in the same way as the region quadtree. The difference is that leaf nodes are either empty (i.e., WHITE) or contain a data



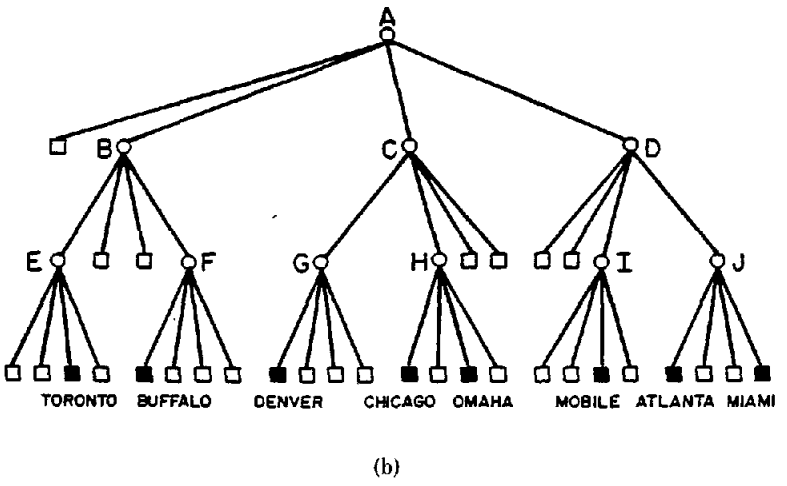


Figure 42. A MX quadtree (b) and the records it represents (a).

point (i.e., BLACK) and its coordinates. A quadrant contains at most one data point. For example, Figure 43 is the PR quadtree corresponding to the data of Figure 2. Orenstein [1982] describes an analogous data structure using binary trees rather than quadtrees. Such a data structure could be called a *k-d PR quadtree* or even better simply a *k-d trie*.

Data points are inserted into PR quadtrees in a manner analogous to that used to insert in a point quadtree; that is, a search is made for them. Actually, the search is for the quadrant in which the data point,

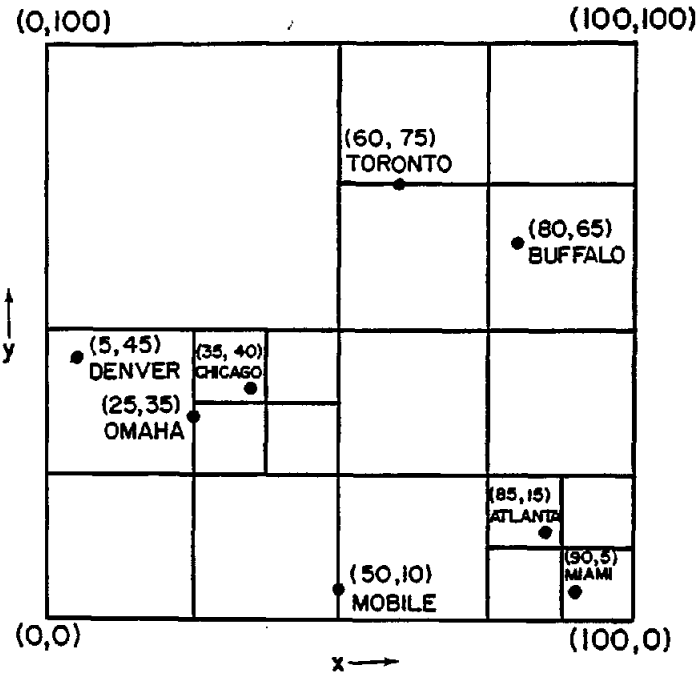


Figure 43. A PR quadtree (b) and the records it represents (a).

say A, belongs (i.e., a leaf node). If the quadrant is already occupied by another data point with different x and y coordinates, say B, then the quadrant must repeatedly be subdivided (termed *splitting*) until nodes A and B no longer occupy the same quadrant. This may result in many subdivisions, especially if the Euclidean distance between A and B is very small. The shape of the resulting PR quadtree is independent of the order in which data points are inserted into it. Deletion of nodes is more complex and may require collapsing of nodes, that is, the direct counterpart of the node-splitting process outlined above.

Matsuyama et al. [1984] discuss the use of a PR quadtree in partitioning a point space into "buckets" of a finite capacity. As

a bucket overflows, a partition into four, equal-sized squares is made. Anderson [1983] makes use of a PR quadtree (termed a *uniform* quadtree) to store endpoints of line segments to be drawn by a plotter. The goal is to minimize pen plotting time by choosing the line segment to be output next whose end point is closest to the current pen position. Samet and Webber [1983] represent polygonal maps, for example, Voronoi diagrams, using a variant of the PR quadtree. It has the advantage that edges are represented exactly, thereby avoiding the edge width problem associated with the methods of Hunter and Steiglitz [1979a] for polygons.

3.3 Comparison of Point Quadtrees and Region-Based Quadtrees

The comparison of the MX, PR, and point quadtrees reduces, in part, to a comparison of their respective decomposition methods. A major difference among the three data structures is in the size of the regions associated with each data point. For the point quadtree there is no a priori constraint on the size of the space spanned by the quadtree (i.e., the x and y coordinates of the data points). For both the MX and PR quadtrees the space spanned by the quadtree is constrained to a maximum width and height. All three quadtrees result in the association of one rectangular region with each data point. The point quadtree produces a rectangle that may, at times, be of infinite width and height. For the MX quadtree this region must be a square with a particular size associated with it. This size is fixed at the time the MX quadtree is defined and is the minimum permissible separation between two data points in the domain of the MX quadtree (equivalently, it is the maximum number of elements permitted in each row and column of the corresponding matrix). The PR quadtree also has a square region, and its size depends on what other data points are currently represented by nodes in the quadtree. In the case of the MX quadtree there is a fixed discrete coordinate system associated with the space spanned by the quadtree, whereas no such limitation exists for the PR quadtree. The advantage of such a fixed coordinate system is that there is no need to store coordinate information with a data point's leaf node. The disadvantage is that the discretization of the domain of the data points limits the differentiation between data points.

The size and shape of a quadtree are important from the standpoint of efficiency of both storage and search operations. The size and shape of the point quadtree are extremely sensitive to the order in which data points are inserted into it during the process of building it. This means that for a point quadtree of M records, its maximum depth is $M - 1$ (i.e., one record is stored at each level in the tree), whereas its minimum depth is $\lfloor \log_4(3 \cdot M) \rfloor$ (i.e., each level in the tree is completely full), where we assume that the root of the tree has a depth of 0. In contrast, the shape and size of the MX and PR quadtrees are independent of the insertion order. For the MX quadtree all nodes corresponding to data points appear at the same depth in the quadtree. The depth of the MX quadtree depends on the size of the space spanned by the quadtree and the maximum number of elements permitted in each row and column of the corresponding matrix. For example, for a 2^n by 2^n matrix, all data points will appear as leaf nodes at a depth of n. The size and shape of the PR quadtree depend on the data points currently in the quadtree. The minimum depth of a PR quadtree for $M > 1$ data points is $\lceil \log_4(M - 1) \rceil$ (i.e., all the data points are at the same level), whereas there is no upper bound on the depth in terms of the number of data points. In particular, for a square region of side length s, such that the minimum Euclidean distance separating two points is d, the maximum depth of the quadtree can be as high as $\lceil \log_2((s/d) \cdot \sqrt{2}) \rceil$.

The volume of data also affects the comparison among the three quadtrees. When the volume is very high, the MX quadtree loses some of its advantage since an array representation may be more economical in terms of space, as there is no need for links. Whereas the size of the PR quadtree was seen to be affected by clustering of data points, especially when the number of data

points is relatively small, this is not a factor in the size of a point quadtree. However, when the volume of data is large and is uniformly distributed, the effect of clustering is lessened and there should not be much difference in storage efficiency between the point and PR quadtrees.

3.4 CIF Quadtrees

The *MX-CIF quadtree* is a quadtreelike data structure devised by Kedem [1981] (and called a *quad-CIF tree*, where CIF denotes Caltech Intermediate Form) for representing a large set of very small rectangles for application in VLSI design rule checking. The goal is to locate rapidly a collection of all objects that intersect a given rectangle. An equivalent problem is to insert a rectangle into the data structure under the restriction that it does not intersect existing rectangles. The MX-CIF quadtree is organized in a similar way to the region quadtree. A region is repeatedly subdivided into four equal-sized quadrants until blocks are obtained that do not contain rectangles. As the subdivision takes place, a set containing all of the rectangles that intersect the lines passing through it is associated with each subdivision point. For example, Figure 44 contains a set of rectangles and its corresponding MX-CIF quadtree. Once a rectangle is associated with a subdivision point, say P, it is not considered to be a member of any of the sons of the node corresponding to P. For example, in Figure 44, node D spans a space that contains both rectangles 11 and 12. However, only rectangle 11 is associated with node D, whereas rectangle 12 is associated with node F.

Our definition of an MX-CIF quadtree is very similar to that of an MX quadtree with the following differences. First, data are associated with both terminal and nonterminal nodes. Nevertheless, the analog of a WHITE node is present and is a NIL pointer in the direction of a quadrant that contains no rectangles. Second, we are representing rectangles rather than points. This is fortunate because it provides a termination condition for the subdivision process in forming an MX quadtree. Section 3.2 demonstrates that MX quadtrees are defined for a domain whose elements must be subdivision points of the space being represented. The nonzero width of the rectangles ensures that they overlap with the subdivision points.

The set of the rectangles that intersect the lines passing through a subdivision point is subdivided into two sets. For example, consider subdivision point P centered at (CX, CY) that partitions a $2 \cdot LX$ by $2 \cdot LY$ rectangular area. All input rectangles that intersect the line $x = CX$ form one set, and all input rectangles that intersect the line $y = CY$ form the other set. Equivalently, these sets correspond to the rectangles intersecting the y and x axes, respectively, passing through (CX, CY). If a rectangle intersects both axes (i.e., it contains the subdivision point P), then we adopt the convention that it is stored with the set associated with the y axis. These subsets are implemented as binary trees, which in actuality are one-dimensional analogs of the MX quadtree. For example, Figure 45 illustrates the binary tree associated with the x and y axes passing through A, the root of the MX-CIF quadtree of Figure 44.

Rectangles are inserted into an MX-CIF quadtree by searching for the position that they are to occupy. We assume that the input rectangle does not overlap any of the existing rectangles. This position is determined by a two-step process. First, the first subdivision point must be located such that at least one of its axis lines (i.e., the quadrant lines emanating from the subdivision point) intersects the input rectangle. Second, having found such a point and an axis, say point P and axis V, the subdivision process is repeated for the V axis until the first subdivision point that is contained within the rectangle is located. During the process of locating the destination position for the input rectangle, the space spanned by the MX-CIF quadtree may have to be repeatedly subdivided (termed *splitting*), creating new nodes in the process. As was the case for the MX quadtree, the shape of the resulting MX-CIF quadtree is independent of the order in which the rectangles are inserted into it. Deletion of nodes is more complex and may require collapsing

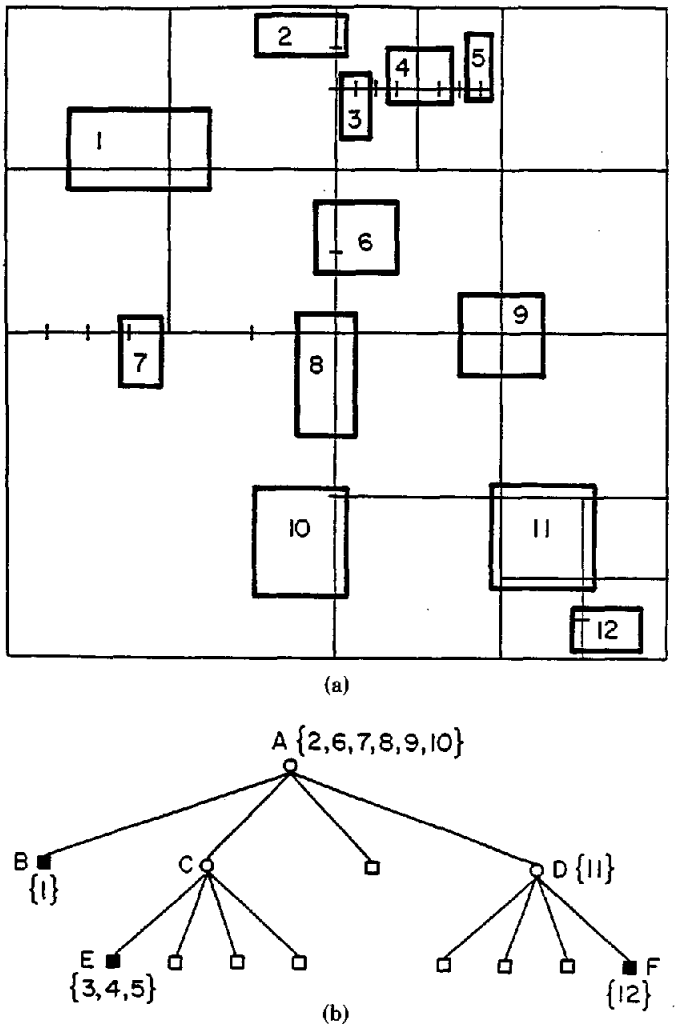


Figure 44. A MX-CIF quadtree (b) and the rectangles it represents (a).

of nodes, that is, the direct counterpart of the node-splitting process outlined above.

The most common search query is one that seeks to determine whether a given rectangle overlaps (i.e., intersects) any of the existing rectangles. It is a prerequisite to the successful insertion of a rectangle. Range queries can also be performed. However, they are more usefully cast in terms of finding all the rectangles within a given area. Another popular query seeks to determine whether one collection of rectangles can be overlaid on another collection without any of the component rectangles intersecting one another. These two operations can be implemented by using variants of algorithms developed for handling set operations (i.e., union and intersection) in region-based quadtrees [Hunter and Steiglitz 1979a; Shneier 1981a]. In particular, the range query can be answered by intersecting the query rectangle with the MX-CIF quadtree. The overlay query can be answered by a two-step process. The two MX-CIF quadtrees are first intersected. If the result is empty, then they can be safely

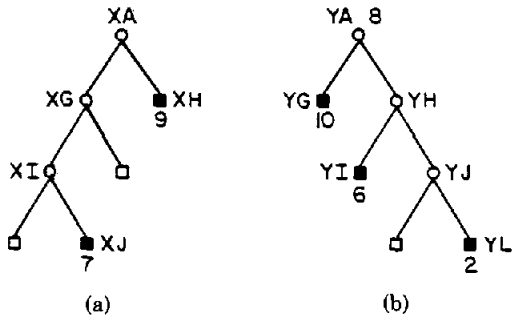


Figure 45. Binary trees for the (a) x axis and (b) y axis passing through node A in Figure 44.

overlaid and all that is needed is to perform a union of the two MX-CIF quadtrees. Boolean queries can also be handled easily.

The problem of determining whether a given rectangle overlaps any of an existing set of rectangles by use of quadtrees is also addressed by Abel and Smith [1983]. Each rectangle is associated with the node corresponding to the smallest block that totally encloses it (i.e., its minimum bounding "quadrant"). This is the same representation used by Kedem, except that Kedem uses binary trees to organize the rectangle associated with each quadtree node. Abel and Smith do not use a pointer-based quadtree representation. Instead, they represent each rectangle by the locational code (see Section 2.2) of its node. These codes are subsequently organized in a B^+-tree [Comer 1979]. Note that many rectangles have identical locational codes, and thus the rectangle dimensions must also be stored along with the locational codes in the B^+-tree. For example, for the set of rectangles in Figure 44, rectangles 2, 6, 7, 8, 9, and 10 have the same locational code as do rectangles 3, 4, and 5, albeit a different one. Hinrichs and Nievergelt [1983] describe another approach to this problem that is based on use of the Grid File, a hierarchical organization of point data, discussed in Section 3.5.

3.5 Bucket Methods

All of the data structures discussed above are primarily designed for in core applications, although their uses can be extended elsewhere. The problem is that when data are stored in external storage, the need to follow pointers may lead to page faults. To overcome this, methods have been designed that collect the points into sets (termed *buckets*) corresponding to the storage unit (i.e., page) of the disk. The remaining task is to organize the access to these buckets; this is often done by replacing the tree structure with an array, thereby facilitating address computation. We term such techniques *bucket methods*, and their aim is to ensure efficient access to disk data. The simplest bucket method is the fixed-grid (or cell) method [Knuth 1973, p. 554; Bentley and Friedman 1979], which is popular among cartographers. It divides the space into equal-sized cells (i.e., squares and cubes for two- and three-dimensional data, respectively) having the width equal to the search radius. If data are sought by using only a fixed-search radius, then the fixed grid is an efficient structure. It is also efficient when points are uniformly distributed (it corresponds to hashing [Knuth 1973]). For a nonuniform distribution it is less efficient, because buckets may be unevenly filled, leading to nearly empty pages as well as long overflow chains. The data structure is essentially a directory in the form of a k-dimensional array with one entry per cell. Each cell may be implemented as a linked list to represent the points within it. Figure 46 is an example in which a grid representation for the data of Figure 2 is shown for a search radius consisting of a square of size 20 by 20; that is, by assuming a 100 by 100 coordinate space, we have 25 squares of equal size. Its deficiency is a fixed size for the blocks, which results in both overflow and underflow. The methods presented below are examples of attempts to address this deficiency from both hierarchical and nonhierarchical viewpoints. We conclude with a discussion of some related work from the hashing area.

The *Grid File* of Nievergelt et al. [1984] is a variation of the grid method, which relaxes the requirement that cell division lines be equidistant. Its goal is to retrieve records with at most two disk accesses. This is done by using a grid directory consisting of grid blocks, which are analogous to the

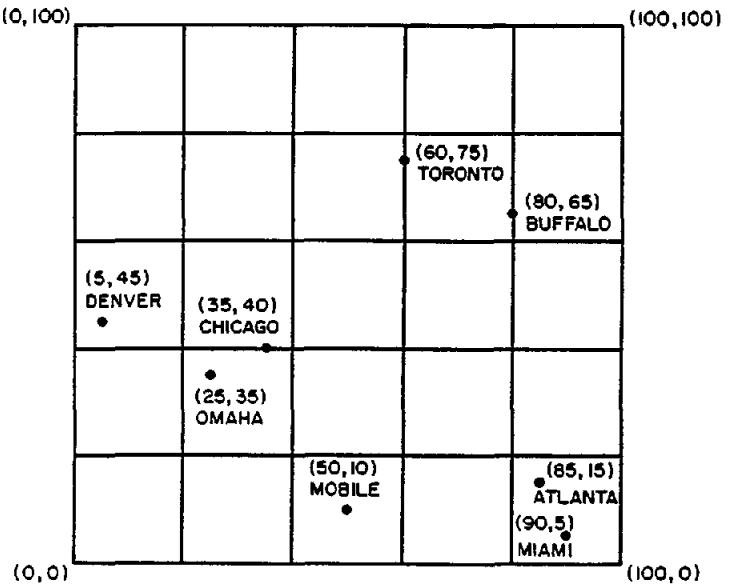


Figure 46. Grid representation corresponding to Figure 2 with a search radius of 20.

cells of the fixed-grid method. All records in one grid block are stored in the same bucket. However, several grid blocks can share a bucket as long as the union of these grid blocks forms a k-dimensional rectangle (i.e., a convex region) in the space of records. Although the regions of the buckets are piecewise disjoint, together they span the space of records.

The purpose of the grid directory is to maintain a dynamic correspondence between the grid blocks in the record space and the data buckets. The grid directory consists of two parts. The first is a dynamic k-dimensional array, which contains one entry for each grid block. The values of the elements are pointers to the relevant data buckets. Usually buckets will have a capacity of 10–1000 records. Thus the entry in the grid directory is small in comparison to a bucket. We are not concerned with how records are organized within a bucket (e.g., linked list and tree). The grid directory may be kept on disk. The second part of the grid directory is a set of k one-dimensional arrays called *linear scales*. These scales define a partition of the domain of each attribute and enable the accessing of the appropriate grid blocks by aiding in the computation of their address on the basis of the value of the relevant attributes. The linear scales are kept in core. It should be noted that the linear scales are useful in guiding a range query by indicating the grid directory elements that overlap the query range.

As an example, consider Figure 47, which shows the Grid File representation for the data in Figure 2. The bucket capacity is two records. There are $k = 2$ different attributes. The grid directory consists of nine grid blocks and six buckets labeled A–F. We refer to grid blocks as if they are array elements; that is, grid block (i, j) is the element in row i (starting at the bottom) and column j (starting at the left) of the grid directory. Grid blocks (2, 2), (3, 1), and (3, 3) are empty; however, they do share buckets with other grid blocks. In particular, grid block (3, 1) shares bucket D with grid block (2, 1), grid blocks (3, 2) and (3, 3) share bucket B, and grid blocks (2, 2) and (2, 3) share bucket E. The sharing is indicated by the broken lines. Figure 48 contains the linear scales for the two attributes (i.e., the x and y coordinates). For example, executing a FIND command with $x = 80$ and $y = 65$ causes the access of the bucket associated with the grid block in row 2 and

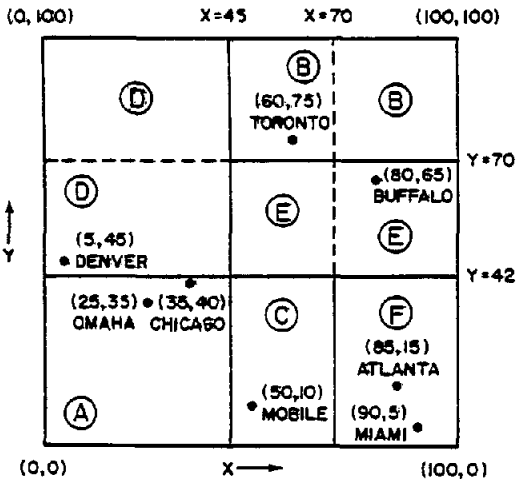


Figure 47. Grid directory for the data of Figure 2.

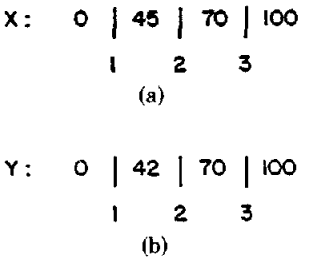


Figure 48. Linear scales for (a) x and (b) y corresponding to the grid directory of Figure 47.

column 3 of the grid directory of Figure 47.

The Grid File is attractive, in part, because of its graceful growth as more and more records are inserted. As the buckets overflow, a splitting process is applied that results in the creation of new buckets and a movement of records. Two types of bucket splits are possible. The first, and most common, occurs when several grid blocks share a bucket that has overflowed. For example, suppose Boise at (10, 80) and Fargo at (15, 75) are inserted in sequence in Figure 47. Boise is inserted in bucket D because it belongs in grid block (3, 1), which currently shares bucket D with grid block (2, 1). Fargo also belongs to grid block (3, 1); however, bucket D is now full. In this case, we merely need to allocate a new bucket and adjust the mapping between grid blocks and buckets. The second type of a split arises when we must refine a grid partition. It is triggered by an overflowing bucket, all of whose records lie in a single grid block (e.g., the overflow of bucket A upon insertion of Kansas City at (30, 30) in Figure 47). In this case there exists a choice with respect to the dimension (i.e., axis) and the location of the splitting point (i.e., we do not have to split at the midpoint of an interval).

The counterpart of splitting is merging. There are two possible instances when merging is appropriate: (1) *Bucket merging*, the most common instance, arises when a pair of neighboring buckets are empty or nearly empty and their coalescing has resulted in a convex bucket region; (2) *directory merging* arises when two adjacent cross sections in the grid directory each have identical bucket values. For example, in the case of the two-dimensional grid directory of Figure 49, where all grid blocks in column 2 are in bucket C and all grid blocks in column 3 are in bucket D, if the merging threshold is satisfied, then buckets C and D can be merged and the linear scales modified to reflect this change. Generally, directory merging is of little practical interest since, even if merging is allowed to occur, it is probable that splitting will soon have to take place.

Merrett and Otoo describe a technique termed *multipaging* [Merrett 1978; Merrett and Otoo 1982] which is very similar to the Grid File. It also uses a directory and maintains a set of linear scales called *axial arrays*. In fact, the Grid File uses multipaging as an index to a paged data structure. A database that is organized by using multipaging differs from the Grid File in that it requires bucket overflow areas. This means that it has a different bucket overflow criterion. Thus it does not guarantee that every record can be retrieved with two disk accesses. In particular, multipaging makes use of a load factor and a probe factor, which are related to the number of overflowing data items. This makes insertion and deletion (as well as bucket splitting and merging) somewhat more complicated than when the Grid File is used.

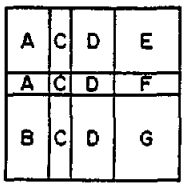


Figure 49. Example grid directory illustrating directory merging.

The EXCELL method of Tamminen [1981] is a bintree together with a directory array providing access by address computation. It can also be viewed as an adaptation of extendible hashing [Fagin et al. 1979] to multidimensional point data. It implements EXHASH, the extendible hashing hash function, by interleaving the most significant bits of the data (analogous to the locational codes discussed in Section 2.2). Similar in spirit to the Grid File, it is based on a regular decomposition and is useful in providing efficient access to, and an efficient representation of, geometric data. It also makes use of a grid directory; however, all grid blocks are of the same size. The principal difference is that grid refinement for the Grid File splits only one interval in two and results in the insertion of a $(k - 1)$-dimensional cross section. In contrast, a grid refinement for the EXCELL method splits all intervals in two (thus the partition points are fixed) for the particular dimension and results in doubling the size of the grid directory. Therefore the grid directory grows more gradually when the Grid File is used, whereas use of EXCELL reduces the need for grid refinement operations at the expense of larger directories in general because of a sensitivity to the distribution of the data. However, a large bucket size reduces the effect of nonuniformity unless the data consist entirely of a few clusters. The fact that all grid blocks define equal-sized regions (and convex as well) means that EXCELL does not require a set of linear scales to access the grid directory as is needed for the Grid File.

An example of the EXCELL method is considered in Figure 50, which shows the representation for the data in Figure 2.

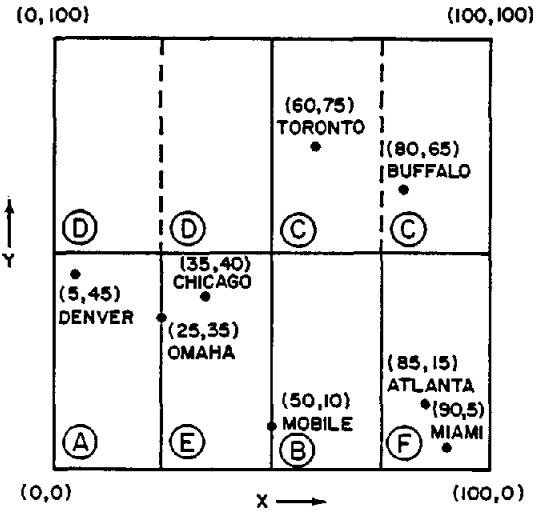


Figure 50. EXCELL representation corresponding to Figure 2.

Again, the convention is adopted that a rectangle is open with respect to its upper and right boundaries and closed with respect to its lower and left boundaries. The capacity of the bucket is two records. There are $k = 2$ different attributes. The grid directory is implemented as an array and in this case it consists of eight grid blocks (labeled in the same way as for the Grid File) and six buckets labeled A–F. Note that grid blocks (2, 3) and (2, 4) share bucket C, whereas grid blocks (2, 1) and (2, 2), despite being empty, share bucket D. The sharing is indicated by the broken lines. Furthermore, when a bucket size of 1 is used, the partition of space induced by EXCELL equals that of a PR k-d tree [Orenstein 1982].

As a database represented by the EXCELL method grows, buckets will overflow. This leads to the application of a splitting process that results in the creation of new buckets and a movement of records. As in the case of the Grid File, two types of bucket splits are possible. The first, and most common, occurs when several grid blocks share a bucket that has overflowed. In this case, a new bucket is allocated and the mapping between grid blocks and buckets is adjusted. The second type of a split arises when a grid partition must be re-

fined; this causes a doubling of the directory. It is triggered by an overflowing bucket that is not shared among several grid blocks (e.g., the overflow of bucket A upon insertion of Kansas City (30, 30) in Figure 50). The split occurs along the different attributes in a cyclic fashion (first split along attribute x, then y, then x, etc.). For both types of bucket splits, a situation may arise in which none of the elements in the overflowing buckets belongs to the newly created bucket, with the result that the directory will have to be doubled more than once. This results from the fact that the splitting points are fixed for EXCELL. For example, this will occur when we attempt to insert Kansas City at (30, 30) in Figure 50 since the first directory doubling at $y = 25$ and $y = 75$ will still have Chicago, Omaha, and Kansas City in the same grid block. Thus we see that the size of the EXCELL grid directory is sensitive to the distribution of the data. However, a large bucket size reduces the effect of nonuniformity unless the data consist entirely of a few clusters.

The counterpart of splitting is merging. However, it is considerably more limited in scope for EXCELL than for the Grid File. Also, it is less likely to arise because EXCELL has been designed primarily for use in geometrical applications in which deletion of records is not so prevalent. As with the Grid File, however, there are two cases where merging is appropriate, that is, bucket merging and directory merging.

Both the Grid File and EXCELL organize space into buckets and use directories in the form of arrays to access them. The similarity to the quadtree lies in the mappings induced by the directories (i.e., EXCELL with the region quadtree and the Grid File with the point quadtree). Trees can also be used to access the buckets [Knott 1971]. Matsuyama et al. [1984] compare a technique of accessing buckets by use of a PR quadtree with one that uses an adaptive k-d tree. Robinson [1981] introduces the *k-d-B-tree*, which is a generalization of the B-tree to allow multiattribute access. O'Rourke [1981; O'Rourke and Sloan 1984] makes use of an adaptive k-d tree, which he calls a *dynamically quantized space*, to access buckets of data for use in multidimensional histogramming to aid in the focusing of the Hough transform. Sloan [1981; O'Rourke and Sloan 1984] addresses the same problem as O'Rourke, albeit with a different data structure, which he calls a *dynamically quantized pyramid*. It is based on the pyramid data structure (see Section 2.12). Here the number of buckets is fixed. It differs from the conventional pyramid in that the partition points at the various levels are allowed to vary rather than being fixed and are adjusted as data are entered. The result is somewhat related to a complete point quadtree [Knuth 1975, p. 401] with buckets.

There has also been considerable work on representing multidimensional point data by use of linear hashing. Linear hashing [Litwin 1980] methods are attractive because they provide for linear growth of the file (i.e., one bucket at a time) without requiring a directory. In contrast, extendible hashing [Fagin et al. 1979] (e.g., EXCELL) and the Grid File methods require extra storage in the form of directories. When a bucket overflows, the directory doubles in size in the case of extendible hashing, whereas in the case of the Grid File it results in the insertion of a $(k-1)$-dimensional cross section. Neither EXCELL nor the Grid File need overflow pages, whereas methods based on linear hashing generally do, although this may be unnecessary. Bit interleaving (e.g., attributed by Bentley [1975a] to McCreight, but see the discussion of the Morton matrix [Morton 1966] in Section 2.2) is used by a number of researchers [Burkhardt 1983; Orenstein 1983; Orenstein and Merrett 1984; Ouksel and Scheuermann 1983; Tropf and Herzog 1981] to create a linear order on the multidimensional domain of the data. Tropf and Herzog [1981] and Orenstein and Merrett [1984] discuss its use in range searching. Burkhardt [1983] terms it *shuffle* order, and adapts it to linear hashing in the same way that EXCELL adapts it to extendible hashing, and uses it to evaluate range queries. Ouksel and Scheuermann [1983] call it z order. Orenstein [1983] discusses the problems associated with such an approach. He points out

that the resulting file may contain a number of sparsely filled buckets, which will result in poor performance for sequential access. He goes on to propose a modification that unfortunately, unlike linear hashing, does not result in a bucket retrieval cost of one or two disk reads (for the hash operations). In contrast, directory-based methods such as the Grid File and EXCELL do not suffer from such a problem to the same extent because, since the directory consists of grid blocks and several grid blocks can share a bucket, the sparseness issue can be avoided.

4. CURVILINEAR DATA

Section 2 was devoted to approaches to region representation that are based on descriptions of their interiors. In this section we focus on representations that specify boundaries of regions. This is done in the more general context of data structures for curvilinear data. The simplest representation is the polygon in the form of vectors [Nagy and Wagle 1979], which are usually specified in the form of lists of pairs of x and y coordinate values corresponding to their start and end points. One of the most common representations is the chain code [Freeman 1974] (described in Section 2.3), which is an approximation of a polygon. Other popular representations include raster-oriented methods [Merrill 1973; Peuquet 1979], as well as a combination of vectors and rasters (e.g., vasters [Peuquet 1983]). There has also been considerable interest recently in hierarchical representations. These are primarily based on rectangular approximations to the data [Ballard 1981; Burton 1977; Peucker 1976]. In particular, Burton [1977] uses upright rectangles, Ballard [1981] uses rectangular strips of arbitrary orientation, and Peucker [1976] uses sets of bands. There also exist methods that are based on a regular decomposition in two dimensions, as reported by Hunter and Steiglitz [1979a], Shneier [1981c], and Martin [1982]. Note that our primary focus is on the facilitation of set operations and not ease of display, which is a characterization of B-splines and Bezier methods [Cohen et al. 1980].

In many applications polygons are not unrelated, but together form a partition of the study area (termed a *polygonal map*). It is possible to use the above representations for each curve that bounds two adjacent regions. However, it is often preferable to represent the complete network of boundaries with a single hierarchical data structure. Some examples include the line quadtree of Samet and Webber [1984], the PM quadtree of Samet and Webber [1983], and the edge variant of the EXCELL method of Tamminen [1981]. In order to avoid confusion with the point space formulation of the EXCELL method, discussed in Section 3.5, we shall use the term *edge-EXCELL*. In the remainder of this section, we elaborate further on the strip tree and also on the representations that are based on a regular decomposition, concluding with a brief comparison of these methods.

4.1 Strip Trees

The *strip tree* is a hierarchical representation of a single curve that is obtained by successively approximating segments of it by enclosing rectangles. The data structure consists of a binary tree whose root represents the bounding rectangle of the entire curve. For example, consider Figure 51, where the curve between points P and Q, at locations (x_P, y_P) and (x_Q, y_Q), respectively, is modeled by a strip tree. The rectangle associated with the root, A in this example, corresponds to a rectangular strip of maximum width, enclosing the curve, whose sides are parallel to the line joining the endpoints of the curve (i.e., P and Q). Next, this rectangle is decomposed into two parts at one of the points (termed a *splitting point*) on the rectangle that is also part of the curve. There is at least one such splitting point. If there are more, then the decomposition is performed by using the point that is at a maximum distance from the line joining the endpoints of the curve. If the curve is both continuous and differentiable at the splitting point, then, of course, the boundaries of the rectangle that pass through these points are tangents to the curve. This splitting process is recur-

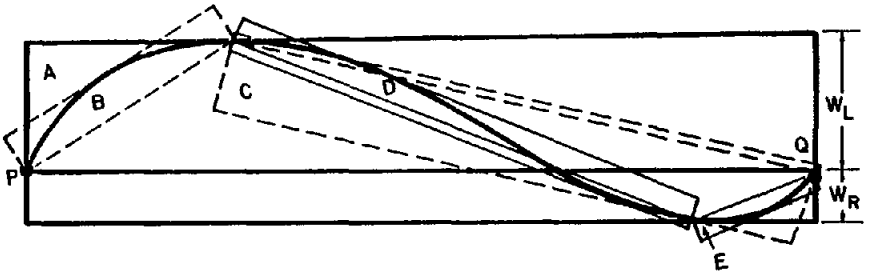


Figure 51. A curve and its decomposition into strips.

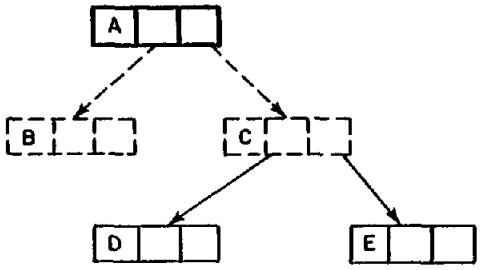


Figure 52. Strip tree corresponding to Figure 51.

sively applied to the two sons until every strip is of a width less than a predetermined value. For Figure 51, the first splitting operation results in the creation of strips B and C. Strip C is further split, creating strips D and E, at which point the splitting process ceases. Figure 52 shows the resulting binary tree. Note that each node in the strip tree is implemented as a record with eight fields. Four fields contain the x and y coordinates of the endpoints, two fields contain pointers to the two sons of the node, and two fields contain information about the width of the strip (i.e., w_L and w_R of Figure 51).

Figure 51 is a relatively simple example. In order to be able to cope with more complex curves, the notion of a strip tree must be extended. In particular, closed curves (e.g., Figure 53) and curves that extend past their endpoints (Figure 54) require some special treatment. The general idea is that these curves are enclosed by rectangles that are split into two rectangular strips, and from now on the strip tree is used as before. Note that the strip tree concept and the related algorithms are regarded by Ballard

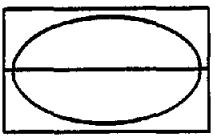


Figure 53. Modeling a closed curve by a strip tree.

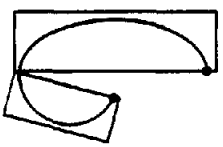


Figure 54. Modeling a curve that extends past its endpoints by a strip tree.

as completely expanded down to a primitive level of unit line segments on a discrete grid, even when the underlying curves are collinear. In order to be able to handle curves that consist of disconnected segments, strips are classified as either regular or not and a special bit is associated with each strip to indicate its status. Formally, a curve is said to be regular if it is connected and has its endpoints touching the ends of the strip.

Like point and region quadtrees, strip trees are useful in applications that involve search and set operations. For example, suppose that we wish to determine whether a road crosses a river. By using a strip tree representation for these features, answering this query means basically that we perform an intersection of the corresponding strip trees. Three cases are possible, as is shown in Figure 55. Figure 55a and b correspond to the answers NO and YES, re-

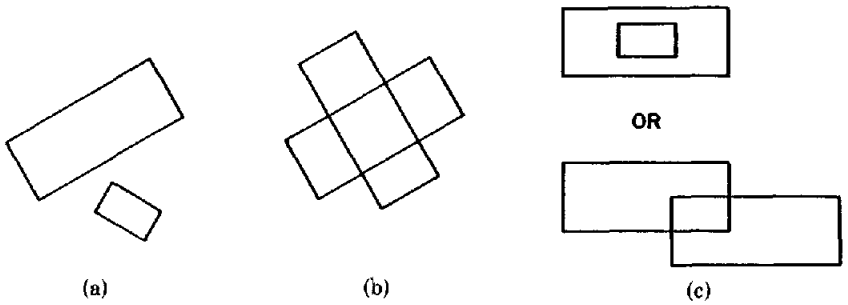


Figure 55. Three possible results of intersecting two strip trees. (a) Null. (b) Clear. (c) Possible.

spectively, whereas Figure 55c requires us to descend further down the strip tree. Other operations that can be performed efficiently by using the strip tree data structure include the computation of the union of two curves, length of a curve, areas of closed curves, intersection of curves with areas, point membership, etc. [Ballard 1981]. In particular, for closed curves that are well behaved, intersection and point membership have an expected execution time of $O(\log v)$, where v is the number of points describing the curve. Strip trees are also used by Gaston and Lozano-Perez [1984] in robotic tactile recognition and localization.

The strip tree can be characterized as a top-down approach to curve approximation. Burton [1977] defines a related structure termed a *BSPR* (binary searchable polygonal representation), which is a bottom-up approach to curve approximation. Once again, the primitive unit of approximation is a rectangle; however, in the case of the BSPR all rectangles are upright (i.e., they have a single orientation). The curve to be approximated is decomposed into a set of simple sections, where each simple section corresponds to a segment of the curve that is monotonic in both the x and y values of the points comprising it. The tree is built by combining pairs of adjacent simple sections to yield compound sections. This process is repeatedly applied until the entire curve is approximated by one compound section. Thus we see that terminal nodes correspond to simple sections and nonterminal nodes correspond to compound sections. For a curve with 2^n simple sections, the corresponding BSPR has n levels.

As an example of a BSPR consider the regular octagon in Figure 56a having vertices A–H. It can be decomposed into four simple sections, that is, ABCD, DEF, FGH, and HA. Figure 56b shows a level 1 approximation to the four simple sections consisting of rectangles AIDN, DJFN, HMFK, and AMHL, respectively. Pairing up adjacent simple sections yields compound sections AIJF corresponding to AIDN and DJFN, and AFKL corresponding to HMFK and AMHL (see Figure 56c). More pairing yields the rectangle for compound section IJKL (see Figure 56d). The resulting BSPR tree is shown in Figure 56e. By using the BSPR, Burton shows how to perform point in polygon determination and polygon intersection. These operations are implemented by tree searches and splitting operations.

Both the strip tree and the BSPR share the property of being independent of the grid system in which they are embedded. An advantage of the BSPR representation over the strip tree is the absence of a need for special handling of closed curves. However, the BSPR is not as flexible as the strip tree. In particular, the resolution of the approximation is fixed (i.e., once the width of the rectangle is selected, it cannot be varied). In fact, this is similar to the advantage of quadtree-based decomposition methods over hexagon-based systems

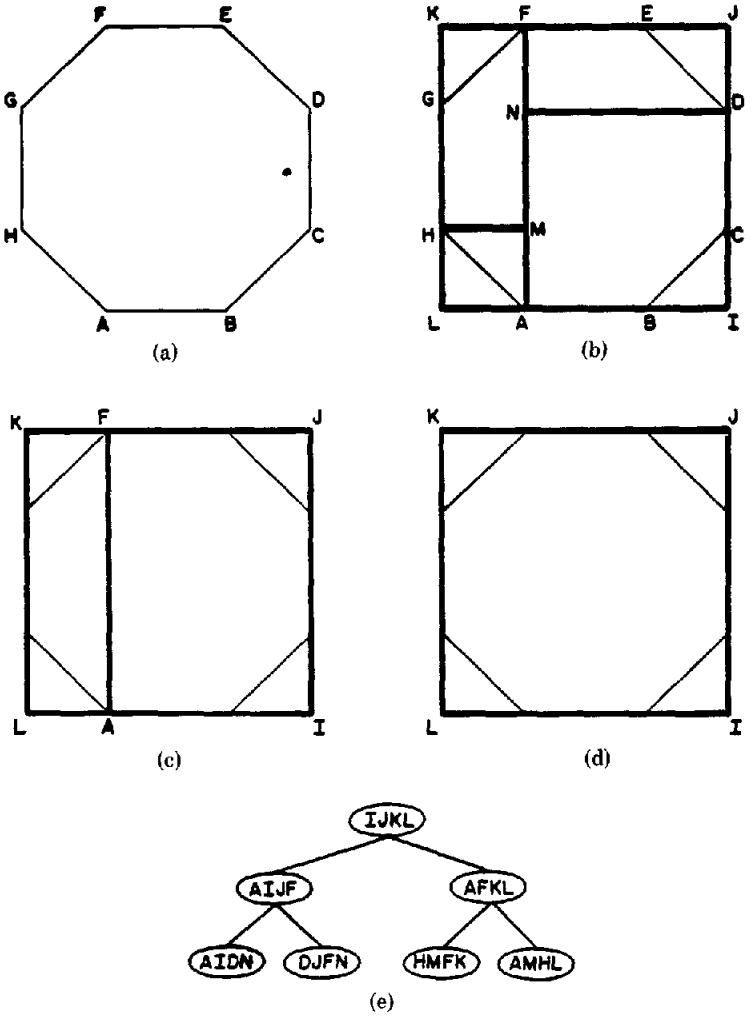


Figure 56. (*a*) A regular octagon and (b)–(d) the three successive approximations resulting from the use of BSPR. (e) The resulting BSPR.

pointed out in Section 2; that is, for the hexagon, we must decide a priori on the finest resolution.

4.2 Methods Based on a Regular Decomposition

Strip tree methods approximate curvilinear data with rectangles. Quadtree methods achieve similar results by use of a collection of disjoint squares having sides of length power of two. A number of variants of quadtrees are currently in use, and can be differentiated by the type of data that they are designed to represent. All but the PM quadtree of Samet and Webber [1983] and the edge-EXCELL of Tamminen [1981] are pixel based and yield approximations whose accuracy is constrained in part by the resolution of the data that they represent. They can be used to represent both linear and nonlinear curves. The latter need not be continuous or differentiable. In contrast, the PM quadtree and the edge-EXCELL yield an exact representation of polygons or collections of polygons.

The *edge quadtree* of Shneier [1981c] is an attempt to store linear feature infor-

mation (e.g., curves) for an image (binary and gray scale) in a manner analogous to that used for storing region information. A region containing a linear feature or part thereof is subdivided into four squares repeatedly until a square is obtained that contains a single curve that can be approximated by a single straight line. Each leaf node contains the following information about the edge passing through it: magnitude (i.e., 1 in the case of a binary image or the intensity in case it is a gray-scale image), direction, intercept, and a directional error term (i.e., the error induced by approximating the curve by a straight line using a measure such as least squares). If an edge terminates within a node, then a special flag is set and the intercept denotes the point at which the edge terminates. Application of this process leads to quadtrees in which long edges are represented by large leaves or a sequence of large leaves. However, small leaves are required in the vicinity of corners or intersecting edges. Of course, many leaves will contain no edge information at all. As an example of the decomposition that is imposed by the edge quadtree, consider Figure 57, which is the edge quadtree corresponding to the polygon of Figure 18 when represented on a 2^4 by 2^4 grid. Note that the edge quadtree in Figure 57 requires fewer blocks than Figure 18, which is the representation of the polygon when using the methods of Hunter and Steiglitz [1979a].

Closely related to the edge quadtree is the *least square quadtree* of Martin [1982]. In that representation, leaf nodes correspond to regions that contain a single curve that can be approximated by k (fixed a priori) straight lines within a least square tolerance. This enables the handling of curved lines and uses fewer nodes than the edge quadtree with greater precision. A cruder method is described by Omolayole and Klinger [1980], where all parts of the image that contain edge data are repeatedly decomposed until a 2 by 2 quadrant is obtained in which they store templatelike representations of the edges. This is quite similar to the MX quadtree, except that the data are edges rather than points. However, it is too low a level of representation in that it does not take advantage of the hierarchical nature of the data structure.

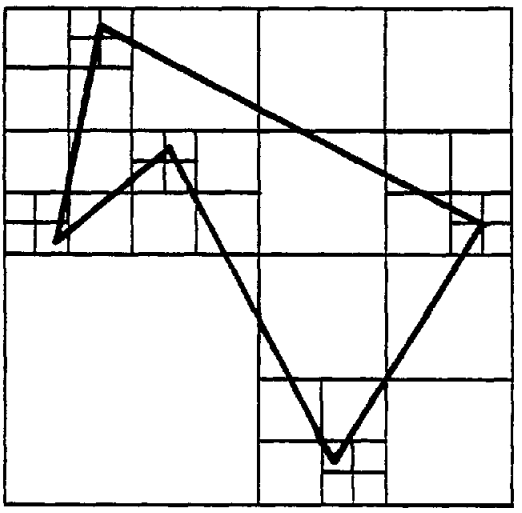


Figure 57. The edge quadtree corresponding to the polygon of Figure 18.

The *line quadtree* of Samet and Webber [1984] addresses the issue of hierarchically representing images that are segmented into a number of different regions rather than mere foreground and background, as is the case for conventional quadtrees. In particular, it encodes both the area of the individual regions and their boundaries in a hierarchical manner. This is in contrast to the region quadtree, which encodes only areas hierarchically, and the strip tree, which encodes only curves hierarchically. The line quadtree partitions the set of regions (termed a *map*) via a recursive decomposition technique that successively subdivides the map until blocks (possibly single pixels) that have no line segments passing through their interior are obtained. With each leaf node, a code is associated that indicates which of its four sides form a boundary (not a partial boundary) of any single region. Thus, instead of distinguishing leaf nodes on the basis of being BLACK or WHITE, boundary adjacency information is used. This boundary information is hierarchical in that it is also associated with nonterminal nodes. In essence, wherever a nonterminal node does not form a T-junction with any of the boundaries of its descendants along a particular side, this side

is then marked as being adjacent to a boundary.

As an illustration of a line quadtree, consider the polygonal map of Figure 58 whose corresponding line quadtree (i.e., block decomposition) is shown in Figure 59. The bold lines indicate the presence of boundaries. Note that the south side of the block corresponding to the root is on a boundary that is also the border of the image. As another example, the western side of the SW son of the root in Figure 59 does not indicate the presence of a boundary (i.e., the side is represented by a light line) even though it is adjacent to the border of the image. The problem is that the SW son of the root has its NW and SW sons in different regions, as is signaled by the presence of a T-junction along its western side. Having the boundary information at the nonterminal nodes enables boundary following algorithms to be performed quickly, in addition to facilitating the process of superimposing one map on top of another. Observe also that the line quadtree has the same number of nodes as a conventional quadtree representation of the image. Boundaries of leaf nodes that are partially on the boundary between two regions can have their boundaries reconstructed by examining their neighbors along the shared boundary. For example, the southern side of the NW son of the SW son of the root in Figure 59, say A, represents a partial boundary. The exact nature of the boundary is obtained by examining the NE and NW sons of the southern brother of A.

The PM quadtree of Samet and Webber [1983] and the edge-EXCELL of Tamminen [1981] are attempts to overcome some of the problems associated with the following three structures: the line quadtree, the edge quadtree, and the quadtree formulation of Hunter and Steiglitz (termed an *MX quadtree*) for representing polygonal maps (i.e., collections of straight lines). In general, all three of these representations correspond to approximations of a map. The line quadtree is based on the approximation that results from digitizing the areas of the polygons comprising a polygonal map. For the edge and MX quadtrees, the result is still an approximation because the vertices

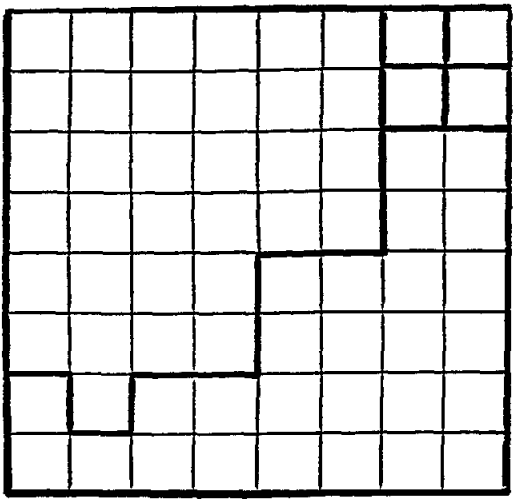


Figure 58. Example polygonal map to illustrate line quadtrees.

are represented by pixels in the edge quadtree and boundaries are represented by BLACK pixels in the MX quadtree. In other words, the MX quadtree is based on a digitization of the boundaries of the polygons, whereas the edge quadtree is based on a piecewise linear approximation of the boundaries of the polygons. Another disadvantage of these three representations is that certain properties of polygonal maps cannot be directly represented by them. For example, it is impossible for five line segments to meet at a vertex. In the case of the edge and MX quadtrees, we would have difficulty in detecting the vertex and for the line quadtree the situation cannot be handled because all regions comprising the map must be rectilinear. Note that it is impossible for five rectilinear regions to meet at a point. Other problems include a sensitivity to shift and rotation, which may result in a loss of accuracy in the original approximation. Finally, owing to their approximate nature, these data structures will most likely require a considerable amount of storage, since each line is frequently approximated at the pixel level, thereby resulting in quadtrees that are fairly deep.

The *PM quadtree* represents a polygonal map by using the PR quadtree discussed in Section 3.2. Each vertex in the map corresponds to a data point in the PR quadtree.

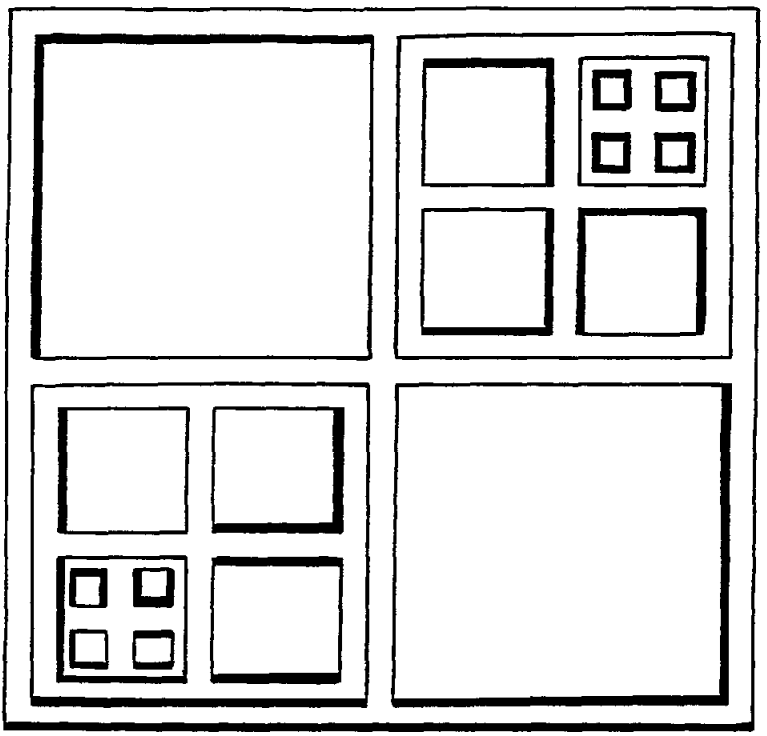


Figure 59. Line quadtree corresponding to Figure 58.

We define a *q-edge* to be a segment of an edge of the map that either spans an entire block in the PR quadtree (e.g., segment RS in Figure 61) or extends from a boundary of a block to a vertex within the block (i.e., when the block contains a vertex, e.g., segment CV in Figure 61).

For every leaf in the PR quadtree we partition all of its q-edges into seven classes. Each of these classes is stored in a balanced binary tree [Aho et al. 1974]. One class corresponds to the set of q-edges that meet at a vertex within the block's region. This class is ordered in an angular manner. The remaining q-edges that pass through the block's region must enter at one side and exit via another. This yields six classes: NW, NS, NE, EW, SW, and SE, where SW denotes q-edges that intersect both the southern and western boundaries of the block's region. Note that these classes are often empty. The q-edges of these classes are ordered by the position of their intercepts along the perimeter of the block's region. A q-edge that coincides with the boundary of a leaf's region is placed in either class NS or EW as is appropriate. For example, consider the polygonal map of Figure 60 and its corresponding PM quadtree in Figure 61. The block containing vertex C has one balanced binary tree for the q-edges intersecting vertex C (three balanced binary tree nodes for q-edges CM, CN, and CV) and one balanced binary tree for the q-edges intersecting the NW boundary (one balanced binary tree node for q-edge ST). In total, the PM quadtree of Figure 61 contains seven quadtree leaf nodes, and nine nonempty balanced binary trees containing seventeen nodes.

The PM quadtree provides a convenient, reasonably efficient data structure for performing a variety of operations. Point-in-polygon determination is achieved by finding a bordering q-edge with respect to each of the seven classes and then selecting the closest of the seven as the true bordering q-edge. The execution time of this proce-

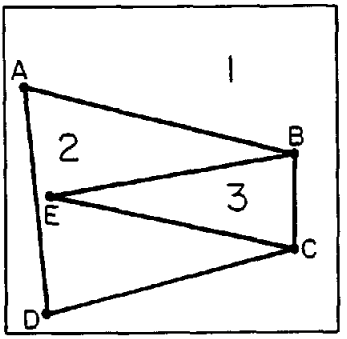


Figure 60. Example polygonal map to illustrate PM quadtrees.

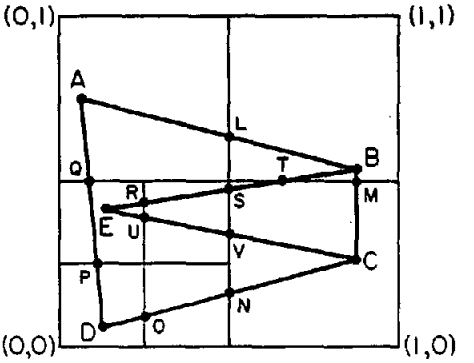


Figure 61. PM quadtree corresponding to Figure 60.

dure is proportional to the depth of the PM quadtree. It should be noted that the depth of the PM quadtree is inversely proportional to the log of the minimum separation between two vertices plus the log of the number of edges in the polygonal map [Samet and Webber 1983]. Besides point-in-polygon determination, there exist efficient algorithms for insertion of an edge into the map, overlaying two maps, clipping and windowing a map, and range searching (i.e., determining all polygons within a given distance of a point).

The *edge-EXCELL* method [Tamminen 1981] is an application of the EXCELL method for point data (described in Section 3.5) to polygonal maps. It is based on a regular decomposition. The principles guiding the decomposition process and the data structure are identical to those used for representing points. The only difference is that now the data consist of straight-line segments that intersect the cells (i.e., grid blocks). Once again, a grid directory is used that maps the cells into storage areas of a finite capacity (e.g., buckets), which often reside on disk. As the buckets overflow (i.e., the number of line segments intersecting them exceeds the capacity of the bucket), buckets are split into two equal-sized grid blocks, which may also lead to a doubling in the size of the directory. If the polygonal map contains a vertex at which m lines intersect, and m is greater than the bucket capacity, then no matter how many times the bucket is split, it will be impossible to store all the line segments in one bucket. In such a case, edge-EXCELL makes use of overflow buckets. This is a disadvantage of edge-EXCELL when compared with the PM quadtree.

Using edge-EXCELL, point-in-polygon determination is achieved by a two-step process. First, the cell in which the point lies is located. Second, the corresponding polygon is determined by finding the closest polygon boundary in any given direction by use of a technique known as ray casting [Roth 1982] (similar to searching for the closest q-edge when using the PM quadtree). Tamminen [1983] has shown that, in practice, this requires on the average little more than one cell access. Edge-EXCELL has also been used to do hidden line elimination [Tamminen 1982].

4.3 Comparison

The chain code is the most compact of the representations. However, it is a very local data structure and is thus rather cumbersome when attempting to perform set operations such as intersection of two curves represented using it. In addition, like the strip tree, and to a lesser extent the BSPR, it is not tied to a particular coordinate system. This is a problem with methods based on a regular decomposition, although it is somewhat reduced for the PM quadtree and edge-EXCELL.

Representations based on a regular decomposition have a number of advantages over the strip tree. First, more than one curve can be represented with one instance

of the data structure—a very important feature for maps. Second, they are unique. In contrast, only one curve can be represented by a single strip tree. Also, the strip tree is not unique when the curve is closed, not regular, or contains more than one endpoint. However, the strip tree is invariant under shifts and rotations. The line quadtree is better than the MX quadtree of Hunter and Steiglitz [1979a], which represents boundary lines as narrow BLACK regions, for two reasons. First, narrow regions are costly in terms of the number of nodes in the quadtree. Second, it commits the use to a specific thickness of the boundary line, which may be unfortunate, for example, when regenerating the picture on an output device. Also such arbitrary decisions of representation accuracy are not very appropriate when data are to be stored in a database.

At this point it might be appropriate to speculate on some other data structures for curvilinear data. Representations based on regular decomposition are attractive because they enable efficient computation of set operations. In particular, at times it is convenient to perform the operations on different kinds of geometric entities (e.g., intersecting curves with areas). The strip tree is an elegant data structure from the standpoint of approximation. However, it has the disadvantage that decomposition points are independent of the coordinate system of the image (i.e., they are at arbitrary points dependent on the data rather than at predetermined positions as is the case with a data structure that is based on a regular decomposition). Thus answering a query such as, "Find all wheat growing regions within 20 miles of the Mississippi River," is not easy to do when the river is represented as a strip tree and the wheat-growing regions are represented by region quadtrees. The problem is that, whereas quadtree methods merely require pointer-chasing operations, strip tree methods may lead to complex geometric computations. What is desired is a regular decomposition strip tree or variant thereof.

The data structures discussed in this section are rooted in the image-processing area and were designed primarily to represent curves and lines. Computational geometry is another area where similar problems arise [Edelsbrunner 1984; Toussaint 1980]. This is a rapidly changing field, which has its roots in the work of Shamos and Hoey [1975; Shamos 1978] and focuses on problems of asymptotical computational complexity of geometric algorithms. However, a full presentation of this field is beyond the scope of this survey. Nevertheless, in the following we do give a brief sample of the types of results attainable for similar problems. Many of the solutions (e.g., Lipton and Tarjan [1977]) are based on the representation of line segments as edges and vertices in a graph.

For example, an alternative to the PM quadtree and edge-EXCELL is the *K-structure* of Kirkpatrick [1983]. It is a hierarchical structure based on triangulation rather than a regular decomposition. The notion of hierarchy in the K-structure is radically different from that of a quadtree, in that instead of replacing a group of triangles by a single triangle at the next higher level, a group of triangles is replaced by a smaller group of triangles. Triangles are grouped for replacement because they share a common vertex. The smaller group results from eliminating the common vertex and then retriangulating. Kirkpatrick [1983] shows that at each level of the hierarchy, at least one twenty-fourth of the vertices can be eliminated in this manner. The vertices that have been eliminated are guaranteed to have degree of 11 or less, thus bounding the cost of retriangulating. Let v denote the number of vertices in a polygonal map. Then, the size of a K-structure is guaranteed to be $O(v)$ although the worst-case constant of proportionality is 24 times the amount of information stored at a node. It also leads to an $O(\log v)$ query time for point-in-polygon determination. The construction process has a worst-case execution time of $O(v)$ for a triangular subdivision and $O(v \log v)$ for a general one. The latter is dominated by the cost of triangulating the original polygonal map [Hertel and Mehlhorn 1983]. Since a triangulation constitutes a convex map, that is, a planar subdivision formed of convex regions, the work of Nievergelt and Preparata [1982] is

relevant. They show that the cost of performing a map overlay operation is $O(v \log v + s)$, where s is the number of intersections of all line segments in the two maps. Finally, it is worth noting that the hierarchical nature of the K-structure may lead to an efficient range-searching algorithm.

Another alternative to the K-structure is the *layered dag* of Edelsbrunner et al. [1985]. This structure is a modification of a binary tree, which is used to store a y-monotone subdivision of a polygonal map. A y-monotone subdivision of a polygonal map is created by partitioning the regions of a map until no vertical line intersects a region's boundary more than twice. Note that the resulting map need not be convex; however, the asymptotic worst-case analysis of the layered dag is identical to that of the K-structure. The number of line segments that must be inserted to check a y-monotone subdivision is usually considerably fewer than the number needed for triangulation.

When the K-structure and the layered dag are compared with the PM quadtree (and to some extent edge-EXCELL), the qualitative comparison is analogous to that of a point quadtree with a PR quadtree. All of these structures have their place; the one to use depends on the nature of the data and the importance of guaranteed worst-case performance. The K-structure and the layered dag organize the data to be stored, whereas the PM quadtree organizes the embedding space from which the data are drawn. The K-structure and the layered dag have better worst-case execution time bounds for similar operations compared with those considered for the PM quadtree. However, considerations such as ease of implementation and integration with representations of other data types must also be taken into account in making an evaluation. In the case of dynamic files, at present it would seem to be more convenient to use the PM quadtree since a general updating algorithm for the K-structure or the layered dag have not been reported.

5. CONCLUSIONS

In this paper we have attempted to give a survey of a large number of hierarchical data structures and show how they are interrelated. Undoubtedly, some data structures may have been overlooked, but any such omissions were unintentional. The idea of recursive decomposition is the underlying basis of each structure discussed. In the past, there has been considerable confusion between these similar representations (e.g., the point quadtree and the region quadtree). Consequently, readers of the literature are cautioned about comparisons involving "quadtrees." Our focus has been on execution time efficiencies although storage requirements have also been taken into consideration.

For the future we foresee the following *quad* trends. Complexity measures for images represented by quadtrees are likely to be proposed [Creuzburg 1981]. Quadtrees and their variants will be used in computational domains [Rheinboldt and Mesztenyi 1980; Samet and Krishnamurthy 1983; Yerry and Shepard 1983]. With the current interest in VLSI, it is certain that increasingly these data structures will find themselves being implemented in hardware [Besslich 1982; Dyer 1981; Ibrahim 1984; Milford and Willis 1984; Woodwark 1982]. Another trend is the development of integrated geographical databases that permit the interaction between data of differing types [Matsuyama et al. 1984; McKeown and Denlinger 1984; Rosenfeld et al. 1982, 1983]. This would facilitate efficiently answering queries like, "Find all cities with a population in excess of 5000 people in wheat-growing regions within 20 miles of the Mississippi River." At this point we have traveled a full circle in our survey and if we still do not know how to answer this query, then, by now, maybe we at least know why—an appropriate point for us to get off and for you, the reader, to get on!

ACKNOWLEDGMENTS

I have benefited greatly from comments by Bernard Diaz, Gary Knott, Azriel Rosenfeld, Deepak Sherlekar, Markku Tamminen, and Robert E. Webber. The referees are also to be commended for the thoroughness of their reviews. This work was supported in part by the National Science Foundation under Grant MCS-83-02118.

REFERENCES

ABEL, D. J. 1984. A B$^+$-tree structure for large quadtrees. *Comput. Vision Gr. Image Process. 27,* 1 (July), 19–31.

ABEL, D. J., AND SMITH J. L. 1983. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Comput. Vision Gr. Image Process. 24,* 1 (Oct.), 1–13.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass.

AHUJA, N. 1983. On approaches to polygonal decomposition for hierarchical image representation. *Comput. Vision Gr. Image Process. 24,* 2 (Nov.) 200–214.

AHUJA, N., AND NASH, C. 1984. Octree representations of moving objects. *Comput. Vision Gr. Image Process. 26,* 2 (May), 207–216.

ANDERSON, D. P. 1983. Techniques for reducing pen plotting time. *ACM Trans. Gr. 2,* 3 (July), 197–212.

BALLARD, D. H. 1981. Strip trees: A hierarchical representation for curves. *Commun. ACM 24,* 5 (May), 310–321. See also corrigendum, *Commun. ACM 25,* 3 (Mar. 1982), 213.

BELL, S. B. M., DIAZ, B. M., HOLROYD, F., AND JACKSON, M. J. 1983. Spatially referenced methods of processing raster and vector data. *Image Vision Comput. 1,* 4 (Nov.), 211–220.

BENTLEY, J. L. 1975a. A survey of techniques for fixed radius near neighbor searching. SLAC Rep. No. 186, Stanford Linear Accelerator Center, Stanford University, Stanford, Calif., Aug.

BENTLEY, J. L. 1975b. Multidimensional binary search trees used for associative searching. *Commun. ACM 18,* 9 (Sept.), 509–517.

BENTLEY, J. L., AND FRIEDMAN, J. H. 1979. Data structures for range searching. *ACM Comput. Surv. 11,* 4 (Dec.), 397–409.

BENTLEY, J. L., AND STANAT, D. F. 1975. Analysis of range searches in quad trees. *Inf. Process. Lett. 3,* 6 (July), 170–173.

BESSLICH, P. W. 1982. Quadtree construction of binary images by dyadic array transformations. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing* (Las Vegas, Nev., June). IEEE, New York, pp. 550–554.

BLUM, H. 1967. A transformation for extracting new descriptors of shape. In *Models for the Perception of Speech and Visual Form,* W. Wathen-Dunn, Ed. M.I.T. Press, Cambridge, Mass., pp. 362–380.

BROOKS, R. A., AND LOZANO-PEREZ, T. 1983. A subdivision algorithm in configuration space for findpath with rotation. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence* (Karlsruhe, West Germany, Aug.). Kaufmann, Los Altos, Calif., pp. 799–806.

BURKHARDT, W. A. 1983. Interpolation-based index maintenance. *BIT 23,* 3, 274–294.

BURT, P. J. 1980. Tree and pyramid structures for coding hexagonally sampled binary images. *Comput. Graphics Image Process. 14,* 3 (Nov.), 249–270.

BURT, P. J., HONG, T. H., AND ROSENFELD, A. 1981. Segmentation and estimation of image region properties through cooperative hierarchical computation. *IEEE Trans. Syst. Man Cybern. 11,* 12 (Dec.), 802–809.

BURTON, W. 1977. Representation of many-sided polygons and polygonal lines for rapid processing. *Commun. ACM 20,* 3 (Mar.), 166–171.

BURTON, F. W., AND KOLLIAS, J. G. 1983. Comment on the explicit quadtree as a structure for computer graphics. *Comput. J. 26,* 2 (May), 188.

CARLSON, W. E. 1982. An algorithm and data structure for 3D object synthesis using surface patch intersections. *Comput. Gr. SIGGRAPH 82 Conf. Proc. 16,* 3, 255–264.

CHIEN, C. H., AND AGGARWAL, J. K. 1984. A normalized quadtree representation. *Comput. Vision Gr. Image Process. 26,* 3 (June), 331–346.

COHEN, E., LYCHE, T., AND RIESENFELD, R. 1980. Discrete B-splines and subdivision techniques in computer-aided geometric design and computer graphics. *Comput. Gr. Image Process. 14,* 3 (Oct.), 87–111.

COMER, D. 1979. The ubiquitous B-tree. *ACM Comput. Surv. 11,* 2 (June), 121–137.

CONNOLLY, C. I. 1984. Cumulative generation of octree models from range data. In *Proceedings of the International Conference on Robotics* (Atlanta, Ga., Mar.). IEEE, New York, pp. 25–32.

COOK, B. G. 1978. The structural and algorithmic basis of a geographic data base. In *Proceedings of the First International Advanced Study Symposium on Topological Data Structures for Geographic Information Systems,* G. Dutton, Ed., Harvard Papers on Geographic Information Systems, Harvard University Press, Cambridge, Mass.

CREUTZBURG, E. 1981. Complexities of quadtrees and the structure of pictures. Tech. Rep. N/81/74, Computer Science Dept., Friedrich-Schiller University, Jena, East Germany.

DAVIS, L. S., AND ROUSSOPOULOS, N. 1980. Approximate pattern matching in a pattern database system. *Inf. Syst. 5,* 2, 107–119.

DECOULON, F., AND JOHNSEN, U. 1976. Adaptive block schemes for source coding of black-and-white facsimile. *Electron. Lett. 12,* 3, 61–62. See also erratum, *Electron. Lett. 12,* 6 (1976), 152.

DEFLORIANI, L., FALCIDIENO, B., NAGY, G., AND PIENOVI, C. 1982. Yet another method for triangulation and contouring for automated cartography. In *Proceedings of the American Congress on Surveying and Mapping* (Hollywood, Fla.), F. S. Cardwell, R. Black, and B. M. Cole, Eds. American Society of Photogrammetry, pp. 101–110.

DEMILLO, R. A., EISENSTAT, S. C., AND LIPTON, R. J. 1978. Preserving average proximity in arrays. *Commun. ACM 21,* 3 (Mar.), 228–231.

DOCTOR, L. J., AND TORBORG, J. G. 1981. Display techniques for octree-encoded objects. *IEEE Comput. Gr. Appl. 1,* 1 (July), 39–46.

DUDA, R. O., AND HART, P. E. 1973. *Pattern Classification and Scene Analysis.* Wiley, New York.

DYER, C. R. 1980. Computing the Euler number of an image from its quadtree. *Comput. Gr. Image Process. 13,* 3 (July), 270–276.

DYER, C. R. 1981. A VLSI pyramid machine for parallel image processing. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing* (Dallas, Tex.). IEEE, New York, pp. 381–386.

DYER, C. R. 1982. The space efficiency of quadtrees. *Comput. Gr. Image Process. 19,* 4 (Aug.), 335–348.

DYER, C. R., ROSENFELD, A., AND SAMET, H. 1980. Region representation: Boundary codes from quadtrees. *Commun. ACM 23,* 3 (Mar.), 171–179.

EASTMAN, C. M. 1970. Representations for space planning. *Commun. ACM 13,* 4 (Apr.), 242–250.

EDELSBRUNNER, H. 1984. Key-problems and key-methods in computational geometry. In *Proceedings of the Symposium of Theoretical Aspects of Computer Science* (Paris, France), Lecture Notes in Computer Science, vol. 166. Springer Verlag, New York, pp. 1–13.

EDELSBRUNNER, H., AND VAN LEEUWEN, J. 1983. Multidimensional data structures and algorithms: A bibliography. Rep. F104, Institute for Information Processing, Technical University of Graz, Graz, Austria, Jan.

EDELSBRUNNER, H., GUIBAS, L. J., AND STOLFI, J. 1985. Optimal point location in a monotone subdivision. *SIAM J. Comput.* in press.

FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. 1979. Extendible hashing—A fast access method for dynamic files. *ACM Trans. Database Syst. 4,* 3 (Sept.), 315–344.

FAUGERAS, O. D., AND PONCE, J. 1983. Prism trees: A hierarchical representation for 3-d objects. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence* (Karlsruhe, West Germany, Aug.). Kaufmann, Los Altos, Calif., pp. 982–988.

FAUGERAS, O. D., HEBERT, M., MUSSI, P., AND BOISSONNAT, J. D. 1984. Polyhedral approximation of 3-d objects without holes. *Comput. Vision Gr. Image Process. 25,* 2 (Feb.), 169–183.

FAVERJON, B. 1984. Obstacle avoidance using an octree in the configuration space of a manipulator. In *Proceedings of the International Conference on Robotics* (Atlanta, Ga., Mar.). IEEE, New York, pp. 504–512.

FINKEL, R. A., AND BENTLEY, J. L. 1974. Quad trees: A data structure for retrieval on composite keys. *Acta Inf. 4,* 1, 1–9.

FREDKIN, E. 1960. Trie memory. *Commun. ACM 3,* 9 (Sept.), 490–499.

FREEMAN, H. 1974. Computer processing of line-drawing images. *ACM Comput. Surv. 6,* 1 (Mar.), 57–97.

FRIEDMAN, J. H., BENTLEY, J. L., AND FINKEL, R. A. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3,* 3 (Sept.), 209–226.

GARGANTINI, I. 1982a. An effective way to represent quadtrees. *Commun. ACM 25,* 12 (Dec.), 905–910.

GARGANTINI, I. 1982b. Linear octtrees for fast processing of three dimensional objects. *Comput. Gr. Image Process. 20,* 4 (Dec.), 365–374.

GARGANTINI, I. 1983. Translation, rotation, and superposition of linear quadtrees. *Int. J. Man-Mach. Stud. 18,* 3 (Mar.), 253–263.

GASTON, P. C., AND LOZANO-PEREZ, T. 1984. Tactile recognition and localization using object models: The case of polyhedra on a plane. *IEEE Trans. Pattern Anal. Mach. Intell. 6,* 3 (May), 257–266.

GIBSON, L., AND LUCAS, D. 1982. Vectorization of raster images using hierarchical methods. *Comput. Gr. Image Process. 20,* 1 (Sept.), 82–89.

GILLESPIE, R., AND DAVIS, W. A. 1981. Tree data structures for graphics and image processing. In *Proceedings of the 7th Conference of the Canadian Man-Computer Communications Society* (Waterloo, Canada, June), pp. 155–161.

GOMEZ, D., AND GUZMAN, A. 1979. Digital model for three-dimensional surface representation. *Geo-Process. 1,* 53–70.

GROSKY, W. I., AND JAIN, R. 1983. Optimal quadtrees for image segments. *IEEE Trans. Pattern Anal. Mach. Intell. 5,* 1 (Jan.), 77–83.

HARARY, F. 1969. *Graph Theory.* Addison-Wesley, Reading, Mass.

HERTEL, S., AND MEHLHORN, K. 1983. Fast triangulation of simple polygons. In *Proceedings of the 1983 International Foundations of Computation Theory Conference* (Borgholm, Sweden, Aug.), Lecture Notes in Computer Science, vol. 158. Springer Verlag, New York, pp. 207–218.

HINRICHS, K., AND NIEVERGELT, J. 1983. The Grid File: A data structure to support proximity queries on spatial objects. Rep. 54, Institut fur Informatik, ETH, Zurich, July.

HOARE, C. A. R. 1972. Notes on data structuring. In *Structured Programming,* O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds. Academic Press, London, p. 154.

HOROWITZ, S. L., AND PAVLIDIS, T. 1976. Picture segmentation by a tree traversal algorithm. *J. ACM 23,* 2 (Apr.), 368–388.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. *Proc. IRE 40,* 9 (Sept.), 1098–1101.

HUNTER, G. M. 1978. Efficient computation and data structures for graphics. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N. J.

HUNTER, G. M., AND STEIGLITZ, K. 1979a. Operations on images using quad trees. *IEEE Trans. Pattern Anal. Mach. Intell. 1,* 2 (Apr.), 145–153.

HUNTER, G. M., AND STEIGLITZ, K. 1979b. Linear transformation of pictures represented by quad

trees. *Comput. Gr. Image Process. 10,* 3 (July), 289–296.

IBRAHIM, H. A. H. 1984. The connected component labeling algorithm on the NON-VON supercomputer. In *Proceedings of the Workshop on Computer Vision: Representation and Control* (Annapolis, Md., Apr.). IEEE, New York, pp. 37–45.

ISMAIL, M. G. B., AND STEELE, R. 1980. Adaptive pel location coding for bilevel facsimile signals. *Electron. Lett. 16,* (May), 361–363

JACKINS, C., AND TANIMOTO, S. L. 1980. Oct-trees and their use in representing three-dimensional objects. *Comput. Gr. Image Process. 14,* 3 (Nov.), 249–270.

JACKINS, C., AND TANIMOTO, S. L. 1983. Quad-trees, oct-trees, and *k*-trees—A generalized approach to recursive decomposition of Euclidean space. *IEEE Trans. Pattern Anal. Mach. Intell. 5,* 5 (Sept.), 533–539.

JONES, L., AND IYENGAR, S. S. 1984. Space and time efficient virtual quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 6,* 2 (Mar.), 244–247.

KAWAGUCHI, E., AND ENDO, T. 1980. On a method of binary picture representation and its application to data compression. *IEEE Trans. Pattern Anal. Mach. Intell. 2,* 1 (Jan.), 27–35.

KAWAGUCHI, E., ENDO, T., AND YOKOTA, M. 1980. DF-expression of binary-valued picture and its relation to other pyramidal representations. In *Proceedings of the 5th International Conference on Pattern Recognition* (Miami Beach, Fla., Dec.). IEEE, New York, pp. 822–827.

KAWAGUCHI, E., ENDO, T., AND MATSUNAGA, J. 1983. Depth-first expression viewed from digital picture processing. *IEEE Trans. Pattern Anal. Mach. Intell. 5,* 4 (July), 373–384.

KEDEM, G. 1981. The Quad-CIF tree: A data structure for hierarchical on-line algorithms. Tech. Rep. 91, Computer Science Dept., The University of Rochester, Rochester, New York, Sept.

KELLY, M. D. 1971. Edge detection in pictures by computer using planning. *Mach. Intell. 6,* 397–409.

KIRKPATRICK, D. 1983. Optimal search in planar subdivisions. *SIAM J. Comput. 12,* 1 (Feb.), 28–35.

KLINGER, A. 1971. Patterns and search statistics. In *Optimizing Methods in Statistics,* J. S. Rustagi, Ed. Academic Press, New York, pp. 303–337.

KLINGER, A., AND DYER, C. R. 1976. Experiments in picture representation using regular decomposition. *Comput. Gr. Image Process. 5,* 1 (Mar), 68–105.

KLINGER, A., AND RHODES, M. L. 1979. Organization and access of image data by areas. *IEEE Trans. Pattern Anal. Mach. Intell. 1,* 1 (Jan.), 50–60.

KNOTT, G. D. 1971. Expandable open addressing hash table storage and retrieval. In *Proceedings of SIGFIDET Workshop on Data Description, Access, and Control* (San Diego, Calif., Nov.). ACM, New York, pp. 187–206.

KNOWLTON, K. 1980. Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes. *Proc. IEEE 68,* 7 (July), 885–896.

KNUTH, D. E. 1973. *The Art of Computer Programming,* vol. 3, *Sorting and Searching.* Addison-Wesley, Reading, Mass.

KNUTH, D. E. 1975. *The Art of Computer Programming,* vol. 1, *Fundamental Algorithms,* 2nd ed. Addison-Wesley, Reading, Mass.

LAUZON, J. P., MARK, D. M., KIKUCHI, L., AND GUEVARA, J. A. 1984. Two-dimensional run-encoding for quadtree representation. Unpublished manuscript, Department of Geography, State University of New York at Buffalo, Buffalo, New York.

LEE, D. T., AND SHACTER, B. J. 1980. Two algorithms for constructing a Delaunay triangulation. *Int. J. Comput. Inf. Sci. 9,* 3 (June), 219–242.

LEE, D. T., AND WONG, C. K. 1977. Worst-case analysis for region and partial region searches in multidimensional binary search trees and quad trees. *Acta Inf. 9,* 1, 23–29.

LETELIER, P. 1983. Transmission d'images à bas débit pour un système de communication téléphonique adapté aux sourds. Thèse de docteur-ingénieur, Université de Paris-Sud, Paris, Sept.

LI, M., GROSKY, W. I., AND JAIN, R. 1982. Normalized quadtrees with respect to translations. *Comput. Gr. Image Process. 20,* 1 (Sept.), 72–81.

LINN, J. 1973. General methods for parallel searching. Tech. Rep. 81, Digital Systems Laboratory, Stanford University, Stanford, Calif., May.

LIPTON, R. J., AND TARJAN, R. E. 1977. Application of a planar separator theorem. In *Proceedings of the 18th Annual IEEE Symposium on the Foundations of Computer Science* (Providence, R. I., Oct.). IEEE, New York, pp. 162–170.

LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Montreal, Oct.). IEEE, New York, pp. 212–223.

LOZANO-PEREZ, T. 1981. Automatic planning of manipulator transfer movements. *IEEE Trans. Syst. Man Cybern. 11,* 10 (Oct.), 681–698.

LUMIA, R. 1983. A new three-dimensional connected components algorithm. *Comput. Vision Gr. Image Process. 23,* 2 (Aug.), 207–217.

LUMIA, R., SHAPIRO, L., AND ZUNIGA, O. 1983. A new connected components algorithm for virtual memory computers. *Comput. Vision Gr. Image Process. 22,* 2 (May), 287–300.

MARTIN, J. J. 1982. Organization of geographical data with quad trees and least square approximation. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing* (Las Vegas, Nev., June). IEEE, New York, pp. 458–463.

MATSUYAMA, T., HAO, L. V., AND NAGAO, M. 1984. A file organization for geographic information systems based on spatial proximity. *Com-*

put. Vision Gr. Image Process. 26, 3 (June), 303–318.

MCCLUSKEY, E. J. 1965. *Introduction to the Theory of Switching Circuits.* McGraw-Hill, New York, pp. 60–61.

MCKEOWN, D. M., JR., AND DENLINGER, J. L. 1984. Map-guided feature extraction from aerial imagery. In *Proceedings of the Workshop on Computer Vision: Representation and Control* (Annapolis, Md., Apr.). IEEE, New York, pp. 205–213.

MEAGHER, D. 1982. Geometric modeling using octree encoding. *Comput. Gr. Image Process. 19,* 2 (June), 129–147.

MERRETT, T. H. 1978. Multidimensional paging for efficient database querying. In *Proceedings of the International Conference on Management of Data* (Milan, Italy, June), pp. 277–289.

MERRETT, T. H., AND OTOO, E. J. 1981. Dynamic multipaging: A storage structure for large shared data banks. In *Improving Database Usability and Responsiveness,* P. Scheuermann, Ed. Academic Press, New York, pp. 237–254.

MERRILL, R. D. 1973. Representations of contours and regions for efficient computer search. *Commun. ACM 16,* 2 (Feb.), 69–82.

MILFORD, D. J., AND WILLIS, P. C. 1984. Quad encoded display. *IEE Proc. 131,* E3 (May), 70–75.

MINSKY, M., AND PAPERT, S. 1969. *Perceptrons: An Introduction to Computational Geometry.* M.I.T. Press, Cambridge, Mass.

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

MUDUR, S. P., AND KOPARKAR, P. A. 1984. Interval methods for processing geometric objects. *IEEE Comput. Gr. Appl. 4,* 2 (Feb.), 7–17.

NAGY, G., AND WAGLE, S. 1979. Geographic data processing. *ACM Comput. Surv. 11,* 2 (June), 139–181.

NIEVERGELT, J., AND PREPARATA, F. P. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM 25,* 10 (Oct.), 739–746.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. 1984. The Grid File: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst. 9,* 1 (Mar.), 38–71.

NILSSON, N. J. 1969. A mobile automaton: An application of artificial intelligence techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington D.C.). Kaufmann, Los Altos, Calif., pp. 509–520.

OLIVER, M. A., AND WISEMAN, N. E. 1983a. Operations on quadtree-encoded images. *Comput. J. 26,* 1 (Feb.), 83–91.

OLIVER, M. A., AND WISEMAN, N. E. 1983b. Operations on quadtree leaves and related image areas. *Comput. J. 26,* 4 (Nov.), 375–380.

OMOLAYOLE, J. O., AND KLINGER, A. 1980. A hierarchical data structure scheme for storing pictures. In *Pictorial Information Systems,* S. K. Chang and K. S. Fu, Eds. Springer-Verlag, Berlin and New York, 1980.

ORENSTEIN, J. A. 1982. Multidimensional tries used for associative searching. *Inf. Process. Lett. 14,* 4 (June), 150–157.

ORENSTEIN, J. A. 1983. A dynamic hash file for random and sequential accessing. In *Proceedings of the 6th International Conference on Very Large Data Bases* (Florence, Italy, Oct.). IEEE, New York, pp. 132–141.

ORENSTEIN, J. A., AND MERRETT, T. H. 1984. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT–SIGMOD Symposium on Principles of Database Systems* (Waterloo, Ontario, Apr.). ACM, New York, pp. 181–190.

O'ROURKE, J. 1981. Dynamically quantized spaces for focusing the Hough Transform. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence* (Vancouver, B.C., Aug.). Kaufmann, Los Altos, Calif., pp. 737–739.

O'ROURKE, J., AND SLOAN, K. R., JR. 1984. Dynamic quantization: Two adaptive data structures for multidimensional squares. *IEEE Trans. Pattern Anal. Mach. Intell. 6,* 3 (May), 266–280.

OUKSEL, M., AND SCHEUERMANN, P. 1983. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of the 2nd ACM SIGACT–SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., Mar.). ACM, New York, pp. 90–105.

OVERMARS, M. H. 1983. *The Design of Dynamic Data Structures,* Lecture Notes in Computer Science, vol. 156. Springer-Verlag, New York.

OVERMARS, M. H., AND VAN LEEUWEN, J. 1982. Dynamic multi-dimensional data structures based on quad- and *k*-d trees. *Acta Inf. 17,* 3, 267–285.

PETERS, F. 1984. An algorithm for transformations of pictures represented by quadtrees, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

PEUCKER, T. 1976. A theory of the cartographic line. *Int. Yearb. Cartogr. 16,* 34–43.

PEUQUET, D. J. 1979. Raster processing: An alternative approach to automated cartographic data handling. *Am. Cartogr. 2* (Apr.), 129–239.

PEUQUET, D. J. 1983. A hybrid data structure for the storage and manipulation of very large spatial data sets. *Comput. Vision Gr. Image Process. 24,* 1 (Oct.), 14–27.

PFALTZ, J. L., AND ROSENFELD, A. 1967. Computer representation of planar regions by their skeletons. *Commun. ACM 10,* 2 (Feb.), 119–122.

PIETIKAINEN, M., ROSENFELD, A., AND WALTER, I. 1982. Split-and-link algorithms for image segmentation. *Pattern Recognition 15,* 4, 287–298.

RAMAN, V., AND IYENGAR, S. S. 1983. Properties and applications of forests of quadtrees for pictorial data representation. *BIT 23,* 4, 472–486.

RANADE, S. 1981. Use of quadtrees for edge enhancement. *IEEE Trans. Syst. Man, Cybern. 11,* 5 (May), 370–373.

RANADE, S., AND SHNEIER, M. 1981. Using quadtrees to smooth images. *IEEE Trans. Syst. Man Cybern. 11,* 5 (May), 373–376.

RANADE, S., ROSENFELD, A., AND PREWITT, J. M. S. 1980. Use of quadtrees for image segmentation. TR-878, Dept. of Computer Science, University of Maryland, College Park, Md., Feb.

RANADE, S., ROSENFELD, A., AND SAMET, H. 1982. Shape approximation using quadtrees. *Pattern Recognition 15,* 1, 31–40.

REDDY, D. R., AND RUBIN, S. 1978. Representation of three-dimensional objects. CMU-CS-78-113, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Apr.

REQUICHA, A. A. G. 1980. Representations of rigid solids: Theory, methods, and systems. *ACM Comput. Surv. 12,* 4 (Dec.), 437–464.

RHEINBOLDT, W. C., AND MESZTENYI, C. K. 1980. On a data structure for adaptive finite element mesh refinements. *ACM Trans. Math. Softw. 6,* 2 (June), 166–187.

RISEMAN, E. M., AND ARBIB, M. A. 1977. Computational techniques in the visual segmentation of static scenes. *Comput. Gr. Image Process. 6,* 3 (June), 221–276.

ROBINSON, J. T. 1981. The *k*-d-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the SIGMOD Conference* (Ann Arbor, Mich., Apr.). ACM, New York, pp. 10–18.

ROSENFELD, A., ED. 1983. *Multiresolution Image Processing and Analysis.* Springer-Verlag, Berlin and New York.

ROSENFELD, A. 1984. Picture processing 1984. *Comput. Vision Gr. Image Process. 26,* 3 (June), 347–384.

ROSENFELD, A., AND KAK, A. C. 1982. *Digital Picture Processing,* 2nd ed. Academic Press, New York.

ROSENFELD, A., AND PFALTZ, J. L. 1966. Sequential operations in digital image processing. *J. ACM 13,* 4 (Oct.), 471–494.

ROSENFELD, A., SAMET, H., SHAFFER, C., AND WEBBER, R. E. 1982. Application of hierarchical data structures to geographical information systems. TR-1197, Computer Science Dept., University of Maryland, College Park, Md., June.

ROSENFELD, A., SAMET, H., SHAFFER, C., AND WEBBER, R. E. 1983. Application of hierarchical data structures to geographical information systems phase II. TR-1327, Computer Science Dept., University of Maryland, College Park, Md., Sept.

ROTH, S. D. 1982. Ray casting for modeling solids. *Comput. Gr. Image Process. 18,* 2 (Feb.), 109–144.

RUTOVITZ, D. 1968. Data structures for operations on digital images. In *Pictorial Pattern Recognition,* G. C. Cheng et al., Eds. Thompson Book Co., Washington D.C., pp. 105–133.

SAMET, H. 1980a. Region representation: Quadtrees from boundary codes. *Commun. ACM 23,* 3 (Mar.), 163–170.

SAMET, H. 1980b. Region representation: Quadtrees from binary arrays. *Comput. Gr. Image Process. 13,* 1 (May), 88–93.

SAMET, H. 1980c. Deletion in two-dimensional quad trees. *Commun. ACM 23,* 12 (Dec.), 703–710.

SAMET, H. 1981a. An algorithm for converting rasters to quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 3,* 1 (Jan.), 93–95.

SAMET, H. 1981b. Connected component labeling using quadtrees. *J. ACM 28,* 3 (July), 487–501.

SAMET, H. 1981c. Computing perimeters of images represented by quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 3,* 6 (Nov.), 683–687.

SAMET, H. 1982a. Neighbor finding techniques for images represented by quadtrees. *Comput. Gr. Image Process. 18,* 1 (Jan.), 37–57.

SAMET, H. 1982b. Distance transform for images represented by quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 4,* 3 (May), 298–303.

SAMET, H. 1983. A quadtree medial axis transform. *Commun. ACM 26,* 9 (Sept.), 680–693. See also corrigendum, *Commun. ACM 27,* 2 (Feb. 1984), 151.

SAMET, H. 1984. Algorithms for the conversion of quadtrees to rasters. *Comput. Vision Gr. Image Process. 26,* 1 (Apr.), 1–16.

SAMET, H. 1985a. A top-down quadtree traversal algorithm. *IEEE Trans. Pattern Anal. Mach. Intell. 7,* 1 (Jan.) in press. Also TR-1237, Computer Science Dept., University of Maryland.

SAMET, H. 1985b. Reconstruction of quadtrees from quadtree medial axis transforms. *Comput. Vision Gr. Image Process. 29,* 2 (Feb.) in press. Also TR-1224, Computer Science Dept., University of Maryland.

SAMET, H. 1985c. Data structures for quadtree approximation and compression. *Commun. ACM 28* in press. Also TR-1209, Computer Science Dept., University of Maryland.

SAMET, H., AND KRISHNAMURTHY, E. V. 1983. A quadtree-based matrix manipulation system. Unpublished manuscript of work in progress.

SAMET, H., AND ROSENFELD, A. 1980. Quadtree structures for image processing. In *Proceedings of the 5th International Conference on Pattern Recognition* (Miami Beach, Fla., Dec.). IEEE, New York, pp. 815–818.

SAMET, H., AND SHAFFER, C. A. 1984. A model for the analysis of neighbor finding in pointer-based quadtrees. TR-1432, Computer Science Dept., University of Maryland, College Park, Md., Aug.

SAMET, H., AND TAMMINEN, M. 1984. Efficient image component labeling. TR-1420, Computer Science Dept., University of Maryland, College Park, Md., July.

SAMET, H., AND TAMMINEN, M. 1985. Computing geometric properties of images represented by

linear quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 7,* 1 (Jan.) in press. Also TR-1359, Computer Science Dept., University of Maryland.

SAMET, H., AND WEBBER, R. E. 1982. On encoding boundaries with quadtrees. TR-1162, Computer Science Dept., University of Maryland, College Park, Md., Feb.

SAMET, H., AND WEBBER, R. E. 1983. Using quadtrees to represent polygonal maps. In *Proceedings of Computer Vision and Pattern Recognition 83* (Washington, DC, June). IEEE, New York, pp. 127–132. Also TR-1372, Computer Science Dept., University of Maryland.

SAMET, H., AND WEBBER, R. E. 1984. On encoding boundaries with quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 6,* 3 (May), 365–369.

SHAMOS, M. I., 1978. Computational geometry. Ph.D. dissertation, Dept. of Computer Science, Yale University, New Haven, Conn.

SHAMOS, M. I., AND HOEY, D. 1975. Closest-point problems. In *Proceedings of the 16th Annual IEEE Symposium on the Foundations of Computer Science* (Berkeley, Calif., Oct.). IEEE, New York, pp. 151–162.

SHNEIER, M. 1981a. Calculations of geometric properties using quadtrees. *Comput. Gr. Image Process. 16,* 3 (July), 296–302.

SHNEIER, M. 1981b. Path-length distances for quadtrees. *Inf. Sci. 23,* 1 (Feb.), 49–67.

SHNEIER, M. 1981c. Two hierarchical linear feature representations: Edge pyramids and edge quadtrees. *Comput. Gr. Image Process. 17,* 3 (Nov.), 211–224.

SLOAN, K. R., JR. 1981. Dynamically quantized pyramids. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence* (Vancouver, B.C., Aug.). Kaufmann, Los Altos, Calif., pp. 734–736.

SLOAN, K. R., JR., AND TANIMOTO, S. L. 1979. Progressive refinement of raster images. *IEEE Trans. Comput. 28,* 11 (Nov.), 871–874.

SRIHARI, S. N. 1981. Representation of three-dimensional digital images. *ACM Comput. Surv. 13,* 1 (Dec.), 399–424.

SUTHERLAND, I. E., SPROULL, R. F., AND SCHUMACKER, R. A. 1974. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv. 6,* 1 (Mar.), 1–55.

TAMMINEN, M. 1981. The EXCELL method for efficient geometric access to data. *Acta Polytech. Scand.* Mathematics and Computer Science Series No. 34, Helsinki.

TAMMINEN, M. 1982. Hidden lines using the EXCELL method. *Comput. Gr. Forum 11,* 3, 96–105.

TAMMINEN, M. 1983. Performance analysis of cell based geometric file organizations. *Comput. Vision Gr. Image Process. 24,* 2 (Nov.), 168–181.

TAMMINEN, M. 1984a. Comment on quad- and oct-trees. *Commun. ACM 27,* 3 (Mar.), 248–249.

TAMMINEN, M. 1984b. Encoding pixel trees. *Comput. Vision Gr. Image Process. 28,* 1 (Oct.), 44–57.

TAMMINEN, M., AND SAMET, H. 1984. Efficient octree conversion by connectivity labeling. In *Proceedings of the SIGGRAPH 84 Conference* (Minneapolis, Minn., July). ACM, New York, pp. 43–51.

TANIMOTO, S. 1976. Pictorial feature distortion in a pyramid. *Comput. Gr. Image Process. 5,* 3 (Sept.), 333–352.

TANIMOTO, S. 1979. Image transmission with gross information first. *Comput. Graph. Image Process. 9,* 1 (Jan.), 72–76.

TANIMOTO, S., AND KLINGER, A. EDS. 1980. *Structured Computer Vision.* Academic Press, New York.

TANIMOTO, S., AND PAVLIDIS, T. 1975. A hierarchical data structure for picture processing. *Comput. Gr. Image Process. 4,* 2 (June), 104–119.

TARJAN, R. E. 1975. Efficiency of a good but not linear set union algorithm. *J. ACM 22,* 2 (Apr.), 215–225.

TOUSSAINT, G. T. 1980. Pattern recognition and geometrical complexity. In *Proceedings of the 5th International Conference on Pattern Recognition* (Miami Beach, Fla., Dec.). IEEE, New York, pp. 1324–1346.

TROPF, H., AND HERZOG, H. 1981. Multidimensional range search in dynamically balanced trees. *Angew. Inf. 2,* 71–77.

TUCKER, L. W. 1984a. Control strategy for an expert vision system using quadtree refinement. In *Proceedings of the Workshop on Computer Vision: Representation and Control* (Annapolis, Md., Apr.). IEEE, New York, pp. 214–218.

TUCKER, L. W. 1984b. Computer vision using quadtree refinement. Ph.D. dissertation, Dept. Electrical Engineering and Computer Science, Polytechnic Institute of New York, Brooklyn, N.Y., May.

UHR, L. 1972. Layered "recognition cone" networks that preprocess, classify, and describe. *IEEE Trans. Comput. 21,* 7 (July), 758–768.

UNNIKRISHNAN, A., AND VENKATESH, Y. V. 1984. On the conversion of raster to linear quadtrees. Department of Electrical Engineering, Indian Institute of Science, Bangalore, India, May.

VAN LEEUWEN, J., AND WOOD, D. 1981. The measure problem for rectangular ranges in d-space. *J. Algorithms 2,* 3 (Sept.), 282–300.

VAN LIEROP, M. L. P. 1984. Transformations on pictures represented by leafcodes. Dept. of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands.

WARNOCK, J. L. 1969. A hidden surface algorithm for computer generated half tone pictures. TR 4-15, Computer Science Dept., University of Utah, Salt Lake City, June.

WEBBER, R. E. 1984. Analysis of quadtree algorithms. Ph.D. dissertation, Computer Science Dept., University of Maryland, College Park,

Md., Mar. Also TR-1376, Computer Science Dept., University of Maryland.

WEBER, W. 1978. Three types of map data structures, their ANDs and NOTs, and a possible OR. In *Proceedings of the 1st International Advanced Study Symposium on Topological Data Structures for Geographic Information Systems*, G. Dutton, Ed. Harvard Papers on Geographic Information Systems, Harvard Univ. Press, Cambridge, Mass.

WILLARD, D. E. 1982. Polygon retrieval. *SIAM J. Comput. 11*, 1 (Feb.), 149–165.

WOODWARK, J. R. 1982. The explicit quadtree as a structure for computer graphics. *Comput. J. 25*, 2 (May), 235–238.

WU, A. Y., HONG, T. H., AND ROSENFELD, A. 1982. Threshold selection using quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell. 4*, 1 (Jan.), 90–94.

YAMAGUCHI, K., KUNII, T. L., FUJIMURA, K., AND TORIYA, H. 1984. Octree-related data structures and algorithms. *IEEE Comput. Gr. Appl. 4*, 1 (Jan.), 53–59.

YAU, M. 1984. Generating quadtrees of cross-sections from octrees. *Comput. Vision Gr. Image Process. 27*, 2 (Aug.), 211–238.

YAU, M., AND SRIHARI, S. N. 1983. A hierarchical data structure for multidimensional digital images. *Commun. ACM 26*, 7 (July), 504–515.

YERRY, M. A., AND SHEPARD, M. S. 1983. A modified quadtree approach to finite element mesh generation. *IEEE Comput. Gr. Appl. 3*, 1 (Jan./Feb.), 39–46.

Received November 1983; final revision accepted June 1984.