72. B. J. Schachter, L. S. Davis and A. Rosenfeld, Some experiments in image segmentation by clustering of local feature values, *Pattern Recognition* **11**, 1978, 19–28.
73. B. J. Schachter and A Rosenfeld, Some new methods of detecting step edges in digital pictures, *Comm. ACM* **21**, 1978, 172–176.
74. S. D. Shapiro, Feature space transforms for curve detection, *Pattern Recognition* **10**, 1978, 129–143.
75. J. M. Tenenbaum and H. G. Barrow, IGS: A paradigm for integrating image segmentation and interpretation, *in* "Pattern Recognition and Artificial Intelligence" (C. H. Chen, ed.), pp. 472–507. Academic Press, New York, 1976.
76. W. B. Thompson, Textural boundary analysis, *IEEE Trans. Comput.* **26**, 1977, 272–276.
77. J. T. Tou, Zoom-thresholding technique for boundary determination, *J. Comput. Informat. Sci.* **8**, 1979, 3–8.
78. G. J. VanderBrug, Semilinear line detectors, *Comput. Graphics Image Processing* **4**, 1975, 287–293.
79. G. J. VanderBrug and A. Rosenfeld, Linear feature mapping, *IEEE Trans. Systems Man Cybernet.* **8**, 1978, 768–774.
80. H. Wechsler and M. Kidode, A new edge detection technique and its implementation, *IEEE Trans. Systems Man Cybernet.* **7**, 1977, 827–836.
81. J. S. Weszka, A survey of threshold selection techniques, *Comput. Graphics Image Processing* **7**, 1978, 259–265.
82. J. S. Weszka and A. Rosenfeld, Threshold evaluation techniques, *IEEE Trans. System Man. Cybernet.* **8**, 1978, 622–629.
83. J. S. Weszka and A. Rosenfeld, Histogram modification for threshold selection, *IEEE Trans. Systems Man Cybernet.* **9**, 1979, 38–52.
84. Y. Yakimovsky, Boundary and object detection in real world images, *J. ACM* **23**, 1976, 599–618.
85. S. W. Zucker, Region growing: childhood and adolescence, *Comput. Graphics Image Processing* **5**, 1976, 382–399.

Chapter 11

Representation

Segmentation decomposes a picture into subsets or regions. Geometrical properties of these subsets—connectedness, size, shape, etc.—are often important in picture description. As we shall see, there are many methods of measuring such properties; the preferred method usually depends on how the subsets are represented. This chapter discusses various representation schemes and their application to geometric property measurement.

In this chapter, Σ denotes a picture; subsets of Σ are denoted by $S, T, \ldots$, and points by $P, Q, \ldots$.

In general, a segmented picture Σ is partitioned into a collection of nonempty subsets $S_1, \ldots, S_m$ such that $\bigcup_{i=1}^{m} S_i = \Sigma$ and $S_i \cap S_j = \varnothing$ for all $i \neq j$. An important special case is $m = 2$, i.e., Σ consists of a set S and its complement $\bar{S}$. (In the general case, we can take any of the S_i as S, so that $\bigcup_{j=1, j\neq i}^{m} S_j = \bar{S}$.) Note that the S's are not necessarily connected; connectedness will be treated in Sections 11.1.7 and 11.3.1.

11.1 REPRESENTATION SCHEMES

Any subset S of a digital picture Σ can be represented by a binary picture (or "bit plane") χ_S of the same size as Σ, having 1's at the points of S and 0's elsewhere. If Σ is $n \times n$, the χ_S representation requires n^2 bits. χ_S can be

regarded as the *characteristic function* of the subset S; this is the function that maps points of S into 1 and points of $\bar{S}$ into 0.

More generally, any partition of Σ into $S_1, \ldots, S_m$ can be represented by an m-valued picture having i's at the points of S_i, $1 \leqslant i \leqslant m$. In particular, if Σ is any picture, then in this sense, Σ represents its own partition into sets of constant gray level, i.e., S_i is the set of points of Σ having gray level i. For an $n \times n$ picture, this representation requires $n^2 \log_2 m$ bits.

The storage requirements of this trivial representation are the same for all partitions of Σ into a given number of sets. Our main interest in this chapter is to study representations which are more economical for "simple" partitions. In the following sections we will define a variety of such representations.

Exercise 1. If the subsets $S_1, \ldots, S_{m-1}$ consist of only a few points, they can be specified by listing the coordinates of these points. For example, if $m = 2$, and $S = S_1$ consists of k points, the list requires $2k \log_2 n$ bits ($2 \log_2 n$ bits for the coordinates of each point) for an $n \times n$ picture, as compared with n^2 bits for the bit plane representation, and is more economical if $k < n^2/2 \log_2 n$. How many bits are required for arbitrary m when there are k points? ▮

11.1.1 Rows

a. Runs

Each row of a picture consists of a sequence of maximal runs of points such that the points in each run all have the same value. Thus the row is completely determined by specifying the lengths and values of these runs. If there are only a few runs, this representation is very economical; for this reason, *run length coding* is sometimes used for picture compression, as mentioned in Section 5.8. For example, suppose that the row has length n, and there are r runs. Since it takes $\log_2 n$ bits to specify the length of a run (it may have any length between 1 and n), the number of bits needed to specify all the run lengths is $r \log_2 n$. Thus if there are m possible values, this representation of the row requires $r(\log_2 n + \log_2 m)$ bits, as compared with the $n \log_2 m$ bits that are required when the row is treated as a string of length n.

When $m = 2$, we need only specify the value of the first run in the row, since the values must alternate. Thus the run length specification of a row of a binary picture requires $1 + r \log_2 n$ bits, as compared with the n bits required to represent the row as a bit string. Note that these savings are only one-dimensional; for an $n \times n$ picture, if the average number of runs in each row is r, the total number of bits required by the run length representation is $n(1 + r \log_2 n)$ as compared with n^2 in the binary case, or $nr(\log_2 m + \log_2 n)$ as compared with $n^2 \log_2 m$ in the general case.

b. Binary trees

Suppose, for simplicity, that the row length is a power of 2, say $n = 2^k$. We shall now describe a method of representing the row by a binary tree, each of whose leaves corresponds to a (not necessarily maximal) run of constant value whose length is a power of 2, say 2^i, and whose position coordinate is a multiple of 2^i.

The root node of the tree represents the entire row. If the row all has one value, we label the root node with that value and stop; in this case, the tree consists only of the root node. Otherwise, we add two descendants to the root node, representing the two halves of the row. The process is then repeated for each of these new nodes: if its half of the row has constant value, we label it with that value and do not give it any descendants; if not, we give it two descendants corresponding to the two halves of its half. In general, at level h in the tree (where the root is at level 0), the nodes (if any) represent pieces of the row of length 2^{k-h}, in positions which are multiples of 2^{k-h}. If a piece has constant value, its node is a leaf node (i.e., it has no descendants), labeled with that value; otherwise, that node has two descendants, corresponding to the two halves of the piece. At level k, the nodes (if any) correspond to single pixels, and are all leaf nodes, labeled with the values of their pixels.

A simple example of a string and the corresponding binary tree is shown in Fig. 1. Note that the blocks of constant value corresponding to the leaf nodes of the tree are not necessarily maximal runs; if the length of a run is not

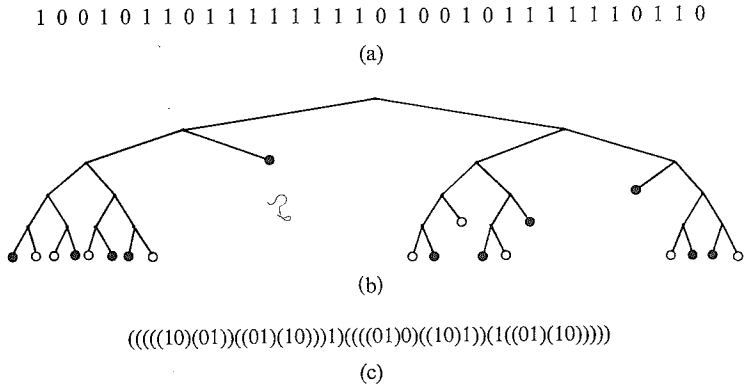


Fig. 1 Binary tree representation of a string. (a) Binary string of length 32. (b) Tree; solid and open circles are leaf nodes corresponding to blocks of 1's and 0's, respectively. There are 39 nodes, 20 of which are leaf nodes. (c) Parenthesis string representation of the tree. The run length representation of this string is 1:1,2,1,1,2,1,8,1,1,2,1,1,6,1,2,1, where the initial 1 represents the fact that the string begins with a run of 1's; there are 16 runs.

a power of 2, or if its starting position is not a multiple of the proper power of 2, it is represented by more than one leaf node.

The space required to store the tree is proportional to the number of nodes. (For example, the tree can be represented by a parenthesis string in which each node maps into a pair of parentheses enclosing the substring representing the subtree rooted at that node, as illustrated in Fig. 1c. The leaf nodes can be represented by their associated values.) If the row consists of only a few runs, the tree will have relatively few nodes, but the exact number depends on the positions and lengths of the runs.

The binary tree representation of the rows of a picture does not seem to have been used in practice. It was presented here as a preliminary to the two-dimensional quadtree representation in Section 11.1.2b.

11.1.2 Blocks

a. MATs

With each point P of the picture Σ, let us associate the set of upright squares of odd side lengths n centered at P. (Our discussion generalizes to any family of mutually similar shapes, but for simplicity we will assume that they are upright squares and that they all have P at their centers.) Let S_P be the largest such square that is contained in Σ and has constant value, and let r_P be the radius of S_P. There may exist other points Q such that S_P is contained in S_Q; if no such Q exists, we call S_P a *maximal block.*

It is easy to see that if we specify the set of centers P, radii r_P, and values v_P of the maximal blocks, Σ is completely determined, since any point of Σ lies in at least one maximal block. In fact, we need only do this for $m - 1$ of the values; the points not covered by any of these blocks must have the omitted value. Thus in the case where there are only two values, we need only specify the blocks for one value. The maximal blocks for a simple picture are shown in Fig. 2.

The set of centers and radii (and values) of the maximal blocks is called the *medial axis* (or *symmetric axis*) *transformation* of Σ, abbreviated MAT or SAT. It has this name because the centers are located at midpoints, or along local symmetry axes, of the regions of constant value in Σ. Intuitively, if a block S_P is maximal and is contained in the constant-value region S, it must touch the border of S in at least two places; otherwise we could find a neighbor Q of P that was farther away than P from the border of S, and then S_Q would contain S_P. A more rigorous treatment of these ideas will be given in Section 11.2.1.

Evidently, if Σ consists of only a few constant-value regions which have simple shapes (i.e., which are unions of only a few blocks), its MAT representation will be quite compact. For an $n \times n$ picture, $2 \log_2 n$ bits are needed to specify the coordinates of each block center, and $(\log_2 n) - 1$ bits to specify the radius; thus if there are b blocks in the MAT, the total number of bits required to specify it is $b(3 \log_2 n + \log_2(m - 1) - 1)$.

0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	0
1	1	1	1	1	1	0	0
1	1	1	1	1	1	0	0

(a)

x	y	r
3	3	2
3	7	0
4	3	2
6	4	1
6	6	0

(b)

Fig. 2 Simple example of a MAT representation. (a) 8×8 binary picture, with centers of MAT blocks (of 1's) underlined. (b) Center coordinates and radii of MAT blocks (lower left corner has coordinates (1,1)).

Note that the MAT representation may still be redundant, i.e., some blocks may be contained in unions of others. There does not seem to be any simple way of reducing this redundancy without carrying out a lengthy search process.

b. Quadtrees

Maximal blocks can be of any size and in any position; they are analogous to runs in the one-dimensional case. (Maximal connected regions of constant value are also analogous to runs, but they are not good primitive elements for representation purposes, since they themselves cannot be specified compactly; a block, on the other hand, is defined by specifying its center and radius.) We next describe a two-dimensional representation based on trees of degree 4; it is analogous to the binary tree representation for rows. We assume for simplicity that the size of the picture Σ is $2^k \times 2^k$.

The root node of the tree represents the entire picture. If the picture has all one value, we label the root node with that value and stop. Otherwise, we add four descendants to the root node, representing the four quadrants of the picture. The process is then repeated for each of these new nodes; and so on. In general, the nodes at level h (if any) represent blocks of size $2^{k-h} \times 2^{k-h}$, in positions whose coordinates are multiples of 2^{k-h}. If a block has constant value, its node is a leaf node; otherwise, its node has four descendants at level

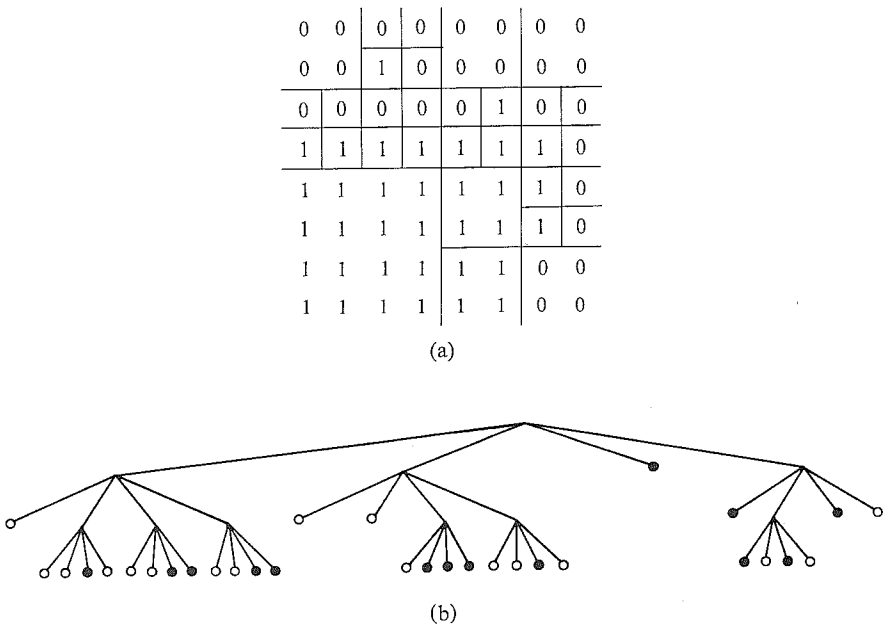


Fig. 3 Quadtree for the picture in Fig. 2. (a) Picture, with quadtree blocks marked. (b) Tree; notation is as in Fig. I. There are 41 nodes, 31 of which are leaf nodes. The order of the sons on each row is NW, NE, SW, SE.

$h + 1$, corresponding to the four quadrants of the block. The nodes at level k, if any, are all leaf nodes corresponding to single pixels.

The tree constructed in this way is called a *quadtree*, since its nonleaf nodes all have degree 4. A simple example of a picture and its corresponding quadtree is shown in Fig. 3. The space required to store the tree is proportional to the number of nodes, as in the one-dimensional case. Note that, unlike the MAT, the quadtree representation is not redundant.

The chief disadvantage of the quadtree representation is that, unlike the nontree representations considered here, it is shift-variant. Two pictures that differ only by a translation may give rise to very different quadtrees. Thus it is hard to tell from their quadtree representations whether two pictures are congruent.

11.1.3 Borders

All of the representations considered up to now are based on maximal runs or blocks of constant value, possibly restricted as to size and position. These

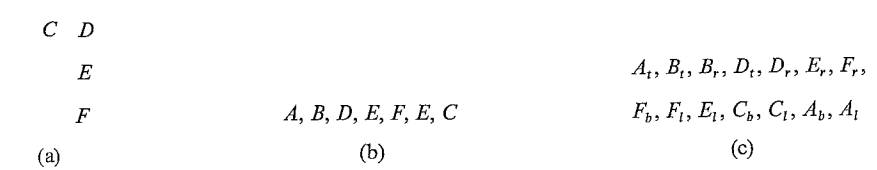


Fig. 4 Methods of defining border sequences. (a) Set S; each point is labeled with a different letter. In this simple example there are no interior points. (b) Clockwise sequence of points around the border, beginning with A. Note that point E is visited twice. (c) Clockwise sequence of "cracks" around the border, beginning with the crack A_t at the top of A. The subscripts t, r, b, l denote top, right, bottom, and left, respectively. Note that if we delete the subscripts, and eliminate consecutive repetitions of the same point, we obtain the point sequence (b).

approaches represent each S_i as the union of the maximal runs or blocks that are contained in it.

Another class of approaches to representation makes use of the fact that the sets S_i are determined by specifying their borders. (Compare the discussion of contour coding in Section 5.8.) The border S' of a set S is the set of points of S that are adjacent to points of the complement $\bar{S}$. We can regard S' as consisting of a set of closed curves; each of these curves contains the points that belong to a particular connected component of S and are adjacent to a particular connected component of $\bar{S}$.[§] These concepts will be defined more precisely in Sections 11.1.7 and 11.2.2, and algorithms given for finding the border curves of a given S and for reconstructing S from its borders. In this section we will only discuss how to specify border curves.

A border curve is determined by specifying a starting point and a sequence of moves around the border. Figure 4 illustrates two ways in which this can be done; one approach moves along a sequence of border points of the set S, while the other moves along a sequence of "cracks" between the points of S and the adjacent points of $\bar{S}$.

In moving from point to point around a border, we always go from a point to one of its eight neighbors. Let us number the neighbors as follows:

3 2 1
4 0
5 6 7

(mnemonic: neighbor i is in direction $45i°$, measured counterclockwise from the positive x-axis). Thus each move is defined by one of the digits $0, 1, \ldots, 7$,

[§] More generally, the points of any S_i that are adjacent to any given S_j consist of a set of arcs.

i.e., by an octal digit. For example, the sequence of moves used in Fig. 4b corresponds to 0766233. A sequence of moves represented by octal digits in this way is called a *chain code*. A border is thus defined by giving the coordinates of a starting point together with a chain code representing a sequence of moves.

If we follow the cracks around a border, at each move we are going either left, right, up, or down; if we denote direction $90i°$ by i, these moves can be represented by a sequence of 2-bit numbers (0, 1, 2, 3). For example, the sequence in Fig. 4c is represented by 00303332112121. We shall call this representation a *crack code*. A border is specified by giving the coordinates of a starting crack together with a crack code.

Each move in a crack code is represented by a 2-bit number, while chain code moves require 3-bit numbers; but the number of moves in crack following is somewhat greater than that in chain code, since several moves may be required to traverse the cracks around a single point. If we assume that at most half of the border points of S have two neighbors in $\bar{S}$ (i.e., are corners or "waists" of S), and that border points having three neighbors in $\bar{S}$ are rare, then the average number of cracks per border point is at most $1\frac{1}{2}$. Thus the number of bits required to represent a border by a crack code should be no greater than the number required to represent it by a chain code.

The storage requirements of these border code schemes depend on the total border length of the sets S_i. In an $n \times n$ picture, there are $2n(n+1)$ cracks, including those around the edges of the picture. If fraction β of these cracks are border cracks, we need about $8\beta n(n+1)$ bits to represent all the crack codes (two bits per move), since each border crack—ignoring the edges of the picture—belongs to two border curves.§ As indicated in the preceding paragraph, chain codes would require a similar number of bits. In addition, for each border curve we need to specify starting point coordinates ($2 \log_2 n$ bits), as well as which set S_i lies on (say) the right as the border is traversed [$\log_2 m$ or $\log_2(m-1)$ bits.] Thus if there are B borders, the total number of bits required for this type of representation is $8\beta n(n+1) + B(\log_2 m + 2 \log_2 n)$. Note that B itself may be on the order of n^2, if there are many small connected components; in this case, border representations would not be economical.

For any given chain code, we can construct a *difference chain code* whose values represent the successive changes in direction, e.g., let 0 represent no turn, ± 1 represent 45° right or left turns, ± 2 and ± 3 similarly represent 90°

§ It suffices to specify the borders for $m-1$ of the subsets S_i, so that some of these border curves can be omitted. In particular, if $m = 2$, we need only specify the border curves of S, so that each border crack is on only one border curve, and the crack codes require only about $4\beta n(n+1)$ bits.

and 135° turns, and 4 represent a 180° turn. Thus the difference chain code, like the chain code, has eight possible values and requires three bits per move. However, the values are no longer equally likely, e.g., 0 and 1 should be very common, while 4 is rare. (What does this imply about the compressibility of the difference code?) Evidently, a border is determined by specifying the coordinates of the starting point, the starting direction, and the difference chain code. The case of the *difference crack code* is analogous; here there are only three possible difference values, 0° and $\pm 90°$.

Exercise 2. If borders often contain long straight segments, their chain or crack codes can be further compressed by run length coding. Analyze the savings that can be obtained in this way. Also, discuss the possibilities for compressing difference codes using run length coding (note, in particular, that turns in a given direction cannot occur in long runs). ∎

Chain codes can also be used for the digital representation of plane curves. The chain code of a given curve C can be constructed as follows: Imagine a Cartesian grid superimposed on C. As we move along C, whenever C enters a grid square, we take the nearest corner of that square as a point on the digitization of C. (If C enters at the midpoint of a side of the square, use any standard rounding convention to pick the "nearest" corner.) When C enters and then leaves a grid square, the two successive grid points (= grid square corners) on the digitization are either the same or are neighbors in the grid, since they are corners of the same square. Thus the sequence of grid points constituting the digitization can be represented by a starting point and a chain code defining the sequence of moves from neighbor to neighbor. Chain codes of digital curves will be discussed further in Sections 11.1.5c and 11.2.3.

11.1.4 Representations of Derived Sets

We often need to define new subsets or partitions of a picture in terms of given ones, e.g., by performing set-theoretic operations on the given ones, or by resegmenting them. If we are using a particular type of representation, we would like to be able to derive the representations of the new sets directly from the representations of the original sets. Methods of doing this in particular situations will be described during the course of this chapter; e.g., see Sections 11.3.1a and 11.3.1b on connected component labeling using various representations. In this section we briefly discuss the problem of performing set-theoretic operations on representations of a given type. For simplicity, we deal only with partitions into a set S and its complement.

If S and T are represented as binary arrays χ_S and χ_T, it is trivial to obtain $\bar{S}$, $S \cup T$, $S \cap T$, etc., by pointwise Boolean operations on these arrays. Such

operations can be done in a single parallel step on a cellular array computer, or they can be done in a single scan of χ_S and χ_T on a conventional computer.

Given the run length representation of S (and T), we get that of $\bar{S}$ by simply reversing the designation of the first value on each row. In the remainder of this paragraph we describe an algorithm that creates the run list L for $S \cap T$ on a given row from the lists L_1 and L_2 of S and T. Analogous algorithms can be given for other Boolean functions. If L_1 and L_2 both begin with runs of 1's, so does L; otherwise, L begins with 0. We will now describe how to successively add runs to L by examining the initial parts of L_1 and L_2; each time we do this, we delete or truncate the initial run(s) in L_1 and L_2. The process is then repeated using the shortened L_1 and L_2. When they are empty, we have the desired L. A counter is also associated with L, and is initially set at 0; its role will be explained in step (b) of the algorithm.

(a) Suppose L_1 and L_2 (currently) both begin with runs of 1's, ρ_1 and ρ_2, say of lengths $|\rho_1| \leqslant |\rho_2|$. In this case we add a run of 1's of length $|\rho_1|$ to b; delete the initial run from L_1; and shorten the initial run of L_2 to length $|\rho_2| - |\rho_1|$ (or delete it, if $|\rho_1| = |\rho_2|$). Note that at least one of L_1 and L_2 now begins with a run of 0's.

(b) Suppose L_1 begins with a run ρ_1 of 0's, and L_2 begins with a run of 1's or with a shorter run of 0's. Add up the run lengths in L_2 until $|\rho_1|$ is reached, i.e.,

$$|\rho_{21}| + |\rho_{22}| + \cdots + |\rho_{2,k-1}| \leqslant |\rho_1| < |\rho_{21}| + |\rho_{22}| + \cdots + |\rho_{2k}|$$

(b1) If ρ_{2k} is a run of 1's, truncate it to length

$$|\rho_{2k}| - (|\rho_1| - |\rho_{21}| - |\rho_{22}| - \cdots - |\rho_{2,k-1}|);$$

delete ρ_1 from L_1; add a run of 0's of length $|\rho_1| + C$ to L, where C is the value in the counter; and reset the counter to 0.

(b2) If ρ_{2k} is a run of 0's, truncate it as above; delete ρ_1 from L_1; and add $|\rho_1|$ to the counter. In this case L_2 still begins with a run of 0's, so we are still in case (b), possibly with the roles of L_1 and L_2 reversed.

Given the MATs of S and T, we get a redundant MAT for $S \cup T$ by simply taking their union. A procedure for constructing a MAT for $S \cap T$ by finding maximal intersections of blocks in the MATs of S and T is described in [49]; the details will not be given here.

The quadtree of $\bar{S}$ is the same as that of S with "black" leaf nodes (= nodes corresponding to blocks of 1's) changed to "white" and vice versa. To get the quadtree of $S \cup T$ from those of S and T, we traverse the two trees simultaneously. Where they agree, the new tree is the same. If S has a gray (= nonleaf) node where T has a black node, the new tree gets a black node; if T has a white node there, we copy the subtree of S at that gray node into the new tree; if S has a white node and T a black node, the new tree gets a black node. The algorithm for $S \cap T$ is exactly analogous, with the roles of black and white reversed. The time required for these algorithms is proportional to the number of nodes in the smaller of the two trees [27, 72].

The crack codes of the borders of $\bar{S}$ are the same as those of S (but in reverse order, if we want to maintain a convention as to the order of border following). Constructing chain codes of the borders of $\bar{S}$ from those of S is somewhat more complex; the details will be omitted here. For either crack or chain codes, it is quite complicated to construct the codes of $S \cup T$ of $S \cap T$ from those of S and T; in particular, this involves finding which pairs of borders surround one another, and also finding all the intersections of each pair of borders. Border representations are not well suited for performing set-theoretic operations.

11.1.5 Approximate Representations

The representations considered so far in this section are all exact, i.e., they completely determine the given partition of Σ. In the following paragraphs we discuss methods of obtaining approximations to a partition based on these representations. Here again, for simplicity, we will usually assume that the partition consists of a set S and its complement.

It should be pointed out that the representations of a set S considered in this chapter are quite sensitive to noise. For example, if we make a tiny hole in S, its various types of maximal-block representations may change substantially (see Fig. 5), and it also acquires a new border. To minimize this problem, one can noise-clean the picture (Section 6.4) before segmenting it; one can noise-clean S by a shrinking and expanding process (see Section 11.3.2d); or one can use approximate representations of S, which should be less sensitive to the presence of noise.

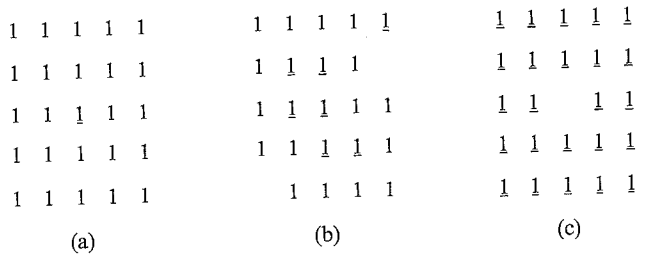


Fig. 5 Effects of noise on the MAT representation: A simple object (a) and two noisy versions (b) and (c), with centers of MAT blocks underlined.

a. *Arrays*

The simplest method of approximation is simple resolution reduction. We can reduce the resolution of a picture by resampling it coarsely; this is equivalent to digital demagnification (see Section 9.3 on geometric transformations of digital pictures). We can either demagnify Σ, obtaining Σ' (say), and then segment Σ' the same way that we segmented Σ; or we can demagnify the segmented Σ (i.e., χ_S). Note that in Σ', each gray level is (in general) a weighted average of a block of gray levels of Σ; thus if we demagnify χ_S, the result may no longer be binary, but we can make it binary again by thresholding. If desired, we can remagnify to make the simplified picture the same size as the original one for display purposes.

On the advantages of using reduced-resolution pictures to obtain preliminary information about which parts of Σ to analyze, see Sections 9.4.4 and 10.2.3a. A "pyramid" of successive reductions, e.g., each half the size of the preceding (n by n, $n/2$ by $n/2$, $n/4$ by $n/4$, ...) provides a wide range of resolutions, at least one of which should be approximately right for any desired purpose. The total storage space required for this pyramid is less than $1\frac{1}{3}$ times that required for Σ alone ($1 + \frac{1}{4} + (\frac{1}{4})^2 + \cdots = 1\frac{1}{3}$).

Other types of picture approximation, not involving reduced resolution, were discussed in Section 10.4.3 in connection with picture partitioning.

b. *Blocks*

When S is represented by maximal blocks (or runs, etc.) we can simplify it by eliminating some of the blocks. For example, if we eliminate small blocks from the MAT, we retain the gross features of S and lose only some of the details (see Fig. 6). A related method of simplifying S by shrinking or expanding it will be discussed in Section 11.3.2d. Eliminating low-level nodes from a quadtree representation may result in substantial simplification, since it may lead to the elimination of nodes at higher levels.

Another possibility is to use a representation based on blocks whose values are only approximately constant (e.g., have variances less than some threshold). To generalize the MAT [1], we consider the set of squares centered at each point P, and let S_P be such that all the smaller squares have below-threshold variances, while the next larger square does not. Maximal

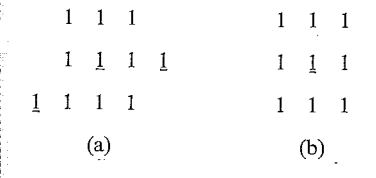


Fig. 6 Simplification by elimination of small MAT blocks. (a) Object, with centers of MAT blocks underlined. (b) Simplified object resulting from ignoring blocks having radius 0.

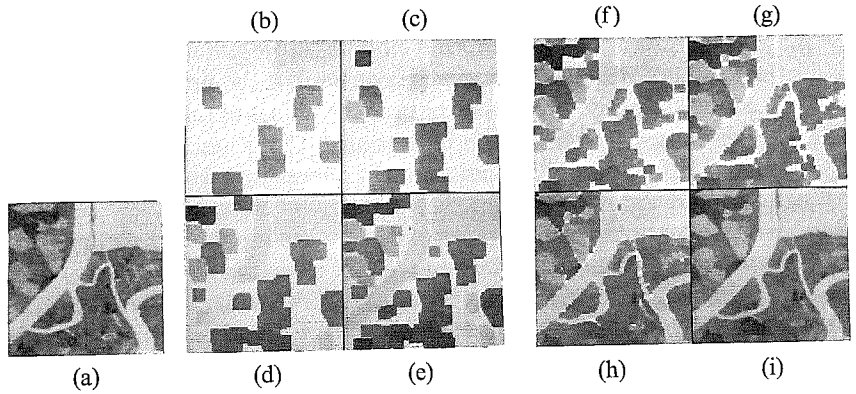


Fig. 7 Picture approximations by maximal low-variance blocks. (a) Picture; (b)–(i) blocks of radii 7, 7 and 6, 7 and 6 and 5, ..., 7 and 6 and ··· and 0, displayed with their mean gray levels. When blocks having different radii overlap, the mean gray level of the smaller block is used; when blocks of the same radius overlap, the maximum of their gray levels is used.

blocks defined in this way are illustrated in Fig. 7. The other maximal-block representations (runs, trees) can be generalized analogously.

If S is not too noisy, the points of its MAT (in brief: its *medial axis*, MA) will tend to lie along a set of arcs or curves. Thus we can represent S, at least approximately, by specifying these curves (e.g., by chain codes) and defining a "radius function" along each curve [7]. This defines a set of "generalized ribbons" (i.e., arbitrarily curved ribbons of varying width) whose union is S. Unfortunately, it is not obvious how to find a simple set of curves that contain the MA; see Section 11.2.1a.

c. *Borders and curves*

Borders can be approximated piecewise linearly, i.e., by polygons (or, analogously, using pieces of higher-order curves) in various ways. Some simple methods of constructing and refining polygonal approximations to a border or curve are briefly described in Section 11.3.3c.

Given a set of successive approximations to a border, these approximations define a tree structure in which each node represents a polygon side, and the sons of a node are its immediate refinements; the sides of the coarsest polygon are the sons of the root node. If desired, we can associate with each polygon side a rectangular strip that just contains the arc; these strips define the zone in which the border might lie [5].

A *line drawing* is a picture that can be segmented into a set of everywhere elongated subsets and a background. (This is a rather imprecise definition; elongatedness will be defined more precisely in Section 11.3.4b.) In this case

the subsets themselves can be approximated by unions of arcs and curves; this is a special case of the "generalized ribbon" representation in which the radius function is constant (or piecewise constant, if the lines have different thicknesses). "Thinning" processes that reduce elongated regions to unions of arcs and curves will be described in Section 11.2.3. These unions can then be further separated into individual arcs and curves by breaking them apart at junctions, and the individual curves can be represented by chain codes. (Here crack codes are not appropriate, since we are representing a one pixel thick curve, not a region border; but chain codes are applicable, since successive points on a curve are neighbors.) Alternatively, the curves can be approximated by polygons (etc.), as above; or one can use generalized chain codes based on moves from neighbor to neighbor in which a larger neighborhood is allowed.

11.1.6 Representation of Three-Dimensional Objects

In analyzing the three-dimensional structure of a scene from one or more pictures, it is often necessary to represent how the scene is divided into objects of various types and empty space. In principle, this could be done using a 3-d array in which the values represent the types of objects; for example, in a binary array 1's could represent points occupied by objects and 0's could represent unoccupied points. However, except for very small sizes, 3-d arrays require a prohibitively large amount of storage; even a $64 \times 64 \times 64$ binary array requires $2^{18} > 250{,}000$ bits. Thus compact representations and approximations become especially important in the 3-d case.

One approach to 3-d representation is to regard the 3-d array as a sequence of 2-d arrays ("slices" or "serial sections"), just as in the two-dimensional case we regarded a picture as a sequence of rows. Any of the representation schemes described in this chapter can then be used to represent the individual slices.

Another possibility is to use 3-d blocks. The 3-d analog of the MAT is based on maximal upright cubes (or blocks of some other standard shape) of constant value centered at each point. The analog of the quadtree is the "octree," obtained by recursive subdivision of the 3-d array (which we assume to be $2^k \times 2^k \times 2^k$) into octants until octants of constant value are reached.

A class of approximate representations known as *generalized cones* (or *cylinders*) has been widely used to represent three-dimensional objects [42]. A generalized cone is defined by an axis, a cross-section shape, and a size function; it is the volume swept out as the shape moves along the axis (perpendicular to it, or at some other fixed angle, and with the axis passing through it at a specified reference point), changing size from position to

position as specified by the size function. (Compare the remarks on "generalized ribbons" in Section 11.1.5b.)

The border of a 3-d region consists of a set of surfaces. The slope of a surface at a point is defined by a pair of angles that specify how the normal to the surface at that point is oriented. Thus the analog of chain or crack code, in the case of a surface, would be an array of quantized values representing the pair of slopes for each unit patch of the surface. On 3-d chain codes for space curves see [19].

Exercise 3. Generalize as many as possible of the techniques in this chapter to the 3-d case. ▮

Various types of three-dimensional information about a scene can be represented in the form of two-dimensional arrays. For example, given a picture of a scene, the range from the observer to each visible point, and the surface slope at each visible point, can be represented as arrays in register with the picture. Note that the slope array is an array of vectors; Marr has called this array the "$2\frac{1}{2}$-d sketch." It should be pointed out that the three-dimensional resolution of such an array varies from point to point of the scene; as the slope of the surface becomes very oblique, the size of the surface patch that maps into a unit area on the picture becomes very large, so that the range and slope information at that point of the array become unreliable.

On the extraction of three-dimensional information about a scene from pictures of the scene, e.g., slope from shading or perspective, and relative range from occlusion cues, see the Appendix to Chapter 12.

11.1.7 Digital Geometry

This section introduces some of the basic geometric properties of subsets of a digital picture. Earlier we mentioned the concept of connectedness for such subsets; we now define this concept more precisely. We also define the borders of a subset, and indicate why they can be regarded as closed curves. Some other concepts of digital geometry, including some useful digital distance functions such as "city block" and "chessboard" distance, will also be introduced.

The results presented in this section are not all self-evident; many of them require rather lengthy proofs. We will not give such proofs here; for a mathematical introduction to the subject see [55].

A point $P = (x, y)$ of a digital picture Σ has four horizontal and vertical neighbors, namely the points

$$(x - 1, y), \quad (x, y - 1), \quad (x, y + 1), \quad (x + 1, y)$$

We will call these points the 4-*neighbors* of P, and say that they are 4-*adjacent* to P. In addition, P has four diagonal neighbors, namely

$$(x-1, y-1), \quad (x-1, y+1), \quad (x+1, y-1), \quad (x+1, y+1)$$

These, together with the 4-neighbors, are called 8-*neighbors* of P (8-*adjacent* to P). Note that if P is on the border of Σ, some of its neighbors may not exist. If S and T are subsets of Σ, we say that S is 4- (8-) adjacent to T if some point of S is 4- (8-) adjacent to some point of T.

Exercise 4. Define the 6-*neighbors* of (x, y) as the 4-neighbors together with

$$(x-1, y-1) \text{ and } (x-1, y+1), \quad \text{if } y \text{ is odd}$$
$$(x+1, y-1) \text{ and } (x+1, y+1), \quad \text{if } y \text{ is even}$$

These can be regarded as the neighbors of (x, y) in a "hexagonal" array constructed from a square array by shifting the even-numbered rows half a unit to the right. Develop "hexagonal" versions of all the concepts in this section. ∎

a. *Connectedness*

A *path* π of length n from P to Q in Σ is a sequence of points $P = P_0, P_1, \ldots, P_n = Q$ such that P_i is a neighbor of P_{i-1}, $1 \leqslant i \leqslant n$. *Note that there are two versions of this and the following definitions, depending on whether "neighbor" means "4-neighbor" or "8-neighbor."* Thus we can speak of π being a 4-*path* or an 8-*path.*

Let S be a subset of Σ, and let P, Q be points of S. We say that P is (4- or 8-) *connected* to Q in S if there exists a (4- or 8-) path from P to Q consisting entirely of points of S. For any P in S, the set of points that are connected to P in S is called a connected *component* of S. If S has only one component, it is called a connected set. For example, if we denote the points of S by 1's, the set

1

1

is 8-connected but not 4-connected.

It is easily seen that "is connected to" is reflexive, symmetric, and transitive, and so is an equivalence relation, i.e., for all points P, Q, R of S:

(a) P is connected to P (hint: a path π can have length $n = 0$);
(b) if P is connected to Q, then Q is connected to P;
(c) if P is connected to P and Q to R, then P is connected to R.

Thus two points are connected to each other in S iff they belong to the same component of S.

b. *Holes and surroundness*

Let $\bar{S}$ be the complement of S. We will assume, for simplicity, that the border Σ' of Σ (i.e., its top and bottom rows and its left and right columns) is in $\bar{S}$. The component of $\bar{S}$ that contains Σ' is called the *background* of S. All other components of $\bar{S}$, if any, are called *holes* in S. If S is connected and has no holes, it is called *simply connected*; if it is connected but has holes, it is called *multiply connected.*

In dealing with connectedness in both S and $\bar{S}$, it turns out to be desirable to use opposite types of connectedness for S and $\bar{S}$, i.e., if we use 4- for S, then we should use 8- for $\bar{S}$, and vice versa. This will allow us to treat borders as closed curves; see Subsection (c). We will adopt this convention from now on.

Let S and T be any subsets of Σ. We say that T *surrounds* S if any path from any point of S to the border of Σ must meet T, i.e., if for any path $P_0, P_1, \ldots, P_n$ such that P_0 is in S and P_n is on the picture border, some P_i must be in T. Evidently the background of S surrounds S. On the other hand, S surrounds any hole in S; if it did not, there would be a path from the hole to the picture border that did not meet S, contradicting the fact that the hole and the border are in different components of $\bar{S}$. The type of path here (4- or 8-) must be the same as the type of connectedness used for $\bar{S}$.

Exercise 5.

(a) Prove that any S surrounds itself, and that if W surrounds V and V surrounds U, then W surrounds U.

(b) Can S and T surround each other without being the same? (Hint: Can a proper subset of S surround S?)

(c) Prove that if S and T are disjoint, then only one of them can surround the other. (Thus for disjoint sets, "surrounds" is a strict partial order relation, i.e., is irreflexive, antisymmetric, and transitive.) ∎

c. *Borders*

As indicated in Section 11.1.3, the *border* S' of a set S is the set of points of S that are adjacent to $\bar{S}$. We will assume, for simplicity, that "adjacent" here means "4-adjacent." The set of nonborder points of S is called the *interior* of S.

Let C be a component of S, and let D be a component of $\bar{S}$ that is adjacent to C. Note that if C and D are 8-adjacent, they are also 4-adjacent; indeed, consider the pattern

$$P \quad X$$
$$Y \quad Q$$

where P is in C and Q in D. If X is in S, it is in C, and if it is in $\bar{S}$, it is in D, and similarly for Y; thus in any case C and D are 4-adjacent.

The set C_D of points of C that are adjacent to D is called the *D-border* of C. Similarly, the set D_C of points of D that are adjacent to C is called the *C-border* of D. It is these component borders that can be regarded as closed curves, and for which we can define border following algorithms, as we shall see in Section 11.2.2.

It can be shown that if C is adjacent to several components of $\bar{S}$, then exactly one of those components, say D_0, surrounds C, and the others are surrounded by C. The D_0-border of C is called its *outer border*, and the other D-borders of C, if any, are called *hole borders*. This surroundness property is what allows us to treat the borders as closed curves. It is true only if we use opposite types of connectedness for S and $\bar{S}$. For example, suppose that we use 4-connectedness for both S and $\bar{S}$, and that the points of S are

```
      1 1
      1 1
1 1         1 1
1 1         1 1
      1 1
      1 1
```

Then each block C of 1's is a 4-component of S, and the block D of 0's that they surround is a 4-component of $\bar{S}$, but C_D and D_C are not closed curves, and D does not surround the C's nor does any one of them surround it. Similarly, if we use 8-connectedness for both S and $\bar{S}$, then S and $\bar{S}$ each consists of a single component but $S_{\bar{S}}$ and $\bar{S}_S$ are not closed curves. On the other hand, if we use 4-connectedness for S and 8- for $\bar{S}$, then each block C of S is a component, all of $\bar{S}$ is a single component, and each $C_{\bar{S}}$ and $\bar{S}_C$ is a closed curve. Similarly, if we use 8- for S and 4- for $\bar{S}$, then S consists of a single component, $\bar{S}$ has the surrounded block D of 0's as a component, and S_D and D_S are closed curves.

Exercise 6. Show that when we use 8-connectedness for S, the outer border and a hole border of S can be the same. Can two borders of a 4-component be the same? Can one be a subset of another? On how many different borders of C can a given point of C lie? ∎

Alternatively, we can define the *(C, D)-border* (of C or D) as the set of pairs (P, Q) such that P is in C, Q is in D, and P, Q are 4-adjacent. This is the same as the set of "cracks" between the points of C and the adjacent points of D. It is evident that any two such borders must be disjoint.

The surroundness and closed curve properties hold only when C and D are components of a set S and its complement, respectively. If we take C and D to be arbitrary connected subsets of Σ, we can say nothing about the D-border of C; C and D can touch in many distinct places, neither of them need surround the other, and the border need not be anything like a closed curve.

d. Distance

The *Euclidean distance* between two points $P = (x, y)$ and $Q = (u, v)$ is

$$d_e(P, Q) = \sqrt{(x-u)^2 + (y-v)^2}$$

It is sometimes convenient to work with simpler "distance" measures on digital pictures. In particular, the *city block distance* between P and Q is defined as

$$d_4(P, Q) = |x-u| + |y-v|$$

and the *chessboard distance* between them is

$$d_8(P, Q) = \max(|x-u|, |y-v|)$$

It can be verified that all three of these measures, d_e, d_4, and d_8, are *metrics*, i.e., for all P, Q, R we have

$$d(P, Q) \geqslant 0, \quad \text{and } = 0 \quad \text{iff } P = Q$$

$$d(P, Q) = d(Q, P)$$

$$d(P, R) \leqslant d(P, Q) + d(Q, R)$$

In other words, each of these d's is *positive definite*, *symmetric*, and satisfies the *triangle inequality*.

It is not hard to see that the points at city block distance $\leqslant t$ from P form a diamond (i.e., a diagonally oriented square) centered at P. For example, if we represent points by their distances from P (so that P is represented by 0), the points at distances $\leqslant 2$ are

```
      2
    2 1 2
  2 1 0 1 2
    2 1 2
      2
```

In particular, the points at distance 1 are just the 4-neighbors of P. It can be shown that $d_4(P, Q)$ is equal to the length of a shortest 4-path from P to Q. (Note that there may be many such paths!)

Analogously, the points at chessboard distance $\leqslant t$ from p form an upright square centered at P; e.g., the points at distances $\leqslant 2$ are

$$\begin{array}{ccccc} 2 & 2 & 2 & 2 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 1 & 0 & 1 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{array}$$

so that the points at distance 1 are the 8-neighbors of P. In general, $d_8(p, q)$ is the length of a shortest 8-path from P to Q, and there are many such paths.

Exercise 7. d_4 and d_8 are integer-valued, but d_e is not. Is $\lfloor d_e \rfloor$ (the greatest integer $\leqslant d_e$) a metric? What about the closest integer to d_e? ∎

The distance between a point P and a set S is defined to be the shortest distance between P and any point of S. The *diameter* of a set is the greatest distance between any two of its points.

11.2 CONVERSION BETWEEN REPRESENTATIONS

Some of the conversions between one representation and another are quite straightforward. For example, it is obvious how to convert from arrays to runs: scan the array row by row; increment a counter as long as the value remains constant; when the value changes, append the (old) value and count to the run list and reset the counter. Conversely, to convert from runs to arrays, we create the array row by row, putting in the current value and decrementing the current run length until it reaches zero, at which point we switch to the next value and length on the run list.

In this section we give conversion algorithms involving blocks (MAT, quadtree) and border representations. We also describe algorithms for thinning line drawings into sets of arcs and curves. As before, we will usually deal only with the case where the given picture Σ is partitioned into a set S and its complement.

Conventionally, the algorithms described here would be implemented on an ordinary (single-processor) digital computer, which would process the given picture row by row. On the other hand, some of the algorithms are very well suited for implementation on a cellular array computer, in which a processor is assigned to each pixel, so that local operations can be performed at all the pixels simultaneously. Cellular computers of reasonable size (100 by

100 processors or more) currently exist or are being built, and it is expected that they will play a major role in picture processing in future years.

11.2.1 Blocks

a. MATs

In Section 11.1.2a we defined the MAT in terms of the set of maximal constant-value blocks (=upright squares of odd side length) in Σ. When Σ is partitioned into S and $\bar{S}$, we can speak of the MAT of S (defined by the maximal blocks that are contained in S) and the MAT of $\bar{S}$.

Let us first observe that the MA of S (= the set of centers of the MAT blocks) consists of those points of S whose chessboard ($= d_8$) distances from $\bar{S}$ are local maxima. Recall that the points within a given chessboard distance of P form an upright square centered at P. If $d_8(P, \bar{S}) = d$, then the square $S_P^{(d-1)}$ of radius $d-1$ centered at P cannot intersect $\bar{S}$ (since there would then be a point of $\bar{S}$ at distance $\leqslant d-1$ from P), but the square of radius d does intersect $\bar{S}$. Suppose $d_8(P, \bar{S})$ is not a local maximum; then P has a neighbor Q such that $d_8(Q, \bar{S}) \geqslant d+1$. Thus a square S_Q of radius $\geqslant d$ centered at Q does not intersect $\bar{S}$; but S_Q contains $S_P^{(d-1)}$, so that $S_P^{(d-1)}$ is not a maximal block. Conversely, if $S_P^{(d-1)}$ is not maximal, it is not hard to see that it must be contained in a square S_Q of constant value centered at some neighbor of P, and S_Q has radius at least d; thus $d_8(Q, \bar{S}) \geqslant d$, so that $d_8(P, \bar{S})$ is not maximal.

Analogously, if we define the MAT using diagonally oriented rather than upright squares, we can show that the MA of S consists of those points of S whose city block (d_4) distances from $\bar{S}$ are local maxima. In this case "local maximum" means that no 4-neighbor of the point has greater distance from $\bar{S}$, whereas in the previous case the MA points were 8-neighbor distance maxima. Figure 8 shows the sets of city block and chessboard distances to $\bar{S}$ when S is a rectangle or a diamond; the local maxima in each case are underlined.

These remarks imply that we can construct the MAT of S by computing the distances to $\bar{S}$ from all points of S and discarding nonmaxima. In the next two subsections we will present some algorithms for computing these distances from a given binary array or row-by-row representation.

Since the MA is a set of local distance maxima, it is generally quite disconnected; indeed, two maxima cannot be adjacent unless their values are equal. We can attempt to make the MA connected by keeping some of the nonmaxima. For example, in the d_8 case, we might keep P if at most one of its 8-neighbors has a larger distance to $\bar{S}$, and if it has a 4-neighbor that is a distance maximum. [The first condition implies that the layer of points at

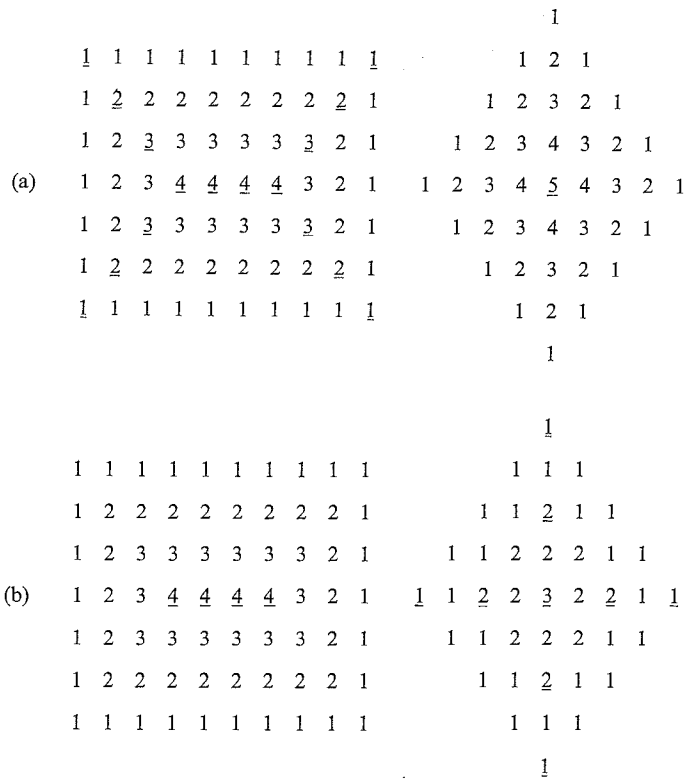


Fig. 8 Distances to $\bar{S}$ for the points of a rectangle and a diamond; the MA points are underlined. (a) d_4; (b) d_8. Note that the d_4 and d_8 values for the rectangle are the same. The MAT radii are 1 less than the distances.

distance $d_8(P)$ from $\bar{S}$ turns sharply at P.] Of course, when we discard fewer points, we have a less economical representation of S. However, if we can find a set of arcs and curves containing the MA, we can represent these curves by chain codes, rather than having to list all the MA points individually, so that the economy of the representation may be improved.

Distance computation

We first give a cellular array computer algorithm for distance computation. This algorithm works for any metric d that is integer-valued and has the following property:

For all P, Q such that $d(P, Q) \geq 2$, there exists a point R, different from P and Q, such that $d(P, Q) = d(P, R) + d(R, Q)$.

Such metrics will be called *regular*. It can be shown that d is regular if and only if, for all distinct P, Q, there exists a point R such that $d(P, R) = 1$ and $d(P, Q) = d(R, Q) + 1$.[§]

Given χ_S, which is 1 at the points of S and 0 elsewhere, we define $\chi_S^{(m)}$ inductively for $m = 1, 2, \ldots$ as follows:

$$\chi_S^{(m)}(P) = \chi_S^{(0)}(P) + \min_{d(Q,P)\leq 1} \chi_S^{(m-1)}(Q) \tag{1}$$

where $\chi_S^{(0)} = \chi_S$. Thus $\chi_S^{(m)}$ can be computed by performing a local operation on the pair of arrays $\chi_S^{(0)}$ and $\chi_S^{(m-1)}$ at every point.

To see how (1) works, note first that 0's (i.e., points of $\bar{S}$) remain 0's, since if $\chi_S^{(0)}(P) = 0$ we have

$$\min_{d(Q,P)\leq 1} \chi_S^{(m-1)}(Q) = \chi_S^{(m-1)}(P) = 0 \qquad \text{for} \quad m = 1, 2, \ldots$$

Similarly, 1's at distance 1 from $\bar{S}$ (i.e., border 1's) remain 1's, since for such points too, the min is 0. On the other hand, on the first iteration of (1), all 1's at distance > 1 from $\bar{S}$ (i.e., interior 1's) become 2's, since for such points the min is 1. On the second iteration, all interior 2's (= all 2's at distance > 2 from $\bar{S}$) become 3's, since the min for such points is 2. On the third iteration, all interior 3's become 4's, and so on. Thus if $d(P, \bar{S}) = k > 0$, the values of $\chi_S^{(m)}(P)$ for $m = 1, 2, \ldots, k$ are $1, 2, \ldots, k$, respectively, and the value remains k at all subsequent iterations. It follows that if m is the greatest distance between $\bar{S}$ and any point in the picture, we have $\chi_S^{(m)}(P) = d(P, \bar{S})$ for all P. It certainly suffices to iterate (1) a number of times equal to the picture's diameter; for an $n \times n$ picture, this is $2(n-1)$ for d_4 and $n-1$ for d_8.

If we want to find a shortest path from each P to $\bar{S}$, then at the iteration when $\chi_S^{(m)}(P)$ stops increasing, we create a pointer from P to one of its neighbors which has the minimum value in (1). This neighbor must be on a shortest path from P to $\bar{S}$, so that if we follow the pointers starting at P, we must move along a shortest path to $\bar{S}$ [61]. Note, incidentally, that P is an MA point iff it does not lie on a shortest path from any other point to S to $\bar{S}$. Indeed, if P were on a shortest path from Q to $\bar{S}$, its predecessor on this path would be a neighbor of P and would have higher distance to $\bar{S}$ than P.

Exercise 8. For a multivalued picture, define the *gray-weighted distance* [61] between P and Q as the smallest possible sum of values along any path from P to Q. (Here a "path" means a sequence of points such that the distance between consecutive points is 1.) Prove that if we apply (1) to a multivalued picture that contains 0's, the value at any point P eventually becomes equal

§ Regularity is a necessary and sufficient condition for a metric to be the distance on a graph; see F. Harary, *Graph Theory*, Addison-Wesley, Reading, Massachusetts, 1969, p. 24, Exercise 2.8.

to the gray-weighted distance between P and the set of 0's. Note that if there are only two values, 0 and 1, the gray-weighted distance to the set of 0's is the same as the ordinary distance. We can also define a gray-weighted MA [33] as the set of points whose gray-weighted distance to the 0's is a local maximum, or equivalently, which do not lie on a minimum-sum path from any other point to the 0's. Is a multivalued picture reconstructible from its gray-weighted MAT? ▮

Metrics d_4 and d_8 are quite non-Euclidean; the set of points at distance $\leqslant k$ from a given point is a square, rather than a circle, for these metrics. We can compute a distance in which this set is an octagon by using d_4 and d_8 at alternate iterations in (1). The points at "octagonal distance" $\leqslant 2$ are

$$\begin{array}{ccccc} & 2 & 2 & 2 & \\ 2 & 2 & 1 & 2 & 2 \\ 2 & 1 & 0 & 1 & 2 \\ 2 & 2 & 1 & 2 & 2 \\ & 2 & 2 & 2 & \end{array}$$

To get the MAT for this distance, we can use 4-neighbor maxima when the distance is odd, and 8-neighbor maxima when it is even, to insure that the argument given in Subsection (a) remains valid.

Algorithm (1) is quite efficient on a cellular array computer, since each iteration can be performed at all points in parallel, and the number of iterations is at most the diameter of the picture. On a conventional computer, however, each iteration requires a number of steps proportional to the area of the picture. We now present an algorithm that computes all the distances (d_4 or d_8) to $\bar{S}$ in only two scans of the picture, so that a large number of iterations is not required. We assume that the border of the picture consists entirely of 0's. Let $N_1(P)$ be the set of (4- or 8-) neighbors that precede P in a row-by-row (left to right, top to bottom) scan of the picture, and let $N_2(P)$ be the remaining (4- or 8-) neighbors of P; i.e., $N_1(x, y)$ consists of the 4-neighbors $(x-1, y)$ and $(x, y+1)$, as well as the 8-neighbors $(x-1, y+1)$ and $(x+1, y+1)$. Then the algorithm is as follows:

$$\chi_S'(P) = \begin{cases} 0 & \text{if } P \in \bar{S} \\ \min\limits_{Q \in N_1} \chi_S'(Q) + 1 & \text{if } P \notin \bar{S} \end{cases}$$

$$\chi_S''(P) = \min_{Q \in N_2} [\chi_S'(P), \chi_S''(Q) + 1] \qquad (2)$$

Thus we can compute χ_S' in a single left to right, top to bottom scan of the picture, since for each P, χ_S' has already been computed for the Q's in N_1.

Similarly, we can compute χ_S'' in a single *reverse* scan (right to left, bottom to top). Then for all P we have $\chi_S''(P) = d_4(P, \bar{S})$ or $d_8(P, \bar{S})$, depending on whether we use 4- or 8-neighbors in the algorithm. As a very simple example, let $\chi_{\bar{S}}$ be

$$\begin{array}{cccc} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{array}$$

surrounded by 0's. Then χ_S' in the 4-neighbor case is

$$\begin{array}{cccc} 0 & 1 & 1 & 1 \\ 1 & 2 & 2 & 0 \\ 1 & 2 & 3 & 1 \end{array}$$

and χ_S'' is

$$\begin{array}{cccc} 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{array}$$

c. Shrinking and expanding

Any S can be "shrunk" by repeatedly deleting its border, and "expanded" by repeatedly adding to it the border of its complement $\bar{S}$. In general, by the border of S we mean the set of points that are at distance 1 from $\bar{S}$. Let S' be the border of S in this general sense; then the successive stages in shrinking S are $S^{(0)} \equiv S$, $S^{(-1)} \equiv S - S'$, $S^{(-2)} \equiv S^{(-1)} - S^{(-1)\prime}$, and so on. Similarly, the successive stages in expanding S are $S^{(1)} \equiv S \cup (\bar{S})'$; $S^{(2)} \equiv S^{(1)} \cup (\overline{S^{(1)}})'$; and so on. It is easily seen that for any k we have $S^{(k)} = \overline{(\bar{S})^{(-k)}}$, or equivalently $S^{(-k)} = \overline{(\bar{S})^{(k)}}$. Given χ_S, we get $\chi_{S^{(1)}}$ by changing 0's to 1's if they have any 1's as neighbors (i.e., at distance 1), and we get $\chi_{S^{(-1)}}$ by changing 1's to 0's if they have any 0's as neighbors.

It should be pointed out that shrinking and expanding do not commute with one another; $(S^{(m)})^{(-n)}$ is not necessarily the same as $(S^{(-n)})^{(m)}$, and neither of them is the same as $S^{(m-n)}$. For example, if S consists of a single point P, then $S^{(1)}$ consists of P and its neighbors, and $(S^{(1)})^{(-1)} = S = \{P\}$; but $S^{(-1)}$ is empty, and so is $(S^{(-1)})^{(1)}$. However, it can be shown that $(S^{(-n)})^{(m)} \subseteq S^{(m-n)} \subseteq (S^{(m)})^{(-n)}$. In particular, we have $(S^{(-k)})^{(k)} \subseteq S \subseteq (S^{(k)})^{(-k)}$ for all k.

In Sections 11.3.2d and 11.3.4b we will show how combinations of shrinking and expanding can be used to define and detect elongated parts, isolated parts, and clusters of parts of a set S and remove "noise" from S. In this

section we are primarily concerned with how shrinking and expanding are related to distance and to the MAT. Since S' is the set of points of S that are at distance 1 from $\bar{S}$, readily $S^{(t)}$ is the set of points at distance $\leqslant t$ from S, and $S^{(-t)}$ is the set of points at distance $> t$ from $\bar{S}$. Thus $\chi_S^{(m)} = \chi_S + \chi_{S^{(-1)}} + \cdots + \chi_{S^{(-m)}}$, which gives us an algorithm for computing the distances to $\bar{S}$ by repeated shrinking and adding.

To obtain the MA of S by shrinking and expanding, we proceed as follows: Let $S_k = S^{(-k)} - (S^{(-k-1)})^{(1)}$ (note that the first of these sets contains the second). Any point P of S_k is at distance exactly $k+1$ from $\bar{S}$, since if it were farther away it would be in $S^{(-k-1)}$, hence in $(S^{(-k-1)})^{(1)}$. Furthermore, P has no neighbors whose distances from $\bar{S}$ are greater than $k+1$, since any such neighbor would be in $S^{(-k-1)}$, so that P would be in $(S^{(-k-1)})^{(1)}$. Thus P's distance from $\bar{S}$ is a local maximum, making it an MA point; in fact, S_k is just the set of MA points that are at distance $k+1$ from $\bar{S}$.

Shrinking and expanding can make use of arbitrary neighborhoods [70]. Let $N(P)$ denote the "neighborhood" of P; this could consist of P together with any set of points having given positions relative to P. Then we can define the N-expansion of S as $\bigcup_{P \in S} N(P)$, and the N-contraction of S as $\{P \in S \mid N(P) \cap \bar{S} = \varnothing\}$. Readily, if neighborhoods are symmetric [i.e., $Q \in N(P)$ iff $P \in N(Q)$], these operations are complementary, i.e. the N-contraction of S is the complement of the N-expansion of $\bar{S}$, and vice versa. These operations can be iterated or combined in various ways, as before. Note that if $N(P)$ consists of the points at distance $\leqslant 1$ from P, these definitions reduce to the ones given earlier.

The analogs of shrinking and expanding for multivalued pictures [41] are local min and local max operations defined over a given neighborhood. Note that if there are only two values, 0 and 1, local min shrinks the 1's, while local max expands them. Just as shrinking and expanding can be used to delete noise from a segmented (two-valued) picture, so local min and max operations can be used to smooth multivalued pictures; compare Section 6.4.4 and Section 11.3.2d.

A single step of shrinking or expanding can be done in a single parallel operation on a cellular array computer, but requires a complete scan of the picture on a conventional computer. On each row, points are marked for changing (from 1 to 0 or from 0 to 1), but they are not actually changed until the next row has been marked. If we want to do a sequence of k steps without having to scan the entire picture k times, we can operate on the rows in a sequence such as 1; 2, 1; 3, 2, 1; 4, 3, 2, 1, ... (Once we have operated on the second row, we can do a second operation on the first row; once we have operated on the third row, we can do a second operation on the second row, and then a third operation on the first row; and so on.) Once we have reached the kth row, and done the sequence $k, (k-1), \ldots, 2, 1$, we are finished with

the first row, since it has been processed k times. The next sequence is $(k+1), k, \ldots, 3, 2$, after which we are finished with the second row; the first row can now be dropped. The next sequence is $(k+2), (k+1), \ldots, 4, 3$, after which the second row can be dropped; and so on. Thus only $k+2$ rows (those currently being processed, plus one before and one after them) need to be available at a time.

d. Reconstruction from MATs

Given the MAT of S, we can construct χ_S by creating solid squares of 1's having the specified centers and radii. Alternatively, we can represent the MAT by a picture Σ_S whose values at the distance maxima are equal to their distances, and consisting of 0's elsewhere. We can then reconstruct the entire set of distances to $\bar{S}$ by applying an iterative algorithm to Σ_S [compare (1)]:

$$\begin{aligned} \Sigma_S^{(m)}(P) &= \max\left[0, \max_{d(P,Q)=1} \Sigma_S^{(m-1)}(Q) - 1\right] && \text{provided } \Sigma_S^{(m-1)}(P) = 0 \\ &= \Sigma_S^{(m-1)}(P) && \text{otherwise} \end{aligned} \tag{3}$$

The number of iterations required is one less than the value of the largest distance maximum. As an example, if Σ_S is

```
1       1
  2   2
    3 3 3
  2   2
1       1
```

(see Fig. 8a), where 0's are represented by blanks, then two iterations of (3) using d_4 give

```
1 1         1 1           1 1 1 1 1 1 1
1 2 2 2 2 2 1             1 2 2 2 2 2 1
  2 3 3 3 2       and     1 2 3 3 3 2 1
1 2 2 2 2 2 1             1 2 2 2 2 2 1
1 1         1 1           1 1 1 1 1 1 1
```

as in Fig. 8a. Alternatively, we can reconstruct the distances using the following two-pass algorithm [compare (2)]:

$$\begin{aligned} \Sigma_S'(P) &= \max_{Q \in N_1}(\Sigma_S(P), \Sigma_S'(Q) - 1) \\ \Sigma_S''(P) &= \max_{Q \in N_2}(\Sigma_S'(P), \Sigma_S''(Q) - 1) \end{aligned} \tag{4}$$

Here we first compute Σ' in a left to right, top to bottom scan, and then compute Σ'' in a right to left, bottom to top scan. In the example given above, using d_4, Σ_S' and Σ_S'' are

$$\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 3 & 3 & 3 & 2 & 1 \\ 0 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} \quad \text{and} \quad \begin{array}{ccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 2 & 3 & 3 & 3 & 2 & 1 \\ 1 & 2 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

respectively. It can be shown that the MAT is the smallest set of distance values from which all of the distance values can be reconstructed using (3) or (4). S itself is then just the set of points that have nonzero values after (3) or (4) has been applied.

e. Quadtrees

In this subsection we briefly describe methods of converting between the quadtree and binary picture representation of a given S. The details of the algorithms can be found in three papers by Samet [63,65,67].

In Section 11.1.2b we described a method of constructing the quadtree corresponding to a given binary picture by recursively subdividing the picture into blocks that are quadrants, subquadrants, ... If a block consists entirely of 1's or 0's, it corresponds to a "black" or "white" leaf node in the tree; otherwise, it corresponds to a "gray" nonleaf node, which has four sons corresponding to its four quadrants. If we do this in a "top–down" fashion, i.e., first examine the entire picture, then its quadrants, then their quadrants, etc., as needed, it may require excessive computational effort, since parts of the picture that contain finely divided mixtures of 0's and 1's will be examined repeatedly.

As an alternative, we can build the quadtree "bottom up" by scanning the picture in a suitable order, e.g., in the sequence

$$\begin{array}{cccccccc} 1 & 2 & 5 & 6 & 17 & 18 & 21 & 22 \\ 3 & 4 & 7 & 8 & 19 & 20 & 23 & 24 \\ 9 & 10 & 13 & 14 & 25 & 26 & 29 & 30 \\ 11 & 12 & 15 & 16 & 27 & 28 & 31 & 32 \\ 33 & \ldots & & & & & & \end{array}$$

where the numbers indicate the order in which the points are examined. As we discover maximal blocks of 0's or of 1's, we add leaf nodes to the tree, together with their needed ancestor gray nodes. This can be done in such a way that leaf nodes are never created until they are known to be maximal, so that it is never necessary to merge four leaves of the same color and change their common parent node from gray to black or white. For the details of this algorithm see [65].

Bottom-up quadtree construction becomes somewhat more complicated if we want to scan the picture row by row. Here we add leaf nodes to the tree as we discover maximal 1×1 or 2×2 blocks of 0's or 1's; if four leaves with a common father all have the same color, they are merged. The details can be found in [67].

Given a quadtree, we can construct the corresponding binary picture by traversing the tree and, for each leaf, creating a block of 0's or 1's of the appropriate size in the appropriate position. A more complicated process can be used if we want to create the picture row by row. Here we must visit each quadtree node once for each row that intersects it (i.e., a node corresponding to a $2^k \times 2^k$ block is visited 2^k times), and, for each leaf, output a run of 0's or 1's of the appropriate length (2^k) in the appropriate position. For the details, see [63].

11.2.2 Borders

We recall that the border S' of a set S is the set of points of S that are 4-adjacent to $\bar{S}$. To find all the border points of S, we can scan χ_S and check the four neighbors of each 1 to see if any of them is 0 (or vice versa). On a cellular array computer, we can find the border by performing a Boolean operation at each point P of χ_S. Let P_n, P_s, P_e, P_w be the four neighbors of P; then we compute

$$P \wedge (\bar{P}_n \vee \bar{P}_s \vee \bar{P}_e \vee \bar{P}_w)$$

where the overbars denote logical negation ($\bar{1} = 0$, $\bar{0} = 1$). Evidently, this yields 1's at border points of S and 0's elsewhere.

Exercise 9. A point of S is called (4- or 8-) *isolated* if it has no neighbors in S. Define a Boolean operation on χ_S that detects 4- or 8- isolated points. ∎

In this section we describe algorithms for finding, following, and coding the individual borders between components of S and $\bar{S}$, and for reconstructing S from these border codes. These algorithms provide the means for converting between the bit plane and border code representations. Conversion between quadtrees and border codes will also be briefly discussed.

We will continue to assume here that S does not touch the edges of the picture. This will make it unnecessary for our algorithms to handle special cases involving neighbors that are outside the picture.

a. Crack following

Let C, D be components of S, $\bar{S}$, respectively, and let P, Q be 4-adjacent points of C and D, so that (P, Q) defines one of the cracks on the (C, D)-border. Let U, V be the pair of points that we are facing when we stand on the crack between P and Q with P on our left, with U, V 4-adjacent to P, Q respectively, i.e.,

$$\begin{matrix} P & U \\ Q & V \end{matrix}' \qquad \begin{matrix} U & V \\ P & Q \end{matrix}' \qquad \begin{matrix} V & Q \\ U & P \end{matrix}' \qquad \text{or} \qquad \begin{matrix} Q & P \\ V & U \end{matrix}$$

(Defining U, V in terms of P on the left implies that outer borders will be followed counterclockwise and hole borders clockwise; the reverse would be true if we had P on the right.) Then the following rules define the next crack (P', Q') along the (C, D)-border:

(1) If we use 8-connectedness for C

U	V	P'	Q'	Turn
–	1	V	Q	right
1	0	U	V	none
0	0	P	U	left

(2) If we use 4-connectedness for C

U	V	P'	Q'	Turn
0	–	P	U	left
1	0	U	V	none
1	1	V	Q	right

The algorithm stops when we come to the initial pair again.

To generate the crack code corresponding to this traversal of the border, note that the four initial configurations of P, Q, U, and V given above correspond to crack codes of 0, 1, 2, and 3, respectively. To determine the code for (P', Q') from that for (P, Q), we simply add 1 (modulo 4) if we turned left, subtract 1 (modulo 4) if we turned right, and use the same code if we made no turn. (Alternatively, the turning rules give us the difference crack code directly.)

b. Border following

The algorithms for following the D-border of C are somewhat more complicated. Let P, Q be as before, and let the eight neighbors of P, in counterclockwise order starting from Q, be $Q = R_1, R_2, \ldots, R_8$. (Using counterclockwise order implies that outer borders will be followed counterclockwise

and hole borders clockwise.) We assume that C does not consist solely of the isolated point $\{P\}$. The following rules define a new pair P', Q':

(1) If we use 8-connectedness for C:

Let R_i be the first of the R's that is 1 (such an R exists since P is not isolated). Then $P' = R_i$, $Q' = R_{i-1}$.

(2) If we use 4-connectedness for C:

Let R_i be the first 4-*neighbor* that is 1 (i.e., the first of R_3, R_5, R_7). If $R_{i-1} = 0$, take $P' = R_i$, $Q' = R_{i-1}$; if $R_{i-1} = 1$, take $P' = R_{i-1}$, $Q' = R_{i-2}$.

The algorithm stops when we come to the initial P again, provided that we find the initial Q (as one of the R's) before finding the next P'. This last condition is necessary because a border can pass through a point twice, as we will see in the example immediately below.

As an example, let C, P, and Q be as in Fig. 9a. The R's and the new P, Q at the first two steps of the algorithm are shown in Figs. 9b and 9c. If we use 8-connectedness for C, the next three steps are shown in Figs. 9d–9f; if we

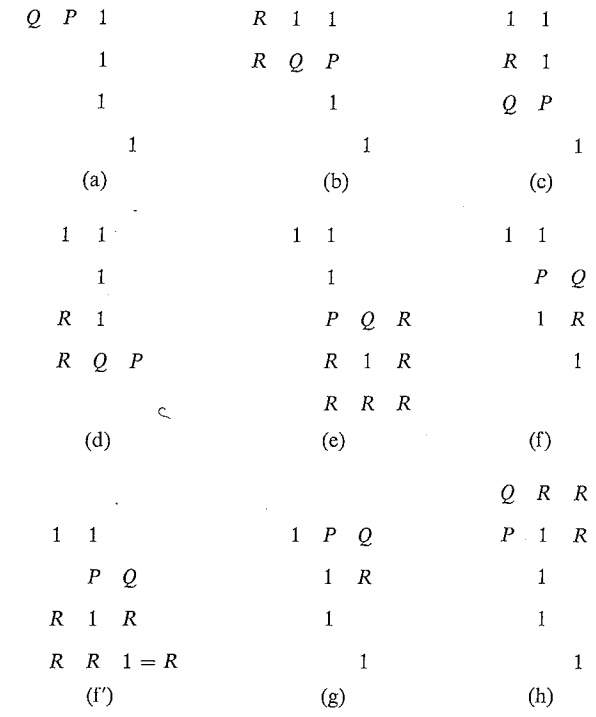


Fig. 9 Border following, showing the initial P and Q in (a), and the R's and new P and Q at successive steps (b)–(h). For detailed explanation see text.

use 4-connectedness, the second step (Fig. 9c) is followed directly by Fig. 9f'. Note that P is the same in Figs. 9e and 9c, but the algorithm would not stop even if this had been the initial P, since the Q of Fig. 9c is not one of the R's encountered at the next step before finding a 1. Analogous remarks apply to Figs. 9b and 9f'. In both the 4- and 8-cases, the last two steps are as shown in Figs. 9g–9h. After step 9h the algorithm stops, since P is the same as the initial P, and the initial Q is one of the R's encountered at the next step before finding a 1. An equivalent stopping criterion would be as follows: The algorithm stops when it finds two successive P's that it had previously found in the same consecutive order.

Any two successive P's found by the algorithm are always 8-neighbors, and in the case where C is 4-connected, they are also 4-connected through 1's; thus all the P's belong to C. Any two successive Q's are always 8-connected through 0's (when C is 4-connected, they may not be 4-connected, since one of the intermediate R's may be a 1 diagonally adjacent to P), and when C is 8-connected, they are also 4-connected through 0's; thus all the Q's belong to D, so that the P's are all on the D-border of C. It is much harder to prove that they constitute the entire D-border; see [55] for a detailed treatment of this. Since successive P's are 8-neighbors, the sequence of P's defines the chain code of the border.

Exercise 10. Define borders using 8-adjacency rather than 4-adjacency, and devise a border following algorithm (for the case where C is 4-connected) in which successive P's are always 4-neighbors. ▮

Exercise 11. Let P_1, P_2 be 4-adjacent points of C, D, respectively, and define P_i for $i > 2$ as follows [13, 36]: If $P_{i-1} = 0$, turn left (relative to the direction from P_{i-2} to P_{i-1}); if $P_{i-1} = 1$, turn right. Work out an example of the operation of this algorithm. How does it handle 8-adjacency? (Hint: Try it on 8-adjacencies in two perpendicular diagonal directions.) ▮

An alternative method of obtaining the successive points of the D-border of C is as follows: Use the crack following algorithm to generate the successive cracks (P_i, Q_i) of the (C, D)-border. If a given point P occurs two or more times in succession as the first term of a crack, eliminate the repetitions. [This situation arises when a border point of C has several consecutive 4-neighbors in D, corresponding to several successive cracks on the (C, D)-border.] The sequence of first terms that remain after these eliminations is just the sequence of points of C_D. In fact, we can generate the chain code of C_D directly from the crack following algorithm. For example, if the current point pairs are

$$\begin{matrix} P & U \\ Q & V \end{matrix}$$

and we next turn right, we add 7 to the chain; if we make no turn, we add 0; while if we turn left, we add nothing to the chain, since the new first term is still P. Since the algorithm for border following is more complicated than that for crack following, it may be more economical to use crack following to generate the sequence of border points and the chain code, as just described, rather than using border following.

c. *Border finding*

Suppose that we want to follow all of the borders of S, one at a time, in order to obtain all their chain codes. (The discussion for crack codes is analogous.) To accomplish this, we might scan the picture systematically, say row by row; when we hit a border (e.g., when we find a 1 immediately following a 0), follow it around; when it has been completely followed, resume the scan. This process will certainly find all the borders, since on any border there must be a 1 with a 0 immediately to its left (e.g., a leftmost point of an outer border or a rightmost point of a hole border). However, it will find the same border repeatedly (on every row that contains points of the component or of the hole), so that each border will be followed many times, which is certainly undesirable. We can eliminate this problem by marking each border while following it, and initiating border following only when we hit an unmarked border; this insures that no border will be followed twice. However, some borders may now be missed; in fact, a hole border may be a subset of an outer border (the simplest examples are

$$\begin{matrix} & 1 & & & & 1 & 1 & 1 \\ 1 & & 1 & \text{and} & & 1 & & 1 \\ & 1 & & & & 1 & 1 & 1 \end{matrix}$$

—see Exercise 6), or at least all its points with 0's on their left may be on an outer border, so that these points get marked when the outer border is followed, and the hole border does not itself get followed. To avoid this problem, we should mark only those border points (e.g., on the D-border of C) that have points of D as left-hand neighbors; this insures that we will never hit the same border by a transition from 0 to an unmarked 1, but it does not interfere with detecting other borders that share points with the given one.

If we want to distinguish between outer borders and hole borders, we can label the connected components of $\bar{S}$ (see Section 11.3.1) before searching for borders; this gives the background a distinctive label, so that outer borders are identifiable by their adjacency to points having that label. Alternatively, we use the following modified process of scanning, border following, and

marking: We use two marks, l and r, for points of the D-border that have D on their left and right, respectively; and we initiate border following at any transition from a 0 to a 1 not marked l, or from a 1 not marked r to a 0. Now an outer border is always first encountered as a transition from 0 to 1 (at the leftmost of the uppermost points of C), and similarly a hole border is always first encountered as a transition from 1 to 0; hence this scheme allows us to identify them as soon as they are encountered.

d. Border tracking

Crack and border following have the disadvantage that they require access to the points of the picture in an arbitrary order, since a border may be of any shape and size. In the following paragraphs we describe a method of constructing crack or chain codes of all the borders of S in a single row-by-row scan of the picture. This allows us to convert a row-wise representation of S (e.g., runs) directly into a border representation. Conversion from borders to runs will also be discussed.

We will now show how to convert runs into crack codes; conversion to chain codes is quite similar and is left as an exercise for the reader.

(1) For each run ρ on the first row, or each run ρ not adjacent to a run on the preceding row, we initialize a code string of the form $1\,2\cdots2\,3$, where the number of 2's is the length of ρ; this represents the top and sides of ρ, which we know must be on a border. The end of this string is associated with the left end of ρ, and its beginning with the right end.

(2) In general, a run ρ on a given row has one end of a code string associated with each of its ends (we will see shortly how the strings at the two ends can be different). Let these strings be α_ρ and β_ρ, where the end of α_ρ is associated with the left end of ρ, and the beginning of β_ρ is associated with its right end. Note that α_ρ always ends with 3 and β_ρ always begins with 1. If no run on the following row is adjacent to ρ, we link the end of α_ρ to the beginning of β_ρ by a string of the form $0\cdots0$, where the number of 0's is the length of ρ. This string represents the bottom of ρ, which must be on a border.

(3) If just one run ρ' on the following row is adjacent to ρ, we add to the end of α_ρ a string of the form $2\cdots2\,3$, if ρ' extends to the left of ρ; a single 3, if ρ' and ρ are lined up at their left ends; and a string of the form $0\cdots0\,3$, if ρ extends to the left of ρ'. The end of the extended α_ρ is now associated with the left end of ρ'. Similarly, we add to the beginning of β_ρ a string of the form $1\,2\cdots2$, if ρ' extends to the right of ρ; a single 1, if ρ' and ρ are lined up at their right ends; and a string of the form $1\,0\cdots0$, if ρ extends to the right of ρ'. The beginning of the extended β_ρ is associated with the right end of ρ'.

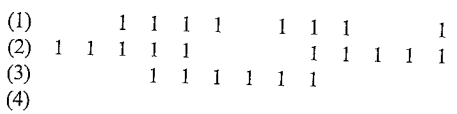
(4) If several runs, say $\rho_1', \ldots, \rho_k'$, on the following row are adjacent to ρ, we add strings to the end of α_ρ and beginning of β_ρ just as in (3), depending

on the positions of ρ_1' relative to the left end of ρ and of ρ_k' relative to its right end. The end and beginning of these strings are associated with the left end of ρ_1' and the right end of ρ_k', respectively. In addition, for each consecutive pair of runs ρ_{i-1}', ρ_i' $(1 < i \leqslant k)$, we create a new string of the form $1\,0\cdots0\,3$, representing the right side of ρ_{i-1}', the bottom portion of ρ between ρ_{i-1}' and ρ_i', and the left side of ρ_i'. The beginning of this new string is associated with the right end of ρ_{i-1}', and its end with the left end of ρ_i'.

(5) Suppose finally that run ρ' is adjacent to several runs, say $\rho_1, \ldots, \rho_k$, on the preceding row. In this case, we add strings to the end of α_{ρ_1} and the beginning of β_{ρ_k} just as in (3); the forms of these strings depend on the positions of ρ' relative to the left end of ρ_1 and the right end of ρ_k. The end and beginning of these extended strings are associated with the left and right ends of ρ', respectively. In addition, for each consecutive pair of runs ρ_{i-1}, ρ_i $(1 < i \leqslant k)$, we link the end of α_i to the beginning of β_{i-1} by a string of the form $2\cdots2$, representing the top portion of ρ' between ρ_{i-1} and ρ_i.

When all rows have been processed, we are left with a set of closed crack codes representing all the borders of S, with outer borders represented counterclockwise and hole borders clockwise. A simple example of the operation of the algorithm is shown in Fig. 10.

If we want to convert crack or chain codes into runs, we must store them in such a way that it is easy to access all the code strings relating to a given row. For example, we can break each code into a set of substrings each of which represents a monotonically nondecreasing or nonincreasing sequence of y-coordinates, and record the coordinates of the endpoints of these segments. We can then generate a row-by-row run representation by scanning the appropriate substrings (in forward order, if y is nonincreasing; in reverse order, if it is nondecreasing) to determine the run ends. Further details will not be given here; see [8].



(b)
(1) 122223, 12223, 1223
(2) 10122223223, 1012232212223003, 12223
(3) 1000010122322122230322210122232230003, 1122233, 103
(4) 10000101223221222303222101222232230003000000, 111222333, 1032
(5) (same), 111222333000, (same)

Fig. 10 Border tracking. (a) Input, with rows numbered. (b) Crack codes at successive rows; new segments on each row are underlined.

e. Reconstruction from borders

Suppose that we are given all the borders of S, e.g., we are given $\chi_{S'}$, which has 1's at border points of S and 0's elsewhere. Assume that we are using 8-connectedness for S and 4-connectedness for $\bar{S}$.[§] We might then try to reconstruct S along the following lines: (1) Label all the 4-components of 0's (see Section 11.3.1 on connected component labeling); let b be the label of the background component. (2) Mark 1's adjacent to b's, say with primes. (3) If $l \neq b$ is any label that occurs adjacent to a primed 1, change all points labeled l to primed 1's. (4) Mark 1's adjacent to primed 1's with stars. (5) If l is any label that occurs adjacent to a starred 1, change all points labeled l to b's. ("Adjacent" in all of these steps means "4-adjacent".) Steps (2)–(5) are repeated until all labels of 0's have been changed to either b's or primed 1's. Unfortunately, this process breaks down if any hole border and outer border of S have a point in common; that point will be adjacent to b's at some stage, and so will become primed, which will result in the 0's inside the hole turning into primed 1's. To avoid this problem, we must be given more than just $\chi_{S'}$. (In fact, $\chi_{S'}$ alone does not determine S; for example, if S' is

```
      1
  1       1
      1
```

is S

```
      1
  1   1   1
      1
```

or is S the same as S'?) In particular, suppose that 1's that lie on more than one border are specially marked. When such 1's are adjacent to b's, we given them stars rather than primes; labels occurring adjacent to them are then turned into b's, not into primed 1's. The successive steps in this algorithm are illustrated schematically in Fig. 11.

If S has only a few borders, a border-following method of reconstructing t from its borders may be more appropriate than the approach just described. Suppose we are given the crack code of the (C, D)-border of S, and the co-ordinates of the initial pair (P, Q); based on the convention that P is always on the left, this allows us to find all the successive pairs. As we find them, we mark the P's as 1's and the Q's as 0's (on an initially blank array). When this

[§] If we are using 4-connectedness for S, the borders used here must be "thick," i.e., must include all points of S that are 8-adjacent to $\bar{S}$. Otherwise, e.g., the interior points of S that are adjacent to the background component (see below) will be labeled as part of it.

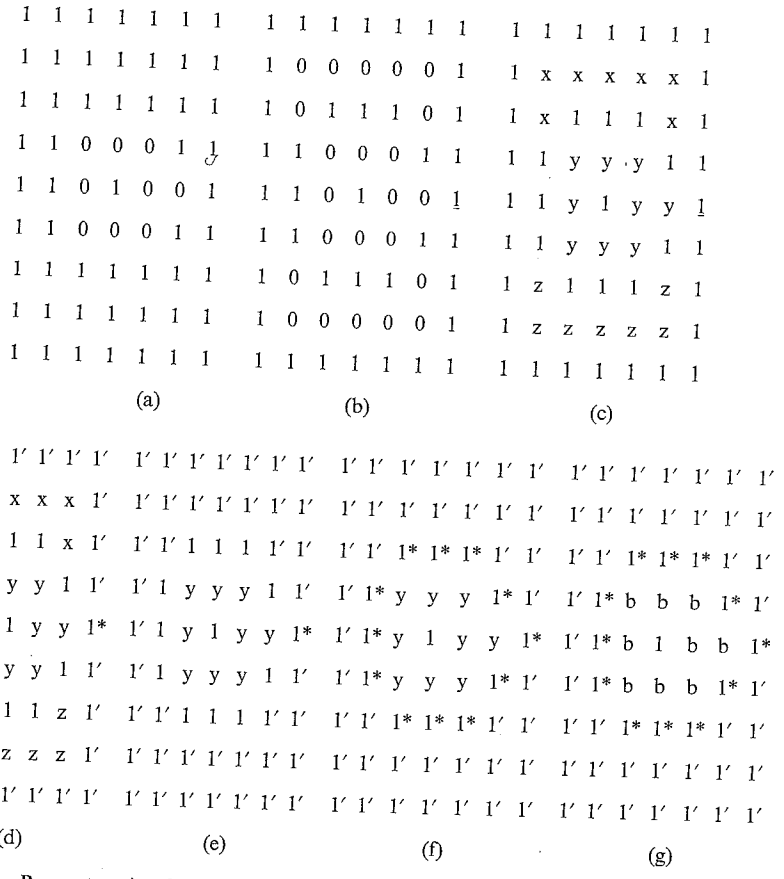


Fig. 11 Reconstruction from borders. (a) S (surrounded by 0's). (b) S' (surrounded by 0's); the 1 that lies on two borders is underlined. (c) All 4-components of $\bar{S}'$ (surrounded by b's) are labeled. (d) 1's adjacent to b's are primed; the previously underlined 1 is starred. (e) The labels x and z occur adjacent to primed 1's; all x's and z's become primed 1's. (f) 1's adjacent to primed 1's are starred. (g) The label y occurs adjacent to starred 1's; all y's become b's. Step (d) is now repeated, and the center 1 is primed; if it were on the outer border of a component of 1's having holes, repetition of steps (e)–(g) would turn that component's interior into primed 1's, its hole borders into starred 1's, and its holes into b's.

process is complete, the D-border of C has been completely marked with 1's, and the C-border of D with 0's. After all the borders of S have been marked in this way, it is easy to "fill in" the rest of S (and $\bar{S}$) by expanding 1's and 0's into 4-neighboring blanks (but not into each other); on a cellular array processor, the number of expansion steps required is less than the diameter of the picture. On a conventional computer, we can "fill in" S and $\bar{S}$ by doing a single row-by-row scan of the picture, turning consecutive blanks into 0's

starting at any 0 or at the picture border, and turning consecutive blanks into 1's starting at any 1.

The reconstruction process is somewhat more complicated if we are given chain codes rather than crack codes. Suppose we know the chain code of the D-border of C and the coordinates of the starting point P; then the chain code gives us the next point P'. Let us first consider the case where we use 8-connectedness for C. We know that the 8-neighbor of P preceding P' (in counterclockwise order) is Q', and that (at least) the 4-neighbor of P preceding P' is in D (it is either Q, or is one of the R's visited before finding a 1; note that if P' is an 8-neighbor of P, this 4-neighbor is the same as Q'). We mark P and P' as 1, and this 4-neighbor—call it Q—and Q' as 0 (on the initially blank array). The chain code now gives us the next point P'', and we know that the 8-neighbor of P' preceding P'' is Q'', and that all the 8-neighbors of P' between Q' and Q'' are also in D. We mark P'' as 1 and all these 8-neighbors (including Q'') as 0's. This process is repeated until we reach the end of the chain; this should get us to P again, say from P^*. At this step, the 8-neighbors of P between Q^* and Q must all be 0's. Note that we may visit a point more than once, but different neighbors will be 0's at each visit. The D-border of C and C-border of D have now all been marked with 1's and 0's, respectively. The procedure is analogous when C is 4-connected, except that now only the 4-neighbors of P_i between Q_i and Q_{i+1} become 0's, since the intermediate 8-neighbors may actually be 1's. Furthermore, if P_{i+1} is an 8-neighbor of P_i, we know that the 4-neighbor following it must also be in C, and we make both of these neighbors 1's. A simple example of reconstruction from a border chain code is given in Fig. 12.

Expanding border 1's and 0's into blanks can be used to "fill in" regions even when the borders do not form perfectly closed curves [75]. For example, suppose that we have detected a set of edges that lie on a region border; we mark the points on the dark sides of the edges as 1's, and the points on the light sides as 0's, as illustrated in Fig. 13a. We then expand both 0's and 1's into blanks; when a blank has both 0's and 1's as neighbors, we mark it 0. Figure 13b shows the results of the first expansion step; it is evident that in a few more steps, the region will be filled with 1's and the background with 0's. This technique can only be used if the edges "surround" the border adequately, and there are no noise edges in the interior or background. If the border is not well surrounded, 1's will "escape" into the background; if noise edges are present, the expansion will create regions of 0's in the interior or 1's in the background.

f. Quadtrees and borders

In this subsection we briefly describe methods of converting between quadtree and border representations. The details can be found in two papers

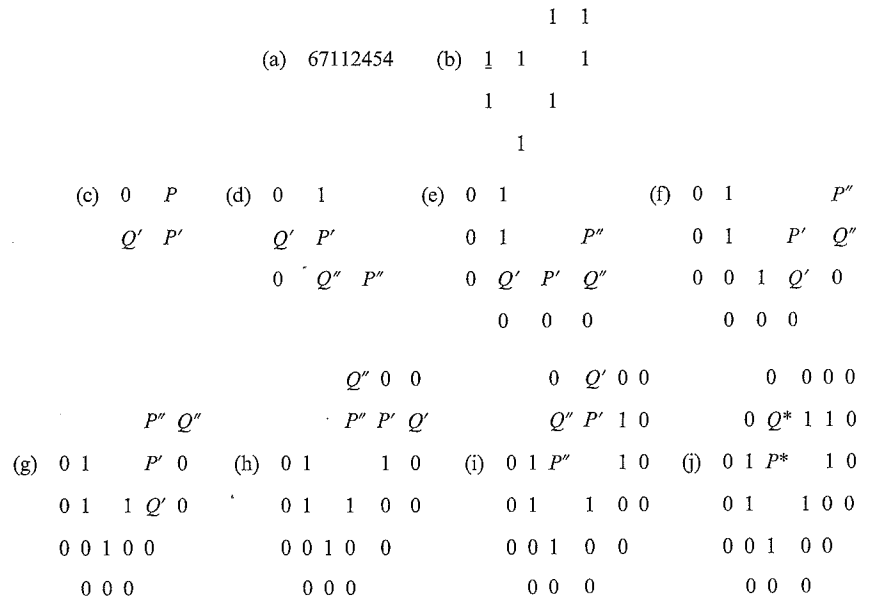


Fig. 12 Reconstruction from border chain codes. (a) Chain code; the border itself is shown in (b), with the starting point underlined. (c) Initial step; (d)–(j) successive steps. At the last step, with $P^* = 1$ and $Q^* = 0$, it is straightforward to fill in the interior by expanding 1's into blanks. The process shown here treats the 1's as 8-connected.

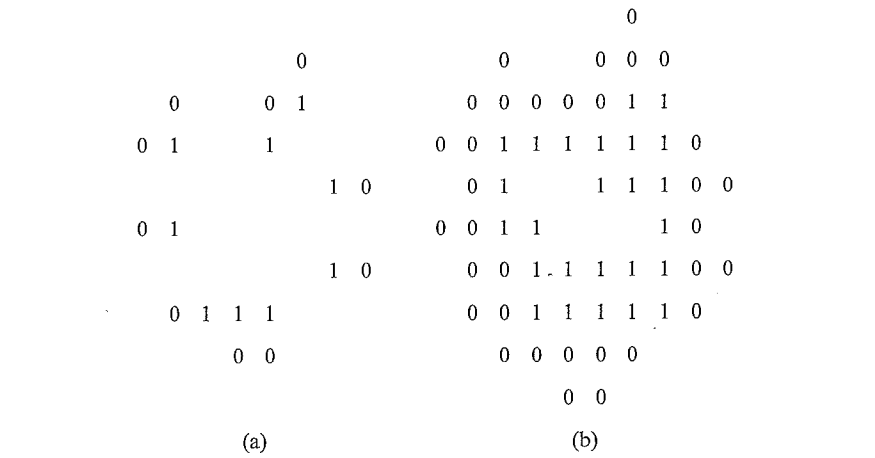
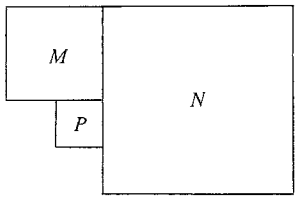


Fig. 13 Reconstruction from incomplete borders. (a) Initial 0's and 1's representing points on the light and dark sides of a border. (b) Result of 4-neighbor expansion of 0's and 1's into blanks (one step); blanks adjacent to both 0 and 1 become 0 (cf. near the lower left). Evidently, on successive steps the hole will fill with 1's and the background with 0's.

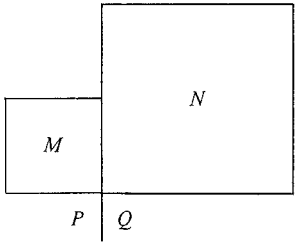
by Samet *et al.* [16, 64]. Conversion between MAT and border representations will not be treated here; on construction of a continuous MAT from a polygonal border see Montanari [38].

In order to determine, for a given leaf node M of a quadtree, whether the corresponding block is on a border, we must visit the leaf nodes that correspond to 4-adjacent blocks and check whether they are black or white. To find the nodes corresponding to, e.g., right-hand neighbor blocks, we move upward from M in the tree until we reach some ancestor node from its northwest or southwest son. (If we reach the root node before this happens, M is on the east edge of the picture and has no right-hand neighbor blocks.) As soon as this occurs, we go back down the tree making the mirror images of the moves made on the way up, i.e., the first move down is to the northeast or southeast son, and the following moves are to northwest or southwest sons. If a leaf node is reached by the time we come to the end of this move sequence, its block is at least as large as M's block, and so is M's sole right-hand neighbor. Otherwise, the nonleaf node reached at the end of the sequence is the root of a subtree whose leftmost leaf nodes correspond to M's right-hand neighbors, and we can find these nodes by traversing that subtree.

Let M, N be black and white leaf nodes whose blocks are 4-adjacent. Thus the pair M, N defines a common border segment of length 2^k (the smaller of the side lengths of M and N) which ends at a corner of M or of N (or both). To determine the next segment along this border, we must find the other leaf P whose block touches the end of this segment:



If the segment ends at a corner of both M and N, we must find the other two leaves P, Q whose blocks meet at that corner:



This can be done by an ascending and descending procedure similar to that described in the preceding paragraph; see [16] for the details. The next border segment is then the common border defined by M and P if P is white, or by N and P if P is black. [In the common corner case, the pair of blocks defining the next border segment is determined exactly as in the crack following algorithm of Subsection (a) above, with M, N, P, Q playing the roles of P, Q, U, V, respectively.] This process is repeated until we come to M, N again, at which stage the entire border has been traversed. The successive border segments constitute a crack code, broken up into pieces whose lengths are powers of 2. The time required for this process is on the order of the number of border nodes times the tree height.

Using the methods described in the last two paragraphs, we can traverse the quadtree, find all borders, and generate their crack codes. As in Subsection (c) above, we should mark each border as we follow it, so that we will not follow it again from another starting point; note that the marking process is complicated by the fact that a node's block may be on many different borders.

To generate a quadtree from a set of crack codes, we first traverse each code and create pairs of leaf nodes having the given border segments, together with the necessary nonleaf nodes. We then generate leaf (and nonleaf) nodes corresponding to the interior blocks. At any stage, if four leaves with a common father all have the same color, they are merged. The details of this algorithm will not be given here; see [64]. The time required is on the order of the perimeter (= total crack code length) times tree height.

11.2.3 Curves

This section discusses methods of "thinning" a given S into a set of arcs and curves. We will assume that S consists entirely of elongated parts, so that the resulting arcs and curves constitute a reasonable approximation to S (Section 11.1.5c). For other types of S's the results of thinning would not be particularly meaningful. (For a definition of elongatedness see Section 11.3.4b.) The result of thinning S will be called the *skeleton* of S.

The MA of an everywhere elongated S can be regarded as a skeleton, but it has two defects. At places where S has even width, its MA is two points thick, since the MA is the set of maxima of the distance to $\bar{S}$, as shown in Section 11.2.1a. Also, as pointed out there, the MA tends to be disconnected, and we would like connected pieces of S to be thinned into connected arcs or curves. In this section we will describe a thinning scheme that yields thin, connected skeletons.

a. Thinning

Our thinning algorithm is a specialized shrinking process which deletes from S, at each iteration, border points whose removal does not locally disconnect their neighborhoods; it can be shown [54] that this guarantees that the connectedness properties of S do not change, even if all such points are deleted simultaneously. To prevent an already thin arc from shrinking at its ends, we further stipulate that points having only one neighbor in S are not deleted.

Unfortunately, if we delete all such border points from S, and S is only two points thick, e.g.,

1 1 1 1 1
1 1 1 1 1

then S will vanish completely. We could avoid this by using an algorithm that examines more than just the immediate neighbors of a point, but such an algorithm would have to be quite complicated. Instead, we delete only the border points that lie on a given side of S, i.e. that have a specific neighbor (north, east, south, or west) in $\bar{S}$, at a given iteration. To insure that the skeleton is as close to the "middle" of S as possible, we use opposite sides alternately, e.g., north, south, east, west .... (It is possible to devise algorithms that remove border points from two adjacent sides at once, e.g., north and east, then west and south; but this approach is somewhat more complicated and will not be described here in detail. Another possibility is to check the neighbors of a point on two sides to determine whether they too will be deleted, and if so, not to delete the given point.)

In order to state the algorithm more precisely, we must give the exact conditions under which a border point can be removed. The border point P of S is called *simple* if the set of 8-neighbors of P that lie in S has exactly one component that is adjacent to P. This last clause means that if we are using 4-connectedness for S, we care only about components that are 4-adjacent to P. If we are using 8-connectedness, the last clause can be omitted. For example, P is 4-simple if its neighborhood is

0 1 1
0 *P* 0
1 0 0

since in this case only one 4-component of 1's is 4-adjacent to P; but P is not 4-simple if its neighborhood is

0 1 1 or 0 1 0
0 *P* 0 or 0 *P* 1
0 1 0 or 0 0 0

On the other hand, P is 8-simple in the third case, but not in the first two cases.

It is easily seen that deleting a simple point from S does not change the connectedness properties of either S or $\bar{S}$; $S - \{P\}$ has the same components as S, except that one of them now lacks the point P, and $\bar{S} \cup \{P\}$ has the same components as $\bar{S}$, except that P is now in one of them. Note that an isolated point (having no neighbor in S) is not simple, and that an end point (having exactly one neighbor in S) is automatically simple.

Our thinning algorithm can now be stated as follows: Delete all border points from a given side of S, provided they are simple and not end points. Do this successively from the north, south, east, west, north, ... sides of S until no further change takes place. A simple example of the operation of this algorithm is shown in Fig. 14.

The deletion of border points from a given side of S should be done "in parallel," i.e. the conditions for deletion of a point should be checked before any other points are deleted. (Suppose we did not do this, but simply performed the deletion row by row. When we deleted north border points, we would strip away layer after layer from the top of S, and the resulting skeleton would not be symmetric; e.g., if S were an upright rectangle, nothing would be left but its bottom row after the first operation.) Thus each iteration of the algorithm can be done as a simple parallel operation on a cellular array computer.

The algorithm can be implemented on a conventional computer, using one scan of the picture for each iteration. On each row, points are marked for deletion, but are not deleted until the points on the following row have been marked. We can avoid repeated scanning of the entire picture by operating on the rows in sequence 1; 2, 1; 3, 2, 1; ... as in Section 11.2.1c. After k steps, where $2k + 1$ is the maximum width of S, no more thinning is needed on the first row, so it can be dropped; thus only about k rows need to be available at a time.

Exercise 12 [3]. Prove that a north border point (=having north neighbor 0) is 8-simple iff

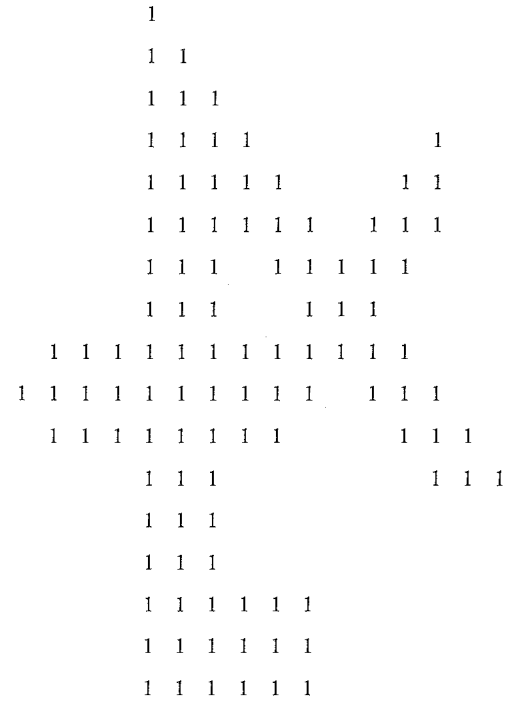
$$\mathrm{w\bar{s}e} + \bar{\mathrm{w}}(\mathrm{nw})\bar{\mathrm{n}} + \bar{\mathrm{n}}(\mathrm{ne})\bar{\mathrm{e}} + \bar{\mathrm{e}}(\mathrm{se})\bar{\mathrm{s}} + \bar{\mathrm{s}}(\mathrm{sw})\bar{\mathrm{w}} = 0$$

where n, e, s, w, (nw), (ne), (se), (sw) denote the north, south, east, west, northwest, northeast, southeast, and southwest neighbors of the point, respectively, and the overbars denote negation ($\bar{0} = 1$, $\bar{1} = 0$). Can you formulate an analogous Boolean condition for 4-simplicity? ▮

b. Alternative thinning schemes

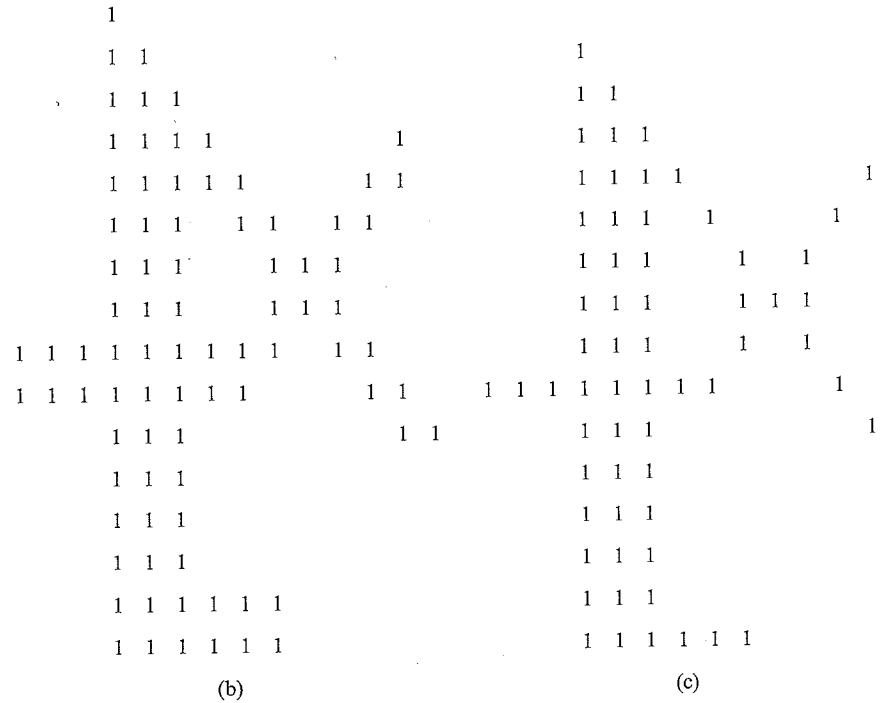
Simplified approaches to thinning can sometimes be used. For example, if S has everywhere essentially constant thickness (see Section 11.3.2d), say

$2k + 1$, it can be thinned (at least roughly) by shrinking it k times. Note, however, that the resulting skeleton may occasionally be thick or broken. The thinning processes described in (a) will handle S's of variable width. Another method [32] that can be used if S is an arc of constant thickness is to initiate two border-following processes at one end of S that traverse the opposite sides of S. (Analogously, if S is a closed curve of constant thickness, we initiate two border-following processes at points just across the width of S from each other, which traverse the two borders of S.) If the distance between the border followers gets significantly larger than the width, we stop one of them until the other one catches up; thus they always remain approximately



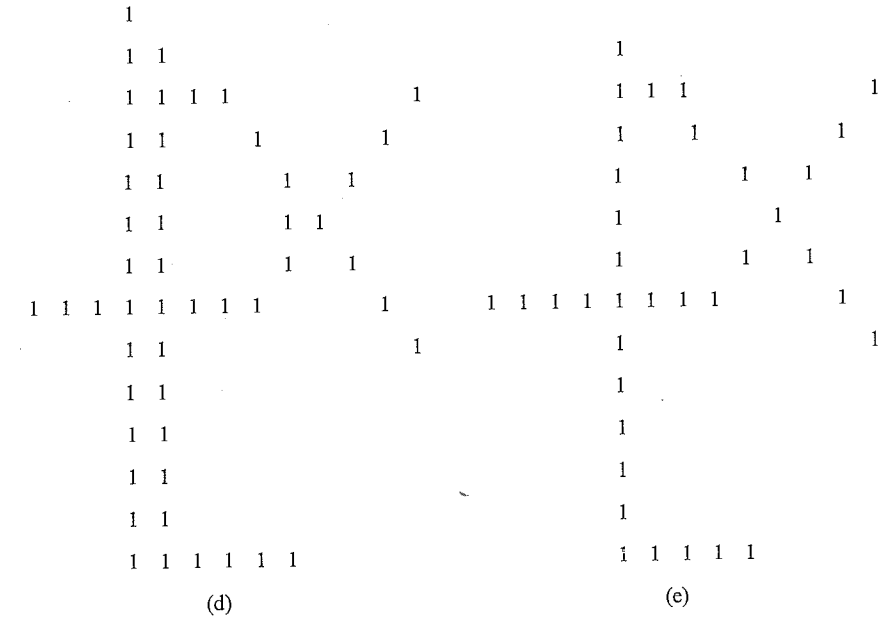
(a)

Fig. 14 Thinning. (a) Original S; (b)–(e) results of thinning successively from north, south, east, and west, treating S as 8-connected. At the next north step, the uppermost point of (e) will be deleted; at the next south step, the leftmost of the lowermost points; and at the next west step, the leftmost of the uppermost points. There will be no other changes. Note how the angle at the upper left shrinks down to the same height as the diagonal line at the upper right. (b′)–(e′) Analogous results treating S as 4-connected, and defining end points as having only one 4-neighbor in S. One more point will be removed at the next north step; there will be no other changes.



(b)

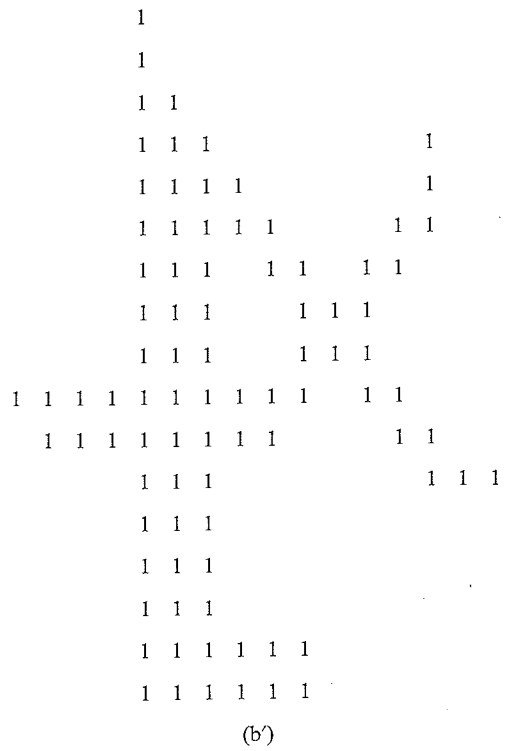
(c)



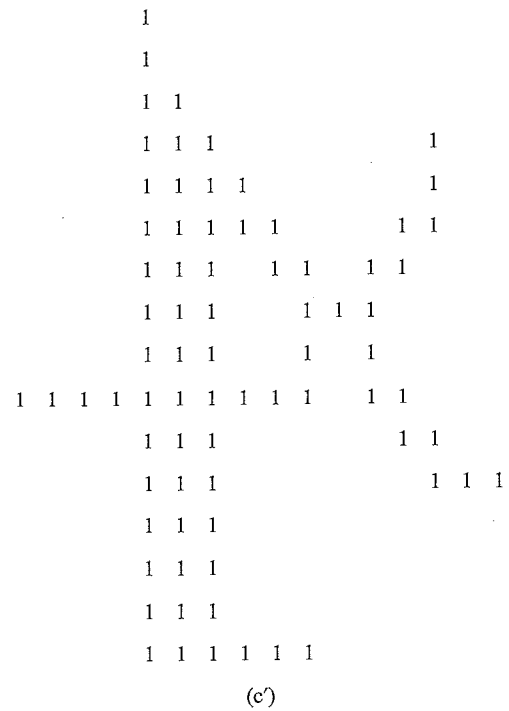
(d)

(e)

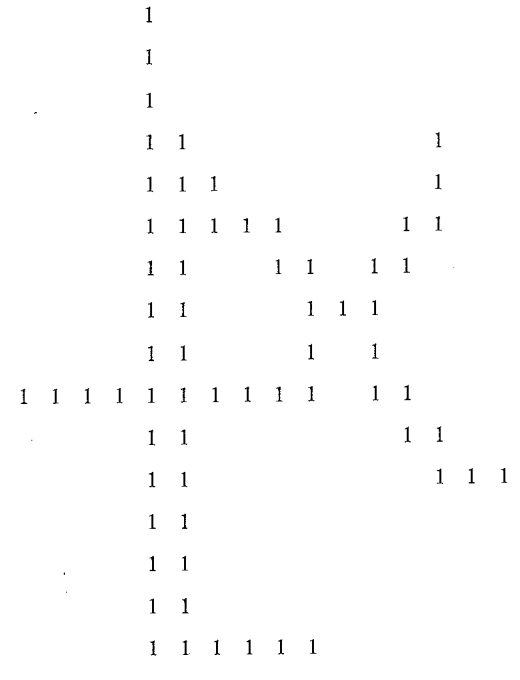
Fig. 14 (*Continued*)



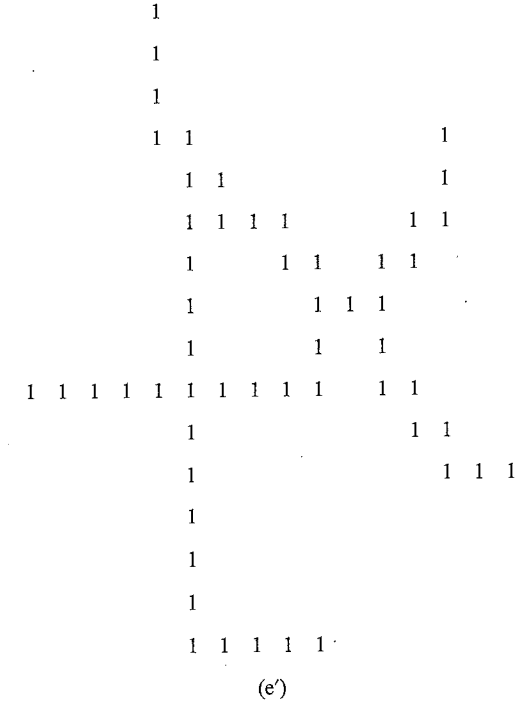
(b′)



(c′)



(d′)



(e′)

Fig. 14 *(Continued)*

ılongside one another. The midpoint of the line segment joining the border ollowers thus traces out a skeleton of S.

Thinning algorithms can be defined for multivalued pictures in various vays. One approach [15] is to generalize the definition of a simple point as ollows: Define the strength of a path $P_1, \ldots, P_n$ as the minimum value of any oint on the path, and the degree of connectedness of P and Q as the maximum trength of any path from P to Q. We call P "simple" if replacing it by the ninimum of its neighbors does not decrease the degree of connectedness of ıy pair of points within its 8-neighborhood. It can be verified that this eneralizes the two-valued definition given in (a). Thinning is then defined ıs a specialized local minimum operation: we repeatedly replace points by he minima of their neighbors, provided they are simple and have more than ne higher-valued neighbor (this generalizes the condition that isolated and nd points are not deleted). At each iteration we do this from only one side, .e., we do it only to points that have a lower-valued neighbor on a specific ide (north, south, ...). The results of applying this process to a set of pictures ıre shown in Fig. 15.

The output of edge detection operators is often thick (see Section 10.2). t can be thinned by suppressing nonmaxima in the gradient direction at ach point; this discards all but the steepest edge value on each cross section f the edge, but does not allow points along the edge to compete with one nother. Another possibility is to increase or decrease the value at each point n proportion to how much greater or smaller it is than its neighbors in the radient direction. This causes the maximum value on each edge cross section o grow at the expense of the lower values, so that eventually it absorbs all f the responses on its cross section [17].

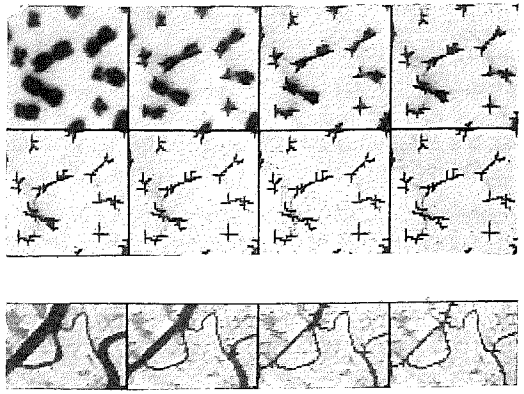


Fig. 15 Examples of gray-weighted thinning.

c. The results of thinning

Ideally, we call a subset A of the set T a simple digital *arc* if it is a connected component of T each point of which has exactly two neighbors in T, except for the two end points, which have one neighbor each. Note that this is two definitions in one, depending on whether we mean 4- or 8-neighbors. Note also that an arc cannot branch, cross itself or even touch itself, since otherwise it would contain points having more than two neighbors in S. A simple digital closed *curve* is a connected component C of T each point of which has exactly two neighbors in T; here too we have two definitions. Note that a border is not always a simple closed curve, since it may pass through some points twice.

Exercise 13. Show that A cannot be both a 4-arc and an 8-arc unless it is a horizontal line segment, and that C can never be both a 4-curve and an 8-curve. ∎

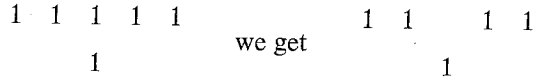
The goal of thinning S is to produce a T which is a union of such arcs and curves, perhaps after a few crossing or branching points—i.e., points having more than two neighbors—have been deleted. Note that such points will often occur in clusters; e.g., consider

```
      1                          1     1
      J                        J J
1 J J J 1      and               J J
      J                    1       1
      1
```

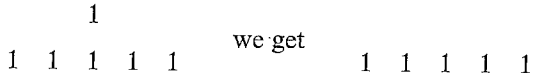
where we have indicated the junction points by J's. If many curve branchings and intersections occur close together, it becomes difficult to identify and classify individual junctions. A thinned S may even have interior points; an 8-connected example is

```
  1   1   1
    1 1 1
  1 1 1 1 1
    1 1 1
  1   1   1
```

in which every border point is either an end point or nonsimple, so that thinning has no effect on it. The results of thinning depend on orientation; for example, when we 8-thin



but when we 8-thin



(Under 4-thinning, neither of these patterns changes.)

Once points having more than two neighbors have been eliminated, it is straightforward to construct the chain code of an arc by moving from neighbor to neighbor, starting at one of the end points and ending at the other; or of a closed curve by beginning at an arbitrary starting point and moving in one direction until the starting point is reached again.

Thinning sometimes produces "spiky" skeletons, especially when S has "hairy" borders that contain many end points, or when end points arise prematurely in the course of the thinning process; this is especially likely in the 4-connected version of the algorithm. One way [56] to detect an inappropriate skeleton branch is to consider the distances of the points of the branch from $\bar{S}$, or, equivalently, the iterations at which they become border points. On a true skeleton branch these distances should be approximately constant, but on a spike branch they should increase steadily as we move in from the tip of the spike.

11.3 GEOMETRIC PROPERTY MEASUREMENT

In this section we describe how to measure geometrical properties of and relationships among subsets of a digital picture, using the various representations introduced earlier. We also discuss methods of deriving new segmentations from a given one, based on geometrical properties or relations.

11.3.1 Topology

We first consider properties and relationships involving the concepts of adjacency and connectedness. These properties are "topological" in the sense that they do not depend on size or shape; in the continuous case, they

remain unchanged under arbitrary "rubber-sheet" distortions. For the basic definitions of these concepts see Section 11.1.7.

a. Component labeling of arrays

We often want to treat the individual connected components of a set S as separate objects. The array representation of these objects is a multivalued picture in which the points of each component have a unique nonzero label, and the points of $\bar{S}$ have value zero. In the following paragraphs we discuss how to construct this representation from a given representation of S.

If S is represented by a binary picture, we can label its components by performing two row-by-row scans. Let us first assume that we want to label 4-connected components. During the first scan, for each point P having value 1, we examine the upper and left-hand neighbors of P; note that if they exist, they have already been visited by the scan, so that if they are 1's, they have already been labeled. If both of them are 0's, we give P a new label; if only one is 0, we give P the other one's label; and if neither is 0, we give P (say) the left one's label, and if their labels are different, we record the fact that they are equivalent, i.e., belong to the same component. When this scan is complete, every 1 has a label, and no label has been assigned to points that belong to different 4-components; but many different labels may have been assigned to points in the same component (see Figs. 16a and 16b). We now sort the equivalent pairs into equivalence classes, and pick a label to represent each class. Finally, we do a second scan of the picture and replace each label by the representative of its class (Fig. 16c); each component has now been uniquely labeled.

To label 8-connected components, we also examine the two upper diagonal neighbors of each 1; these have also already been visited by the scan. If all four neighbors are 0, the current point P gets a new label; if one of them is 1, P gets the same label; if two or more of them are 1, P gets one of their labels, and the equivalences are noted. (We need only note those equivalences that have not already been detected in previous positions of P; the details are left to the reader.) The equivalence processing and relabeling are done just as in the 4-connected case. Alternatively, we can examine the left, upper, and upper left neighbors of every point P, whether P is 0 or 1; if P is 1, we proceed as above, and if P is 0, but its upper and left neighbors are 1, we note the equivalence of their labels. Figures 16d and 16e show results of the first scans using these two schemes.

The component labeling process seems to be basically sequential. Algorithms can be devised for labeling components using a cellular array computer, but they are not very efficient. For example, suppose that we first give each 1 a unique label, e.g., its coordinates in the picture; this can be done

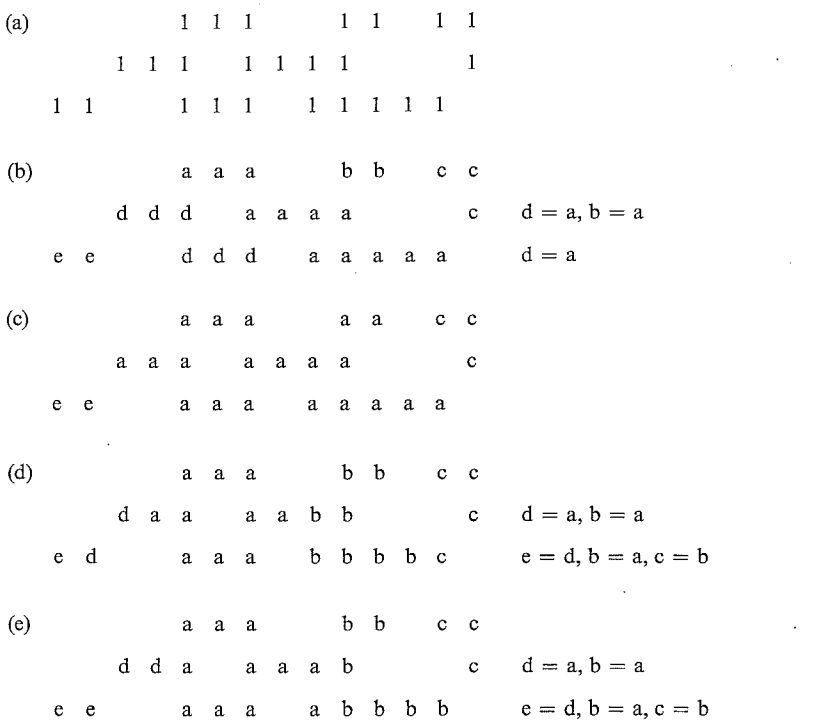


Fig. 16 Component labeling. (a) Input S. (b) Result of first scan, using 4-connectedness for S; equivalences discovered on each row are indicated. (c) Result of second scan, replacing all equivalent labels by a representative one. (d)–(e) Results of first scan using two versions of the 8-connectedness algorithm (see text).

in a number of steps equal to the picture diameter. We can then repeatedly perform a local maximum operation in parallel, where the maximum is defined by the lexicographic ordering of the coordinate pairs; points labeled 0 remain 0. We use the 4-neighbor maximum if we want to label 4-components, and the 8-neighbor for 8-components. When this is iterated until no further change takes place, every point of a given component is labeled with the coordinates of the uppermost of its rightmost points. However, the number of iterations required may be on the order of the picture area (consider a snakelike component).

Component labeling can be carried out simultaneously for the components of S and $\bar{S}$, or in fact for the components of the sets in any partition $S_1, \ldots, S_m$ (e.g., the partition of an arbitrary picture into sets of constant gray level), using 4- or 8-connectedness for the S_i's in any combination.

b. Component labeling in other representations

A row-by-row approach to labeling the components of S (or of an arbitrary partition) can easily be defined in the case where S is given by its run length representation. On the first row, each run of 1's gets a new label. On subsequent rows, we compare the position of each run ρ with those of the runs on the preceding row. If ρ is adjacent (4- or 8-) to no run on the preceding row, it gets a new label. If it is adjacent to just one such run, it gets that run's label. If it is adjacent to two or more such runs, it gets (say) the first of their labels, and we note that all of their labels are equivalent. When the rows have all been processed, we sort out the equivalences and then do a second scan to give the runs their final labels.

If S is represented by a quadtree, we can label its components by traversing the tree in a standard order, say NW, NE, SW, SE. Whenever we come to a black leaf node, we visit the leaf nodes whose blocks adjoin M's block on its south and east sides (or at its southeast corner, if we are labeling 8-components); see Section 11.2.2f on how to find these nodes. If we find unlabeled black leaf nodes, we give them the same label as M; if we find black leaf nodes that already have labels, we note that their labels are equivalent. When the traversal is complete, we sort out the equivalences, retraverse the tree, and give the black leaf nodes their final labels. The time required is on the order of the number of nodes in the tree times the tree height. For the details of this algorithm see [66].

To label components based on the MAT representation, we must check all pairs of blocks that overlap or are adjacent (i.e., whose centers are no farther apart than the sum of their radii) in order to determine label equivalences. The details of this process are left to the reader. If we are given a border representation, and we know which are the outer borders, we can mark the borders in an array as in Section 11.2.2e, and give the points of each outer border a unique label; these labels can then be expanded into the interiors.

c. Component counting

Once we have labeled the components of S, we know how many components it has, since this is just the number of final labels used. In this subsection we describe a method of counting components without labeling them, based on a parallel shrinking operation [34, 37] suitable for implementation on a cellular array computer.

Suppose that we use 4-connectedness for S and 8- for $\bar{S}$. If the right, lower, and lower right neighbors of P are

$$\begin{matrix} P & X \\ Y & Z \end{matrix}$$

we define the operation Ψ as taking P into 1 iff $P + X = 2$, $P + Y = 2$, or $X + Y + Z = 3$. Readily, this is equivalent to saying that

$$\text{if } P = 1, \quad \text{we have } \Psi(P) = 0 \quad \text{iff } X = Y = 0$$

$$\text{if } P = 0, \quad \text{we have } \Psi(P) = 1 \quad \text{iff } X = Y = Z = 1$$

For any subset T of the picture, let T_1 be the set of 1's that are either in T or immediately above and to the left of T after Ψ is applied, and let T_0 be the set of such 0's. It can be shown that if C is a 4-component of 1's, so is C_1, and if D is an 8-component of 0's, so is D_0 (unless C or D consists of only one point, in which case C_1 or D_0 is empty). Moreover, if C is 4-adjacent to D, then C_1 is 4-adjacent to D_0. Thus, except for components consisting of single points, Ψ preserves the connectedness properties of both S and $\bar{S}$.

When Ψ is applied repeatedly, it can be shown that any component C of S shrinks to the single point (x_c, y_c), where x_c is the coordinate of the leftmost point of C and y_c is the coordinate of its uppermost point. This single point then vanishes. The number of steps required for this to happen is just the largest city block distance between (x_c, y_c) and any point of C. The same is true for any component D of $\bar{S}$ other than the background component. A component can shrink to a point even if it originally has holes, because the holes shrink to points (which then vanish) before the component does. An example of the operation of Ψ is shown in Fig. 17.

If we use 8-connectedness for S and 4- for $\bar{S}$, Ψ is defined to take P into 1 iff $P + Z = 2$ or $P + X + Y \geqslant 2$. Readily, this is equivalent to

$$\text{if } P = 1, \quad \text{we have } \Psi(P) = 0 \quad \text{iff } X = Y = Z = 0$$

$$\text{if } P = 0, \quad \text{we have } \Psi(P) = 1 \quad \text{iff } X = Y = 1$$

The results of applying this Ψ repeatedly are exactly analogous to those above. Of course, we could have defined Ψ using other 2×2 neighborhoods of P; such Ψ's would shrink components toward the northeast, southeast, or southwest rather than toward the northwest. A symmetric shrinking algorithm, based on a 3×3 neighborhood, is described in [51].

To count components using a Ψ operation, we modify it so that, instead of vanishing, isolated points turn into special marks, which do not interfere with the effect of Ψ on 1's and 0's. These marks shift leftward and upward to a counter at the northwest corner of the picture. The number of iterations required to shrink all components to points and count them in this way is less than the diameter of the picture.

d. The genus

If all the components of S are simply connected (i.e., have no holes), special methods can be used to count the components.§ More generally, for any S, special methods can be used to compute the number of components of S minus the number of holes; this number is called the *genus* or *Euler number* of S.

If we repeatedly delete simple points from S in parallel, it can be shown that any simply connected component either shrinks to a single point or vanishes. (For example, any component of the form

1 1 1 ⋯ 1
1 1 1 ⋯ 1

vanishes immediately, since all its points are simple.) By making the deletion direction-dependent, we can guarantee that all simply connected components shrink to single points; the details will not be given here. Thus we can count the simply connected components of S by applying this process until there is no further change and then counting the resulting isolated points by shifting them to the upper left corner.

There are several simple methods of computing the genus $g(S)$ by counting local patterns of various types in the picture Σ. Let us use the following notation (blanks can be either 0's or 1's):

Pattern	No. of those patterns in Σ	Pattern	No. of those patterns in Σ
1	v	$\begin{matrix}1&0\\0&0\end{matrix}$, $\begin{matrix}0&0\\0&0\end{matrix}$, $\begin{matrix}1&0\\0&0\end{matrix}$, $\begin{matrix}0&0\\1&0\end{matrix}$ or $\begin{matrix}0&0\\0&1\end{matrix}$	v'
1 or 1 1 1	e	$\begin{matrix}1&0\\0&1\end{matrix}$ or $\begin{matrix}0&1\\1&0\end{matrix}$	d'
1 1 or 1 1	d	$\begin{matrix}0&1\\1&1\end{matrix}$, $\begin{matrix}1&0\\1&1\end{matrix}$, $\begin{matrix}1&1\\0&1\end{matrix}$ or $\begin{matrix}1&1\\1&0\end{matrix}$	t'
$\begin{matrix}1&1\\1&1\end{matrix}$, $\begin{matrix}1&1\\1&1\end{matrix}$, $\begin{matrix}1&1\\1&\end{matrix}$ or $\begin{matrix}1&1\\&1\end{matrix}$	t		
$\begin{matrix}1&1\\1&1\end{matrix}$	q		

§ One should be careful about assuming that all components are simply connected; a single pinhole adjacent to the border is just as much of a hole as a completely hollowed out interior.

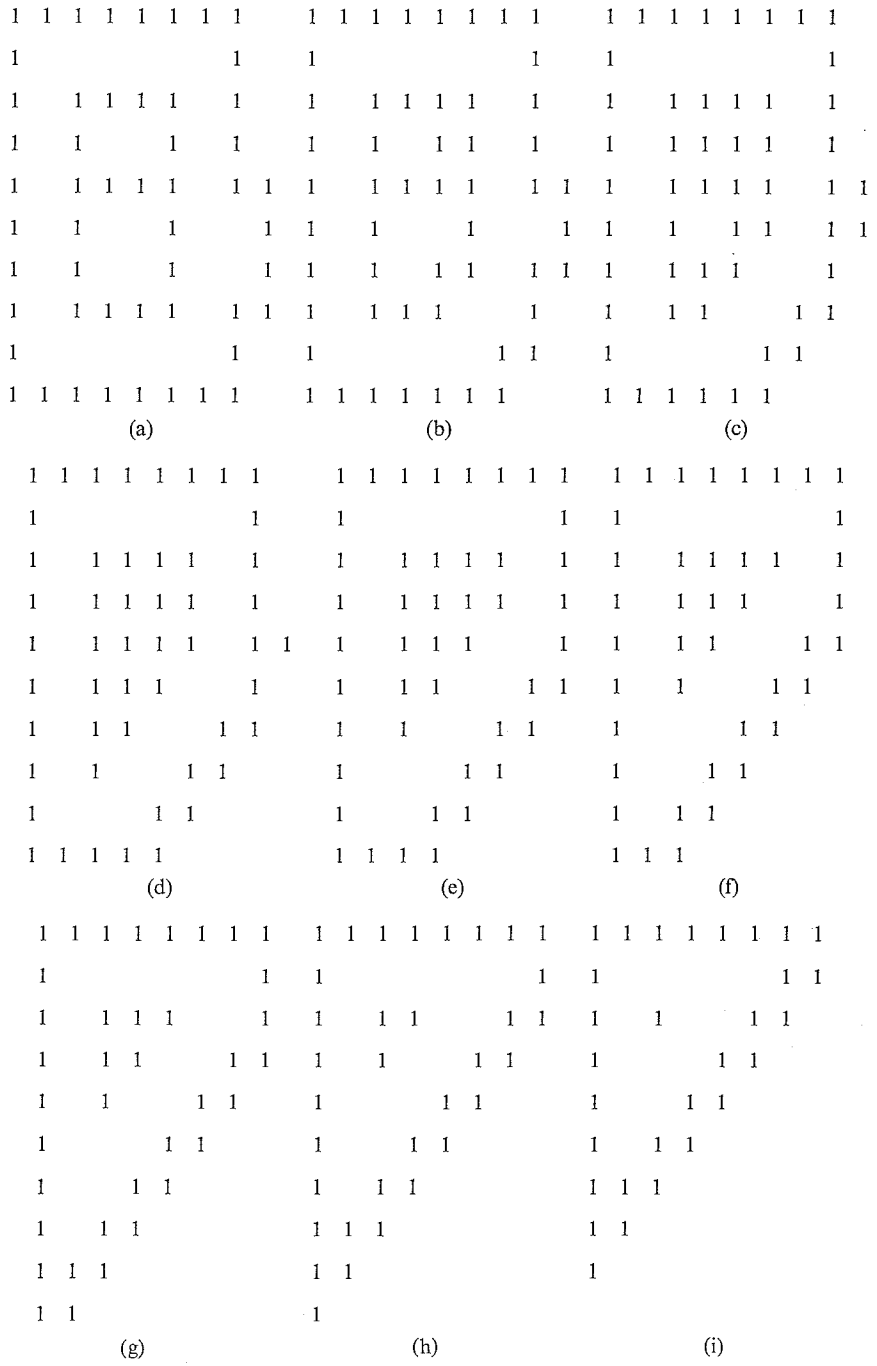


Fig. 17 Shrinking operation Ψ. (a) Input S. (b)–(q) Results of successive applications.

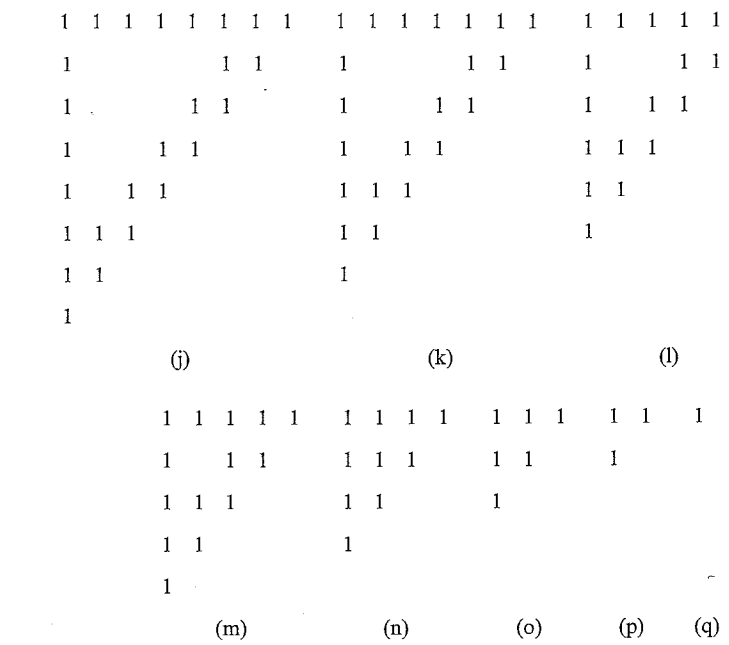


Fig. 17 (*Continued*)

If we use 4-connectedness for S and 8- for $\bar{S}$, then it can be shown [23, 37] that

$$g(S) = v - e + q = \tfrac{1}{4}(v' - t' + 2d')$$

In the reverse case we have

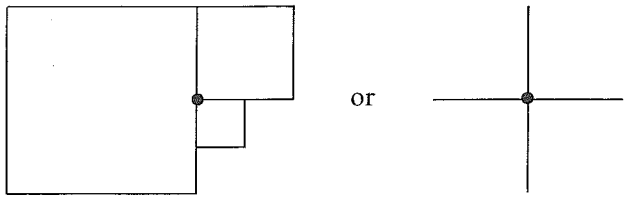
$$g(S) = v - e - d + t - q = \tfrac{1}{4}(v' - t' - 2d')$$

These patterns can be counted in a row-by-row scan by comparing each row with the preceding one, or they can be converted to special marks by local parallel logical operations, and the marks can then be shifted and counted.

The primed formulas just given provide a very simple method of computing the genus from the crack codes of the borders of S. In fact, both formulas imply that $4g(S)$ is the number of convex corners of S minus the number of concave corners. To see this, note that each v' pattern has a convex corner at its center point; each t' pattern has a concave corner; and each d' pattern has two convex corners if diagonally adjacent 1's are not connected, but two concave corners if they are connected. Of course, if we know which borders of S are outer borders and which are hole borders, we can obtain

$g(S)$ by simply subtracting the number of hole borders from the number of outer borders.

The unprimed formulas can be generalized to the case where S is represented by a quadtree [14]. In particular, let v be the number of black leaf nodes; e the number of pairs of such nodes whose blocks are horizontally or vertically adjacent; and q the number of triples or quadruples of such nodes whose blocks meet at and surround a common point, e.g.,



Then $g(S) = v - e + q$. These adjacencies can be found (see Section 11.2.2f) by traversing the tree; the time required is on the order of the number of nodes in the tree times the tree height.

Finally, we give a very simple formula for computing the genus from the run length representation of S. For each run ρ, let $k(\rho)$ be the number of runs on the preceding row to which ρ is adjacent; then $g(S) = \sum_\rho (1 - k(\rho))$.

e. Adjacency, surroundedness, and nesting

Given any partition of a picture Σ, say into $S_1, \ldots, S_m$, we define the (4- or 8-) *adjacency graph* of the partition as the graph whose nodes are the S_i's, and in which nodes S_i and S_j are joined by an arc iff S_i is adjacent to S_j. It is straightforward to construct this graph by scanning Σ and checking the neighbors of each point to find all the adjacencies. This information is also easy to derive from the run length representation: check all consecutive pairs of runs on each row, and check all pairs of dissimilar-valued runs on successive rows for possible adjacency. Given the MAT representation, we check all pairs of dissimilar-valued MAT points for adjacency (distance between centers equal to sum of radii). In the quadtree representation, for each leaf node, we check the leaf nodes whose blocks are adjacent to it (Section 11.2.2f). In a border representation the information would be immediately available, since for each border arc we must specify which pair of regions defines it.

In the preceding paragraph the sets S_i may or may not be connected. (Given any partition $S_1, \ldots, S_m$ we can define a refined partition $C_1, \ldots, C_M$ consisting of the connected components of the S_i's.) An important special case is that in which the S_i's are the connected components of a set S and its

complement $\bar{S}$, where we use 4-connectedness for S and 8- for $\bar{S}$ or vice versa. Construction of the graph is still straightforward, once we have labeled all the components. It makes no difference whether we use 4- or 8-adjacency between components, since as already pointed out, if a component of S and a component of $\bar{S}$ are 8-adjacent, they are also 4-adjacent. In this case, as we have seen, the borders are all closed curves. Moreover, it can be shown [55] that in this case the adjacency graph is a *tree*. (*Corollary*: If C_1, C_2 are any two components of S, and D_1, D_2 any two components of $\bar{S}$, we cannot have C_1, C_2 both adjacent to each of D_1, D_2, since otherwise the graph would contain the cycle C_1, D_1, C_2, D_2.)

We recall (Section 11.1.7b) that T *surrounds* S if any path from S to the border of Σ meets T. We also saw (Section 11.1.7c) that if C, D are adjacent components of S, $\bar{S}$, respectively, then one of them surrounds the other. Hence the adjacency tree of the components of S and $\bar{S}$ becomes a directed tree if we use the additional relationship of surroundness. Evidently, this tree is rooted at the background component of $\bar{S}$, which contains the border of Σ and so surrounds all the other components of S and $\bar{S}$. Just below the root we have "continent" components of S that are adjacent to the background; just below them, "lake" components of $\bar{S}$; just below them, "island" components of S; and so on.

We often need to know whether one of two sets surrounds the other, and in particular, whether a given set S surrounds a given point P of $\bar{S}$. (In cartography, this is known as the *point-in-polygon* problem.) If we are given the array representation of S, this problem can be solved for all P by simply labeling the background component of $\bar{S}$, say with b's; now, if $P = 0$ it is surrounded by S, and if $P = b$ it is not. Maximal-block representations of S (runs, MAT, quadtree) do not seem to be very useful in connection with this problem.§ For border representations, on the other hand, there are various algorithms [44] for deciding whether P is inside or outside a given closed curve; using such algorithms, we can test whether P is inside the outer border of each component of S. We mention here only one standard approach: Move to the right (say) from P, and count the number of times that the curve is crossed (but do not count times that the curve is only touched without being crossed), until the border of Σ is reached. If the number of crossings is odd, P is inside the curve; if it is even, P is outside.

If $S_1, \ldots, S_m$ and $T_1, \ldots, T_n$ are partitions of Σ, we say that $T_1, \ldots, T_n$ is a *refinement* of $S_1, \ldots, S_m$ if every T_i is contained in some S_j—in fact, in a

§ On the other hand, if we are given a maximal-block representation of S, it is relatively easy to tell whether a given point P lies in S or in $\bar{S}$—e.g., given the MAT, we check whether P is at a distance from each block's center not exceeding its radius; but it is harder to tell this from a border representation, where we would need to check whether P is inside some outer border of S but not inside a hole border of the same component of S.

unique one, since the S's are disjoint. (*Corollary*: Any S_j is a union of T_i's.) The same definition is used in the more general case where T is a subset of S, and $S_1, \ldots, S_m$ and $T_1, \ldots, T_n$ are partitions of S and T, respectively. As an important example, it is easy to show that if $T \subseteq S$, and $S_1, \ldots, S_m$ and $T_1, \ldots, T_n$ are the components of S and T, respectively, then $T_1, \ldots, T_n$ is a refinement of $S_1, \ldots, S_m$. Thus, e.g., if we threshold Σ at s and at t, where $s \leqslant t$, and let S, T denote the sets of above-threshold points, every component of T is contained in a unique component of S.

Suppose that we have a sequence of partitions $T_{h1}, \ldots, T_{hn_h}, h = 1, 2, \ldots, k$, each of which is a refinement of the preceding one, and where the first partition is the trivial one consisting of Σ itself. Thus each T_{hi} is contained in a unique $T_{h-1,j}$. Let us define a directed graph whose nodes are the T_{hi}'s, and where each node is joined to the node of the preceding partition which contains it. Readily, this graph is a tree rooted at $T_1 = \Sigma$. This *refinement tree* was introduced in [29] for the case where the partitions are the components of above-threshold points at a series of thresholds $0 = t_1 \leqslant \cdots \leqslant t_k$ (note that since $t_1 = 0$, the first partition consists of just Σ).

11.3.2 Size

In this section we consider size properties of a set S—area, perimeter, extent in a given direction, etc. Shape properties such as straightness, convexity, elongatedness, etc., will be considered in the next two sections.

a. Area

The *area* of S is simply the number of points of S. The areas of the sets in any partition $S_1, \ldots, S_m$ of Σ can be counted in a single scan of Σ, using one counter for each S_i. On a cellular array computer, marks corresponding to the points in each S_i can be shifted, e.g., leftward and upward, and counted at the upper left corner of Σ; the time required is on the order of the diameter of Σ. The areas of the connected components of S, or of any partition, can be measured in the process of labeling them (Sections 11.3.1a and 11.3.1b); each label occurrence adds 1 to the appropriate counter, and if two labels are found to be equivalent, the contents of their counters are combined.

Area computation from various other representations is also easy. Given the run length representation of S, we simply add the lengths of all the runs of 1's to obtain the area of S; given the quadtree representation, we add the areas of all the black leaf nodes (i.e., we add 4^{k-h} for each black leaf node at level h of the tree). On the other hand, it is not straightforward to obtain the area of S from its MAT representation, since there may be multiple overlaps of blocks.

Given a border representation of S, we get its area by adding the areas enclosed by all the outer borders and subtracting the areas enclosed by all the hole borders. The area enclosed by a given border can be obtained using standard integration formulas. For example, suppose we are given the crack code $\varepsilon_1 \cdots \varepsilon_r$, where each ε_i is 0, 1, 2, or 3. If we take the starting point $(x_0, y_0) = (0, 0)$ at the origin, then the y-coordinate at the end of the kth segment is $y_k = \sum_{i=1}^{k} \Delta y_i$, where $\Delta y_i = 1$ if $\varepsilon_i = 1$, 0 if $\varepsilon_i = 0$ or 2, and -1 if $\varepsilon_i = 3$. Let $\Delta x_i = 1$ if $\varepsilon_i = 2$, 0 if $\varepsilon_i = 1$ or 3, and -1 if $\varepsilon_i = 0$. Then it can be verified that the enclosed area is $\sum_{i=1}^{n} y_{i-1} \Delta x_i$. The analogous formula for a chain code is left as an exercise for the reader [19].

The area of S and the areas of other sets that can be derived from S (e.g., its connected components, its points at a given distance from $\bar{S}$, its convex hull, etc.) are all useful descriptive properties of S. Area-based criteria can also be used to derive new sets from a given S; e.g., we can discard all connected components of less than a given area—or, less trivially, if they are adjacent to only one other set in the given partition, we can merge them with that set. It should be realized that for some partitions, most of the connected components will be very tiny; for example, it has been found empirically [43] that the number of components of constant gray level having area N is proportional to $1/N^4$.

In the foregoing, we have treated the points of S as unit squares. Alternatively, we can regard S as a set of lattice points, and define the area of S as the sum of the areas of the polygons obtained by joining the outer border lattice points of each component of S, less the sum of the areas of the polygons obtained by joining the hole border lattice points. *Pick's theorem* states that if a simply connected S has b border points and i interior points, then its area is $\frac{1}{2}b + i - 1$ (rather than $b + i$, when we treat the lattice points as unit squares). More generally [69], if S has n holes, its area is $\frac{1}{2}b + i + n - 1$.

b. Perimeter and arc length

The *arc length* of a digital arc or curve is obtained from its chain code representation by counting horizontal and vertical moves as 1 and diagonal moves as $\sqrt{2}$. This gives reasonable values for an 8-arc or 8-curve, but it gives values that are somewhat too high in the 4-case, where a diagonal move is represented by a horizontal plus a vertical move and so is treated as having length 2.

The *perimeter* of S can be defined in a number of different ways. Some possible definitions are

(1) The sum of the lengths of the crack codes of all the borders of S.
(2) The sum of the arc lengths of all these borders, regarded as 8-curves.
(3) The sum of the areas of the borders of S.

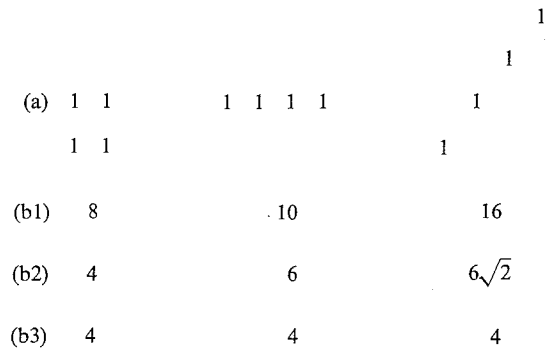


Fig. 18 Digital perimeter. (a) Three S's. (b1)–(b3) Their perimeters according to definitions (1)–(3).

Simple examples of the results obtained using these definitions are shown in Fig. 18.

It is straightforward to define algorithms for computing perimeter, as measured in any of these ways, from the array representation of S. We can easily compute the contribution of each border point and sum these contributions during a scan of S; or we can compute the contributions in parallel on a cellular array computer, and sum them by shifting and adding. Each border representation of S (crack code, chain code, χ_S) yields one of these measures directly; the problem of obtaining each of them from the other two representations (e.g., chain code length or border area from the crack codes) is left as an exercise for the reader.

Perimeter is easily measured from the run length representation of S; e.g., the crack code length is the sum of the number of run ends and the lengths of the subruns that are not overlapped by runs on the row above or on the row below. To compute crack code length from the quadtree representation, we visit each leaf node, and check the colors of the nodes whose blocks are adjacent to its block on two sides, say bottom and right, to determine which of these adjacencies contributes to the perimeter; the time required for this is proportional to the number of nodes times the tree height [68]. It is not straightforward to compute perimeter from the MAT representation; multiple overlaps of blocks make it complicated to determine how much each block contributes to the perimeter.

It should be pointed out that the perimeter of a digitized region often grows exponentially as the digitization becomes finer [35]; the area, on the other hand, tends to a finite limit. On the accuracy of estimating the lengths of straight lines and curves from digital representations see [18, 50].

If we regard S as a set of lattice points, its perimeter is the sum of the perimeters of the polygons formed by joining its border points. A formula for the perimeter computed in this way is obtained as follows [69]: Let $P_0, \ldots, P_7$ be the eight neighbors of P, numbered counterclockwise starting from the right-hand neighbor. Let

$$b_1(P) = \sum_{k=0}^{3} [(P \wedge P_{2k}) - (P_{2k} \wedge (P_{2k-1} \vee P_{2k-2}) \wedge (P_{2k+1} \vee P_{2k+2}))]$$

$$b_2(P) = \sum_{k=0}^{3} [(P \wedge P_{2k+1}) - (P_{2k} \wedge P_{2k+1} \wedge P_{2k+2})]$$

where the subscripts are all modulo 8. Then the perimeter of S is

$$\frac{1}{2} \sum_{P \in S} [b_1(P) + b_2(P)\sqrt{2}]$$

c. Extent and cross sections

The *height* of S is the vertical distance between its highest and lowest points, and similarly its *width* is the horizontal distance between its leftmost and rightmost points. [Width should not be confused with *thickness*, which remains essentially the same when S is rotated or bent; see subsection (d).] More generally, the *extent* of S in a given direction θ is the distance between its extreme points as measured parallel to θ. Equivalently: If we drop a perpendicular from each point S onto a line l of slope θ, the extent of S in direction θ is the distance along l from the foot of the first perpendicular to the foot of the last one. In other words, the extent of S is the length of its *projection* on a line of slope θ. Alternatively, if we bring together a pair of parallel lines of slope $\theta + \pi/2$ until they hit S from opposite sides, the distance between them is the extent of S in direction θ.

Measuring the height or width of S is straightforward; measuring the extent in an arbitrary direction θ is more complicated, since it is harder to identify the extremal points. A brute-force solution is to rotate S by $-\theta$ and measure the width of the rotated S. In the next paragraph we indicate how to measure width from a given representation of S.

Given the array representation χ_S, we scan it to find the leftmost and rightmost columns containing 1's; the difference between the coordinates of these columns is the width. (A counting algorithm can be devised to compute this difference on a cellular array computer in time proportional to the picture diameter.) Given the run length representation, we scan it to find the coordinates of the leftmost and rightmost ends of runs of 1's; similarly, given the quadtree representation, we traverse the tree to find the leftmost and rightmost blocks of 1's. Given a border representation, we can scan the codes of the outer borders to find the leftmost and rightmost border points of S

(which are evidently also the leftmost and rightmost points of S). For example, given a crack code, we compute $x_k = \sum_{i=1}^{k} \Delta x_i$ for each k [see (a) above], and keep track of its greatest positive and negative values over all k; these indicate how far to the right and left of the starting point the given border extends.

The extent of S in a given direction is an orientation-sensitive property of S. Properties of this type are useful primarily when the orientation is known. Alternatively, we can *normalize* the orientation of S, e.g., by finding its greatest extent and rotating it to make it vertical, or by finding its smallest-area circumscribing rectangle (i.e., finding a pair of perpendicular directions in which the product of the extents of S is smallest) and rotating to make its long side vertical. Methods of normalizing the orientation of an arbitrary picture will be discussed in Section 12.1.1c; these methods can also be used for segmented pictures.

We obtain more detailed directional information about S if we examine its individual *cross sections* in a given direction (or along a given family of curves; but we will consider here for simplicity the cross sections defined by the rows of the picture). Each such cross section (of χ_S) consists of runs of 1's separated by runs of 0's. Various properties of these runs, as functions of row position, can provide useful descriptive properties of S; e.g., we can use the extent of the set of runs, their total length, the number of runs, etc. We can also use comparisons between runs on successive rows as a basis for segmenting S into pieces of simple shape [24]. In particular, we can break S into parts consisting of successions of single runs that do not change radically in length or position; whenever runs split or merge, or grow, shrink, or shift significantly, new pieces of S are defined. Thus each piece is a strip having approximately constant or slowly changing width. Of course, properties or segments derived from these runs are quite orientation-sensitive; they too should be applied only when the orientation is known, or after normalizing it.

d. *Distances and thickness*

The greatest extent of S in any direction is called its *diameter*. Readily, this is equal to the greatest distance between any two points of S. The extents of S in various directions, or the distances between points of S, are sometimes useful as descriptive properties.

Knowing the set of all its interpoint distances does not determine S up to congruence; for example [37],

```
1 1 1          1 1 1
  1     and    1   1
  1
```

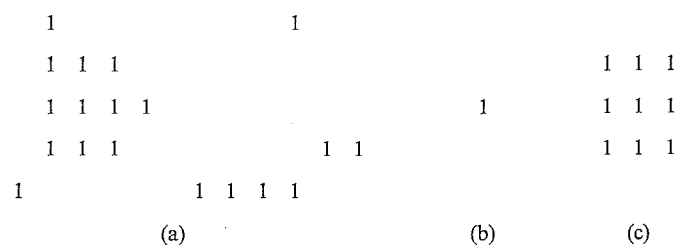



Fig. 19 Simplification by shrinking and reexpanding. (a) S; (b) $S^{(-1)}$; (c) $(S^{(-1)})^{(1)}$.

have the same set of distances (four 1's, two 2's, two $\sqrt{2}$'s, and two $\sqrt{5}$'s). On the other hand, if we have a one-to-one correspondence between S and T such that corresponding pairs of points have the same distance, S and T must be congruent; this generalizes the familiar theorem (corresponding to the case where S and T have three points each) that a triangle is determined up to congruence by specifying the lengths of its three sides.

One can also use the notion of distance to define new sets from a given S, e.g., the set of points of Σ at (or within) a given distance of S or of $\bar{S}$. We recall (Section 11.2.1c) that the k-step expansion $S^{(k)}$ of S is the set of points within distance k of S, while the k-step contraction $S^{(-k)}$ is the set of points further than k from $\bar{S}$. In the following paragraphs we show how combinations of expanding and shrinking can be used to extract new sets from S, and to "clean" it.

Note first that shrinking followed by expanding wipes out small parts of S; it can thus be used as a noise cleaning operation. A simple example is given in Fig. 19. We observe that this process also wipes out thin parts of S such as curves; it should not be used if such parts are significant. In fact, we can define the *thickness* of S as twice the number of shrinking steps required to wipe it out. Methods of detecting thin or elongated parts of S by shrinking, expanding, and comparing with the original S will be discussed in Section 11.3.4b.

We now show how expanding followed by shrinking can be used to identify isolated parts and clustered parts of S. Suppose that we expand S k times and then shrink the result k times, i.e., we compute $(S^{(k)})^{(-k)}$; we recall that this set always contains S. A small isolated piece of S (where "isolated" means "more than $2k$ away from the rest of S") simply expands and shrinks back, so that it gives rise to a small connected component of $(S^{(k)})^{(-k)}$. On the other hand, a cluster of pieces of S (less than $2k$ apart) "fuses" under the expansion, and only shrinks back at its edges, so that it yields a large component of $(S^{(k)})^{(-k)}$, as illustrated in Fig. 20. Thus to detect clusters or isolated pieces of S, we compute $(S^{(k)})^{(-k)}$ (for various k's) and examine its connected

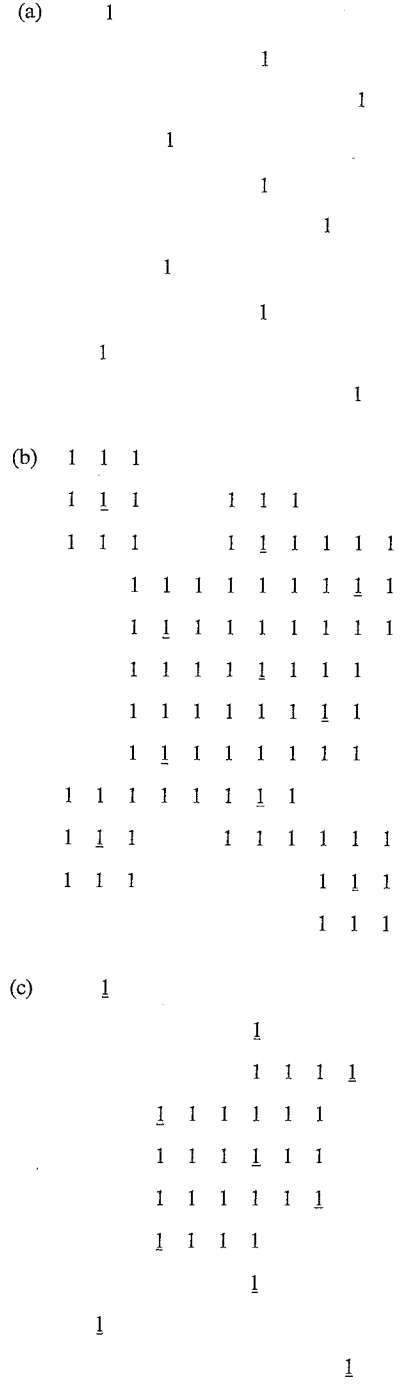


Fig. 20 Cluster detection by expanding and reshrinking. (a) S; (b) $S^{(1)}$; (c) $(S^{(1)})^{(-1)}$. Eight-neighbor expansion and shrinking were used. In (b) and (c) the original points of S are underlined.

components; if the area of a component is large relative to k^2, it must have arisen from a cluster of pieces of S, while if it is small, it must have arisen from an isolated piece.

11.3.3 Angle

In this section we discuss properties involving slope and curvature, as measured for curves or borders. Most of this material assumes that the given curve or border is specified by a chain code. We also briefly discuss relationships of relative position (e.g., "to the left of") between objects.

a. Slope and curvature

The slope of a chain code at any point is a multiple of 45°, and the slope of a crack code is always a multiple of 90°. In order to measure a more continuous range of slopes, we must use some type of smoothing. For example, we can define the *left* and *right k-slopes* at a point P as the slopes of the lines joining P to the points k steps away along the curve on each side. Alternatively, we might define the k-slope at P as the average of the unit slopes (i.e., codes) at a sequence of k points centered at P.

Curvature is ordinarily defined as rate of change of slope. Here again, if we use differences of successive unit slopes, we always have a multiple of 45° for chain code, and a multiple of 90° for crack code. To obtain a more continuous range of values, we must again use smoothing. For example, we can define the *k-curvature* of P as the difference between its left and right k-slopes. (We would not want to use the average of the unit curvatures (i.e., differences of consecutive codes) at a sequence of k points centered at P, since curvatures should be cumulated rather than averaged.)

The k-slopes (and analogously for the k-curvature) can take on angular values whose tangents are rational numbers with denominator $\leqslant k$. Of course, k-slope is not defined if P is less than k away from an end of the arc. We have not specified here how to choose k; its choice depends on the particular application. Ordinarily, k should not be too large a fraction of the total arc length or perimeter. (Consider what happens in the case of a closed curve when k is half the perimeter or more!)

As a simple example, consider the arc in Fig. 21a and assume it to be continued indefinitely in both directions. Here the right 1-slope is 0° at some points and 45° at others, but the right 3-slope is $\tan^{-1}(\frac{1}{3})$ at every point (this is because the arc is periodic with period 3). For $k > 3$, the right k-slopes are not all equal, but as k increases, they all approach $\tan^{-1}(\frac{1}{3})$. Similarly, the k-curvatures fluctuate for small k (except that they are all zero for $k = 3$), and approach zero as k gets large.

For a border, say of a region of 1's, we can also measure curvature at a point P from the array representation by counting the number of 1's in a neighborhood of P. If this is about half the neighborhood size, the border is relatively straight at P; if it is much higher or much lower, the border has a sharp convex or concave curvature at P. The neighborhood size used determines the amount of smoothing in this definition.

b. Straightness

It is of interest to characterize those digital arcs which could arise from the digitization of a straight edge or line segment. We give a characterization here in terms of chain code; the crack code case is left to the reader. As examples, Figs. 21a and 21b are digital straight line segments, but Figs. 21c–21e are not.

The following conditions on the chain code are necessary for straightness [53]:

(1) At most two slopes occur in the chain, and if there are two, they differ by 45°. This is violated by Fig. 21c.

(2) At least one of the two slopes occurs in runs of length 1. This is violated by Fig. 21d.

(3) The other slope occurs in runs of at most two lengths (except possibly at the ends of the arc, where the runs may be truncated), and if there are two lengths, they differ by 1. This is violated by Fig. 21e.

These conditions are not sufficient. In fact, it turns out that at least one of the two run lengths occurs in runs of length 1; the other occurs in runs of at

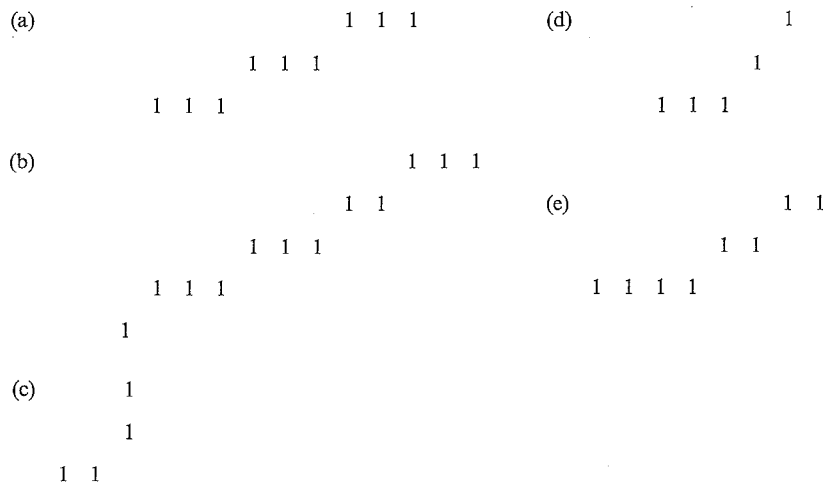
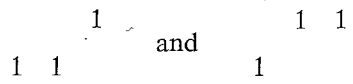


Fig. 21 Digital straightness. (a) and (b) are digital straight-line segments, but (c)–(e) are not.

most two lengths (except at the ends) which differ by 1; and so on. These conditions ensure that the runs of length 1 are spaced as evenly as possible in the chain. In any case, for many purposes one would not bother to check these conditions exactly; it would suffice to check that the smoothed slope is approximately constant, or that the fit of the arc to a real straight line is good.

Note that there may be more than one straight-line segment between two points, if the direction from one to the other is not a multiple of 45°. As a very simple example, consider

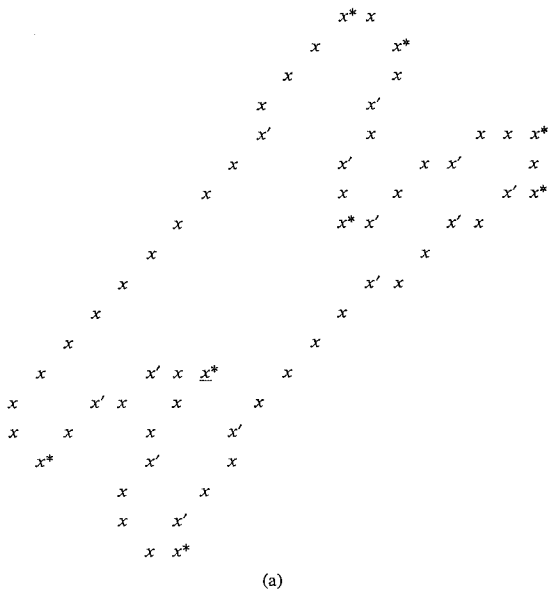


c. Curve segmentation

There are many ways of segmenting a curve into parts. Some of these are analogous to techniques for picture segmentation (Chapter 10), with slope (or smoothed slope) playing the role of gray level. Others do not depend on slope; for example, we can segment a curve at local or global extremum points, e.g., leftmost, rightmost, uppermost, or lowermost. It is assumed here that segmentation at junctions, if any, has already been done, so that we are dealing with a simple arc, curve or border.

The *slope histogram* of a curve tells us how often each slope occurs on the curve. Of course, it does not tell us how these slopes are arranged along the curve; a squiggle consisting of equal numbers of horizontal and vertical segments may have the same slope histogram as a square. However, for noncomplex curves it is reasonable to assume that a peak on the slope histogram provides information about overall orientation. It should also be pointed out that if the curve is the border of a convex object, its slope histogram does determine it, since in this case the sequence of slopes around the curve must be monotonic. Figures 22a–22c show a closed curve, its chain code, and its histogram of 1-slopes; the two peaks, 180° apart, correspond to the predominant orientation of the curve. The slope histogram is sometimes called the "directionality spectrum."

Information about the "wiggliness" of a curve can be obtained from its *curvature histogram.* For a smooth curve, the curvatures will be concentrated near 0, whereas for a wiggly curve they will be more spread out. This is analogous to the use of histograms of gray level difference magnitudes to describe the coarseness or busyness of a texture; see Section 12.1.5c. The 1-curvature histogram for Fig. 22a is shown in Fig. 22d; since the curve is smooth, the absolute values are nearly all $\leqslant 1$. One can use the curvature histogram to determine a threshold for segmenting a curve into straight



(a)

55656 70211 21111 10110 10224 45455
42212 12345 55655 55555 55671 10100

(b)

Slope	0	1	2	3	4	5	6	7
Frequency	7	15	8	1	5	18	4	2

(c)

Difference	−3	−2	−1	0	+1	+2	+3
Frequency	1	1	13	23	18	4	0

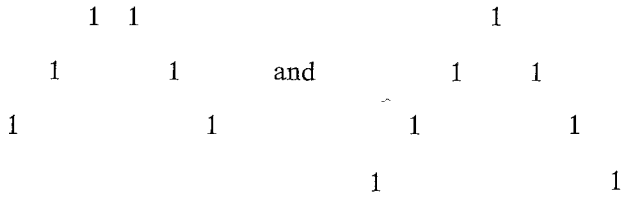
(d)

Fig. 22 Digital curve segmentation. (a) Curve; starting point underlined, curvature maxima starred, inflections primed. (b) Chain code. (c) Histogram of 1-slopes. (d) Histogram of 1-curvatures.

and curved parts; and similarly, one can use the slope histogram as a guide in segmenting a curve into relatively straight parts ("strokes") in various directions.

The analog of edges for curves are *corners*; these are points where the slope changes abruptly, i.e., where the absolute curvature is high. Thus the simplest types of "corners" are maxima of the (smoothed) curvature [57]. The maxima of 1-curvature in Fig. 22a are starred; generally speaking, they

represent a reasonable set of corners. This simple definition, however, does not distinguish very well between smooth and sharp turns, e.g., between



as long as the total amount of rotation is the same. One way to make this distinction is to compare the k-curvatures for several values of k; if small k's give the same value as large ones at the maximum, the corner must be sharp [20, 60].

Inflections are points where the sign of the curvature changes; they separate the curve into convex and concave parts. The 1-inflections in Fig. 22a are primed.

Various types of *local features* on a curve can be detected by drawing chords. The (maximum) distance between a chord and its arc is high (relative to the chord length) when the arc is sharply curved and low when it is relatively straight; but it might also be wiggly or S-curved. Another measure is the ratio of arc length to chord length; this is high for a sharply curved or wiggly arc, and low for a relatively straight one. Both of these measures are especially high for "spurs" or "spikes" on the curve. Of course, the sizes of the features that are detected in this way depend on the lengths of the arcs that are used (i.e., on k). (Note that this is a length-based, rather than slope- or curvature-based, curve segmentation technique.)

More generally, arbitrary curvature sequences can be detected on a curve by a *matching* process. For example, we can compute a sum of absolute slope differences between corresponding points, just as in picture matching, to obtain measures of the mismatch of a "template" with a curve in various positions. Much of the discussion in Chapter 9 about picture matching applies to curve matching as well, with appropriate modifications.

The curve segmentation methods mentioned up to now are all "parallel," i.e., they are applied independently at all positions along the curve. One can also use sequential methods (Section 10.4) analogous to tracking or region growing, e.g., keep extending an arc as long as it remains a relatively good fit to a straight line. Splitting techniques can also be used—for example, if a chord is not a good fit to its arc, subdivide the arc (e.g., at the point farthest from the chord), draw the chords of the two arcs, and repeat the process for each of them. Of course, splitting and merging can be combined; for a detailed discussion of this approach to the approximation of curves see [46],

Chapters 2 and 7, as well as [9]. When constructing polygonal approximations to a curve in this way, a good place to put the initial polygon vertices (i.e., the initial segmentation points) is at the curvature maxima, since the curve is turning rapidly at these points and cannot be fit well there by a single line segment.

Iterative "relaxation" methods (Section 10.5) can also be used in curve segmentation. For example, suppose that at each point we estimate initial corner and no corner (= straight) probabilities at each point, based on the k-curvature for some k. With each corner probability we associate the amount of turn (e.g., the k-curvature itself, which is the difference between the left and right k-slopes), and with each straight probability we associate the average of these two k-slopes. Straights then reinforce nearby straights to the extent that their slopes agree; a corner reinforces nearby straights on each side, to the extent that its slope on that side agrees with the slope of the straight, and it competes with nearby corners. When this reinforcement process is iterated, the corner probabilities become high at sharp curvature maxima, and the straight probabilities become high elsewhere [12].

d. *Curve equations and transforms*

In the real plane, various types of equations can be used to define curves. *Parametric equations* specify the coordinates of the points on the curve as functions of a parameter, i.e., $x = f(t)$, $y = g(t)$. The *slope intrinsic equation* specifies slope as a function of arc length along the curve; this determines the curve once a starting point is given. Similarly, the *curvature intrinsic equation* specifies curvature as a function of arc length; this determines the curve, given a starting point and initial slope. (The parameter t in the parametric equations might also be arc length.) A curve can sometimes be defined by specifying one coordinate as a function of the other, e.g., $y = f(x)$ or $r = f(\theta)$; but this is only possible when the function is single-valued.

All of these types of equations can be used for digital curves. Here the arc length and the coordinates are all discrete-valued. The functions can be specified by listing their values, or we can specify them analytically and obtain the discrete values by quantization. Evidently, the chain code is a digital version of the slope intrinsic equation, since it specifies slope as a function of position along the curve; note, however, that the positions used are not evenly spaced, since diagonal steps are $\sqrt{2}$ times as long as horizontal or vertical steps. Similarly, the difference chain code (Section 11.1.3) is a digital version of the curvature intrinsic equation.

Given a string of numbers specifying a curve, we can compute a one-dimensional discrete transform of the string, e.g., its discrete Fourier transform. (Note that for closed curves, the string can be regarded as periodic, which simplifies its Fourier analysis.) In the case of parametric equations, we can still use a one-dimensional transform if we combine the two real coordinates into a single complex coordinate, i.e., $x + iy = f(t) + ig(t)$.

The transform can provide useful descriptive information about the curve. For example, a wiggly curve should have stronger high-frequency content than a smooth curve; compare the use of Fourier-based features to measure texture coarseness/busyness in Section 12.1.5d. Symmetries in the curve should be detectable as peaks in its transform, and the lower-frequency transform values should be good descriptors of the gross shape of the curve. If we derive these features from the magnitudes of the Fourier coefficients (i.e., from the power spectrum), they should be essentially rotation-invariant, since except for quantization effects, rotation of the curve corresponds to a cyclic shift of the string of values that specify the curve. We can get scale invariance by using ratios of transform values.

One can also "filter" a curve by operating on the transform and then inverse transforming, e.g., smooth the curve by suppressing high frequencies from the transform. Note, however, that the inverse transform may not be a simple closed curve unless the changes made to the transform are small.

Exercise 14. Prove that a crack code defines a closed curve iff $n_0 = n_2$ and $n_1 = n_3$, where n_i is the number of i's in the code. Similarly, prove that a chain code defines a closed curve iff $n_7 + n_0 + n_1 = n_3 + n_4 + n_5$ and $n_1 + n_2 + n_3 = n_5 + n_6 + n_7$. ▮

e. *Relative position*

In describing a picture, we often want to make statements about the relative positions of objects in the picture, e.g., "A is to the left of B," "A is above B," "A is near B," or "C is between A and B." Note that except for "near" (and "far"), these are relations of relative bearing, i.e., they involve the direction of A as seen from B, or the relative directions of A and B as seen from C.

For point objects, it is not hard to give fuzzy definitions for these relations, e.g., "A is to the left of B" has value 1 for the 180° direction (where B is at the origin), and drops off to zero in some appropriate way as we go from 180° to ±90°. For extended objects, however, defining these relations becomes quite complicated, as Fig. 23 shows. If we required that every point of A be to the left of every point of B, we exclude all but cases (a) and (b); but excluding case (e) seems unreasonable. If we require only that every point of A be to the left of some point of B, we include all but case (f); but including (d) seems unreasonable. If we require that A's centroid be to the left of B's centroid, we include all but case (d); but including (f) seems unreasonable. One proposed definition [76] requires that two conditions be satisfied: A's centroid must

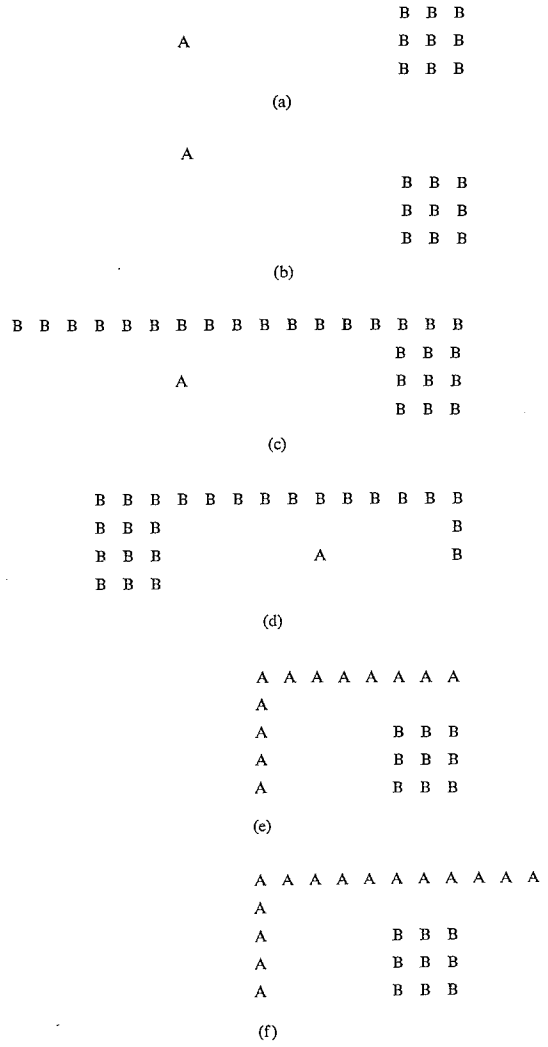


Fig. 23 The difficulty of defining "to the left of." In which of these cases is the object composed of *A*'s to the left of the object composed of *B*'s?

be to the left of *B*'s leftmost point, and *A*'s rightmost point must be to the left of *B*'s rightmost point. This definition excludes (c), (d), and (f), which is defensible. However, a purely coordinate-based rule of this type will not be adequate in all cases; our judgements about "to the left of" depend on our three-dimensional interpretation of the given picture, and may even depend on the meanings of the objects that appear in the picture.

11.3.4 Shape

Border features such as corners provide information about shape at a local level. In this section we discuss global shape properties such as complexity, elongatedness, and convexity. Other measures which provide global information about shape, such as moments and Fourier coefficients, will be discussed in Chapter 12, since they are applicable to unsegmented pictures. In this section, *S* is usually a connected set.

a. Complexity

Human judgments of shape complexity depend on several factors. Of course, topological factors play an important role; the numbers of components and holes in *S* affect its judged complexity. We will assume in the following paragraphs that *S* is simply connected, i.e., is connected and has no holes, so that it has only a single border.

The *wiggliness* or *jaggedness* of *S*'s border is an important complexity factor. This can be measured by the total absolute curvature summed over the border [77]. Alternatively, we might simply count the curvature maxima ("corners"), perhaps weighted by their sharpnesses.

Another frequently proposed complexity measure is p^2/A, $(perimeter)^2/area$. (p is squared in this expression to make the ratio independent of size; when we magnify *S* by the factor m, p is multiplied by m and A is multiplied by m^2.) In the real plane, the "isoperimetric inequality" states that $p^2/A \geqslant 4\pi$ for any shape, with equality holding iff the shape is a circle; the ratio increases when the shape becomes elongated or irregular, or if its border becomes wiggly. In a digital picture, it turns out [52] that, depending on how perimeter is measured (see Section 11.3.2b), p^2/A is smaller for certain octagons than for digitized circles. However, we can still use it as a (rough) measure of complexity.§

The complexity of *S* also depends on how much *information* is required to specify *S*. Thus the presence of *equal parts*, *periodicities*, or *symmetries* in *S* reduce its complexity. As indicated in Section 3.5, there is a tendency to perceive certain ambiguous pictures in such a way that their descriptions are simplified. For example, Fig. 3.18a is easy to see as a three-dimensional cube, since this interpretation makes the lines and angles all equal, while Figs. 3.18b is easier to see as two-dimensional. The three-dimensional interpretation of Fig. 3.18b is also improbable, since it requires that two corners of the cube be exactly in line with the eye.

§ Of course, if we want to measure *circularity*, we can compute, e.g., the standard deviation of the distances of the border points of *S* from its centroid [25]; this is small iff *S* is approximately circular.

The measurement of curvature, perimeter, and area have already been discussed. Periodicity or symmetry of (parts of) a picture can be detected using (approximate) matching techniques (Chapter 9); in the case of symmetry, we must rotate by 180° (for symmetry relative to a point) or reflect (for symmetry relative to a line) before matching. It is easiest to perceive symmetry of a picture about a vertical or horizontal axis; in these cases, symmetry of a shape is relatively easy to detect using any of the standard representations—array, run length, MAT, quadtree [2], or border—or from approximations [10]. The details will not be given here. On the use of moments of odd order as asymmetry measures see Section 12.1.3.

b. *Elongatedness*

As in Section 11.2.1c, let $S^{(k)}$ denote the result of expanding S k times, and $S^{(-k)}$ the result of shrinking it k times. Suppose that $S^{(-k)}$ is empty, i.e., S vanishes when we shrink it k times. Thus every point of S is within distance k of $\bar{S}$, so we can say that the *thickness* t of S is at most $2k$, as in Section 11.3.2d.

Suppose that the area A of S is large relative to k^2, say $A \geq 10k^2$. Then we can call S elongated, since its intrinsic "length" $(=A/t) \geq 10k^2/2k = 5k$ is at least $2\frac{1}{2}$ times its thickness. In general, we can define the *elongatedness* of a simply connected S as A/t^2, where A is the area of S and t is twice the number of shrinking steps required to make S disappear.

Note that this measure of elongatedness is unreliable for small values of t; for example,

```
1 1                    1 1 1 1
1 1      and
```

both have $A = 4$ and $t = 2$, but we would only call the second one elongated. We can use this measure even when S has holes, but its reasonableness is less obvious; we might call a ring elongated, but what about a sieve? If S is noisy (i.e., has pinholes), it should be cleaned (e.g., by expanding and reshrinking; see Section 11.3.2d) before applying the methods of this subsection.

Somewhat more information about the elongatedness of S can be obtained by studying how S's area changes as we shrink it, rather than merely using A and t. In particular, for an everywhere elongated S the rate of decrease of area should be relatively constant, since the areas of the borders of S, $S^{(-1)}$, $S^{(-2)}, \ldots$ are nearly the same; whereas for a compact S, the rate of decrease should steadily decline, since the border areas decrease.

Overall measures of the elongatedness of S are only of limited usefulness, since S may be partly elongated and partly not, and the elongated parts may have different thicknesses (e.g., S consists of streams, rivers, and lakes). We

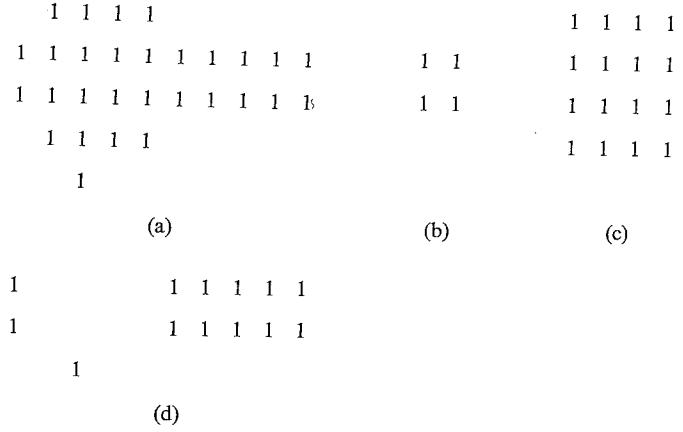


Fig. 24 Elongated part detection. (a) S; (b) $S^{(-1)}$; (c) $(S^{(-1)})^{(1)}$; (d) $S - (S^{(-1)})^{(1)}$. The 10-point component is large, hence is elongated.

will now show how shrinking and reexpanding can be used to detect elongated parts of S.

Consider $(S^{(-k)})^{(k)}$, which is the result of shrinking S k times and then reexpanding it k times. As mentioned in Section 11.2.1c, this is always contained in S. Let C be any connected component of the difference set $S - (S^{(-k)})^{(k)}$. Since C vanishes under k steps of shrinking, its thickness is at most $2k$; thus if its area is large relative to k^2, it is elongated. By doing this analysis for various values of k, we can detect elongated parts of S that have various thicknesses. A simple example, using only $k = 1$, is shown in Fig. 24.

The methods described in this subsection are designed for use with the array representation of S. They are especially efficient on a cellular array computer, but can also be implemented on a conventional computer, as discussed in Section 11.2.1c. Elongatedness is relatively easy to detect using maximal-block representations, e.g., S must be elongated if the number of MAT blocks is high and their radii are small, or the number of black quadtree leaves is large and their sizes are small. It is harder to detect from a border representation.

c. *Convexity*

In this and the next two subsections we discuss properties of S that are defined in terms of the intersections of straight lines with S.

In the real plane, S is called *convex* if any straight line meets S at most once, i.e., in only one run of points. Evidently, a convex set must be connected and can have no holes; and an arc can be convex only if it is a straight-line segment.

It is easily seen that each of the following properties is equivalent to convexity:

(1) For any points P, Q of S, the straight-line segment from P to Q lies in S.
(2) For any points $P = (x, y)$, $Q = (u, v)$ of S, the midpoint $((x + u)/2, (y + v)/2)$ of P and Q lies in S.

In fact, it suffices, in these definitions, to assume that P and Q are border points of S.

We can use analogous definitions for digital pictures, but we must be careful to allow for quantization effects. The digital straight-line segment from P to Q is not always uniquely defined (see Section 11.3.3b); we might require, e.g., that at least one of the possible segments lie entirely in S. The midpoint of P and Q may not have integer coordinates; here we might require that for at least one rounding of each half-integer coordinate, either up or down, the result lies in S.

It should be pointed out that even if S is digitally convex [e.g., it has the digital version of property (2)], t may not be the digitization of any convex object. (Conversely, however, if S is such a digitization it can be shown [26] that S does have the digital property (2).) Note that any S can be the digitization of a concave object, e.g., one having tiny concavities that are missed by the sampling process.

There are various ways of subdividing a given S into convex pieces; for a detailed treatment of this subject see [46], Chapter 9. A useful heuristic is to make cuts in S that join deepest points of concavities (e.g., concave corners on the border of S).

Convexity is most readily detected from the border representation of S; in fact, we can define S to be convex if the curvature of its border never changes sign. (Compare the remarks in Section 11.3.3c.) It is much harder to detect convexity from the maximal block representations. Some methods of determining convexity from the array representation are described in the next subsection.

d. *The convex hull*

In the real plane, there is a smallest convex set S_H containing any given set S. (*Proof*: Readily, any intersection of convex sets is convex; in particular, the intersection of all the convex sets containing S is convex.) S_H is called the *convex hull* of S; it is schematically illustrated in Fig. 25. It can be shown that S_H is the intersection of all the half-planes containing S; and if S is connected, S_H is the union of all the line segments whose end points are in S.

Clearly S is convex iff $S_H = S$. Thus one way to decide whether S is convex is to construct its convex hull and see whether it properly contains S. In any

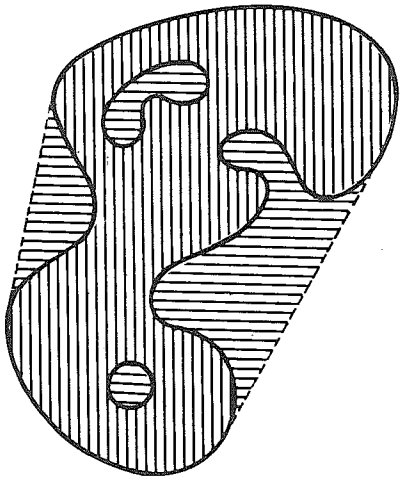


Fig. 25 A set (vertical shading) and its convex hull (includes the horizontally shaded parts).

case, we can define the *concavities* of S as the connected components of the difference set $S_H - S$.

The half-plane definition leads to the following construction for S_H [62], which can also be used to construct (and define) a convex hull in the digital case. Let P_1 be the leftmost of the uppermost points of S, and let L_1 be the horizontal line through P_1. Rotate L_1 counterclockwise about P_1 until it hits S; call the resulting rotated line L_2, and let P_2 be the point of S farthest from P_1 along L_2. Rotate L_2 counterclockwise about P_2 until it hits S; let L_3 be this rotated line, and let P_3 be the point farthest from P_2 along L_3. It is not hard to show that when this process is repeated, we eventually have $P_n = P_1$ and $L_n = L_1$. The polygon whose vertices are $P_1, \ldots, P_{n-1}$ is then S_H. Note that the L's bound half-planes that just contain S. Another characterization of the P's is as follows: For any border points P, Q of S, let $\bar{S}_{PQ}$ be the part of $\bar{S}$ surrounded by S and by the line segment PQ. Then $\bar{S}_{PQ}$ is maximal (i.e., is not contained in any other $\bar{S}_{P'Q'}$) iff P and Q are two consecutive P_i's (modulo $n - 1$).

The union of line segments definition leads [4] to a digital convex hull construction (and definition) that is more appropriate for a cellular array computer. Let S_θ denote the result of "smearing" S in direction θ, i.e., it is the union of all possible shifts of S by amounts 0, 1, 2, ... in direction θ; we need not use shifts greater than the diameter of S. Then $S_\theta \cap S_{\theta+\pi}$ is the set of points that have points of S on both sides of them in direction θ, i.e., these are just the points that are on line segments of slope θ between two points of S.

Thus $\bigcup_\theta (S_\theta \cap S_{\theta+\pi}) = S_H$, if S is connected. We cannot use just a few directions θ in this construction; for example,

1

1 1

1 1 1 1 1 1

contains every line segment between a pair of its points whose slope is a multiple of 45°, but it is not convex.

Certain simple properties of S_H can be estimated without actually constructing it. For example, it can be shown [45], using methods of *integral geometry*, that the expected number of times that a random line meets S is proportional to the perimeter of S_H; thus we might estimate this perimeter by drawing a large number of random lines and counting intersections. Incidentally, the expected length of intersection of a random line with S (i.e., the sum of the lengths of the runs in which the line meets S) is proportional to the area of S.

e. Generalizations of convexity

If S is in a known orientation, or if its orientation has been normalized, the intersections with S of lines in specific directions may provide useful descriptive properties of S. For example, we call S *row convex* if every row of the picture meets it only once; *column convex* is defined analogously. The S shown in the preceding subsection is both row and column convex, as well as diagonally convex for both diagonal directions.

These concepts can also be used to define new subsets in terms of S. For example, the "row convex hull" of a connected S might be defined as the union of all horizontal line segments whose end points are in S, i.e., as $S_0 \cap S_\pi$; and similarly for other directions. Analogously, the *shadow* of S from direction θ is the set of points from which a half-line in direction θ meets S; these are the points that would be in shadow if S were illuminated by a parallel beam of light in direction $\theta + \pi$. [22]. More generally, one can define sets of points from which the half-line in direction θ meets S a certain number of times, or in runs of a certain total length, etc.

Similar concepts can be defined if we use families of lines emanating from a given point, rather than in a given direction. In the real plane, for any point P of S, we call S *starshaped from* P if every line through P meets S exactly once. [Equivalently: (1) for all points Q of S, the straight-line segment from P to Q lies in S; (2) for all such Q, the midpoint of P and Q lies in S.] Readily, a starshaped S must be connected and can have no holes, but it need not be convex (a star is starshaped from its center). Evidently S is convex iff it is

starshaped from every one of its points. In general, the set of points of S from which all of S is visible is called the *kernel* of S; note that it may be empty.

If S is starshaped from P, every point of S, and in particular every point of the border of S, is visible from P; thus this border has a single-valued polar equation $r = f(\theta)$ when we take the origin at P. If S is convex, this is true for any $P \in S$. For arbitrary sets, the visible part of S may vary from point to point. If [11] we associate the area (or some other property) of this visible part with each point of S, we can segment S into parts by classifying the points on the basis of these values; this is analogous to segmenting S on the basis of the distances of its points from $\bar{S}$.

11.4 BIBLIOGRAPHICAL NOTES

Only a few general references are mentioned here. Selected references on particular methods were given in the text; no attempt has been made to give a complete bibliography.

The MAT representation is due to Blum [6]; on its theory in the digital case see Rosenfeld and Pfaltz [58] and Mott-Smith [40]. The quadtree representation was introduced by Klinger [30, 31]; details of the algorithms for converting between quadtrees and other representations, and for computing geometric properties from quadtrees, can be found in a series of papers by Samet *et al.* [14, 16, 63–68, 72] (see also Alexandridis and Klinger [2] and Hunter and Steiglitz [27, 28]). Chain codes and their generalizations have been extensively studied by Freeman, who reviews them in [19]. On the representation of borders by sequences of circular arcs defined by successive triples of border points see Shapiro and Lipkin [71].

The theory of digital connectedness was developed by Rosenfeld, and is reviewed in [55]. On the theory of digital distance see Rosenfeld and Pfaltz [59]; on early work using shrinking and expanding for shape analysis see Moore [39].

Digital convexity has been extensively studied by Sklansky (e.g., [73]; see also [74] on a parallel approach to filling concavities by iteratively adding "concavity points" to S.) Another approach to convex hull construction based on a succession of inscribed polygons is described in [21]. Many fast algorithms have been proposed for solving various geometric problems involving intersections (e.g., constructing the convex hull of a set of points by intersecting a set of half-planes) and distances (e.g., finding pairs of points that are nearest neighbors).

Shape analysis techniques are reviewed by Pavlidis [47, 48]; for an extensive review of shape perception see Zusne [78].

REFERENCES

1. N. Ahuja, L. S. Davis, D. L. Milgram, and A. Rosenfeld, Piecewise approximation of pictures using maximal neighborhoods, *IEEE Trans. Comput.* **27**, 1978, 375–379.
2. N. Alexandridis and A. Klinger, Picture decomposition, tree data-structures, and identifying directional symmetries as node combinations, *Comput. Graphics Image Processing* **8**, 1978, 43–77.
3. C. Arcelli, A condition for digital points removal, *Signal Processing* **1**, 1979, 283–285.
4. C. Arcelli and S. Levialdi, Concavity extraction by parallel processing, *IEEE Trans. Systems Man Cybernet.* **1**, 1971, 394–396.
5. D. H. Ballard, Strip trees: a hierarchical representation for map features, *Proc. IEEE Conf. on Pattern Recognition and Image Processing* August 1979, 278–285.
6. H. Blum, A transformation for extracting new descriptors of shape, *in* "Models for the Perception of Speech and Visual Form" (W. Wathen-Dunn, ed.), pp. 362–380. MIT Press, Cambridge, Massachusetts, 1967.
7. H. Blum and R. Nagel, Shape description using weighted symmetric axis features, *Pattern Recognition* **10**, 1978, 167–180.
8. R. L. T. Cederberg, Chain-link coding and segmentation for raster scan devices, *Comput. Graphics Image Processing* **10**, 1979, 224–234.
9. L. S. Davis, Understanding shape: angles and sides, *IEEE Trans. Comput.* **26**, 1977, 236–242.
10. L. S. Davis, Understanding shape: symmetry, *IEEE Trans. Systems Man Cybernet.* **7**, 1977, 204–212.
11. L. S. Davis and M. L. Benedikt, Computational models of space: isovists and isovist fields, *Comput. Graphics Image Processing* **11**, 1979, 49–72.
12. L. S. Davis and A. Rosenfeld, Curve segmentation by relaxation labeling, *IEEE Trans. Comput.* **26**, 1977, 1053–1057.
13. I. De Lotto, Un inseguitore di contorno, *Alta Frequenza* **32**, 1963, 703–705.
14. C. R. Dyer, Computing the Euler number of an image from its quadtree, *Comput. Graphics Image Processing* **13**, 1980, 270–276.
15. C. R. Dyer and A. Rosenfeld, Thinning algorithms for grayscale pictures, *IEEE Trans. Pattern Anal. Machine Intelligence* **1**, 1979, 88–89.
16. C. R. Dyer, A. Rosenfeld, and H. Samet, Region representation: boundary codes from quadtrees. *Comm. ACM* **23**, 1980, 171–179.
17. R. B. Eberlein, An iterative gradient edge detection algorithm, *Comput. Graphics Image Processing* **5**, 1976, 245–253.
18. T. J. Ellis, D. Proffitt, D. Rosen, and W. Rutkowski, Measurement of the lengths of digitized curved lines. *Comput. Graphics Image Processing* **10**, 1979, 333–347.
19. H. Freeman, Computer processing of line-drawing images, *Comput. Surveys* **6**, 1974, 57–97.
20. H. Freeman and L. S. Davis, A corner-finding algorithm for chain-coded curves, *IEEE Trans. Comput.* **26**, 1977, 297–303.
21. H. Freeman and R. Shapira, Determining the minimum-area encasing rectangle for an arbitrary closed curve, *Comm. ACM* **18**, 1975, 409–413.
22. H. A. Glucksman, A parapropagation pattern classifier, *IEEE Trans. Electron. Comput.* **14**, 1965, 434–443.
23. S. B. Gray, Local properties of binary images in two dimensions, *IEEE Trans. Comput.* **20**, 1971, 551–561.
24. R. L. Grimsdale, F. H. Sumner, C. J. Tunis, and T. Kilburn, A system for the automatic recognition of patterns, *Proc. IEEE* **106B**, 1959, 210–221.

25. R. M. Haralick, A measure for cirularity of digital figures, *IEEE Trans. Systems Man Cybernet.* **4**, 1974, 394–396.
26. L. Hodes, Discrete approximation of continuous convex blobs, *SIAM J. Appl. Math.* **19**, 1970, 477–485.
27. G. M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Trans. Pattern Anal. Machine Intelligence* **1**, 1979, 145–153.
28. G. M. Hunter and K. Steiglitz, Linear transformation of pictures represented by quad trees, *Comput. Graphics Image Processing* **10**, 1979, 289–296.
29. R. A. Kirsch, Resynthesis of biological images from tree-structured decomposition data, *in* "Graphic Languages" (F. Nake and A. Rosenfeld, eds.), pp. 1–19. North-Holland Publ., Amsterdam, 1972.
30. A. Klinger, Data structures and pattern recognition, *Proc. Internat. Joint Conf. Pattern Recognition, 1st* 1973, 497–498.
31. A. Klinger and C. R. Dyer, Experiments on picture representation using regular decomposition, *Comput. Graphics Image Processing* **5**, 1976, 68–105.
32. R. S. Ledley, J. Jacobsen, and M. Belson, BUGSYS: a programming system for picture processing—not for debugging, *Comm. ACM* **9**, 1966, 79–84.
33. G. Levi and U. Montanari, A grey-weighted skeleton, *Informat. Control* **17**, 1970, 62–91.
34. S. Levialdi, On shrinking binary picture patterns, *Comm. ACM* **15**, 1972, 7–10.
35. B. B. Mandelbrot, "Fractals: Form, Chance, and Dimension." Freeman, San Francisco, California, 1977.
36. S. J. Mason and J. K. Clemens, Character recognition in an experimental reading machine for the blind, *in* "Recognizing Patterns" (P. A. Kolers and M. Eden, eds.), pp. 156–167. MIT Press, Cambridge, Massachusetts, 1968.
37. M. L. Minsky and S. Papert, "Perceptrons—An Introduction to Computational Geometry." MIT Press, Cambridge, Massachusetts, 1969.
38. U. Montanari, Continuous skeletons from digitized images, *J. ACM* **16**, 1969, 534–549.
39. G. A. Moore, Automatic scanning and computer processes for the quantitative analysis of micrographs and equivalent subjects, *in* "Pictorial Pattern Recognition" (G. C. Cheng *et al.*, eds.), pp. 275–326. Thompson, Washington, D.C., 1968.
40. J. C. Mott-Smith, Medial axis transformations, *in* Picture Processing and Psychopictorics" (B. S. Lipkin and A. Rosenfeld, eds.), pp. 267–283. Academic Press, New York, 1970.
41. Y. Nakagawa and A. Rosenfeld, A note on the use of local min and max operations in digital picture processing. *IEEE Trans. Systems Man Cybernet.* **8**, 1978, 623–635.
42. R. Nevatia and T. O. Binford, Description and recognition of curved objects, *Artificial Intelligence* **8**, 1977, 77–98.
43. S. Nishikawa, R. J. Massa, and J. C. Mott-Smith, Area properties of television pictures, *IEEE Trans. Informat. Theory* **11**, 1965, 348–352.
44. S. Nordbeck and B. Rystedt, Computer cartography—point-in-polygon programs, *BIT* **7**, 1967, 30–64.
45. A. B. J. Novikoff, Integral geometry as a tool in pattern perception, *in* "Principles of Self-Organization" (H. von Foerster and G. W. Zopf, eds.), pp. 347–368. Pergamon, Oxford, 1962.
46. T. Pavlidis, "Structural Pattern Recognition." Springer, New York, 1977.
47. T. Pavlidis, A review of algorithms for shape analysis, *Comput. Graphics Image Processing* **7**, 1978, 243–258.
48. T. Pavlidis, Algorithms for shape analysis of contours and waveforms, *Proc. Internat. Joint Conf. on Pattern Recognition, 4th* 1978, 70–85.
49. J. L. Pfaltz and A. Rosenfeld, Computer representation of planar regions by their skeletons, *Comm. ACM* **10**, 1967, 119–122, 125.

50. D. Proffitt and D. Rosen, Metrication errors and coding efficiency of chain-encoding schemes for the representation of lines and edges, *Comput. Graphics Image Processing* **10**, 1979, 318–332.
51. C. V. Kameswara Rao, B. Prasada, and K. R. Sarma, A parallel shrinking algorithm for binary patterns, *Comput. Graphics Image Processing* **5**, 1976, 265–270.
52. A. Rosenfeld, Compact figures in digital pictures, *IEEE Trans. Systems Man Cybernet.* **4**, 1974, 211–223.
53. A. Rosenfeld, Digital straight line segments, *IEEE Trans. Comput.* **23**, 1974, 1264–1269.
54. A. Rosenfeld, A characterization of parallel thinning algorithms, *Informat. Control* **29**, 1975, 286–291.
55. A. Rosenfeld, "Picture Languages: Formal Models for Picture Recognition," Chapter 2. Academic Press, New York, 1979.
56. A. Rosenfeld and L. S. Davis, A note on thinning, *IEEE Trans. Systems Man Cybernet.* **6**, 1976, 226–228.
57. A. Rosenfeld and E. Johnston, Angle detection on digital curves, *IEEE Trans. Comput.* **22**, 1973, 875–878.
58. A. Rosenfeld and J. L. Pfaltz, Sequential operations in digital picture processing, *J. ACM* **13**, 1966, 471–494.
59. A. Rosenfeld and J. L. Pfaltz, Distance functions on digital pictures, *Pattern Recognition* **1**, 1968, 33–61.
60. A. Rosenfeld and J. Weszka, An improved method of angle detection on digital curves, *IEEE Trans. Comput.* **24**, 1975, 940–941.
61. D. Rutovitz, Data structures for operations on digital images, *in* "Pictorial Pattern Recognition" (G. C. Cheng *et al.*, eds.), pp. 105–133. Thompson, Washington, D.C., 1968.
62. D. Rutovitz, An algorithm for in-line generation of a convex cover, *Comput. Graphics Image Processing* **4**, 1975, 74–78.
63. H. Samet, Region representation: quadtree-to-raster conversion, Computer Science Center TR-768, Univ. of Maryland, College Park, Maryland, June 1979.
64. H. Samet, Region representation: quadtrees from boundary codes, *Comm. ACM* **23**, 1980, 163–170.
65. H. Samet, Region representation: quadtrees from binary arrays, *Comput. Graphics Image Processing* **13**, 1980, 88–93.
66. H. Samet, Connected component labelling using quadtrees, *J. ACM* **28**, 1981, 487–501.
67. H. Samet, An algorithm for converting rasters to quadtrees, *IEEE Trans. Pattern Anal. Machine Intelligence* **3**, 1981, 93–95.
68. H. Samet, Computing perimeters of images represented by quadtrees, *IEEE Trans. Pattern Anal. Machine Intelligence* **3**, 1981, 683–687.
69. P. V. Sankar and E. V. Krishnamurthy, On the compactness of subsets of digital pictures, *Comput. Graphics Image Processing* **8**, 1978, 136–143.
70. J. Serra, Theoretical basis of the Leitz texture analysis system, *Leitz Sci. Tech. Informat. Suppl.* **1** (4), 1974, 125–136.
71. B. Shapiro and L. Lipkin, The circle transform, an articulable shape descriptor, *Comput. Biomed. Res.* **10**, 1977, 511–528.
72. M. Shneier, Calculations of geometric properties using quadtrees, *Comput. Graphics Image Processing* **16**, 1981, 296–302.
73. J. Sklansky, Recognition of convex blobs, *Pattern Recognition* **2**, 1970, 3–10.
74. J. Sklansky, L. P. Cordella, and S. Levialdi, Parallel detection of concavities in cellular blobs, *IEEE Trans. Comput.* **25**, 1976, 187–196.
75. J. P. Strong III and A. Rosenfeld, A region coloring technique for scene analysis, *Comm. ACM* **16**, 1973, 237–246.

76. P. H. Winston, Learning structural descriptions from examples, *in* "The Psychology of Computer Vision" (P. H. Winston, ed.), pp. 157–209. McGraw-Hill, New York, 1975.
77. L. T. Young, J. E. Walker, and J. E. Bowie, An analysis technique for biological shape, *Informat. Control* **25**, 1974, 357–370.
78. L. Zusne, "Visual Perception of Form." Academic Press, New York, 1970.