2.2

Basic Binary Image Processing

Alan C. Bovik
The University of Texas at Austin

1 Introduction

In this second chapter on basic methods, we explain and demonstrate fundamental tools for the processing of *binary* digital images. Binary image processing is of special interest, since an image in binary format can be processed using very fast logical (Boolean) operators. Often, a binary image has been obtained by abstracting essential information from a gray-level image, such as object location, object boundaries, or the presence or absence of some image property.

As seen in the previous two chapters, a digital image is an array of numbers or sampled image intensities. Each gray level is quantized or assigned one of a finite set of numbers represented by B bits. In a binary image, only one bit is assigned to each pixel: $B = 1$ implying two possible gray-level values, 0 and 1. These two values are usually interpreted as Boolean, hence each pixel can take on the logical values "0" or "1", or equivalently, "true" or "false." For example, these values might indicate the absence or presence of some image property in an associated gray-level image of the same size, where "1" at a given coordinate indicates the presence of the property at that coordinate in the gray-level image, and "0" otherwise. This image property is quite commonly a sufficiently high or low intensity (brightness), although more abstract properties, such as the presence or absence of certain objects, or smoothness/nonsmoothness, etc., might be indicated.

Since most image display systems and software assume images of eight or more bits per pixel, the question arises at to how binary images are displayed. Usually, they are displayed using the two extreme gray tones, black and white, which are ordinarily represented by 0 and 255, respectively, in a gray-scale display environment, as depicted in Fig. 1. There is no established convention for the Boolean values that are assigned to "black" and to "white." In this chapter we will uniformly use "1" to represent "black" (displayed as gray level 0) and "0" to represent "white" (displayed as gray level 255). However, the assignments are quite commonly reversed, and it is important to note that the Boolean values "0" and "1" have no physical significance other than what the user assigns to them.

Binary images arise in a number of ways. Usually, they are created from gray-level images for simplified processing or for printing (see Chapter 8.1 on image halftoning). However, certain types of sensors directly deliver a binary image output. Such devices are usually associated with printed, handwritten or line drawing images, with the input signal being entered by hand on a pressure sensitive tablet, a resistive pad, or a light pen.

In such a device, the (binary) image is first initialized prior to image acquisition:

$$g(\mathbf{n}) = \text{"0"} \tag{1}$$

at all coordinates $\mathbf{n}$. When pressure, a change of resistance, or light is sensed at some image coordinate $\mathbf{n}_0$, then the image is assigned the value "1":

$$g(\mathbf{n}_0) = \text{"1"} \tag{2}$$

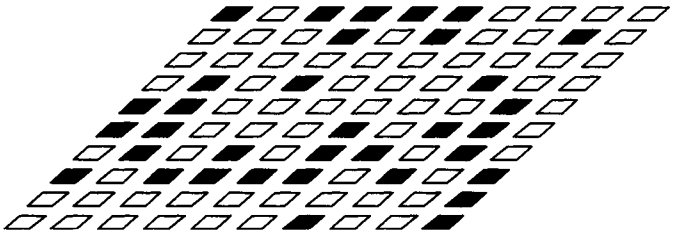


FIGURE 1 A 10×10 binary image.

This continues until the user completes the drawing, as depicted in Fig. 2. These simple devices are quite useful for entering engineering drawings, handprinted characters, or other binary graphics in a binary image format.

2 Image Thresholding

Usually, a binary image is obtained from a gray-level image by some process of information abstraction. The advantage of the B-fold reduction in the required image storage space is offset by what can be a significant loss of information in the resulting binary image. However, if the process is accomplished with care, then a simple abstraction of information can be obtained that can enhance subsequent processing, analysis, or interpretation of the image.

The simplest such abstraction is the process of *image thresholding*, which can be thought of as an extreme form of gray-level quantization. Suppose that a gray-level image f can take K possible gray levels $0, 1, 2, \ldots, K-1$. Define an integer threshold, T, that lies in the gray-scale range: $T \in \{0, 1, 2, \ldots, K-1\}$. The process of thresholding is a process of simple comparison: each pixel value in f is compared to T. Based on this comparison, a binary decision is made that defines the value of the corresponding pixel in an output binary image g:

$$g(\mathbf{n}) = \begin{cases} \text{“0”} & \text{if } f(\mathbf{n}) \geq T \\ \text{“1”} & \text{if } f(\mathbf{n}) < T \end{cases} \tag{3}$$

Of course, the threshold T that is used is of critical importance, since it controls the particular abstraction of information that is obtained. Indeed, different thresholds can produce different valuable abstractions of the image. Other thresholds may produce little valuable information at all. It is instructive to observe the result of thresholding an image at many different levels in sequence. Figure 3 depicts the image “mandrill” (Figure 8 of Chapter 1.1) thresholded at four different levels. Each produces different information, or in the case of Figs. 3(a) and 3(d), very little useful information. Among these, Fig. 3(c) probably contains the most visual information, although it is far from ideal. The four threshold values (50, 100, 150, 200) were chosen without using any visual criterion.

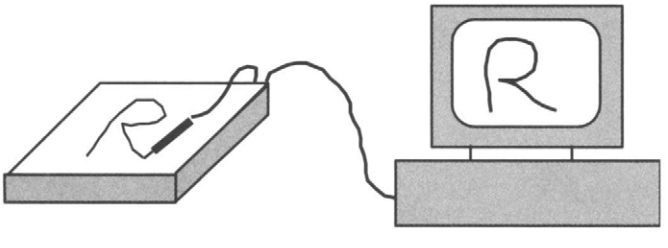


FIGURE 2 Simple binary image device.

As will be seen, image thresholding can often produce a binary image result that is quite useful for simplified processing, interpretation or display. However, some gray-level images do not lead to any interesting binary result regardless of the chosen threshold T.

Several questions arise: given a gray-level image, how does one decide whether binarization of the image by gray-level thresholding will produce a useful result? Can this be decided automatically by a computer algorithm? Assuming that thresholding is likely to be successful, how does one decide on a threshold level T? These are apparently simple questions pertaining to a very simple operation. However, the answers to these questions turn out to be quite difficult to answer in the general case. In other cases, the answer is simpler. In all cases, however, the basic tool for understanding the process of image thresholding is the image histogram, which was defined and studied in Chapter 2.1.

Thresholding is most commonly and effectively applied to images that can be characterized as having *bimodal histograms*. Figure 4 depicts two hypothetical image histograms. The one on the left has two clear modes; the one at the right either has a single mode, or two heavily-overlapping, poorly separated modes.

Bimodal histograms are often (but not always!) associated with images that contain objects and background having significantly different average brightness. This may imply bright objects on a dark background, or dark objects on a bright background. The goal, in many applications, is to separate the objects from the background, and to label them as object or as background. If the image histogram contains well separated modes associated with object and with background, then thresholding can be the means for achieving this separation. Practical examples of gray-level images with well-separated bimodal histograms are not hard to find. For example, an image of machine-printed type (like that being currently read), or of handprinted characters, will have a very distinctive separation between object and background. Examples abound in biomedical applications, where it is often possible to control the lighting of objects and background. Standard bright-field microscope images of single or multiple cells (micrographs) typically contain bright objects against a darker background. In many industry applications, it is also possible to control the relative brightness of objects of

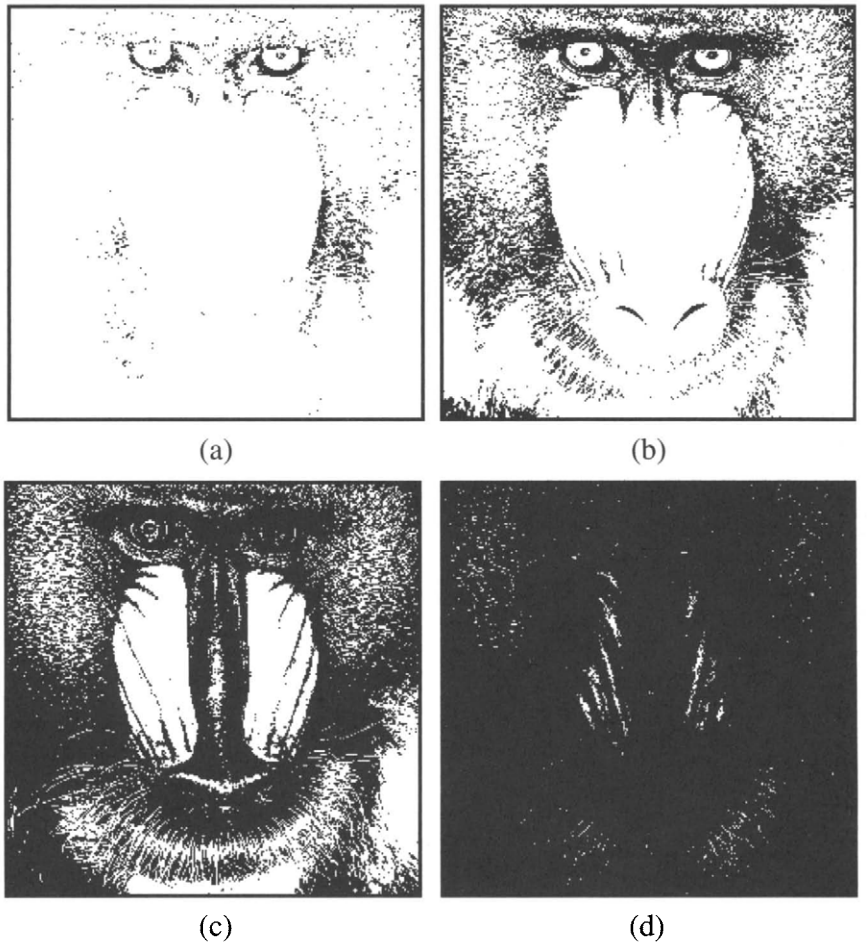


FIGURE 3 Image "mandrill" thresholded at gray-levels (a) 50; (b) 100; (c) 150; and (d) 150.

interest and the backgrounds they are set against. For example, machine parts that are being imaged (perhaps in an automated inspection application) may be placed on a mechanical conveyor that has substantially different reflectance properties than the objects.

Given an image with a bimodal histogram, a general strategy for thresholding is to place the threshold T between the image modes, as depicted in Fig. 4(a). Many "optimal" strategies have been suggested for deciding the exact placement of the threshold between the peaks. Most of these are based on an assumed statistical model for the histogram, and by posing the decision of labeling a given pixel as "object" versus "background" as a statistical inference problem. In the simplest version, two hypotheses are posed:

H_0: The pixel belongs to gray level Population 0

H_1: The pixel belongs to gray level Population 1

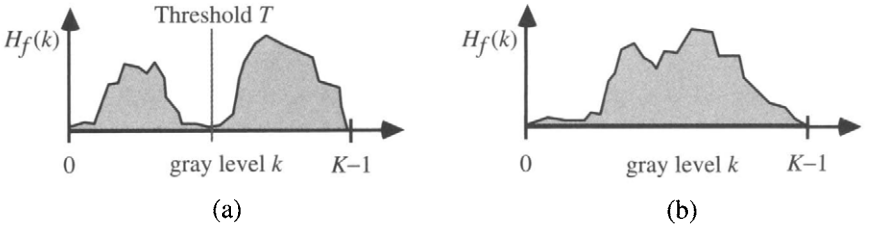


FIGURE 4 Hypothetical histograms. (a) Well-separated modes. (b) Poorly separated or indistinct modes.

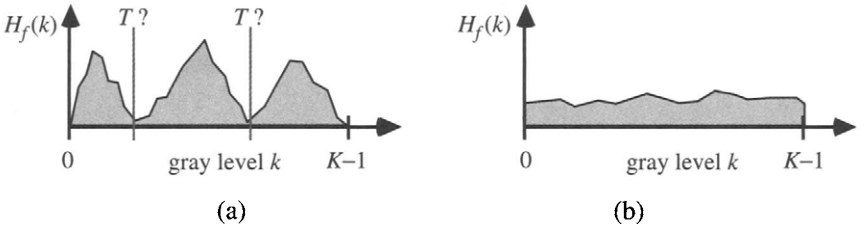


FIGURE 5 Hypothetical histograms. (a) Multimodal histogram, showing difficulty of threshold selection. (b) Non-modal histogram, for which threshold selection is quite difficult or impossible.

where pixels from population 0 and 1 have conditional probability density functions (pdfs) $p_f(a|H_0)$ and $p_f(a|H_1)$, respectively, under the two hypotheses. If it is also known (or estimated) that H_0 is true with probability p_0 and that H_1 is true with probability p_1 ($p_0 + p_1 = 1$), then the decision may be cast as a likelihood ratio test. If an observed pixel has gray-level $f(\mathbf{n}) = k$, then the decision may be rendered according to

$$\frac{p_f(k|H_1)}{p_f(k|H_0)} \begin{matrix} H_1 \\ > \\ < \\ H_0 \end{matrix} \frac{p_0}{p_1} \tag{4}$$

The decision whether to assign logical "0" or "1" to a pixel can thus be regarded as applying a simple statistical test to each pixel. In (4), the conditional pdfs may be taken as the modes of a bimodal histogram. Algorithmically, this means that they must be fit to the histogram using some criterion, such as least-squares. This is usually quite difficult, since it must be decided that there are indeed two separate modes, the locations (centers) and widths of the modes must be estimated, and a model for the shape of the modes must be assumed. Depending on the assumed shape of the modes (in a given application, the shape might be predictable), specific probability models might be applied, e.g., the modes might be taken to have the shape of Gaussian pdfs (Chapter 4.5). The prior probabilities p_0 and p_1 are often easier to model, since in many applications the relative areas of object and background can be estimated or given reasonable values based on empirical observations.

A likelihood ratio test such as (4) will place the image threshold T somewhere between the two modes of the image histogram. Unfortunately, any simple statistical model of the image does not account for such important factors as object/background continuity, visual appearance to a human observer, non-uniform illumination or surface reflectance effects, and so on. Hence, with rare exceptions, a statistical approach such as (4) will not produce as good a result as would a human decision-maker making a manual threshold selection.

Placing the threshold T between two obvious modes of a histogram may yield acceptable results, as depicted in Fig. 4(*a*). The problem is significantly complicated, however, if the image contains multiple distinct modes or if the image is non-modal or level. Multi-modal histograms can occur when the image contains multiple objects of different average brightness on a uniform background. In such cases, simple thresholding will exclude some objects (Fig. 5). Non-modal or flat histograms usually imply more complex images, containing significant gray-level variation, detail, non-uniform lighting or reflection, etc. (Fig. 5). Such images are often not amenable to a simple thresholding process, especially if the goal is to achieve figure-ground separation. However, all of these comments are, at best, rules of thumb. An image with a bimodal histogram might not yield good results when thresholded at any level, while an image with a perfectly flat histogram might yield an ideal result. It is a good mental exercise to consider when these latter cases might occur.

Figures 6–8 shows several images, their histograms, and the thresholded image results. In Fig. 6, a good threshold level for the micrograph of the cellular specimens was taken to be $T = 180$. This falls between the two large modes of the histogram (there are many smaller modes) and was deemed to be visually optimal by one user. In the binarized image, the individual cells are not perfectly separated from the background. The reason for this is that the illuminated cells have non-uniform brightness profiles, being much brighter towards the centers. Taking the threshold higher ($T = 200$), however, does not lead to improved results, since the bright background then begins to fall below threshold.

Figure 7 depicts a *negative* (for better visualization) of a digitized mammogram. Mammography is the key diagnostic tool for the detection of breast cancer, and in the future, digital tools for mammographic imaging and analysis. The image again shows two strong modes, with several smaller modes. The first threshold chosen ($T = 190$) was selected at the minimum point between the large modes. The resulting binary image has the nice result of separating the region of the breast from the background. However, radiologists are often interested in the detailed structure of the breast and in the brightest (darkest in the negative) areas which might indicate tumors or microcalcifications. Figure 7(d) shows the result of thresholding at the lower level of 125 (higher level in the positive image), successfully isolating much of the interesting structure.

Generally, the best binarization results via thresholding are obtained by direct human operator intervention. Indeed, most

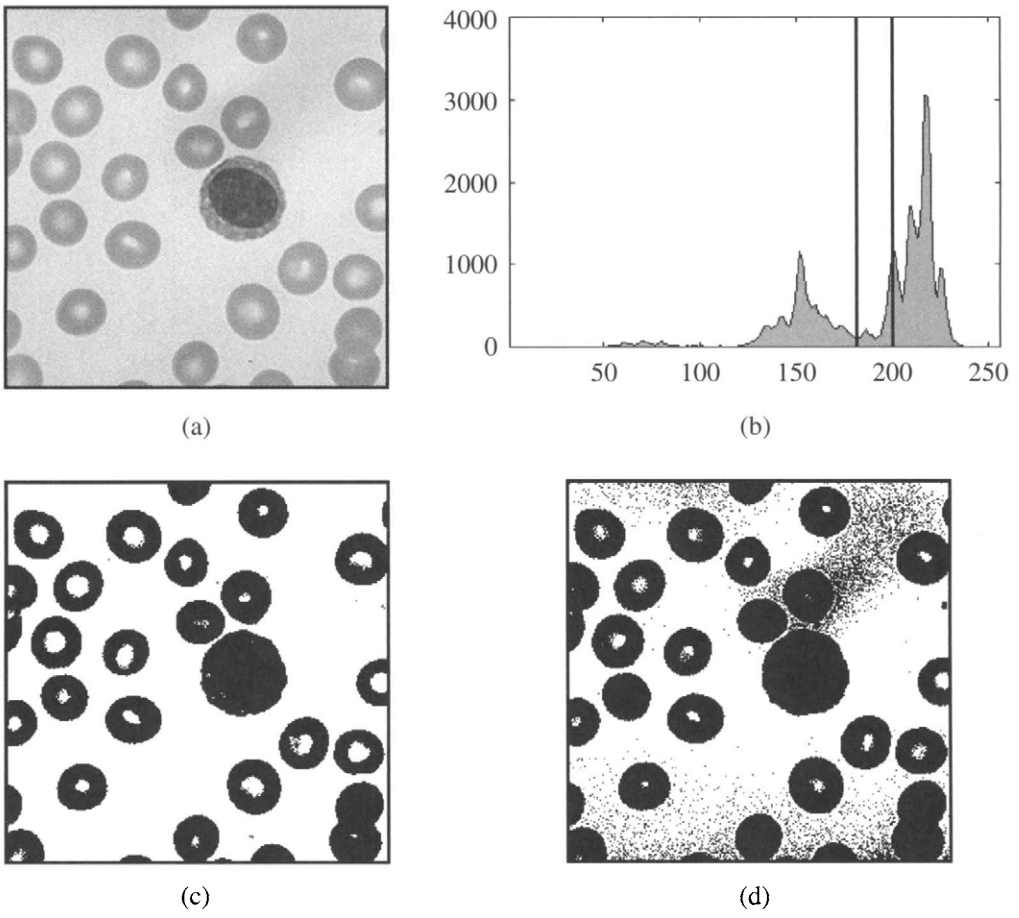


FIGURE 6 Binarization of "micrograph." (a) Original; (b) histogram showing two threshold locations (180 and 200); (c) and (d) resulting binarized images.

general-purpose image processing environments have thresholding routines that allow user interaction. However, even with a human picking a visually "optimal" value of T, thresholding rarely gives "perfect" results. There is nearly always some misclassification of object as background, and vice-versa. For example in the image "micrograph," no value of T is able to successfully extract the objects from the background; instead, most of the objects have "holes" in them, and there is a sprinkling of black pixels in the background as well.

Because of these limitations of the thresholding process, it is usually necessary to apply some kind of *region correction* algorithms to the binarized image. The goal of such algorithms is to correct the misclassification errors that occur. This requires identifying misclassified background points as object points, and vice-versa. These operations are usually applied directly to the binary images, although it is possible to augment the process by also incorporating information from the original gray-scale image. Much of the remainder of this chapter will be devoted to algorithms for region correction of thresholded binary images.

3 Region Labeling

A simple but powerful tool for identifying and labeling the various objects in a binary image is a process called *region labeling, blob coloring,* or *connected component identification.* It is useful since once they are individually labeled, the objects can be separately manipulated, displayed or modified. For example, the term "blob coloring" refers to the possibility of displaying each object with a different identifying color, once labeled.

Region labeling seeks to identify connected groups of pixels in a binary image f that all have the same binary value. The simplest such algorithm accomplishes this by scanning the entire image (left-to-right, top-to-bottom), searching for occurrences of pixels of the same binary value and connected along the horizontal or vertical directions. The algorithm can be made slightly more complex by also searching for diagonal connections, but this is usually unnecessary. A record of connected pixels groups is maintained in a separate label array r having the same dimensions as f, as the image is

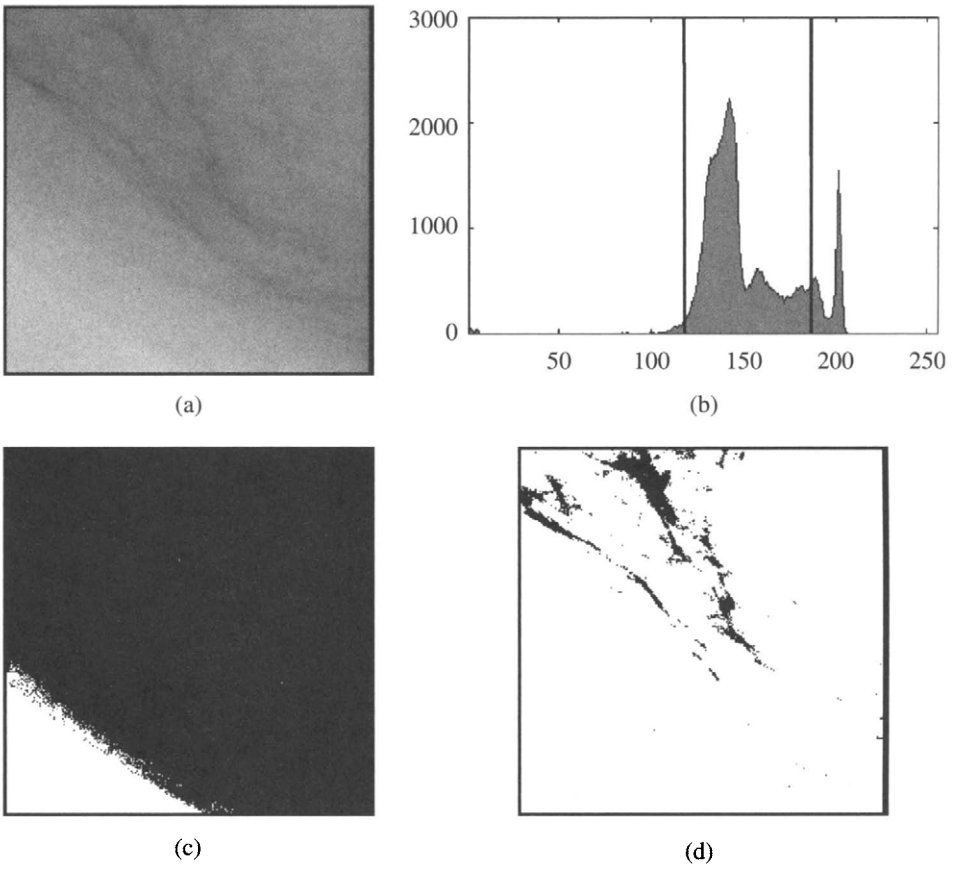


FIGURE 7 Binarization of "mammogram." (a) Original *negative* mammogram; (b) histogram showing two threshold locations (190 and 125); (c) and (d) resulting binarized images.

scanned. The following algorithm steps explain the process, where the region labels used are positive integers.

Region Labeling Algorithm

1. Given an $N \times M$ binary image f, initialize an associated $N \times M$ region label array: $r(\mathbf{n}) = \text{"0"}$ for all $\mathbf{n}$. Also initialize a region number counter: $k = 1$.
   Then, scanning the image from left-to-right and top-to-bottom, for every $\mathbf{n}$ do the following:
2. If $f(\mathbf{n}) = \text{"0"}$ then do nothing.
3. If $f(\mathbf{n}) = \text{"1"}$ and also $f(\mathbf{n} - (1,0)) = f(\mathbf{n} - (0,1)) = \text{"0"}$ (as depicted in Fig. 8(a)), then set $r(\mathbf{n}) = \text{"}k\text{"}$ and $k = k + 1$. In this case the left and upper neighbors of $f(\mathbf{n})$ do not belong to objects.
4. If $f(\mathbf{n}) = \text{"1"}$, $f(\mathbf{n} - (1,0)) = \text{"1"}$ and $f(\mathbf{n} - (0,1)) = \text{"0"}$ (Fig. 8(b)), then set $r(\mathbf{n}) = r(\mathbf{n} - (1,0))$. In this case the upper neighbor $f(\mathbf{n} - (1,0))$ belongs to the same object as $f(\mathbf{n})$.
5. If $f(\mathbf{n}) = \text{"1"}$, $f(\mathbf{n} - (1,0)) = \text{"0"}$ and $f(\mathbf{n} - (0,1)) = \text{"1"}$ (Fig. 8(c)), then set $r(\mathbf{n}) = r(\mathbf{n} - (0,1))$. In this case the left neighbor $f(\mathbf{n} - (0,1))$ belongs to the same object as $f(\mathbf{n})$.
6. If $f(\mathbf{n}) = \text{"1"}$, and $f(\mathbf{n} - (1,0)) = f(\mathbf{n} - (0,1)) = \text{"1"}$ (Fig. 8(d)), then set $r(\mathbf{n}) = r(\mathbf{n} - (0,1))$. If $r(\mathbf{n} - (0,1)) \neq r(\mathbf{n} - (1,0))$, then record the labels $r(\mathbf{n} - (0,1))$ and $r(\mathbf{n} - (1,0))$ as equivalent. In this case both the left and upper neighbors belong to the same object as $f(\mathbf{n})$, although they may have been labeled differently.

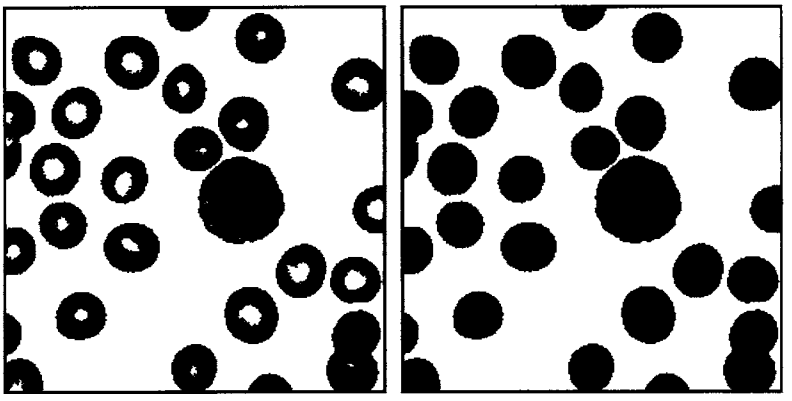
FIGURE 8 Pixel neighbor relationships used in a region labeling algorithm. In each of (a)–(d), $f(\mathbf{n})$ is the lower right pixel.

A simple application of region labeling is the measurement of object area. This can be accomplished by defining a vector $\mathbf{c}$ with elements $c(k)$ that are the pixel area (pixel count) of region k.

Region Counting Algorithm

Initialize $\mathbf{c} = \mathbf{0}$. For every $\mathbf{n}$ do the following:

1. If $f(\mathbf{n}) = \text{"0"}$ then do nothing.



(a) (b)

FIGURE 9 Result of applying the region labeling/counting/removal algorithms to (a) the binarized image in Fig. 6(c); (b) and then to the image in (b), but in polarity-reversed mode.

2. If $f(\mathbf{n}) = \text{"1"}$, then $c[r(\mathbf{n})] = c[r(\mathbf{n})] + 1$.

Another simple but powerful application of region labeling is the removal of minor regions or objects from a binary image. The way on which this is done depends on the application. It may be desired that only a single object should remain (generally, the largest object), or it may be desired that any object with a pixel area less than some minimum value should be deleted. A variation is that the minimum value is computed as a percentage of the largest object in the image. The following algorithm depicts the second possibility:

Minor Region Removal Algorithm

Assume a minimum allowable object size of S pixels. For every $\mathbf{n}$ do the following:

1. If $f(\mathbf{n}) = \text{"0"}$ then do nothing.
2. If $f(\mathbf{n}) = \text{"1"}$ and $c[r(\mathbf{n})] < S$, then set $g(\mathbf{n}) = \text{"0"}$.

Of course, all of the above algorithms can be operated in reverse polarity, by interchanging "0" for "1" and "1" for "0" everywhere.

An important application of region labeling/region counting/minor region removal is in the correction of thresholded binary images. Application of a binarizing threshold to a gray-level image inevitably produces an imperfect binary image, with such errors as extraneous objects or holes or holes in objects. These can arise from noise, unexpected objects (such as dust on a lens), and generally, non-uniformities in the surface reflectances and illuminations of the objects and background.

Figure 9 depicts the result of sequentially applying the region labeling/region counting/minor region removal algorithms to the binarized "micrograph" image in Fig. 6(c). The series of algorithms was first applied to the image 6(c) as above to remove extraneous small black objects, using a size threshold of 500 pixels as shown in Fig. 9(a). It was then applied again to this modified image, but in polarity reversed mode, to remove the many object holes, this time using a threshold of 1,000 pixels. The result shown in Fig. 9(b) is a dramatic improvement over the original binarized result, given that the goal was to achieve a clean separation of the objects in the image from the background.

4 Binary Image Morphology

We next turn to a much broader and more powerful class of binary image processing operations that collectively fall under the name *binary image morphology*. These are closely related to (in fact, are the same as in a mathematic sense) the gray-level morphologic operations described in Chapter 3.3. As the name indicates, these operators modify the *shapes* of the objects in an image.

Logical Operations

The morphologic operators are defined in terms of simple logical operations on local groups of pixels. The logical operators that are used are the simple NOT, AND, OR, and MAJ (majority) operators. Given a binary variable x, NOT(x) is its logical complement. Given a set of binary variables $x_1, \ldots, x_n$, the operation AND($x_1, \ldots, x_n$) returns value "1" if and only if $x_1 = \cdots = x_n = \text{"1"}$ and "0" otherwise. The operation OR($x_1, \ldots, x_n$) returns value "0" if and only if $x_1 = \cdots = x_n = \text{"0"}$ and "1" otherwise. Finally, if n is odd, the operation MAJ($x_1, \ldots, x_n$) returns value "1" if and only if a majority of ($x_1, \ldots, x_n$) equal "1" and "0" otherwise.

We observe in passing the DeMorgan's Laws for binary arithmetic, specifically:

$$\text{NOT}[\text{AND}(x_1, \ldots, x_n)] = \text{OR}[\text{NOT}(x_1), \ldots, \text{NOT}(x_n)] \quad (5)$$

$$\text{NOT}[\text{OR}(x_1, \ldots, x_n)] = \text{AND}[\text{NOT}(x_1), \ldots, \text{NOT}(x_n)] \quad (6)$$

which characterizes the duality of the basic logical operators AND and OR under complementation. However, note that

$$\text{NOT}[\text{MAJ}(x_1,\ldots,x_n)] = \text{MAJ}[\text{NOT}(x_1),\ldots,\text{NOT}(x_n)] \quad (7)$$

hence MAJ is its own dual under complementation.

Windows

As mentioned, morphologic operators change the shapes of objects using *local logical operations.* Since they are local operators, a formal methodology must be defined for making the operations occur on a local basis. The mechanism for doing this is the *window.*

A window defines a geometric rule according to which gray levels are collected from the vicinity of a given pixel coordinate. It is called a window since it is often visualized as a moving collection of empty pixels that is passed over the image. A morphologic operation is (conceptually) defined by moving a window over the binary image to be modified, in such a way that it is eventually centered over every image pixel, where a local logical operation is performed. Usually this is done row-by-row, column-by-column, although it can be accomplished at every pixel simultaneously, if a massively parallel-processing computer is used.

Usually, a window is defined to have an approximate circular shape (a digital circle cannot be exactly realized) since it is desired that the window, and hence, the morphologic operator, be rotation-invariant. This means that if an object in the image is rotated through some angle, then the response of the morphologic operator will be unchanged other than also being rotated. While rotational symmetry cannot be exactly obtained, symmetry across two axes can be obtained, guaranteeing that the response be at least reflection-invariant. Window size also significantly effects the results, as will be seen.

A formal definition of windowing is needed in order to define the various morphologic operators. A window $\mathbf{B}$ is a set of $2P+1$ coordinate shifts $b_i=(n_i, m_i)$ centered around (0, 0):

$$\mathbf{B} = \{b_1,\ldots,b_{2P+1}\} = \{(n_1,m_1),\ldots,(n_{2P+1},m_{2P+1})\}$$

Some examples of common one-dimensional (row and column) windows are

$$\mathbf{B} = \text{ROW}[2P+1] = \{(0,m);\ m = -P,\ldots,P\} \quad (8)$$

$$\mathbf{B} = \text{COL}[2P+1] = \{(n,0);\ n = -P,\ldots,P\} \quad (9)$$

and some common two-dimensional windows are

$$\mathbf{B} = \text{SQUARE}[(2P+1)^2] = \{(n,m);\ n,m = -P,\ldots,P\} \quad (10)$$

$$\mathbf{B} = \text{CROSS}[4P+1] = \text{ROW}(2P+1) \cup \text{COL}(2P+1) \quad (11)$$

with obvious shape-descriptive names. In each of (8)–(11), the quantity in brackets is the number of coordinates shifts in the window, hence also the number of local gray levels that will be collected by the window at each image coordinate. Note that the windows (8)–(11) are each defined with an odd number $2P+1$ coordinate shifts. This is because the operators are symmetrical: pixels are collected in pairs from opposite sides of the center pixel or (0,0) coordinate shift, plus the (0,0) coordinate shift is always included. Examples of each of the windows (8)–(11) are shown in Fig. 10.

The example window shapes in (8)–(11) and in Fig. 10 are by no means the only possibilities, but they are (by far) the most common implementations because of the simple row-column indexing of the coordinate shifts.

The action of gray-level collection by a moving window creates the *windowed set.* Given a binary image f and a window $\mathbf{B}$, the windowed set at image coordinate $\mathbf{n}$ is given by

$$\mathbf{B}f(\mathbf{n}) = \{f(\mathbf{n}-\mathbf{m});\ \mathbf{m} \in \mathbf{B}\} \quad (12)$$

which, conceptually, is the set of image pixels covered by $\mathbf{B}$ when it is centered at coordinate $\mathbf{n}$. Examples of windowed sets associated with some of the windows in (8)–(11) and Fig. 10 are:

$$\mathbf{B} = \text{ROW}(3): \quad \mathbf{B}f(n_1,n_2) = \{f(n_1,n_2-1), f(n_1,n_2), f(n_1,n_2+1)\} \quad (13)$$

$$\mathbf{B} = \text{COL}(3): \quad \mathbf{B}f(n_1,n_2) = \{f(n_1-1,n_2), f(n_1,n_2), f(n_1+1,n_2)\} \quad (14)$$

$$\mathbf{B} = \text{SQUARE (9)}: \quad \mathbf{B}f(n_1,n_2) = \{f(n_1-1,n_2-1), f(n_1-1,n_2), f(n_1-1,n_2+1),$$
$$f(n_1,n_2-1), f(n_1,n_2), f(n_1,n_2+1),$$
$$f(n_1+1,n_2-1), f(n_1+1,n_2), f(n_1+1,n_2+1)\} \quad (15)$$

$$\mathbf{B} = \text{CROSS(5)}: \quad \mathbf{B}f(n_1,n_2) = \{f(n_1-1,n_2), f(n_1,n_2-1), f(n_1,n_2), f(n_1,n_2+1), f(n_1+1,n_2)\} \quad (16)$$

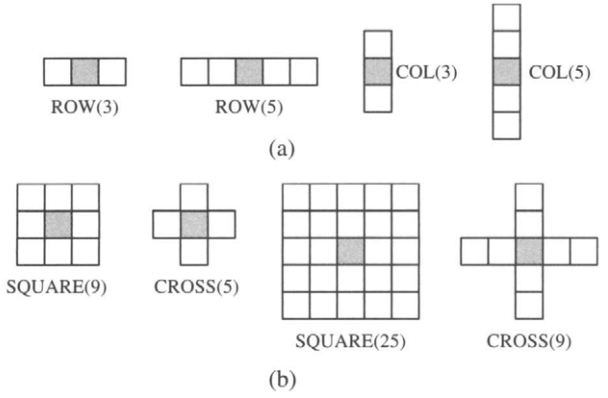


FIGURE 10 Examples of windows. The window is centered over the shaded pixel. (a) One-dimensional windows ROW$(2P+1)$ and COL$(2P+1)$ for $P=1$, 2; (b) Two-dimensional windows SQUARE $[(2P+1)^2]$ and CROSS$[4P+1]$ for $P=1$, 2.

where the elements of (13)–(16) have been arranged to show the geometry of the windowed sets when centered over coordinate $\mathbf{n}=(n_1, n_2)$. Conceptually, the window may be thought of as capturing a series of miniature images as it is passed over the image, row-by-row, column-by-column.

One last note regarding windows involves the definition of the windowed set when the window is centered near the boundary edge of the image. In this case, some of the elements of the windowed set will be undefined, since the window will overlap "empty space" beyond the image boundary. The simplest and most common approach is to use *pixel replication*: set each undefined windowed set value equal to the gray level of the nearest known pixel. This has the advantage of simplicity, and also the intuitive value that the world just beyond the borders of the image probably doesn't change very much. Figure 11 depicts the process of pixel replication.

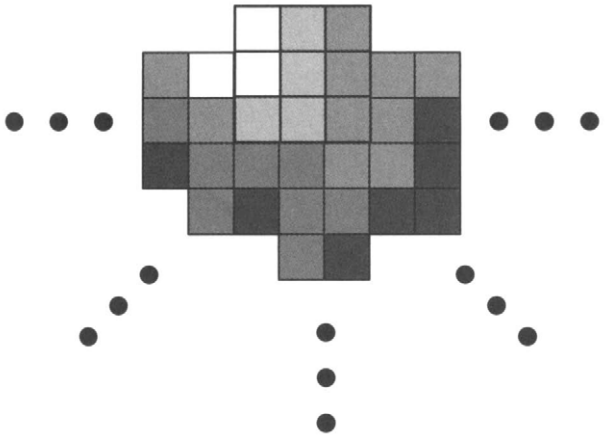


FIGURE 11 Depiction of pixel replication for a window centered near the (top) image boundary.

Morphologic Filters

Morphologic filters are Boolean filters. Given an image f, a many-to-one binary or Boolean function h, and a window $\mathbf{B}$, the Boolean-filtered image $g=h(f)$ is given by

$$g(\mathbf{n}) = h[\mathbf{B}f(\mathbf{n})] \tag{17}$$

at every $\mathbf{n}$ over the image domain. Thus, at each $\mathbf{n}$, the filter collects local pixels according to a geometrical rule into a windowed set, performs a Boolean operation on them, and returns the single Boolean result $g(\mathbf{n})$.

The most common Boolean operations that are used are AND, OR, and MAJ. They are used to create the following simple, yet powerful *morphologic filters*. These filters act on the objects in the image by shaping them: expanding or shrinking them, smoothing them, and eliminating too-small features.

The binary *dilation filter* is defined by

$$g(\mathbf{n}) = \mathrm{OR}[\mathbf{B}f(\mathbf{n})]. \tag{18}$$

and is denoted $g=dilate(f, \mathbf{B})$. The binary *erosion filter* is defined by

$$g(\mathbf{n}) = \mathrm{AND}[\mathbf{B}f(\mathbf{n})]. \tag{19}$$

and is denoted $g=erode(f, \mathbf{B})$. Finally, the binary *majority filter* is defined by

$$g(\mathbf{n}) = \mathrm{MAJ}[\mathbf{B}f(\mathbf{n})]. \tag{20}$$

and is denoted $g=majority(f, \mathbf{B})$. Next we explain the response behavior of these filters.

The *dilate* filter expands the size of the foreground, object, or "1"-valued regions in the binary image f. Here the "1"-valued pixels are assumed to be black because of the convention we have assumed, but this is not necessary. The process of dilation also smoothes the boundaries of objects, removing gaps or bays of too-narrow width, and also removing object holes of too-small size. Generally, a hole or gap will be filled if the dilation window cannot fit into it. These actions are depicted in Fig. 12, while Fig. 13 shows the result of dilating an actual binary image. Note that dilation using $\mathbf{B}=$ SQUARE(9) removed most of the small holes and

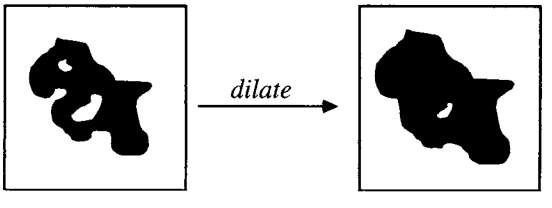


FIGURE 12 Illustration of dilation of a binary "1"-valued object. The smallest hole and gap were filled.

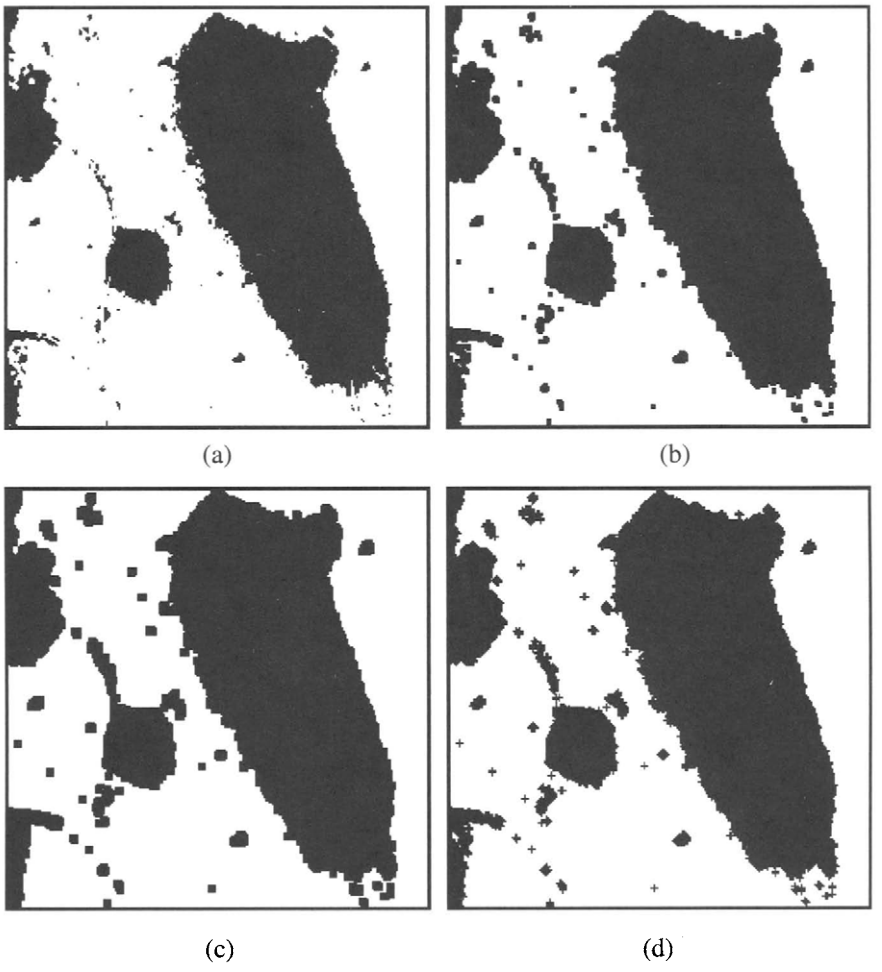


FIGURE 13 Dilation of a binary image. (a) Binarized image "cells." Dilate with: (b) B = SQUARE(9); (c) B = SQUARE(25); (d) B = CROSS(9).

gaps, while using **B** = SQUARE(25) removed nearly all of them. It is also interesting to observe that dilation with the larger window nearly completed a bridge between two of the large masses. Dilation with CROSS(9) highlights an interesting effect: individual, isolated "1"-valued or BLACK pixels were dilated into larger objects having the same shape as the window. This can also be seen with the results using the SQUARE windows. This effect underlines the importance of using symmetric windows, preferably with near rotational symmetry, since then smoother results are obtained.

The *erode* filter shrinks the size of the foreground, object, or "1"-valued regions in the binary image *f*. Alternately, it expands the size of the background or "0"-valued regions. The process of erosion smoothes the boundaries of objects, but in a different way than dilation: it removes peninsulas or fingers of too-narrow width, and also it removes "1"-valued objects of too-small size. Generally, an isolated object will be eliminated if the dilation window cannot fit into it. The effects of *erode* are depicted in Fig. 14.

Figure 15 shows the result of applying the *erode* filter to the binary image "cell." Erosion using **B** = SQUARE(9) removed many of the small objects and fingers, while using **B** = SQUARE(25) removed most of them. As an example of intense smoothing, **B** = SQUARE(81) (a 9×9 square window) was also applied. Erosion with CROSS(9) again produced a good result, except at a few isolated points where isolated "0"-valued or WHITE pixels were expanded into larger "+" -shaped objects.

An important property of the *erode* and *dilate* filters is the relationship that exists between them. In fact, in reality they

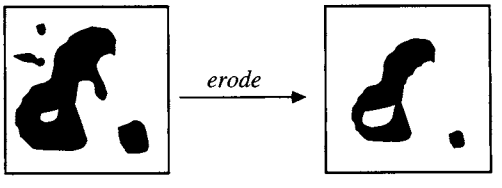


FIGURE 14 Illustration of erosion of a binary "1"-valued object. The smallest objects and peninsula were eliminated.

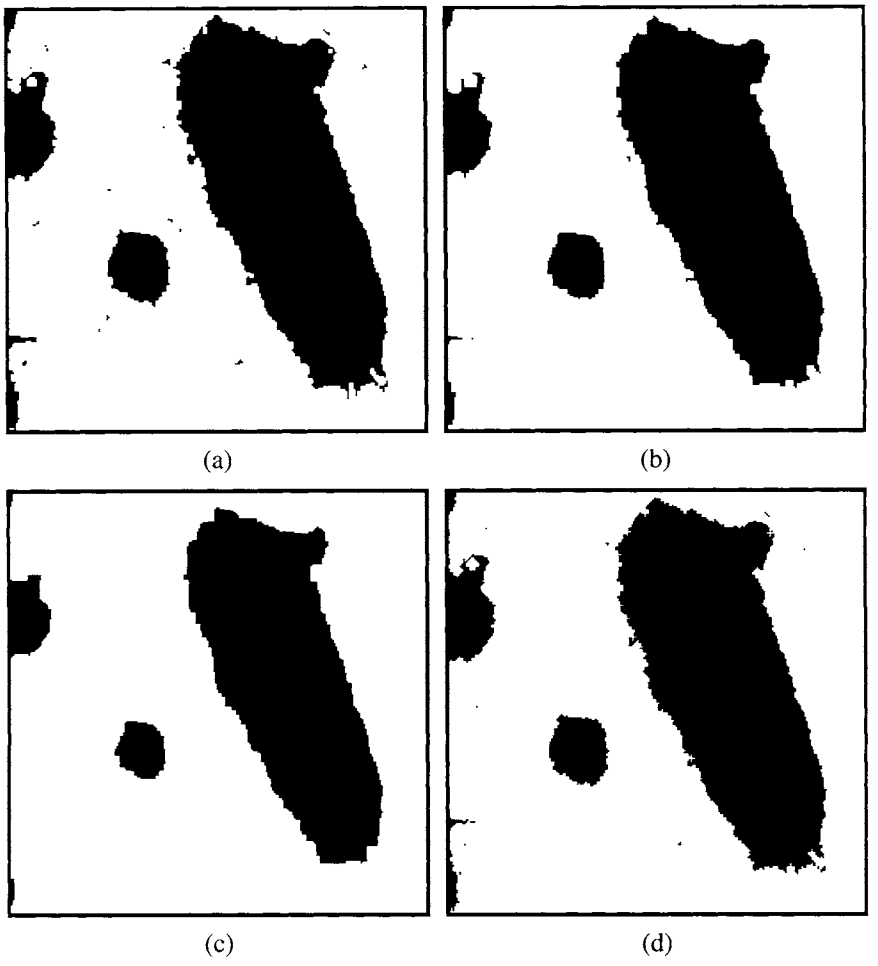


FIGURE 15 Erosion of the binary image "cells." Erode with: (a) B=SQUARE(9); (b) B=SQUARE(25); (c) B=SQUARE(81); (d) B=CROSS(9).

are the same operation, in the dual (complementary) sense. Indeed, given a binary image f and an arbitrary window **B**, it is true that

$$dilate(f, \mathbf{B}) = \text{NOT}\{erode[\text{NOT}(f), \mathbf{B}]\} \quad (21)$$

$$erode(f, \mathbf{B}) = \text{NOT}\{dilate[\text{NOT}(f), \mathbf{B}]\}. \quad (22)$$

Equations (21) and (22) are a simple consequence of the DeMorgan's Laws (5) and (6). A correct interpretation of this is that erosion of the "1"-valued or BLACK regions of an image is the same as dilation of the "0"-valued or WHITE regions—and vice-versa.

An important and common misconception must be mentioned. *Erode* and *dilate* shrink and expand the sizes of "1"-valued objects in a binary image. However, they are *not* inverse operations of one another. Dilating an eroded image (or eroding a dilated image) very rarely yields the original image. In particular, dilation cannot recreate peninsulas, fingers or small objects that have been eliminated by erosion. Likewise, erosion cannot unfill holes filled by dilation or recreate gaps or bays filled by dilation. Even without these effects, erosion generally will not exactly recreate the same shapes that have been modified by dilation, and vice-versa.

Before discussing the third common Boolean filter, the majority, we will consider further the idea of sequentially applying *erode* and *dilate* filters to an image. One reason for doing this is that the *erode* and *dilate* filters have the effect of changing the sizes of objects, as well as smoothing them. For some objects this is desirable, e.g., when an extraneous object is shrunk to the point of disappearing; however, often it is undesirable, since it may be desired to further process or analyze the image. For example, it may be of interest to label the objects and compute their sizes, as in Section 3 of this chapter.

Although *erode* and *dilate* are not inverse operations of one another, they are approximate inverses in the sense that if they are performed in sequence on the same image with the same window **B**, then object and holes that are not eliminated will be returned to their approximate sizes. We thus define the

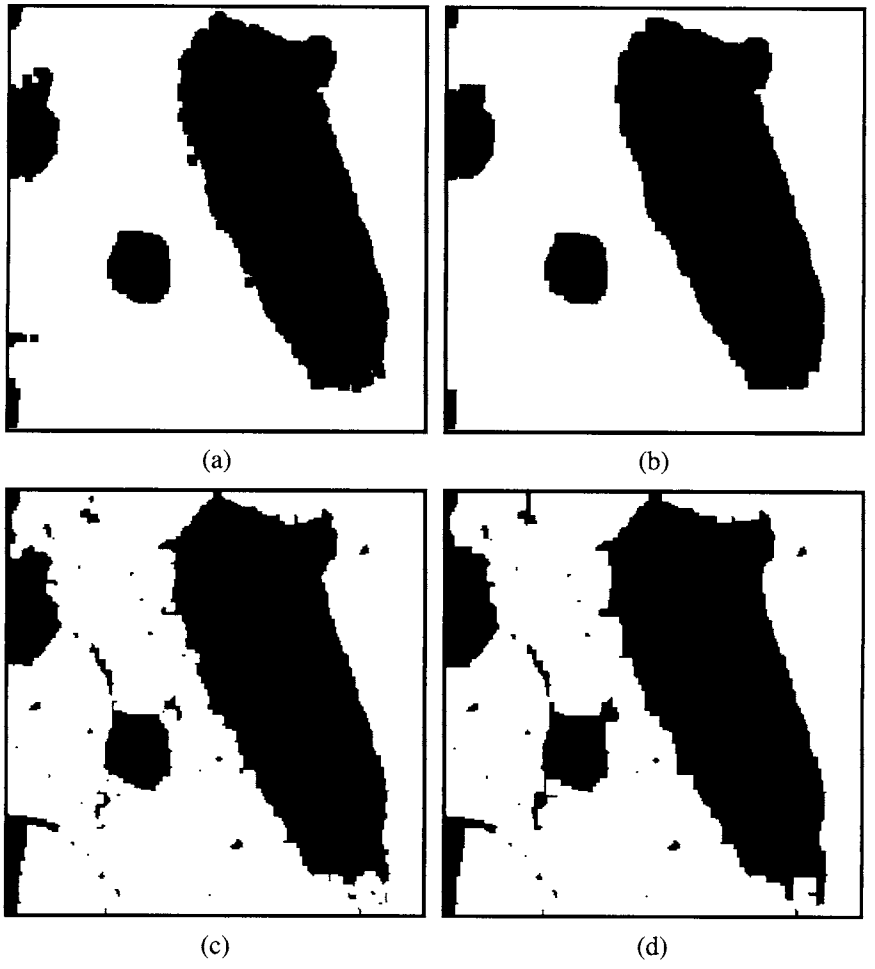


FIGURE 16 *Open* and *close* filtering of the binary image "cells." *Open* with: (a) B = SQUARE(25); (b) B = SQUARE(81). *Close* with: (c) B = SQUARE(25); (d) B = SQUARE(81).

size-preserving smoothing morphologic operators termed *open filter* and *close filter*, as follows:

$$open(f, \mathbf{B}) = dilate[erode(f, \mathbf{B}), \mathbf{B}] \tag{23}$$

$$close(f, \mathbf{B}) = erode[dilate(f, \mathbf{B}), \mathbf{B}]. \tag{24}$$

Hence, the opening (closing) of image f is the erosion (dilation) with window **B** followed by dilation (erosion) with window **B**. The morphologic filters *open* and *close* have the same smoothing properties as *erode* and *dilate*, respectively, but they do not generally effect the sizes of sufficiently large objects much (other than pixel loss from pruned holes, gaps or bays, or pixel gain from eliminated peninsulas).

Figure 16 depicts the results of applying the *open* and *close* operations to the binary image "cell," using the windows **B** = SQUARE(25) and **B** = SQUARE(81). Large windows were used to illustrate the powerful smoothing effect of these morphologic smoothers. As can be seen, the *open* filters did an excellent job of eliminating what might be referred to as "black noise"—the extraneous "1"—valued objects and other features, leaving smooth, connected, and appropriately-sized large objects. By comparison, the *close* filters smoothed the image intensely as well, but without removing the undesirable "black noise." In this particular example, the result of *open* is probably preferable to that of *close*, since the extraneous BLACK structures present more of a problem in the image.

It is important to understand that the *open* and *close* filters are *unidirectional* or *biased* filters in the sense that they remove one type of "noise" (either extraneous WHITE or BLACK features), but not both. Hence, *open* and *close* are somewhat special-purpose binary image smoothers that are used when too-small BLACK and WHITE objects (respectively) are to be removed.

It is worth noting that the *close* and *open* filters are again in fact, the same filters, in the dual sense. Given a binary image f and an arbitrary window **B**:

$$close(f, \mathbf{B}) = \text{NOT}\{open[\text{NOT}(f), \mathbf{B}]\} \tag{25}$$

$$open(f, \mathbf{B}) = \text{NOT}\{close[\text{NOT}(f), \mathbf{B}]\}. \tag{26}$$

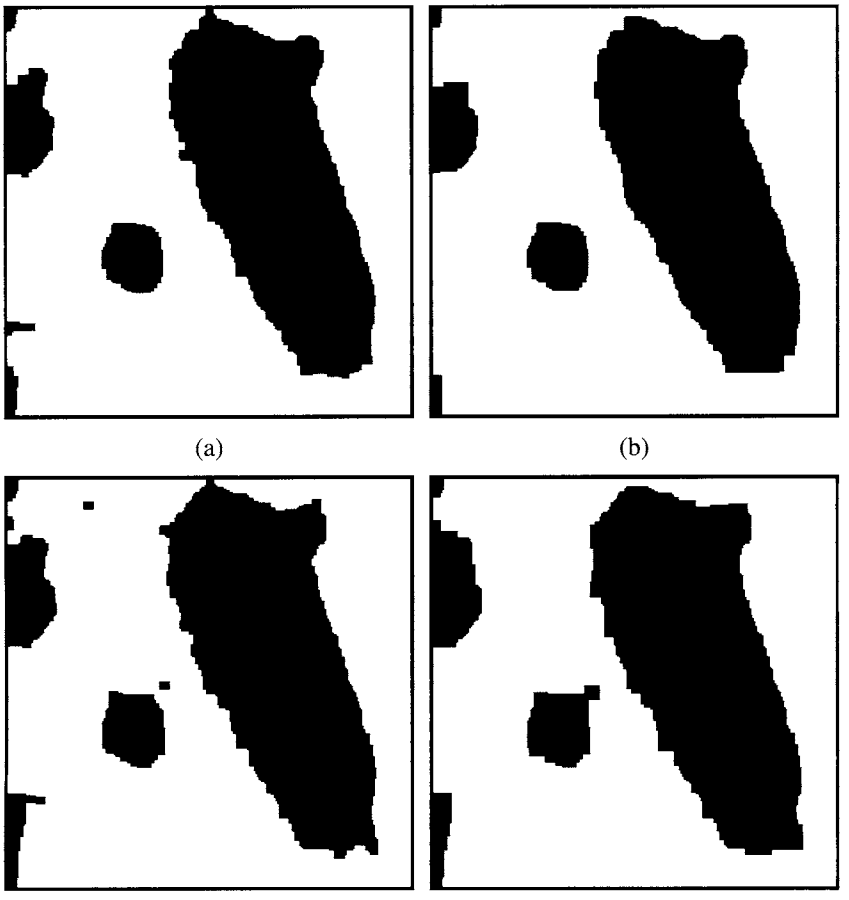


FIGURE 17 *Close-open* and *open-close* filtering of the binary image "cells." *Close-open* with: (a) B = SQUARE(25); (b) B = SQUARE(81). *Open-close* with: (c) B = SQUARE(25); (d) B = SQUARE(81).

In most binary smoothing applications, it is desired to create an unbiased smoothing of the image. This can be accomplished by a further concatenation of filtering operations, applying *open* and *close* operations in sequence on the same image with the same window **B**. The resulting images will then be smoothed *bidirectionally.* We thus define the unbiased smoothing morphologic operators *close-open filter* and *open-close filter,* as follows:

$$close\text{-}open(f, \mathbf{B}) = close[open(f, \mathbf{B}), \mathbf{B}] \quad (27)$$

$$open\text{-}close(f, \mathbf{B}) = open[close(f, \mathbf{B}), \mathbf{B}]. \quad (28)$$

Hence, the *close-open* (*open-close*) of image f is the *open* (*close*) of f with window **B** followed by the *close* (*open*) of the result with window **B**. The morphologic filters *close-open* and *open-close* in (27) and (28) are general-purpose, bi-directional, size-preserving smoothers. Of course, they may each be interpreted as a sequence of four basic morphologic operations (erosions and dilations).

The *close-open* and *open-close* filters are quite similar but are not mathematically identical. Both remove too-small structures without affecting size much. Both are powerful shape smoothers. However, differences between the processing results can be easily seen. These mainly manifest as a function of the first operation performed in the processing sequence. One notable difference between *close-open* and *open-close* is that *close-open* often links together neighboring holes (since *erode* is the first step), while *open-close* often links neighboring objects together (since *dilate* is the first step). The differences are usually somewhat subtle, yet often visible upon close inspection.

Figure 17 shows the result of applying the *close-open* and the *open-close* filters to the ongoing binary image example. As can be seen the results (for **B** fixed) are very similar, although the *close-open* filtered results are somewhat cleaner, as expected. There are also only small differences between the results obtained using the medium and larger windows, because of the intense smoothing that is occurring. To fully appreciate the power of these smoothers, it is worth comparing to the original binarized image "cells" in Fig. 13(a).

The reader may wonder whether further sequencing of the filtered responses will produce different results. If the filters

are properly alternated as in the construction of the *close-open* and *open-close* filters, then the dual filters become increasingly similar. However, the smoothing power can most easily be increased by simply taking the window size to be larger.

Once again, the *close-open* and *open-close* filters are dual filters under complementation.

We now return to the final binary smoothing filter, the *majority filter*. The *majority* filter is also known as the binary *median filter*, since it may be regarded as a special case (the binary case) of the gray-level median filter (Chapter 3.2).

The *majority* filter has similar attributes as the *close-open* and *open-close* filters: it removes too-small objects, holes, gaps, bays and peninsulas (both "1"-valued and "0"-valued small features), and it also does not generally change the size of objects or of background, as depicted in Fig. 18. It is less biased than any of the other morphologic filters, since it does not have an initial *erode* or *dilate* operation to set the bias. In fact, *majority* is its own dual under complementation, since

$$majority(f, \mathbf{B}) = \text{NOT}\{majority[\text{NOT}(f), \mathbf{B}]\} \quad (29)$$

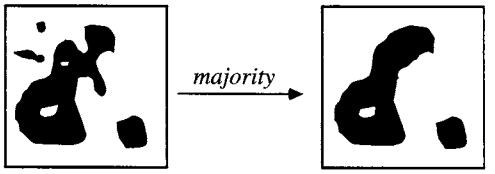


FIGURE 18 Effect of *majority* filtering. The smallest holes, gaps, fingers, and extraneous objects are eliminated.

The *majority* filter is a power, unbiased shape smoother. However, for a given filter size, it does not have the same degree of smoothing power as *close-open* or *open-close*.

Figure 19 shows the result of applying the *majority* or *binary median* filter to the image "cell." As can be seen, the results obtained are very smooth. Comparison with the results of *open-close* and *close-open* are favorable, since the boundaries of the major smoothed objects are much smoother in the case of the median filter, for both window shapes used and for each size. The *majority* filter is quite commonly used for smoothing noisy binary images of this type because of these nice

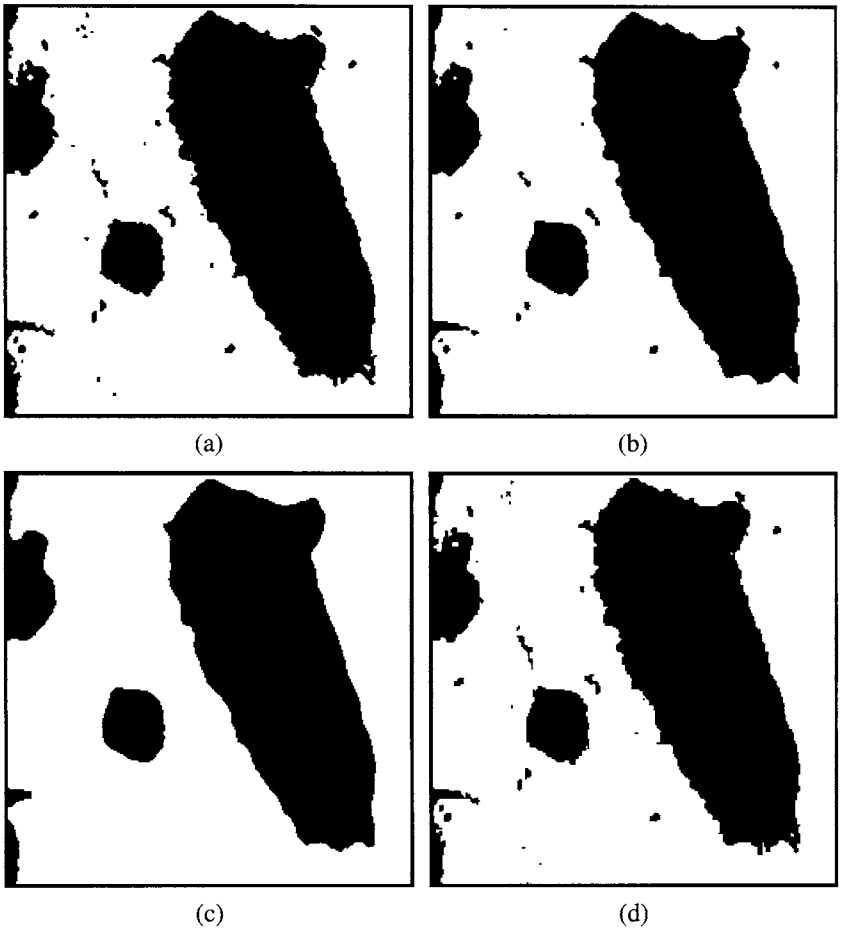
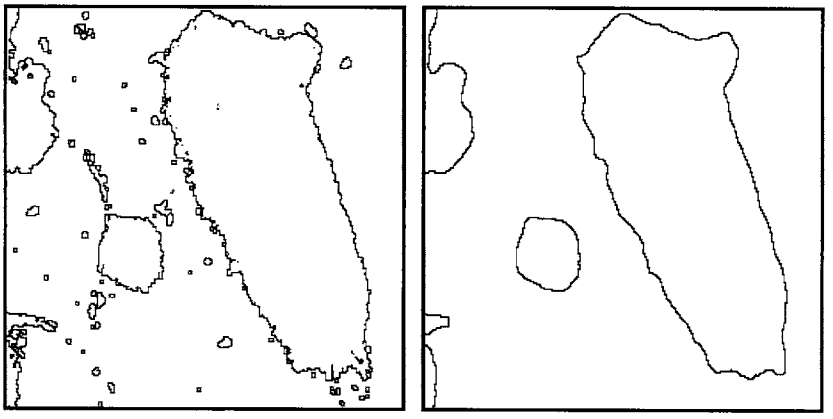


FIGURE 19 *Majority* or *median* filtering of the binary image "cells." *Majority* with: (a) B = SQUARE(9); (b) B = SQUARE(25). *Majority* with: (c) B = SQUARE(81); (d) B = CROSS(9).



(a) (b)

FIGURE 20 Object boundary detection. Application of *boundary* (f, B) to (a) the image "cells"; (b) the *majority*-filtered image in Fig. 19(c).

properties. The more general gray-level median filter (Chapter 3.2) is also among the most-used image processing filters.

Morphologic Boundary Detection

The morphologic filters are quite effective for smoothing binary images but they have other important applications as well. One such application is *boundary detection,* which is the binary case of the more general edge detectors studied in Chapters 4.11 and 4.12.

At first glance, boundary detection may seem trivial, since the boundary points can be simply defined as the transitions from "1" to "0" (and vice-versa). However, when there is noise present, boundary detection becomes quite sensitive to small noise artifacts, leading to many useless detected edges. Another approach which allows for smoothing of the object boundaries involves the use of morphologic operators.

The "difference" between a binary image and a dilated (or eroded) version of it is one effective way of detecting the object boundaries. Usually it is best that the window **B** that is used be small, so that the difference between image and dilation is not too large (leading to thick, ambiguous detected edges). A simple and effective "difference" measure is the two-input exclusive-OR operator XOR. The XOR takes logical value "1" only if its two inputs are different. The boundary detector then becomes simply:

$$boundary(f, \mathbf{B}) = \mathrm{XOR}[f,\ dilate(f, \mathbf{B})] \qquad (30)$$

The result of this operation as applied to the binary image "cells" is shown in Fig. 20(a) using $\mathbf{B} = \mathrm{SQUARE}(9)$. As can be seen, essentially all of the BLACK/WHITE transitions are marked as boundary points. Often, this is the desired result. However, in other instances, it is desired to detect only the major object boundary points. This can be accomplished by first smoothing the image with a *close-open, open-close,* or *majority* filter. The result of this smoothed boundary detection process is shown in Fig. 20(b). In this case, the result is much cleaner, as only the major boundary points are discovered.

5 Binary Image Representation and Compression

In several later chapters, methods for compressing gray-level images are studied in detail. Compressed images are representations that require less storage than the nominal storage. This is generally accomplished by coding of the data based on measured statistics, rearrangement of the data to exploit patterns and redundancies in the data, and (in the case of lossy compression), quantization of information. The goal is that the image, when decompressed, either looks very much like the original despite a loss of some information (lossy compression), or is not different from the original (lossless compression).

Methods for lossless compression of images are discussed in Chapter 5.1. Those methods can generally be adapted to both gray-level and binary images. Here, we will look at two methods for lossless binary image representation that exploit an assumed structure for the images. In both methods the image data is represented in a new format that exploits the structure. The first method is *run-length coding,* which is so-called because it seeks to exploit the redundancy of long run-lengths or runs of constant value "1" or "0" in the binary data. It is thus appropriate for the coding/compression of binary images containing large areas of constant value "1" and "0". The second method, *chain coding,* is appropriate for binary images containing binary contours, such as the boundary images shown in Fig. 20. Chain coding achieves compression by exploiting this assumption. The chain code is also an information-rich, highly manipulable representation that can be used for shape analysis.

Run-Length Coding

The number of bits required to naively store an $N \times M$ binary image is NM. This can be significantly reduced if it is known that the binary image is smooth in the sense that it is composed primarily of large areas of constant "1" and/or "0" value.

The basic method of run-length coding is quite simple. Assume that the binary image f is to be stored or transmitted on a row-by-row basis. Then for each image row numbered m, the following algorithm steps are used:

1. Store the first pixel value ("0" or "1") in row m in a 1-bit buffer as a reference
2. Set the run counter $c = 1$
3. For each pixel in the row:
   - Examine the next pixel to the right
   - If it is the same as the current pixel, set $c = c + 1$
   - If different from the current pixel, store c in a buffer of length b and set $c = 1$
   - Continue until end of row is reached

Thus, each run-length is stored using b bits. This requires that an overall buffer with segments of lengths b be reserved to store the run-lengths. Run-length coding yields excellent lossless compressions, provided that the image contains lots of constant runs. Caution is necessary, since if the image contains only very short runs, then run-length coding can actually increase the required storage.

Figure 21 depicts two hypothetical image rows. In each case, the first symbol stored in a one-bit buffer will be logical "1". The run-length code for Fig. 21(a) would be "1", 7, 5, 8, 3, 1 .... with symbols after the "1" stored using b bits. The first five runs in this sequence have average length $24/5 = 4.8$, hence if $b \leq 4$ then compression will occur. Of course, the compression can be much higher, since there may be runs of lengths in the dozens or hundreds, leading to very high compressions.

In Fig. 21(b), however, in this worst-case example the storage actually increases b-fold! Hence, care is needed when applying this method. The apparent rule of them, if it can be applied *a priori*, is that the average run-length L of the image should satisfy $L > b$ if compression is to occur. In fact, the compression ratio will be approximately L/b.

Run-length coding is also used in other scenarios than binary image coding. It can also be adapted to situations where there are run-lengths of any value. For example, in the JPEG lossy image compression standard for gray-level images (see Chapter 5.5), a form of run-length coding is used to code runs of zero-valued frequency-domain coefficients. This run-length coding is an important factor in the good compression performance of JPEG. A more abstract form of run-length coding is also responsible for some of the excellent compression performance of recently-developed wavelet image compression algorithms (Chapter 5.4).

Chain Coding

Chain coding is an efficient representation of binary images composed of contours. We will refer to these as "contour images." We assume that contour images are composed only of single-pixel width, connected contours (straight or curved). These arise from processes of edge detection or boundary detection, such as the morphologic boundary detection method just described above, or the results of some of the edge detectors described in Chapters 4.11 and 4.12 when applied to gray-scale images.

The basic idea of chain coding is to code contour directions instead of naïve bit-by-bit binary image coding or even coordinate representations of the contours. Chain coding is based on identifying and storing the directions from each pixel to its neighbor pixel on each contour. Before defining this process, it is necessary to clarify the various types of neighbors that are associated with a given pixel in a binary image. Fig. 22 depicts two neighborhood systems around a pixel (shaded). To the left are depicted the *4-neighbors* of the pixel, which are connected along the horizontal and vertical directions. The set of 4-neighbors of a pixel located at coordinate $\mathbf{n}$ will be denoted $N_4(\mathbf{n})$. To the right are the *8-neighbors* of the shaded pixel in the center of the grouping. These include the pixels connected along the diagonal directions. The set of 8-neighbors of a pixel located at coordinate $\mathbf{n}$ will be denoted $N_8(\mathbf{n})$.

If the initial coordinate $\mathbf{n}_0$ of an 8-connected contour is known, then the rest of the contour can be represented without loss of information by the directions along which the contour propagates, as depicted in Fig. 23(a). The initial coordinate can be an endpoint, if the contour is open, or an arbitrary point, if the contour is closed. The contour can be reconstructed from the directions, if the initial coordinate is known. Since there are only eight directions that are possible,

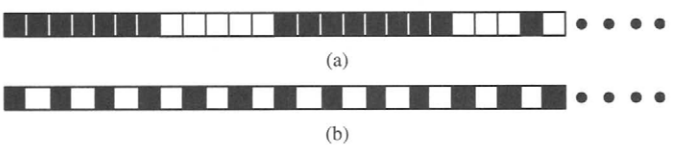


FIGURE 21 Example rows of a binary image, depicting (a) reasonable and (b) unreasonable scenarios for run-length coding.

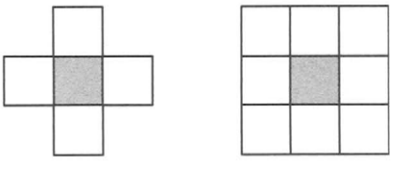


FIGURE 22 Depiction of the 4-neighbors and the 8-neighbors of a pixel (shaded).

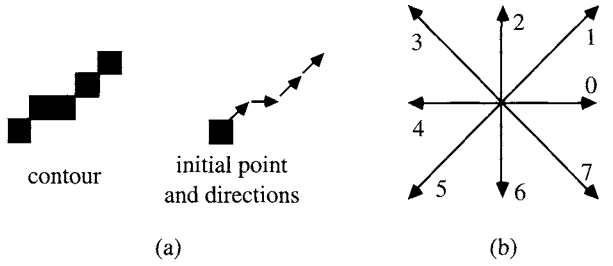


FIGURE 23 Representation of a binary contour by direction codes. (a) A connected contour can be represented exactly by an initial point and the subsequent directions. (b) Only 8 direction codes are required.

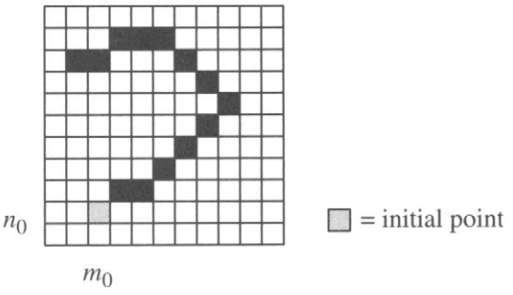


FIGURE 24 Depiction of chain coding.

then a simple 8-neighbor direction code may be used. The integers $\{0, \ldots, 7\}$ suffice for this, as shown in Fig. 23(b).

Of course, the direction codes 0, 1, 2, 3, 4, 5, 6, 7 can be represented by their 3-bit binary equivalents: 000, 001, 010, 011, 100, 101, 110, 111. Hence, each point on the contour *after* the initial point can be coded by three bits. The initial point of each contour requires $\lceil \log_2(MN) \rceil$ bits, where $\lceil \cdot \rceil$ denotes the ceiling function: $\lceil x \rceil$ = the smallest integer that is greater than or equal to x. For long contours, storage of the initial coordinates is incidental.

Figure 24 shows an example of chain coding of a short contour. After the initial coordinate $\mathbf{n}_0 = (n_0, m_0)$ is stored, the chain code for the remainder of the contour is: 1, 0, 1, 1, 1, 3, 3, 3, 4, 4, 5, 4 in integer format, or 001, 000, 001, 001, 001, 001, 011, 011, 011, 100, 100, 101, 100 in binary format.

Chain coding is an efficient representation. For example, if the image dimensions $N = M = 512$, then representing the contour by storing the coordinates of each contour point requires six times as much storage as the chain code.

Acknowledgment

The author thanks Mita Desai for carefully reading this chapter and providing comments.