Chapter 3

Binary Image Analysis

In a number of applications, such as document analysis and some industrial machine vision tasks, binary images can be used as the input to algorithms that perform useful tasks. These algorithms can handle tasks ranging from very simple counting tasks to much more complex recognition, localization, and inspection tasks. Thus by studying binary image analysis before going on to gray-tone and color images, one can gain insight into the entire image analysis process.

In this chapter, the basic operations of binary machine vision are described. First, a simple object-counting algorithm is used to show the reader how a very simple algorithm can be used to accomplish a useful task. Next we discuss the connected components labeling operator, which gives each separate connected group of pixels a unique label and is a predecessor to most later steps of processing. Then a set of thinning and thickening operators is introduced. The operators of mathematical morphology can be used to join and separate components, close up holes, and find features of interest in an image . Once a set of components has been isolated, a number of important properties of each component can be computed for use in higher-level tasks such as recognition and tracking. A set of basic properties is defined and the accuracy of the algorithms that compute them discussed. Finally, the problem of automatically thresholding a gray-scale or color image to produce a useful binary image is studied.

3.1 Pixels and Neighborhoods

A binary image B can be obtained from a gray scale or color image I through an operation that *selects* a subset of the image pixels as *foreground* pixels, the pixels of interest in an image analysis task, leaving the rest as *background pixels* to be ignored. The selection operation can be as simple as the thresholding operator that chooses pixels in a certain range of gray-tones or subspace of color space or it may be a complex classification algorithm. Thresholding will be discussed at the end of this chapter, while more advanced selection operators will appear at various parts of the text. For the beginning of this chapter, we will assume that the binary image B is the initial input to our tasks. Figure 3.1 illustrates the concept with four binary images of hand-printed characters.

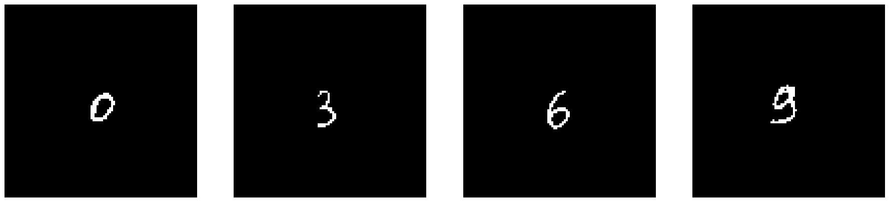


Figure 3.1: Binary images of hand-printed characters.

The pixels of a binary image B are 0's and 1's; the 1's will be used to denote foreground pixels and the 0's background pixels. The term $B[r, c]$ denotes the value of the pixel located at row r, column c of the image array. An $M \times N$ image has M rows numbered from 0 to $M - 1$ and N columns numbered from 0 to $N - 1$. Thus $B[0, 0]$ refers to the value of the upper leftmost pixel of the image and $B[M - 1, N - 1]$ refers to the value of the lower rightmost pixel.

In many algorithms, not only the value of a particular pixel, but also the values of its neighbors are used when processing that pixel. The two most common definitions for neighbors are the *four-neighbors* and the *eight-neighbors* of a pixel. The four-neighborhood $N_4(r, c)$ of pixel (r, c) includes pixels $(r - 1, c)$, $(r + 1, c)$, $(r, c - 1)$, and $(r, c + 1)$, which are often referred to as its north, south, west, and east neighbors, respectively. The eight-neighborhood $N_8(r, c)$ of pixel (r, c) includes each pixel of the four-neighborhood plus the diagonal neighbor pixels $(r-1, c-1)$, $(r-1, c+1)$, $(r+1, c-1)$, and $(r+1, c+1)$, which can be referred to as its northwest, northeast, southwest, and southeast neighbors, respectively. Figure 3.2 illustrates these concepts.

	N	
W	*	E
	S	

a) four-neighborhood N_4

NW	N	NE
W	*	E
SW	S	SE

b) eight-neighborhood N_8

Figure 3.2: The two most common neighborhoods of a pixel.

Either the four-neighborhood or the eight-neighborhood (or some alternate definition) can be used as the *neighborhood* of a pixel in various algorithms. To be general, we will say that a pixel (r', c') *neighbors* a pixel (r, c) if (r', c') lies in the selected type of neighborhood of (r, c).

3.2 Applying Masks to Images

A basic concept in image processing is that of applying a *mask* to an image. The concept comes from the image processing operation of convolution, but is used in a general sense in image analysis as a whole. A mask is a set of pixel positions and corresponding values called *weights*. Figure 3.3 shows three different masks. The first two (a and b)are square

1	1	1
1	1	1
1	1	1

a)

1	2	1
2	4	2
1	2	1

b)

1
1
1
1
1

c)

Figure 3.3: Three masks that can be applied to an image.

masks, one with equal weights, all of value one and one with unequal weights. The third mask (c) is rectangular and has equal weights.

Each mask has an *origin*, which is usually one of its positions. Usually the origins of symmetric masks, such as a) and b) Figure 3.3, are their center pixels. For nonsymmetric masks, any pixel may be chosen as the origin, depending on the intended use. The top pixel of mask c) might be chosen as its origin.

The application of a mask to an input image yields an output image of the same size as the input. For each pixel in the input image, the mask is conceptually placed on top of the image with its origin lying on that pixel. The values of each input image pixel under the mask are multiplied by the weights of the corresponding mask pixels. The results are summed together to yield a single output value that is placed in the output image at the location of the pixel being processed on the input. Figure 3.4 illustrates the application of mask b) of Figure 3.3 to a gray-tone image.

The original gray tone image is shown in Figure 3.4a. Notice that when the center of the mask lies on top of one of the perimeter pixels of the image, some of the pixels of the mask lie outside of the image. In order to make the output image come out the same size as the input, we must add some virtual rows and columns to the input image around the edges. In the example below, we have added two virtual rows (one above the image and one below) and two virtual columns (one to the left and one to the right). The values in these virtual rows and columns can be set arbitrarily to zero or some other constant or, as has been done here, they can merely duplicate the closest row (or column) to them. Thus the virtual row added to the top of the input image would duplicate the values 40, 40, 80, 80, 80; the virtual column on the left would be all 40's; the virtual column on the right would be all 80's; and the virtual row added to the bottom would again have values 40, 40, 80, 80, 80. The output image c) produced by the application of the mask b) is a smoothed version of the input a); however, the values are all much bigger than in the original. To normalize, we divide the value obtained for each pixel by the sum of the weights in the mask, in this case 16, obtaining the final image shown in d). The original and final images are shown in gray tone in e) expanded to 120×120 for visibility. Because of the expansion, each pixel in the final image is shown as a 24-pixel strip; thus the smoothness is at a strip level, instead of at a pixel level.

40	40	80	80	80
40	40	80	80	80
40	40	80	80	80
40	40	80	80	80
40	40	80	80	80

a) original gray-tone image

1	2	1
2	4	2
1	2	1

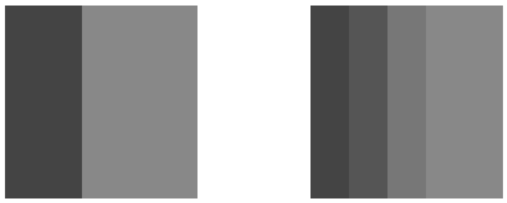
b) 3×3 mask

640	800	1120	1280	1280
640	800	1120	1280	1280
640	800	1120	1280	1280
640	800	1120	1280	1280
640	800	1120	1280	1280

c) result of applying the mask to the image

40	50	70	80	80
40	50	70	80	80
40	50	70	80	80
40	50	70	80	80
40	50	70	80	80

d) normalized result after division by the sum of the weights in the mask (16)



e) original image and result, expanded to 120×120 for viewing

Figure 3.4: Application of a mask with weights to a gray-scale image.

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

a) E: external corners

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

b) I: internal corners

Figure 3.5: The 2 by 2 masks for counting the foreground objects in a binary image. The 1's represent foreground pixels, and the 0's represent background pixels.

3.3 Counting the Objects in an Image

Chapter 1 presented an application where it was important to count the number of holes in an object. Counting the number of foreground objects is an equivalent problem that can be performed with the same algorithm by merely swapping the roles of the two sets: E and I. For counting foreground objects, the external corner patterns are 2 by 2 masks that have three 0's and one 1-pixel. The internal corner patterns are 2 by 2 masks that have three 1's and one 0-pixel. Figure 3.5 illustrates the two sets of masks. Note that the algorithm expects each object to be a 4-connected set of 1-pixels with no interior holes.

The application of one of these masks to a binary image can be visualized as placing the mask on the image so that the top left pixel of the mask lines up with the particular pixel being considered on the image. In this case the mask is defining a neighborhood of the image pixel consisting of the pixel, its neighbor to the right, and the two pixels below them. If all four image pixels that fall under the mask have exactly the same value as the corresponding mask pixel, then the type of corner defined by that mask is identified with that image pixel. Suppose that the function *external_match(L, P)* sequences through the four external masks and returns *true* if the subimage with top left pixel (L,P) matches one of them, false otherwise. Similarly, the function *internal_match(L,P)* returns *true* if the subimage with top left pixel (L,P) matches one of the internal masks and false otherwise. The object-counting function *count_objects(B)* takes in a binary image B, loops through each pixel of the image, excluding pixels of the last row and the last column, where the 2 by 2 mask cannot be placed, and returns the number of objects in the image.

Conventions for defining algorithms Pseudo-code for the object-counting procedure is given below. We will use this syntax for all procedures given in the text. Note that all routines are called *procedures*, but those that are functions include a *return* statement (as in C) to return a value. To keep the procedures short and simple, we will often use utility procedures within them such as *external_match* and *internal_match*. The code for very straightforward utility procedures such as these is usually omitted. We also omit type declarations, which are language-dependent, but we specify the required types in the text and explain important variables in comments. Finally, we use global constants for various sizes rather than clouding the procedure calls with extra arguments.

In the object-counting procedure, the constant MaxRow is the row number of the last row in the image, while MaxCol is the column number of the last column. The first row and the first column are assumed to be row and column zero, the default for C arrays.

```
Compute the number of foreground objects of binary image B.
Objects are 4-connected and simply connected.
E is the number of external corners.
I is the number of internal corners.


     procedure count_objects(B);
     {
     E := 0;
     I := 0;
     for L := 0 to MaxRow - 1
        for P := 0 to MaxCol - 1
           {
           if external_match(L, P) then E := E + 1;
           if internal_match(L, P) then I := I + 1;
           } ;
     return((E - I) / 4);
     }
```

Algorithm 1: Counting Foreground Objects

Exercise 1 Efficiency of counting objects

What is the maxiumum number of times that procedure *count_objects* examines each pixel of the image? How can procedures *external_match* and *internal_match* be coded to be as efficient as possible?

Exercise 2 Driving around corners

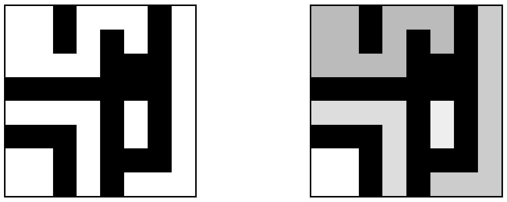
Obtain some graph paper to represent a pixel array and blacken some region of connected squares (keep it small at first). The blackened squares correspond to the foreground pixels and the empty squares correspond to the background. Imagine that the pixels are all city blocks and you are driving around the blackened region in a clockwise direction. Do your right turns correspond to E corners or I corners? What about left turns? Is there a relationship between the number of left turns and the number of right turns made in driving the complete perimeter? If so, what is it? In driving the entire perimeter, did you ever cross over or touch a previously visited intersection? Is that ever possible? Why or why not? Before answering, consider the case of only two blackened blocks touching diagonally across a single shared intersection. Do your left-right counting rules still hold? Does the object-counting formula still hold?

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

a) binary image

1	1	0	1	1	1	0	2
1	1	0	1	0	1	0	2
1	1	1	1	0	0	0	2
0	0	0	0	0	0	0	2
3	3	3	3	0	4	0	2
0	0	0	3	0	4	0	2
5	5	0	3	0	0	0	2
5	5	0	3	0	2	2	2

b) connected components labeling



c) binary image and labeling, expanded for viewing

Figure 3.6: A binary image with five connected components of the value 1.

3.4 Connected Components Labeling

Suppose that B is a binary image and that $B(r,c) = B(r',c') = v$ where either $v = 0$ or $v = 1$. The pixel (r,c) is *connected* to the pixel (r',c') with respect to value v if there is a sequence of pixels $(r,c) = (r_0,c_0),(r_1,c_1),\ldots,(r_n,c_n) = (r',c')$ in which $B(r_i,c_i) = v, i = 0,\ldots,n$, and (r_i,c_i) neighbors (r_{i-1},c_{i-1}) for each $i = 1,\ldots,n$. The sequence of pixels $(r_0,c_0),\ldots,(r_n,c_n)$ forms a connected *path* from (r,c) to (r',c'). A *connected component* of value v is a set of pixels C, each having value v, and such that every pair of pixels in the set are connected with respect to v. Figure 3.6a) shows a binary image with five such connected components of 1's; these components are actually connected with respect to either the eight-neighborhood or the four-neighborhood definition.

1 Definition *A* **connected components labeling** *of a binary image B is a labeled image LB in which the value of each pixel is the label of its connected component.*

A label is a symbol that uniquely names an entity. While character labels are possible, positive integers are more convenient and are most often used to label the connected components. Figure 3.6b) shows the connected components labeling of the binary image of Figure 3.6a).

There are a number of different algorithms for the connected components labeling operation. Some algorithms assume that the entire image can fit in memory and employ a simple, recursive algorithm that works on one component at a time, but can move all over the image while doing so. Other algorithms were designed for larger images that may not fit in memory and work on only two rows of the image at a time. Still other algorithms were designed for massively parallel machines and use a parallel propagation strategy. We will look at two different algorithms in this chapter: the recursive search algorithm and a row-by-row algorithm that uses a special union-find data structure to keep track of components.

A Recursive Labeling Algorithm

Suppose that B is a binary image with $MaxRow + 1$ rows and $MaxCol + 1$ columns. We wish to find the connected components of the 1-pixels and produce a labeled output image LB in which every pixel is assigned the label of its connected component. The strategy, adapted from the Tanimoto AI text, is to first negate the binary image, so that all the 1-pixels become -1's. This is needed to distinguish unprocessed pixels (-1) from those of component label 1. We will accomplish this with a function called *negate* that inputs the binary image B and outputs the negated image LB, which will become the labeled image. Then the process of finding the connected components becomes one of finding a pixel whose value is -1 in LB, assigning it a new label, and calling procedure *search* to find its neighbors that have value -1 and recursively repeat the process for these neighbors. The utility function *neighbors(L,P)* is given a pixel position defined by L and P. It returns the set of pixel positions of all of its neighbors, using either the 4-neighborhood or 8-neighborhood definition. Only neighbors that represent legal positions on the binary image are returned. The neighbors are returned in scan-line order as shown in Figure 3.7. The recursive connected components labeling algorithm is a set of six procedures, including *negate*, *print*, and *neighbors*, which are left for the reader to code.

	1	
2	*	3
	4	

a) four-neighborhood

1	2	3
4	*	5
6	7	8

b) eight-neighborhood

Figure 3.7: Scan-line order for returning the neighbors of a pixel.

Figure 3.8 illustrates the application of the recursive connected components algorithm to the first (top leftmost) component of the binary image of Figure 3.6.

A Row-by-Row Labeling Algorithm

The classical algorithm, deemed so because it is based on the classical connected components algorithm for graphs, was described in Rosenfeld and Pfaltz (1966). The algorithm makes two passes over the image: one pass to record equivalences and assign temporary labels and the second to replace each temporary label by the label of its equivalence class. In between the two passes, the recorded set of equivalences, stored as a binary relation, is processed to determine the equivalence classes of the relation. Since that time, the *union-find* algorithm, which dynamically constructs the equivalence classes as the equivalences are found, has been widely used in computer science applications. The union-find data structure allows efficient construction and manipulation of equivalence classes represented by tree structures. The addition of this data structure is a useful improvement to the classical algorithm.

Compute the connected components of a binary image.
B is the original binary image.
LB will be the labeled connected component image.

```
procedure recursive_connected_components(B, LB);
{
LB := negate(B);
label := 0;
find_components(LB, label);
print(LB);
}

procedure find_components(LB, label);
{
for L := 0 to MaxRow
  for P := 0 to MaxCol
    if LB[L,P] == -1 then
      {
      label := label + 1;
      search(LB, label, L, P);
      }
}

procedure search(LB, label, L, P);
{
LB[L,P] := label;
Nset := neighbors(L, P);
for each (L',P') in Nset
  {
  if LB[L',P'] == -1
  then search(LB, label, L', P');
  }
}
```

Algorithm 2: Recursive Connected Components

Step 1.

-1	-1	0	-1	-1	-1
-1	-1	0	-1	0	0
-1	-1	-1	-1	0	0

Step 2.

1	**-1**	0	-1	-1	-1
-1	-1	0	-1	0	0
-1	-1	-1	-1	0	0

Step 3.

1	1	0	-1	-1	-1
-1	-1	0	-1	0	0
-1	-1	-1	-1	0	0

Step 4.

1	1	0	-1	-1	-1
1	**-1**	0	-1	0	0
-1	-1	-1	-1	0	0

Step 5.

1	1	0	-1	-1	-1
1	1	0	-1	0	0
-1	-1	-1	-1	0	0

Figure 3.8: The first five steps of the recursive labeling algorithm applied to the first component of the binary image of Figure 3.6. The image shown is the (partially) labeled image LB. The boldface pixel of the image is the one being processed by the search procedure. Using the neighborhood orderings shown in Figure 3.7, the first unprocessed neighhbor of the boldface pixel whose value is -1 is selected at each step as the next pixel to be processed.

Union-Find Structure The purpose of the union-find data structure is to store a collection of disjoint sets and to efficiently implement the operations of *union* (merging two sets into one) and *find* (determining which set a particular element is in). Each set is stored as a tree structure in which a node of the tree represents a label and points to its one parent node. This is accomplished with only a vector array $PARENT$ whose subscripts are the set of possible labels and whose values are the labels of the parent nodes. A parent value of zero means that this node is the root of the tree. Figure 3.9 illustrates the tree structure for two sets of labels { 1,2,3,4,8 } and { 5,6,7 } . Label 3 is the parent node and set label for the first set; label 7 is the parent node and set label for the second set. The values in array $PARENT$ tell us that nodes 3 and 7 have no parents, label 2 is the parent of label 1, label 3 is the parent of labels 2, 4, and 8, and so on. Note that element 0 of the array is not used, since 0 represents the background label, and a value of 0 in the array means that a node has no parent.

The *find* procedure is given a label X and the parent array $PARENT$. It merely follows the parent pointers up the tree to find the label of the root node of the tree that X is in. The *union* procedure is given two labels X and Y and the parent array $PARENT$. It modifies the structure (if necessary) to merge the set containing X with the set containing Y. It starts at labels X and Y and follows the parent pointers up the tree until it reaches

PARENT

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
2	3	0	3	7	7	0	3

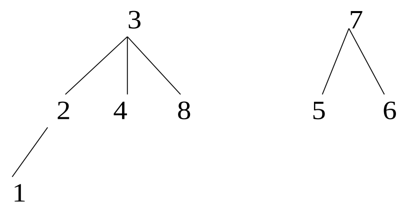


Figure 3.9: The union-find data structure for two sets of labels. The first set contains the labels { 1,2,3,4,8 } , and the second set contains labels { 5,6,7 } . For each integer label i, the value of $PARENT[i]$ is the label of the parent of i or zero if i is a root node and has no parent.

Find the parent label of a set.
X is a label of the set.
PARENT is the array containing the union-find data structure.

```
procedure find(X, PARENT);
{
j := X;
while PARENT[j] <> 0
   j := PARENT[j];
return(j);
}
```

Algorithm 3: Find

Construct the union of two sets.
X is the label of the first set.
Y is the label of the second set.
PARENT is the array containing the union-find data structure.

```
procedure union(X, Y, PARENT);
{
j := X;
k := Y;
while PARENT[j] <> 0
  j := PARENT[j];
while PARENT[k] <> 0
  k := PARENT[k];
if j <> k then PARENT[k] := j;
}
```

Algorithm 4: Union

the roots of the two sets. If the roots are not the same, one label is made the parent of the other. The procedure for *union* given here arbitrarily makes X the parent of Y. It is also possible to keep track of the set sizes and to attach the smaller set to the root of the larger set; this has the effect of keeping the tree depths down.

The Classical Connected Components Algorithm using Union-Find The union-find data structure makes the classical connected components labeling algorithm more efficient. The first pass of the algorithm performs label propagation to propagate a pixel's label to its neighbors to the right and below it. Whenever a situation arises in which two different labels can propagate to the same pixel, the smaller label propagates and each such equivalence found is entered in the union-find structure. At the end of the first pass, each equivalence class has been completely determined and has a unique label, which is the root of its tree in the union-find structure. A second pass through the image then performs a translation, assigning to each pixel the label of its equivalence class.

The procedure uses two additional utility functions: *prior_neighbors* and *labels*. The prior_neighbors function returns the set of neighboring 1-pixels above and to the left of a given one and can be coded for a 4-neighborhood (in which case the north and west neighbors are returned) or for an 8-neighborhood (in which case the northwest, north, northeast, and west neighbors are returned). The labels function returns the set of labels currently assigned to a given set of pixels.

Figure 3.10 illustrates the application of the classical algorithm with union-find to the binary image of Figure 3.6. Figure 3.10a) shows the labels for each pixel after the first pass. Figure 3.10b) shows the union-find data structure indicating that the equivalence classes determined in the first pass are $\{\{1,2\},\{3,7\},4,5,6\}$. Figure 3.10c) shows the final labeling of the image after the second pass. The connected components represent regions of the image for which both shape and intensity properties can be computed. We will discuss

Initialize the data structures for classical connected components.

```
procedure initialize();
"Initialize global variable label and array PARENT."
{
"Initialize label."
label := 0;
"Initialize the union-find structure."
for i := 1 to MaxLab
  PARENT[i] := 0;
}
```

Algorithm 5: Initialization for Classical Connected Components

some of these properties in Section 3.5.

Using Run-Length Encoding for Connected Components Labeling As introduced in Chapter 2, a *run-length encoding* of a binary image is a list of contiguous horizontal runs of 1's. For each run, the location of the starting pixel of the run and either its length or the location of its ending pixel must be recorded. Figure 3.11 shows a sample run-length data structure. Each run in the image is encoded by its starting- and ending-pixel locations. (ROW, START_COL) is the location of the starting pixel and (ROW, END_COL) is the location of the ending pixel, LABEL is the field in which the label of the connected component to which this run belongs will be stored. It is initialized to zero and assigned temporary values in pass 1 of the algorithm. At the end of pass 2, the LABEL field contains the final, permanent label of the run. This structure can then be used to output the labels back to the corresponding pixels of the output image.

3.5 Binary Image Morphology

The word *morphology* refers to form and structure; in computer vision it can be used to refer to the shape of a region. The operations of *mathematical morphology* were originally defined as set operations and shown to be useful for processing sets of 2D points. In this section, we define the operations of binary morphology and show how they can be useful in processing the regions derived from the connected components labeling operation.

3.5.1 Structuring Elements

The operations of binary morphology input a binary image B and a *structuring element* S, which is another, usually much smaller, binary image. The structuring element represents a shape; it can be of any size and have arbitrary structure that can be represented by a binary image. However, there are a number of common structuring elements such as a rectangle of

Compute the connected components of a binary image.
B is the original binary image.
LB will be the labeled connected component image.

```
procedure classical_with_union-find(B,LB);
{
“Initialize structures.”
initialize();
“Pass 1 assigns initial labels to each row L of the image.”
for L := 0 to MaxRow
  {
  “Initialize all labels on line L to zero”
  for P := 0 to MaxCol
    LB[L,P] := 0;
  “Process line L.”
  for P := 0 to MaxCol
    if B[L,P] == 1 then
      {
      A := prior_neighbors(L,P);
      if isempty(A)
      then { M := label; label := label + 1; };
      else M := min(labels(A));
      LB[L,P] := M;
      for X in labels(A) and X <> M
        union(M, X, PARENT);
      }
  }
“Pass 2 replaces Pass 1 labels with equivalence class labels.”
for L := 0 to MaxRow
  for P := 0 to MaxCol
    if B[L,P] == 1
    then LB[L,P] := find(LB[L,P],PARENT);
} ;
```

Algorithm 6: Classical Connected Components with Union-Find

Exercise 3 Labeling Algorithm Comparison

Suppose a binary image has one foreground region, a rectangle of size 1000 by 1000. How many times does the recursive algorithm look at (read or write) each pixel? How many times does the classical procedure look at each pixel?

Exercise 4 Relabeling

Because equivalent labels are merged into one equivalence class, some of the initial labels from Pass 1 are lost in Pass 2, producing a final labeling whose numeric sequence of labels often has many gaps. Write a relabeling procedure that converts the labeling to one that has a contiguous sequence of numbers from 1 to the number of components in the image.

1	1	0	2	2	2	0	3
1	1	0	2	0	2	0	3
1	1	1	1	0	0	0	3
0	0	0	0	0	0	0	3
4	4	4	4	0	5	0	3
0	0	0	4	0	5	0	3
6	6	0	4	0	0	0	3
6	6	0	4	0	7	7	3

a) after Pass 1

1	1	0	1	1	1	0	3
1	1	0	1	0	1	0	3
1	1	1	1	0	0	0	3
0	0	0	0	0	0	0	3
4	4	4	4	0	5	0	3
0	0	0	4	0	5	0	3
6	6	0	4	0	0	0	3
6	6	0	4	0	3	3	3

c) after Pass 2

PARENT

1	**2**	**3**	**4**	**5**	**6**	**7**
0	1	0	0	0	0	3

b) union-find structure showing equivalence classes

Figure 3.10: The application of the classical algorithm with the union-find data structure to the binary image of Figure 3.6:

Exercise 5 Run-Length Encoding

Design and implement a row-by-row labeling algorithm that uses the run-length encoding of a binary image instead of the image itself and uses the LABEL field of the structure to store the labels of the runs.

specified dimensions [BOX(l,w)] or a circular region of specified diameter [DISK(d)]. Some image processing packages offer a library of these primitive structuring elements. Figure 3.12 illustrates some common structuring elements and several nonstandard ones.

The purpose of the structuring elements is to act as probes of the binary image. One pixel of the structuring element is denoted as its *origin*; this is often the central pixel of a symmetric structuring element, but may in principle be any chosen pixel. Using the origin as a reference point, translations of the structuring element can be placed anywhere on the image and can be used to either enlarge a region by that shape or to check whether or not the shape fits inside a region. For example, we might want to check the size of holes by seeing if a smaller disk fits entirely within a region, while a larger disk does not.

3.5.2 Basic Operations

The basic operations of binary morphology are *dilation*, *erosion*, *closing*, and *opening*. As the names indicate, a dilation operation enlarges a region, while an erosion makes it smaller. A closing operation can close up internal holes in a region and eliminate "bays" along the boundary. An opening operation can get rid of small portions of the region that jut out from the boundary into the background region. The mathematical definitions are as follows:

	0	**1**	**2**	**3**	**4**
0	1	1	0	1	1
1	1	1	0	0	1
2	1	1	1	0	1
3	0	0	0	0	0
4	0	1	1	1	1

(a)

	ROW_START	ROW_END
0	1	2
1	3	4
2	5	6
3	0	0
4	7	7

(b)

	ROW	START_COL	END_COL	LABEL
1	0	0	1	0
2	0	3	4	0
3	1	0	1	0
4	1	4	4	0
5	2	0	2	0
6	2	4	4	0
7	4	1	4	0

(c)

Figure 3.11: Binary image (a) and its run-length encoding (b) and (c). Each run of 1's is encoded by its row (ROW) and the columns of its starting and ending points (START_COL and END_COL). In addition, for each row of the image, ROW_START points to the first run of the row and ROW_END points to the last run of the row. The LABEL field will hold the component label of the run; it is initialized to zero.

2 DEFINITION *The* **translation** X_t *of a set of pixels* X *by a position vector* t *is defined by*

$$X_t = \{\ x + t \mid x \in X\} \tag{3.1}$$

Thus the translation of a set of 1's in a binary image moves the entire set of ones by the specified amount. The translation t would be specified as an ordered pair $(\delta r, \delta c)$ where δr is the amount to move in rows and δc is the amount to move in columns.

3 DEFINITION *The* **dilation** *of binary image* B *by structuring element* S *is denoted by* $B \oplus S$ *and is defined by*

$$B \oplus S = \bigcup_{b \in B} S_b \tag{3.2}$$

This union can be thought of as a neighborhood operator. The structuring element S is swept over the image. Each time the origin of the structuring element touches a binary 1-pixel, the entire translated structuring element shape is ORed to the output image, which has been initialized to all zeros. Figure 3.13a) shows a binary image, and Figure 3.13c) illustrates its dilation by the 3 by 3 rectangular structuring element shown in Figure 3.13b).

To follow the mathematical definition, consider the first 1-pixel of the binary image B. Its coordinates are (1,0) meaning row 1, column 0 of the image. The translation $S_{(1,0)}$ means

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

a) BOX(3,5)

	1	1	1	
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
	1	1	1	

b) DISK(5)

	1	1	1	
1				1
1				1
1				1
	1	1	1	

c) RING(5)

1	1		
1	1		
1	1	1	1
1	1	1	1

d)

1	1	1	1	1	1
1		1	1		1
1		1	1		1
1		1	1		1

e)

1
1
1
1

f)

Figure 3.12: Examples of structuring elements (blanks represent 0's).

that the structuring element S is ORed into the output image so that its origin (which is its center) coincides with position (1,0). As a result of this OR, the output image (initially all 0) has 1-pixels at positions (0,0), (0,1), (1,0), (1,1), (2,0), and (2,1), which are real positions, and at positions (0,-1), (1,-1), and (2,-1), which are virtual positions and are ignored. For the next pixel (1,1) of B, the translation $S_{(1,1)}$ is added to the output by ORing in 1-pixels at positions (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2). This continues until a copy of the structuring element has been ORed into the output image for every pixel of the input image, producing the final result of Figure 3.13c).

4 Definition *The* **erosion** *of binary image B by structuring element S is denoted by $B \ominus S$ and is defined by*

$$B \ominus S = \{\ b \mid b + s \in B\ \forall s \in S\} \tag{3.3}$$

The erosion operation also sweeps the structuring element over the entire image. At each position where every 1-pixel of the structuring element covers a 1-pixel of the binary image, the binary image pixel corresponding to the origin of the structuring element is ORed to the output image. Figure 3.13d illustrates an erosion of the binary image of Figure 3.13a by the 3 by 3 rectangular structuring element.

Dilation and erosion are the most primitive operations of mathematical morphology. There are two more common operations that are composed of these two: closing and opening.

5 Definition *The* **closing** *of binary image B by structuring element S is denoted by $B \bullet S$ and is defined by*

$$B \bullet S = (B \oplus S) \ominus S \tag{3.4}$$

1	1	1	1	1	1	1	
			1	1	1	1	
			1	1	1	1	
		1	1	1	1	1	
			1	1	1	1	
		1	1				

a) Binary image B

1	1	1
1	**1**	1
1	1	1

b) Structuring Element S

1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1
	1	1	1	1	1	1	1
	1	1	1	1	1	1	1
	1	1	1	1	1	1	1
	1	1	1	1			

c) Dilation $B \oplus S$

				1	1		
				1	1		
				1	1		

d) Erosion $B \ominus S$

	1	1	1	1	1	1	
		1	1	1	1	1	
		1	1	1	1	1	
		1	1	1	1	1	
		1	1	1	1	1	
		1	1				

e) Closing $B \bullet S$

			1	1	1	1	
			1	1	1	1	
			1	1	1	1	
			1	1	1	1	
			1	1	1	1	

f) Opening $B \circ S$

Figure 3.13: The basic operations of binary morphology. Foreground pixels are shown as 1's. Background pixels, whose value is 0, are shown as blanks.

6 Definition *The* **opening** *of binary image B by structuring element S is denoted $B \circ S$ and is defined by*

$$B \circ S = (B \ominus S) \oplus S \tag{3.5}$$

Figure 3.13e illustrates the closing of the binary image of Figure 3.13a by the 3 by 3 rectangular structuring element; Figure 3.13f illustrates the opening of the binary image by the same structuring element.

Exercise 6 Using elementary operations of binary morphology

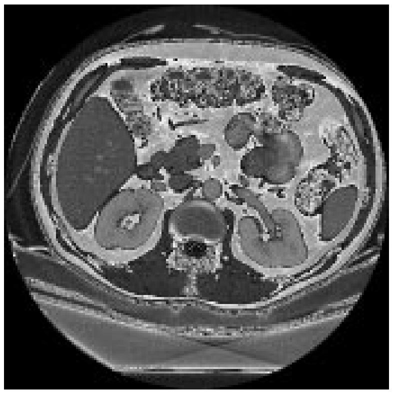
A camera takes an image I of a penny, a dime, and a quarter lying on a white background and not touching one another. Thresholding is used successfully to create a binary image B with 1 bits for the coin regions and 0 bits for the background. You are given the known diameters of the coins D_P, D_D, and D_Q. Using the operations of mathematical morphology (dilation, erosion, opening, closing) and the logical operators AND, OR, NOT, and MINUS (set difference), show how to produce three binary output images: P, D, and Q. P should contain just the penny (as 1 bits), D should contain just the dime, and Q should contain just the quarter.

3.5.3 Some Applications of Binary Morphology

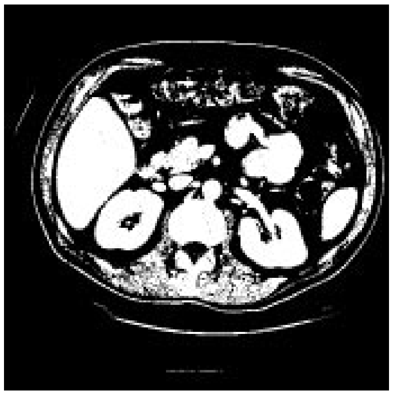
Closings and openings are useful in imaging applications where thresholding, or some other initial process, produces a binary image with tiny holes in the connected components or with a pair of components that should be separate joined by a thin region of foreground pixels. Figure 3.14a is a 512 $\times$ 512 16-bit gray-scale medical image, Figure 3.14b is the result of thresholding to select pixels with gray tones above 1070, and Figure 3.14c is the result of performing an opening operation to separate the organs and a closing to get rid of small holes. The structuring element used in the opening was DISK(13), and the structuring element used in the closing was DISK(2).

Binary morphology can also be used to perform very specific inspection tasks in industrial machine vision. Sternberg (1985) showed how a watch gear could be inspected to check whether it had any missing or broken teeth. Figure 3.15a shows a binary image of a watch gear. The watch gear has four holes inside of the main object and is surrounded by a number of teeth, which are individually visible in the image. In order to process the watch gear images, Sternberg defined several special purpose structuring elements whose shapes and sizes were derived from the physical properties of the watch gear. The following structuring elements are used in the watch-gear inspection algorithm:

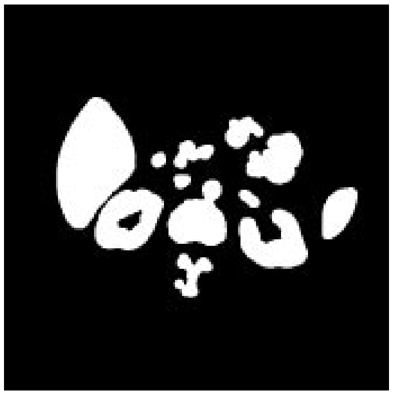
- **hole_ring:** a ring of pixels whose diameter is slightly larger than the diameters of the four holes in the watch gears. It fits just around these holes and can be used to mark a few pixels at their centers.
- **hole_mask:** an octagon that is slightly larger than the holes in the watch gears.
- **gear_body:** a disk structuring element that is as big as the gear minus its teeth.
- **sampling_ring_spacer:** a disk structuring element that is used to move slightly outward from the gear body.



a) Medical image G



b) Thresholded image B



c) Result of morphological operations

Figure 3.14: Use of binary morphology in medical imaging. The 512 $\times$ 512 16-bit medical image shown in a) is thresholded (at 1070) to produce the binary image shown in b). Opening with a DISK(13) structuring element and closing with a DISK(2) gives the results shown in c).

- **sampling_ring_width:** a disk structuring element that is used to dilate outward to the tips of the teeth.
- **tip_spacing:** a disk structuring element whose diameter spans the tip-to-tip space between teeth.
- **defect_cue:** a disk structuring element whose purpose is to dilate defects in order to show them to the user.

Figure 3.15 illustrates the gear-tooth inspection procedure. Figure 3.15a shows the original binary image to be inspected. Figure 3.15b shows the result of eroding the original image with the hole_ring structuring element. The result image has 1 pixels in a tiny cluster in the center of each hole. These are the only pixel locations where the hole_ring structuring element completely overlapped the object region. Figure 3.15c shows the result of dilating the previous image with structuring element hole_mask. The result here is four octagons covering the original four holes. Figure 3.15d shows the result of ORing the four octagons into the original binary image. The result is the gear tooth with the four holes filled in.

The next step is to produce a sampling ring that can be used to check the teeth. It is produced by taking the image of Figure 3.15d, opening it with structuring element gear_body to get rid of the teeth, dilating that with structuring element sampling_ring_spacer to bring it out to the base of the teeth, dilating that with the structuring element sampling_ring_width to bring the next image out to the tip of the teeth, and subtracting second to the last result from the last result to get a ring that just fits over the teeth. The sampling ring is shown in Figure 3.15e.

Once we have the sampling ring, it is ANDed with the original image to produce an image of just the teeth, as shown in Figure 3.15f. The gaps are already visible, but not marked. Dilating the teeth image with the structuring element tip_spacing produces the solid ring image shown in Figure 3.15g which has spaces in the solid ring wherever there are defects in the teeth. Subtracting this result from the sampling ring leaves only the defects, which are dilated by structuring element defect_cue and shown to the user as large blobs on the screen.

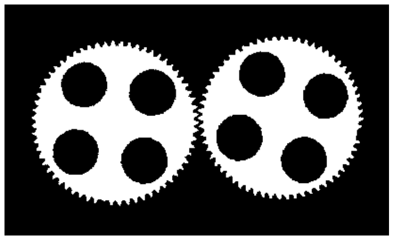
Exercise 7 Structuring element choices

Sternberg used a ring structuring element to detect the centers of the holes in the gear-tooth inspection task. If your system only supports disk and box structuring elements, what can you do to detect the centers of the holes?

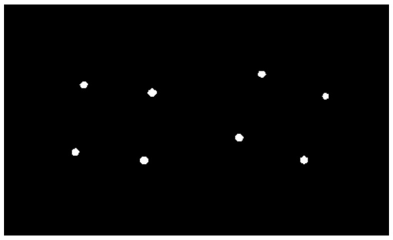
Exercise 8 Morphological processing application

Suppose a satellite image of a region can be thresholded so that the water pixels are 1's. However, bridges across reivers produce thin lines of 0's cutting across the river regions. a) Describe how to restore the bridge pixels to the water region. b) Describe how to detect the thin bridges as separate objects.

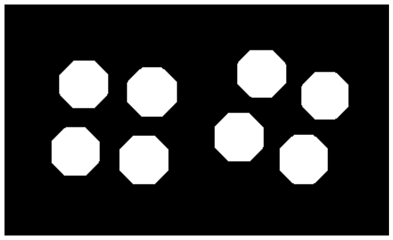
Binary morphology can also be used to extract primitive features of an object that can be used to recognize the object. For instance, the corners of flat two-dimensional objects can be good primitives in shape recognition. If an object with sharp corners is opened



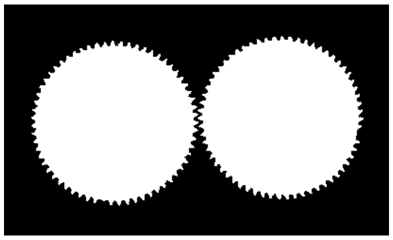
a) original image B



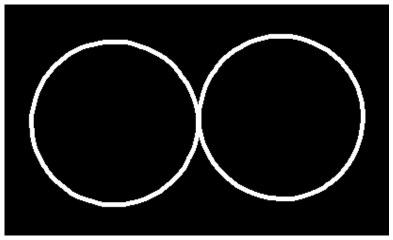
b) B1 = B $\ominus$ hole_ring



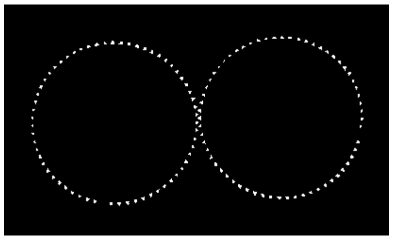
c) B2 = B1 $\oplus$ hole_mask



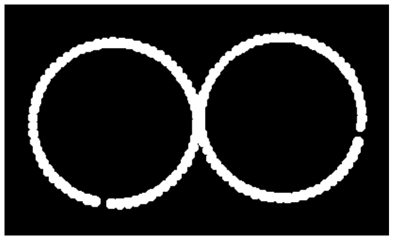
d) B3 = B OR B2



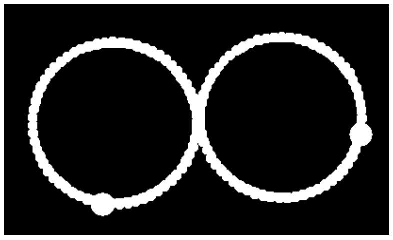
e) B7 (see text)



f) B8 = B AND B7



g) B9 = B8 $\oplus$ tip_spacing



h) RESULT = ((B7 - B9) $\oplus$ defect_cue) OR B9

Figure 3.15: The gear-tooth inspection procedure (courtesy of Stanley R. Sternberg with permission of Academic Press).

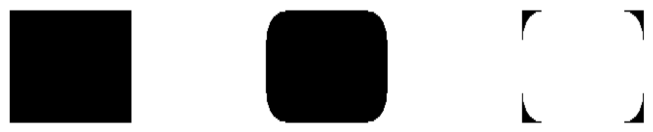


Figure 3.16: The use of binary morphology to extract shape primitives.

with a disk structuring element, the corners are chopped off as shown in Figure 3.16. If the resultant opening is subtracted from the original binary image of the shape, only the corners remain and can be used in a structural recognition algorithm. A shape matching system can use morphological feature detection to rapidly detect primitives that are useful in object recognition.

3.5.4 Conditional Dilation

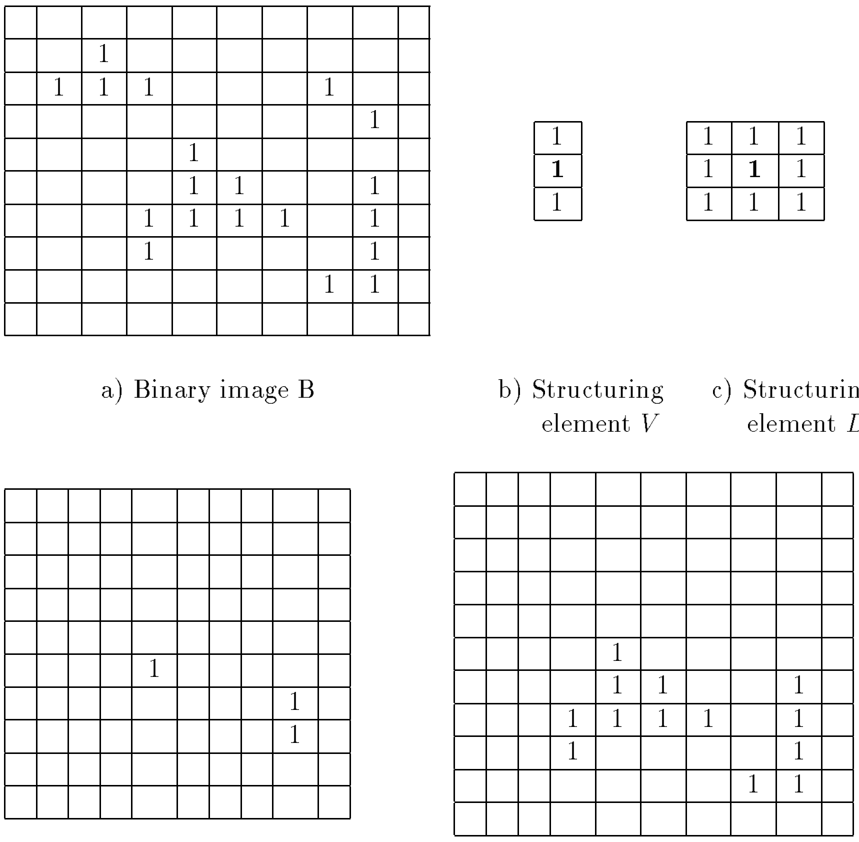
One use of binary morphology is to identify certain components of a binary image that satisfy certain shape and size constraints. It is often possible to derive a structuring element that when applied to a binary image removes the components that do not satisfy the constraints and leaves a few 1-pixels of those components that do satisfy the constraints. But we want the entire components, not just what remains of them after the erosion. The *conditional dilation* operation was defined to solve this problem.

7 DEFINITION *Given an original binary image B, a processed binary image C, and a structuring element S, let $C_0 = C$ and $C_n = (C_{n-1} \oplus S) \cap B$. The* **conditional dilation** *of C by S with respect to B is defined by*

$$C \oplus |_B \, S = C_m \tag{3.6}$$

where the index m is the smallest index satisfying $C_m = C_{m-1}$.

This definition is intended for discrete sets of points arising from finite digital images. It says that the set $C = C_0$ is repeatedly dilated by structuring element S, and each time the result is reduced to only the subset of pixels that were 1's in the original binary image B. Figure 3.17 illustrates the operation of conditional dilation. In the figure, the binary image B was eroded by structuring element V to select components in which 3-pixel long vertical edges could be found. Two of the components were selected, as shown in the result image C. In order to see these entire components, C is conditionally dilated by D with respect to the original image B to produce the results.


a) Binary image B

b) Structuring element V

c) Structuring element D

d) $C = B \ominus V$

e) $C \oplus |_B \; D$

Figure 3.17: The operation of conditional dilation.

3.6 Region Properties

Once a set of regions has been identified, the properties of the regions become the input to higher-level procedures that perform decision-making tasks such as recognition or inspection. Most image processing packages have operators that can produce a set of properties for each region. Common properties include geometric properties such as the area of the region, the centroid, the extremal points; shape properties such as measures of the circularity and elongation; and intensity properties such as mean gray tone and various texture statistics. In this section we give the definitions of some of the most useful geometric and shape properties and explain how they may be used in decision-making tasks. Gray-level properties are covered in Chapter 7 on Image Texture.

In the discussion that follows, we denote the set of pixels in a region by R. The simplest geometric properties are the region's area A and centroid $(\overline{r}, \overline{c})$. Assuming square pixels, we define these properties by

area:

$$A = \sum_{(r,c)\in R} 1 \tag{3.7}$$

which means that the area is just a count of the pixels in the region R.

centroid:

$$\bar{r} = \frac{1}{A} \sum_{(r,c)\in R} r \tag{3.8}$$

$$\bar{c} = \frac{1}{A} \sum_{(r,c)\in R} c \tag{3.9}$$

The centroid $(\bar{r}, \bar{c})$ is thus the "average" location of the pixels in the set R. Note that even though each $(r, c) \in R$ is a pair of integers, $(\bar{r}, \bar{c})$ is generally not a pair of integers; often a precision of tenths of a pixel is justifiable for the centroid.

Exercise 9 Using the area property

The gear-tooth example was designed to use only morphological and logical operations that could be rapidly executed on a specially-designed machine. Given that we are looking for larger-than-normal gaps between the teeth, how could the detection be performed in a way that minimizes the morphological operations for general purpose machines on which they do not run rapidly?

The length of the *perimeter* P of a region is another global property. A simple definition of the perimeter of a region without holes is the set of its interior border pixels. A pixel of a region is a border pixel if it has some neighboring pixel that is outside the region. When 8-connectivity is used to determine whether a pixel inside the region is connected to a pixel outside the region, the resulting set of perimeter pixels is 4-connected. When 4-connectivity is used to determine whether a pixel inside the region is connected to a pixel outside the region, the resulting set of perimeter pixels is 8-connected. This motivates the following definition for the 4-connected perimeter P_4 and the 8-connected perimeter P_8 of a region R.

perimeter:

$$\begin{aligned} P_4 &= \{(r,c) \in R | N_8(r,c) - R \neq \emptyset\} \\ P_8 &= \{(r,c) \in R | N_4(r,c) - R \neq \emptyset\} \end{aligned}$$

Exercise 10 Region from perimeter

Describe an algorithm to generate a binary image of a region without holes, given only its perimeter.

To compute length $|P|$ of perimeter P, the pixels in P must be ordered in a sequence $P =< (r_o, c_o), \ldots, (r_{K-1}, c_{K-1}) >$, each pair of successive pixels in the sequence being neighbors, including the first and last pixels. Then the *perimeter length* $|P|$ is defined by

Exercise 11 Area from perimeter

Design an algorithm to compute the area of a region without holes, given only its perimeter. Is it possible to perform the task without regenerating the binary image?

perimeter length:

$$\begin{aligned} |P| &= |\{k|(r_{k+1}, c_{k+1}) \in N_4(r_k, c_k)\}| \\ &+ \sqrt{2}|\{k|(r_{k+1}, c_{k+1}) \in N_8(r_k, c_k) - N_4(r_k, c_k)\}| \end{aligned} \tag{3.10}$$

where $k+1$ is computed modulo K, the length of the pixel sequence. Thus two vertically or horizontally adjacent pixels in the perimeter cause value 1 to be added to the total, while two diagonally adjacent pixels cause about 1.4 to be added.

With the area A and perimeter P defined, a common measure of the circularity of the region is the length of the perimeter squared divided by the area.

circularity(1):

$$C_1 = \frac{|P|^2}{A} \tag{3.11}$$

However, for digital shapes, $|P|^2/A$ assumes its smallest value not for digital circles, as it would for continuous planar shapes, but for digital octagons or diamonds depending on whether the perimeter is computed as the number of its 4-neighboring border pixels or as the length of the border, counting 1 for vertical or horizontal moves and $\sqrt{2}$ for diagonal moves. To solve this problem, Haralick (1974) proposed a second circularity measure

circularity(2):

$$C_2 = \frac{\mu_R}{\sigma_R} \tag{3.12}$$

where μ_R and σ_R are the mean and standard deviation of the distance from the centroid of the shape to the shape boundary and can be computed according to the following formulas.

mean radial distance:

$$\mu_R = \frac{1}{K} \sum_{k=0}^{K-1} ||(r_k, c_k) - (\bar{r}, \bar{c})|| \tag{3.13}$$

standard deviation of radial distance:

$$\sigma_R = (\frac{1}{K} \sum_{k=0}^{K-1} [||(r_k, c_k) - (\bar{r}, \bar{c})|| - \mu_R]^2)^{1/2} \tag{3.14}$$

where the set of pixels (r_k, c_k), $k = 0, \ldots, K-1$ lie on the perimeter P of the region. The circularity measure C_2 increases monotonically as the digital shape becomes more circular and is similar for digital and continuous shapes.

Figure 3.18 illustrates some of these basic properties on a simple labeled image having three regions: an ellipse, a rectangle, and a 3×3 square.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
2	2	2	2	0	0	0	0	0	1	1	1	1	1	1	0
2	2	2	2	0	0	0	0	1	1	1	1	1	1	1	1
2	2	2	2	0	0	0	0	1	1	1	1	1	1	1	1
2	2	2	2	0	0	0	0	1	1	1	1	1	1	1	1
2	2	2	2	0	0	0	0	0	1	1	1	1	1	1	0
2	2	2	2	0	0	0	0	0	0	1	1	1	1	0	0
2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0
2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0
2	2	2	2	0	0	3	3	3	0	0	0	0	0	0	0
2	2	2	2	0	0	3	3	3	0	0	0	0	0	0	0
2	2	2	2	0	0	3	3	3	0	0	0	0	0	0	0
2	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0

labeled connected-components image

region	**region**	**row of**	**col of**	**perim.**	**circu-**	**circu-**	**radius**	**radius**
num.	**area**	**center**	**center**	**length**	**larity$_1$**	**larity$_2$**	**mean**	**var.**
1	44	6	11.5	21.2	10.2	15.4	3.33	.05
2	48	9	1.5	28	16.3	2.5	3.80	2.28
3	9	13	7	8	7.1	5.8	1.2	0.04

properties of the three regions

Figure 3.18: Basic properties of image regions.

Exercise 12 Using properties

Suppose you have a collection of two-dimensional shapes. Some of them are triangles, some are rectangles, some are octagons, some are circles, and some are ellipses or ovals. Devise a recognition strategy for these shapes. You may use the operations of mathematical morphology and/or the properties defined so far.

bounding box and extremal points:

It is often useful to have a rough idea of where a region is in an image. One useful concept is its *bounding box*, which is a rectangle with horizontal and vertical sides that encloses the region and touches its topmost, bottommost, leftmost, and rightmost points. As shown in Fig. 3.19, there can be as many as eight distinct extremal pixels to a region: topmost right, rightmost top, rightmost bottom, bottommost right, bottommost left, leftmost bottom, leftmost top, and topmost left. Each extremal point has an extremal coordinate value in either its row or column coordinate position. Each extremal point lies on the bounding box of the region.

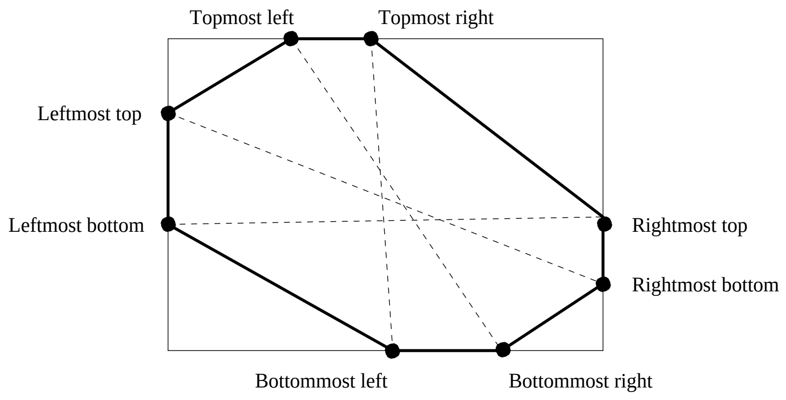


Figure 3.19: The eight extremal points of a region and the normally oriented bounding box that encloses the region. The dotted lines pair together opposite extremal points and form the extremal point axes of the shape.

Extremal points occur in opposite pairs: topmost left with bottommost right; topmost right with bottommost left; rightmost top with leftmost bottom; and rightmost bottom with leftmost top. Each pair of opposite extremal points defines an axis. Useful properties of the axis include its axis length and orientation. Because the extremal points come from a spatial digitization or quantization, the standard Euclidean distance formula will provide distances that are biased slightly low. (Consider, for example, the length covered by two pixels horizontally adjacent. From the left edge of the left pixel to the right edge of the right pixel is a length of 2 but the distance between the pixel centers is only 1.) The appropriate calculation for distance adds a small increment to the Euclidean distance to account for this. The increment depends on the orientation angle θ of the axis and is given by

$$Q(\theta) = \begin{cases} \frac{1}{|\cos\theta|} & : \ |\theta| < 45^\circ \\ \frac{1}{|\sin\theta|} & : \ |\theta| > 45^\circ \end{cases} \tag{3.15}$$

With this increment, the length of the extremal axis from extremal point (r_1, c_1) to extremal point (r_2, c_2) is

extremal axis length:

$$D = \sqrt{(r_2 - r_1)^2 + (c_2 - c_1)^2} + Q(\theta) \tag{3.16}$$

Spatial moments are often used to describe the shape of a region. There are three second order *spatial moments* of a region. They are denoted by μ_{rr}, μ_{rc}, and μ_{cc} and are defined as follows:

second-order row moment:

$$\mu_{rr} = \frac{1}{A} \sum_{(r,c)\in R} (r - \bar{r})^2 \tag{3.17}$$

second-order mixed moment:

$$\mu_{rc} = \frac{1}{A} \sum_{(r,c)\in R} (r - \bar{r})(c - \bar{c}) \tag{3.18}$$

second-order column moment:

$$\mu_{cc} = \frac{1}{A} \sum_{(r,c)\in R} (c - \bar{c})^2 \tag{3.19}$$

Thus μ_{rr} measures row variation from the row mean, μ_{cc} measures column variation from the column mean, and μ_{rc} measures row and column variation from the centroid. These quantities are often used as simple shape descriptors, as they are invariant to translation and scale change of a 2D shape.

The second spatial moments have value and meaning for a region of any shape, the same way that the covariance matrix has value and meaning for any two-dimensional probability distribution. If the region is an ellipse, there is an algebraic meaning that can be given to the second spatial moments.

If a region R is an ellipse whose center is the origin, then R can be expressed as

$$R = \{(r,\ c) \mid dr^2 + 2erc + fc^2 \leq 1\} \tag{3.20}$$

A relationship exists between the coefficients d, e, and f of the equation of the ellipse and the second moments μ_{rr}, μ_{rc}, and μ_{cc}. It is given by

$$\begin{pmatrix} d & e \\ e & f \end{pmatrix} = \frac{1}{4(\mu_{rr}\mu_{cc} - \mu_{rc}^2)} \begin{pmatrix} \mu_{cc} & -\mu_{rc} \\ -\mu_{rc} & \mu_{rr} \end{pmatrix} \tag{3.21}$$

Since the coefficients d, e, and f determine the lengths of the major and minor axes and the orientation of the ellipse, this relationship means that the second moments μ_{rr}, μ_{rc}, and μ_{cc} also determine the lengths of the major and minor axes and the orientation of the ellipse. Ellipses are frequently the result of imaging circular objects. Ellipses also provide a rough approximation to other elongated objects.

*** lengths and orientations of ellipse axes:**

To determine the lengths of the major and minor axes and their orientations from the second-order moments, we must consider the following four cases.

1. $\mu_{rc} = 0$ and $\mu_{rr} > \mu_{cc}$
   The major axis is oriented at an angle of $-90°$ counterclockwise from the column axis and has a length of $4\mu_{rr}^{1/2}$. The minor axis is oriented at an angle of $0°$ counterclockwise from the column axis and has a length of $4\mu_{cc}^{1/2}$.

2. $\mu_{rc} = 0$ and $\mu_{rr} \le \mu_{cc}$
   The major axis is oriented at an angle of $0°$ counterclockwise from the column axis and has a length of $4\mu_{cc}^{1/2}$. The minor axis is oriented at an angle of $-90°$ counterclockwise from the column axis and has a length of $4\mu_{rr}^{1/2}$.

3. $\mu_{rc} \ne 0$ and $\mu_{rr} \le \mu_{cc}$
   The major axis is oriented at an angle of

$$\tan^{-1}\left[-2\mu_{rc}\varnothing\mu_{rr} - \mu_{cc} + \left((\mu_{rr}-\mu_{cc})^2 + 4\mu_{rc}^2\right)^{1/2}\right]$$

   counterclockwise with respect to the column axis and has a length of

$$\left\{8\left(\mu_{rr} + \mu_{cc} + \left[(\mu_{rr}-\mu_{cc})^2 + 4\mu_{rc}^2\right)^{1/2}\right]\right\}^{1/2}$$

   The minor axis is oriented at an angle $90°$ counterclockwise from the major axis and has a length of

$$\left[8\left\{\mu_{rr} + \mu_{cc} - \left[(\mu_{rr}-\mu_{cc})^2 + 4\mu_{rc}^2\right]^{1/2}\right\}\right]^{1/2}$$

4. $\mu_{rc} \ne 0$ and $\mu_{rr} > \mu_{cc}$
   The major axis is oriented at an angle of

$$\tan^{-1}\frac{\left[\left\{\mu_{cc} + \mu_{rr} + \left[(\mu_{cc}-\mu_{rr})^2 + 4\mu_{rc}^2\right]^{1/2}\right)\right\}^{1/2}}{-2\mu_{rc}}$$

   counterclockwise with respect to the column axis and has a length of

$$\left[8\left\{\mu_{rr} + \mu_{cc} + \left[(\mu_{rr}-\mu_{cc})^2 + 4\mu_{rc}^2\right]^{1/2}\right\}\right]^{1/2}$$

   The minor axis is oriented at an angle of $90°$ counterclockwise from the major axis and has a length of

$$\left[8\left\{\mu_{rr} + \mu_{cc} - \left[(\mu_{rr}-\mu_{cc})^2 + 4\mu_{rc}^2\right]^{1/2}\right\}\right]^{1/2}$$

*** best axis:**

Some image regions (objects) have a natural axis; for example, a pencil or hammer, or the characters 'I', '/' and '-'. A *best axis* for an object can be computed as that axis about which the region pixels have least second moment. Using an analogy from mechanics, this

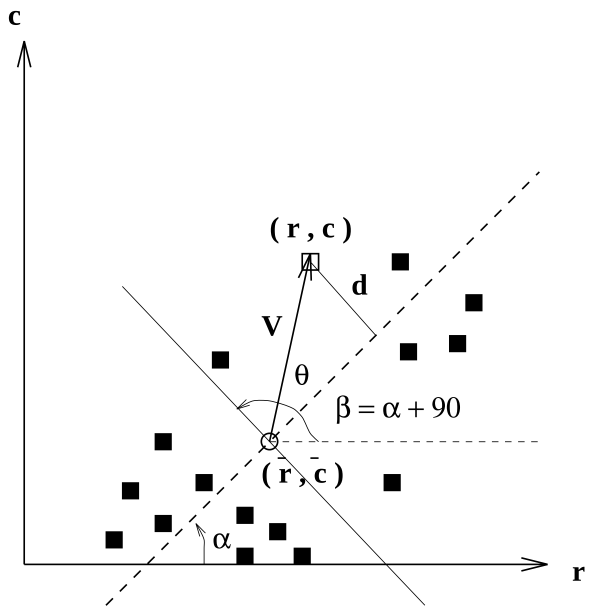


Figure 3.20: Moment about an axis is computed by summing the squared distance of each pixel from the axis.

is an axis of least inertia – an axis about which we could spin the pixels with least energy input. Note that for a circular disk, all axes have equal minimum (and maximum) inertia. It is known that an axis of least inertia must pass through the centroid $(\bar{r}, \bar{c})$ of our set of pixels (unit masses), and we will assume this here. First, we compute the second moment of a point set about an arbitrary axis; then we'll find the axis of least second moment. A set of moments about a selected set of axes might provide a good set of features for recognizing objects, as we shall see in the next chapter. For example, the second moment of character 'I' about a vertical axis through its centroid is very small, whereas that of the character '/' or '-' is not small.

Figure 3.20 shows a set of pixels and an axis making angle α with the row axis. The angle $\beta = \alpha + 90$ is the angle that a perpendicular to the axis makes with the row axis. To compute the second moment of the point set about the axis, we need to sum the squares of the distances d for all pixels: we normalize by the number of pixels to obtain a feature that does not change significantly with the number of pixels making up the shape. Note that, since we are summing d^2, the angles α and β can be changed $+/- \pi$ with no change to the second moment. Equation 3.22 gives the formula for computing the second moment: $\circ$ is the vector scalar product that is used to project the vector $\bar{V}$ onto the unit vector in direction β, giving length d. Any axis can be specified by the three parameters $\bar{r}, \bar{c}$ and α.

*** second moment about axis:**

$$\begin{aligned}\mu_{\bar{r},\bar{c},\alpha} &= \frac{1}{A}\sum_{(r,c)\in R} d^2 \\ &= \frac{1}{A}\sum_{(r,c)\in R} (\bar{V}\circ(\cos\beta,\sin\beta))^2 \\ &= \frac{1}{A}\sum_{(r,c)\in R} ((r-\bar{r})\cos\beta + (c-\bar{c})\sin\beta)^2 \qquad (3.22)\end{aligned}$$

where $\beta = \alpha + \pi/2$.

Exercise 13 Program to compute point set features

Write a program module, or C++ class, that manages a *bag* of 2D points and provides the following functionality. A *bag* is different from a *set* in that duplicate points are allowed.

- construct an initially empty bag of 2D points (r,c)
- add point (r,c) to the bag
- compute the centroid of the current bag of points
- compute the row and column moments of the current bag of points
- compute the bounding box
- compute the best and worst axes and the second moments about them

---

Exercise 14 Program to compute features from images

After creating the feature extraction module of the previous exercise, enhance it to compute the second moments about horizontal, vertical and diagonal axes through the centroid of points. Thus, five different second moments will be available for any bag of points. Create a set of 20x20 binary images of digits from '0' to '9' for test data, or access some existing data. Write a program that scans an image of a digit and computes the five moments. Study whether or not the five moments have potential for recognizing the input digit.

The above formula can be used to compute several moments to capture some information about the shape of the point set; for example, moments about the vertical, horizontal, and diagonal axes are useful for classifying alphabetic characters in standard orientation. The least (and most) inertia is an invariant property of the point set and translates and rotates with the point set. The axis of least inertia can be obtained by minimizing $\mu_{\bar{r},\bar{c},\alpha}$. Assuming now that the best axis must pass through the centroid, we need only differentiate the formula with respect to α to determine the best $\hat{\alpha}$.

*** axis with least second moment:**

$$
\begin{aligned}
\tan 2\hat{\alpha} &= \frac{2 \sum (r-\bar{r})(c-\bar{c})}{\sum (r-\bar{r})(r-\bar{r}) - \sum (c-\bar{c})(c-\bar{c})} \\
&= \frac{\frac{1}{A}\, 2 \sum (r-\bar{r})(c-\bar{c})}{\frac{1}{A}\sum (r-\bar{r})(r-\bar{r}) - \frac{1}{A}\sum (c-\bar{c})(c-\bar{c})} \\
&= \frac{2\,\mu_{rc}}{\mu_{rr} - \mu_{cc}} \qquad (3.23)
\end{aligned}
$$

There are two extreme values for α, a minimum and a maximum, which are 90 degrees apart. We have already seen the method to distinguish the two in the above discussion about the major and minor axes of an ellipse. In fact, the above formula allows us to compute an ellipse that approximates the point set in the sense of these moments. Note that highly symmetrical objects, such as squares and circles, will cause a zero-divide in the above formula; hence the case analysis used with the elliptical data must also be done here.

Exercise 15 Compute the extremes of inertia

Differentiate the formula in Equation 3.22 and show how the best (and worst) axes are obtained in Equation 3.23.

Exercise 16 Verify that the best axis passes through the centroid

Verify that the axis of least inertia must pass though the centroid. Consult the references at the chapter's end or other references on statistical regression or mechanics; or, prove it yourself.

3.7 Region Adjacency Graphs

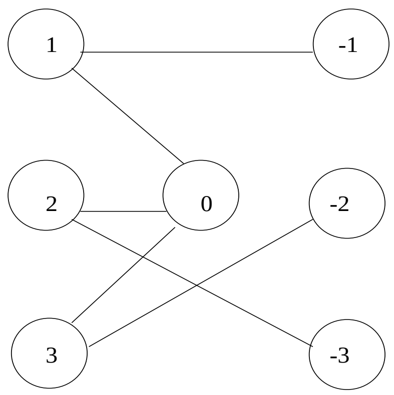
In addition to properties of single regions, relationships among groups of regions are also useful in image analysis. One of the simplest, but most useful relationships is *region adjacency.* Two regions are adjacent if a pixel of one region is a neighbor of a pixel of the second region. In binary images, there are only two kinds of regions: foreground regions and background regions. All of the foreground regions are adjacent to the background and not to one another. If the background is one single, connected region, then there is nothing further to compute. Suppose instead that the foreground regions can have holes in them, each hole belonging to the background. Applying the connected components labeling operation to the foreground pixels yields a labeled image in which the foreground regions each have a numeric label and the background regions all have label zero. But it is also possible to apply the connected components operator to the background. In this case, all the background regions can be assigned labels, too. One of these regions will be large and will start at the top left of the image. This one can be given a special label, such as 0. The rest of the background regions are the holes in the foreground regions. Given the image of foreground labels and the image of background labels, it is useful to determine which background regions are adjacent to each foreground region or vice versa. The structure for keeping track of adjacencies between pairs of regions is called a *region adjacency graph.* It can be used for keeping track of adjacencies between foreground and background regions in the binary case and for keeping track of all adjacencies in the general image segmentation case.

8 Definition *A* **region adjacency graph (RAG)** *is a graph in which each node represents a region of the image, and an edge connects two nodes if the two regions are adjacent.*

Figure 3.21 gives an example of a region adjacency graph for a binary image of foreground and background regions. The foreground regions have been labeled as usual with positive integers. The background regions have been labeled with zero for the large region that starts at the upper left pixel of the image and with negative integers for the hole regions.

0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	0	2	2	0
0	1	-1	-1	-1	1	0	2	2	0
0	1	1	1	1	1	0	2	2	0
0	0	0	0	0	0	0	2	2	0
0	3	3	3	0	2	2	2	2	0
0	3	-2	3	0	2	-3	-3	2	0
0	3	-2	3	0	2	-3	-3	2	0
0	3	3	3	0	2	2	2	2	0
0	0	0	0	0	0	0	0	0	0

a) Labeled image of foreground and background regions



b) Region adjacency graph

Figure 3.21: A labeled image and its region adjacency graph.

The algorithm for constructing a region adjacency graph is straightforward. It processes the image, looking at the current row and the one above it. It detects horizontal and vertical adjacencies, and if 8-adjacency is specified, diagonal adjacencies between points with different labels. As new adjacencies are detected, new edges are added to the region adjacency graph data structure being constructed. There are two issues related to the efficiency of this algorithm. The first is with respect to space. It is possible for an image to have tens of thousands of labels. In this case, it may not be feasible, or at least not suitable in a paging environment, to keep the entire structure in internal memory at once. The second issue relates to execution time. When moving along an image, point by point, the same adjacency (ie. the same two region labels) will be detected over and over again. It is desirable to enter the adjacency into the data structure as infrequently as possible. These issues are addressed in the Exercise.

Exercise 17 Efficient RAG construction

Design a data structure for keeping track of adjacencies while constructing a region adjacency graph. Give algorithms that construct the graph from an arbitrary labeled image and that attempt to minimize references to the data structure. Discuss how you would store the final RAG in permanent storage (on disk) and how you would handle the case where the RAG is too large to keep in internal memory during its construction.

3.8 Thresholding Gray-Scale Images

Binary images can be obtained from gray-scale images by thresholding operations. A thresholding operation chooses some of the pixels as the foreground pixels that make up the objects of interest and the rest as background pixels. Given the distribution of gray tones in a given image, certain gray-tone values can be chosen as threshold values that separate the pixels into groups. In the simplest case, a single threshold value t is chosen. All pixels whose gray-tone values are greater than or equal to t become foreground pixels and all the rest become background. This threshold operation is called *threshold above*. There are many variants including *threshold below*, which makes the pixels with values less than or equal to t the foreground; *threshold inside*, which is given a lower threshold and an upper threshold and selects pixels whose values are between the two as foreground; and *threshold outside*, which is the opposite of threshold inside. The main question associated with these simple forms of thresholding is how to choose the thresholds.

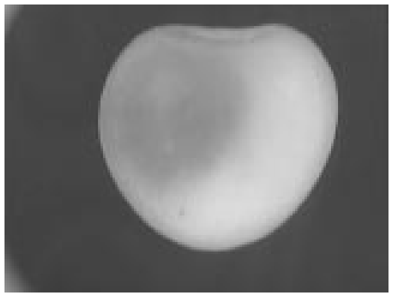
3.8.1 The Use of Histograms for Threshold Selection

Thresholds can be selected interactively by a user of an interactive package, but for image analysis processes that must run automatically, we would like to be able to compute the thresholds automatically. The basis for choosing a threshold is the *histogram* of the gray-tone image.

9 DEFINITION *The* **histogram** *h of gray-tone image I is defined by*

$$h(m) = |\{(r,c) \mid I(r,c) = m\}|,$$

where m spans the gray-level values.


a) Image of a bruised cherry

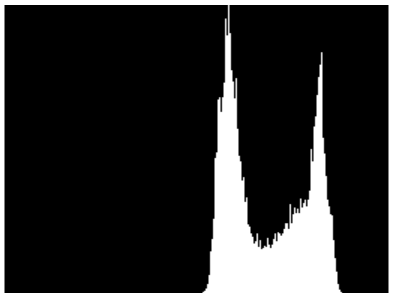

b) Histogram of the cherry image

Figure 3.22: Histogram of the image of a bruised cherry displaying two modes, one representing the bruised portion and the other the nonbruised portion.

Figure 3.22 shows the image of a bruised cherry and its histogram. The histogram has two distinct modes representing the bruised portion and nonbruised portion of the cherry.

A histogram can be computed by using an array data structure and a very simple procedure. Let H be a vector array dimensioned from 0 to MaxVal, where 0 is the value of the smallest possible gray-level value and MaxVal is the value of the largest. Let I be the two-dimensional image array with row values from 0 to MaxRow and column values from 0 to MaxCol as in the previous sections. The histogram procedure is given by the following code.

```
Compute the histogram H of gray-tone image I.

    procedure histogram(I,H);
    {
    "Initialize the bins of the histogram to zero."
    for i := 0 to MaxVal
      H[i] := 0;
    "Compute values by accumulation."
    for L := 0 to MaxRow
      for P := 0 to MaxCol
        {
        grayval := I[r,c];
        H[grayval] := H[grayval] + 1;
        } ;
    }
```

Algorithm 7: Image Histogram

This histogram procedure assumes that each possible gray tone of the image corresponds to a single bin of the histogram. Sometimes we instead want to group several gray tones into a single bin, usually for purposes of displaying the histogram when there are many possible gray tones. In this case the procedures can easily be modified to calculate the bin number as a function of the gray-tone. If *binsize* is the number of gray tones per bin, then *grayval/binsize* truncated to its integer value gives the correct bin subscript.

Given the histogram, automatic procedures can be written to detect peaks and valleys of the histogram function. The simplest case is when we are looking for a single threshold that separates the image into dark pixels and light pixels. If the distributions of dark pixels and bright pixels are widely separated, then the image histogram will be bimodal, one mode corresponding to the dark pixels and one mode corresponding to the bright pixels. With little distribution overlap, the threshold value can easily be chosen as any value in the valley between the two dominant histogram modes as shown in Figure 3.23a. However, as the distributions for the bright and dark pixels become more and more overlapped, the choice of threshold value becomes more difficult, because the valley begins to disappear as the two distributions begin to merge together as shown in Figure 3.23b.

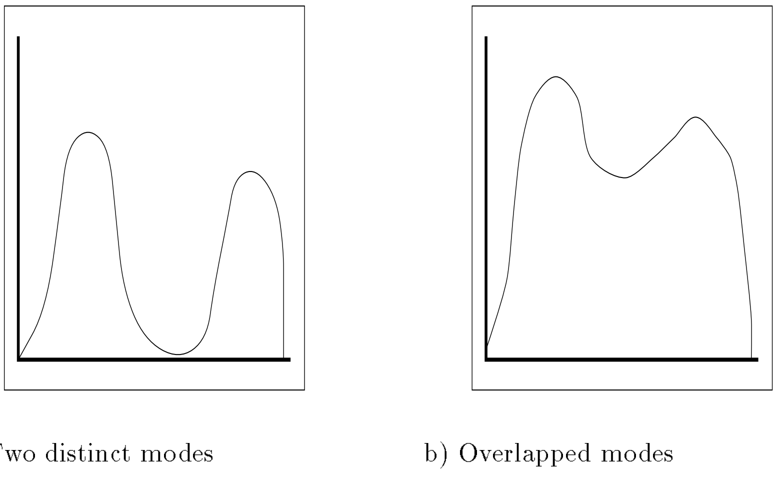


Figure 3.23: Two image histograms. The histogram on the left has two easily-separable modes; the one on the right has overlapped modes that make it more difficult to find a suitable threshold.

3.8.2 * Automatic Thresholding: the Otsu Method

Several different methods have been proposed for automatic threshold determination. We discuss here the Otsu method, which selects the threshold based on the minimization of the within-group variance of the two groups of pixels separated by the thresholding operator. For this discussion, we will specify the histogram function as a probability function P where $P(0), ..., P(I)$ represent the histogram probabilities of the observed gray values $0, ..., I$; $P(i) = |\{(r,c) \mid Image(r,c) = i\}|/|R \times C|$, where $R \times C$ is the spatial domain of the image. If the histogram is bimodal, the histogram thresholding problem is to determine a best threshold t separating the two modes of the histogram from each other. Each threshold

t determines a variance for the group of values that are less than or equal to t and a variance for the group of values greater than t. The definition for best threshold suggested by Otsu is that threshold for which the weighted sum of *within-group variances* is minimized. The weights are the probabilities of the respective groups.

We motivate the within-group variance criterion by considering the situation that sometimes happens at a ski school. A preliminary test of capabilities is given and the histogram of the resulting scores is bimodal. There are advanced skiers and novices. Lessons that are aimed at the advanced skiers go too fast for the others, and lessons that are aimed at the level of the novices are boring to the advanced skiers. To fix this situation, the teacher decides to divide the class into two mutually exclusive and homogeneous groups based on the test score. The question is to determine which test score to use as the dividing criterion. Ideally, each group should have test scores that have a unimodal bell-shaped histogram, one around a lower mean and one around a higher mean. This would indicate that each group is homogeneous within itself and different from the other.

A measure of group homogeneity is variance. A group with high homogeneity will have low variance. A group with low homogeneity will have high variance. One possible way to choose the dividing criterion is to choose a dividing score such that the resulting weighted sum of the within-group variances is minimized. This criterion emphasizes high group homogeneity. A second way to choose the dividing criterion is to choose a dividing score that maximizes the resulting squared difference between the group means. This difference is related to the between-group variance. Both dividing criteria lead to the same dividing score because the sum of the within-group variances and the between-group variances is a constant.

Let σ_W^2 be the weighted sum of group variances, that is, the *within-group variance*. Let $\sigma_1^2(t)$ be the variance for the group with values less than or equal to t and $\sigma_2^2(t)$ be the variance for the group with values greater than t. Let $q_1(t)$ be the probability for the group with values less than or equal to t and $q_2(t)$ be the probability for the group with values greater than t. Let $\mu_1(t)$ be the mean for the first group and $\mu_2(t)$ the mean for the second group. Then the within-group variance σ_W^2 is defined by

$$\sigma_W^2(t) = q_1(t)\ \sigma_1^2(t) + q_2(t)\ \sigma_2^2(t) \tag{3.24}$$

where

$$\begin{aligned} q_1(t) &= \sum_{i=1}^{t} P(i) \\ q_2(t) &= \sum_{i=t+1}^{I} P(i) \qquad (3.25) \\ \mu_1(t) &= \sum_{i=1}^{t} i\ P(i)/q_1(t) \\ \mu_2(t) &= \sum_{i=t+1}^{I} i\ P(i)/q_2(t) \qquad (3.26) \\ \sigma_1^2(t) &= \sum_{i=1}^{t} [i - \mu_1(t)]^2\ P(i)/q_1(t) \end{aligned}$$

$$\sigma_2^2(t) = \sum_{i=t+1}^{I} [i - \mu_2(t)]^2 \, P(i)/q_2(t) \tag{3.27}$$

The best threshold t can then be determined by a simple sequential search through all possible values of t to locate the threshold t that minimizes $\sigma_W^2(t)$. In many situations this can be reduced to a search between the two modes. However, identification of the modes is really equivalent to the identification of separating values between the modes.

There is a relationship between the within-group variance $\sigma_W^2(t)$ and the total variance σ^2 that does not depend on the threshold. The total variance is defined by

$$\sigma^2 = \sum_{i=1}^{I} (i - \mu)^2 P(i)$$

where

$$\mu = \sum_{i=1}^{I} i \; P(i)$$

The relationship between the total variance and the within-group variance can make the calculation of the best threshold less computationally complex. By rewriting σ^2, we have

$$\begin{aligned}
\sigma^2 &= \sum_{i=1}^{t} [i - \mu_1(t) + \mu_1(t) - \mu]^2 \; P(i) + \sum_{i=t+1}^{I} [i - \mu_2(t) + \mu_2(t) - \mu]^2 \; P(i) \\
&= \sum_{i=1}^{t} \left\{ [i - \mu_1(t)]^2 + 2[i - \mu_1(t)][\mu_1(t) - \mu] + [\mu_1(t) - \mu]^2 \right\} P(i) \\
&\quad + \sum_{i=t+1}^{I} \left\{ [i - \mu_2(t)]^2 + 2[i - \mu_2(t)][\mu_2(t) - \mu] + [\mu_2(t) - \mu]^2 \right\} P(i)
\end{aligned}$$

But

$$\begin{aligned}
\sum_{i=1}^{t} [i - \mu_1(t)][\mu_1(t) - \mu] P(i) &= 0 \quad \text{and} \\
\sum_{i=t+1}^{I} [i - \mu_2(t)][\mu_2(t) - \mu)] P(i) &= 0
\end{aligned}$$

Since

$$\begin{aligned}
q_1(t) &= \sum_{i=1}^{t} P(i) \quad \text{and} \quad q_2(t) = \sum_{i=t+1}^{I} P(i) \\
\sigma^2 &= \sum_{i=1}^{t} [i - \mu_1(t)]^2 \, P(i) + [\mu_1(t) - \mu]^2 \, q_1(t)
\end{aligned}$$

$$\begin{aligned} & + \sum_{i=t+1}^{I} [i - \mu_2(t)]^2 P(i) + [\mu_2(t) - \mu]^2 q_2(t) \\ = \quad & [q_1(t)\ \sigma_1^2(t) + q_2(t)\ \sigma_2^2(t)] \\ & + \left\{ q_1(t) [\mu_1(t) - \mu]^2 + q_2(t) [\mu_2(t) - \mu]^2 \right\} \end{aligned} \tag{3.28}$$

The first bracketed term is the within-group variance σ_W^2. It is just the sum of the weighted variances of each of the two groups. The second bracketed term is called the between-group variance σ_B^2. It is just the sum of the weighted squared distances between the means of each group and the grand mean. The between-group variance can be further simplified. Note that the grand mean μ can be written as

$$\mu = q_1(t)\ \mu_1(t) + q_2(t)\ \mu_2(t) \tag{3.29}$$

Using Eq. (3.29) to eliminate μ in Eq. (3.28), substituting $1 - q_1(t)$ for $q_2(t)$, and simplifying, we obtain

$$\sigma^2 = \sigma_W^2(t) + q_1(t)[1 - q_1(t)]\,[\mu_1(t) - \mu_2(t)]^2$$

Since the total variance σ^2 does not depend on t, the t minimizing $\sigma_W^2(t)$ will be the t maximizing the between group variance $\sigma_B^2(t)$,

$$\sigma_B^2(t) = q_1(t)\,[1 - q_1(t)]\,[\mu_1(t) - \mu_2(t)]^2 \tag{3.30}$$

To determine the maximizing t for $\sigma_B^2(t)$, the quantities determined by Eqs. (3.25) to (3.27) all have to be determined. However, this need not be done independently for each t. There is a relationship between the value computed for t and that computed for the next $t : t+1$. We have directly from Eq. (3.25) the recursive relationship

$$q_1(t+1) = q_1(t) + P(t+1) \tag{3.31}$$

with initial value $q_1(1) = P(1)$.

From Eq. (3.26) we obtain the recursive relation

$$\mu_1(t+1) = \frac{q_1(t)\ \mu_1(t) + (t+1)P(t+1)}{q_1(t+1)} \tag{3.32}$$

with the initial value $\mu_1(0) = 0$. Finally, from Eq. (3.29) we have

$$\mu_2(t+1) = \frac{\mu - q_1(t+1)\ \mu_1(t+1)}{1 - q_1(t+1)} \tag{3.33}$$

Automatic threshold-finding algorithms only work well when the images to be thresholded satisfy their assumptions about the distribution of the gray-tone values over the image. The Otsu automatic threshold finder assumes a bimodal distribution of gray-tone values. If the image approximately fits this constraint, it will do a good job. If the image is not at all bimodal, the results are not likely to be useful. Figure 3.24 illustrates the application of the Otsu operator to the gray-tone image of some toy blocks shown in a). The operator returned a threshold of 93 from the possible range of 0 to 255. The pixels below and above

a) original image b) pixels below 93 c) pixels above 93

Figure 3.24: A gray-tone image and the pixels below and above the threshold of 93 (shown in white) found by the Otsu automatic thresholding operator.

the threshold are shown in b) and c), respectively. Only the very dark regions of the image have been isolated.

If the gray-tone values of an image are strongly dependent on the location within the image, for example lighter in the upper left corner and darker in the lower right, then it may be more appropriate to use local instead of global thresholds. This idea is sometimes called *dynamic* thresholding. In some applications, the approximate shapes and sizes of the objects to be found are known in advance. In this case a technique called ***knowledge-based*** thresholding, which evaluates the resultant regions and chooses the threshold that provides the best results, can be employed. Finally, some images are just not thresholdable, and alternate techniques must be used to find the objects in them.

Exercise 18 Automatic threshold determination

Write a program to implement the Otsu automatic threshold finder. Try the program on several different types of scanned images.

3.9 References

There are a number of different algorithms for the connected components labeling operation, each designed to address a certain task. Tanimoto (1990) assumes that the entire image can fit in memory and employs a simple, recursive algorithm that works on one component at a time, but can move all over the image. Other algorithms were designed for larger images that may not fit in memory and work on only two rows of the image at a time. Rosenfeld and Pfalz (1966) developed the two-pass algorithm that uses a global equivalence table and is sometimes called the 'classical' connected components algorithm. Lumia, Shapiro, and Zuniga (1983) developed another two-pass algorithm that uses a local equivalence table to avoid paging problems. Danielsson and Tanimoto (1983) designed an algorithm for massively parallel machines that uses a parallel propagation strategy. Any algorithms that keep track of equivalences can use the union-find data structure (Tarjan, 1975) to efficiently perform set-union operations.

Serra (1982) produced the first systematic theoretical treatment of mathematical morphology. Sternberg (1985) designed a parallel pipeline architecture for rapidly performing the operations and applied it to problems in medical imaging and industrial machine vision. He also extended the binary morphology operations to gray-scale morphology (1986), which has become a standard image filtering operation. Haralick, Sternberg, and Zhuang (1987) published a tutorial paper on both binary and gray-scale morphology that has helped to show their value to the computer vision community. Shapiro, MacDonald, and Sternberg (1987) showed that morphological feature detection can be used for object recognition.

Automatic thresholding has been addressed in a number of papers. The method described in this text is due to Otsu (1979). Other methods have been proposed by Kittler and Illingworth (1986) and by Cho, Haralick, and Yi (1989). Sahoo *et al.* (1988) give a general survey of thresholding techniques.

1. S. L. Tanimoto, *The Elements of Artificial Intelligence Using Common LISP*, W. H. Freeman and Company, New York, 1990.

2. A. Rosenfeld and J. L. Pfaltz, "Sequential Operations in Digital Picture Processing," *Journal of the Association for Computing Machinery,* Vol. 13, 1966, pp. 471–494.

3. R. Lumia, .G. Shapiro, and O. Zuniga, "A New Connected Components Algorithm for Virtual Memory Computers," *Computer Vision, Graphics, and Image Processing,* Vol. 22, 1983, pp. 287–300.

4. P.-E. Danielsson and S.L. Tanimoto, "Time Complexity for Serial and Parallel Propagation in Images," in *Architecture and Algorithms for Digital Image Processing*, A. Oosterlinck and P.-E. Danielsson (eds.), *Proceedings of the SPIE*, Vol. 435, 1983, pp. 60-67.

5. R. E. Tarjan, "Efficiency of a Good but not Linear Set Union Algorithm," *Journal of the Association for Computing Machinery*, Vol. 22, 1975, pp. 215-225.

6. J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, New York, 1982.

7. S. R. Sternberg, "An Overview of Image Algebra and Related Architectures," *Integrated Technology for Parallel Image Processing*, Academic Press, London, 1985, pp. 79-100.

8. S. R. Sternberg, "Grayscale Morphology," *Computer Vision, Graphics, and Image Processing*, Vol. 35, 1986, pp. 333-355.

9. R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image Analysis Using Mathematical Morphology," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PMI-9, 1987, pp. 523-550.

10. L. G. Shapiro, R. S. MacDonald, and S. R. Sternberg, "Ordered Structural Shape Matching with Primitive Extraction by Mathematical Morphology," *Pattern Recognition*, Vol 20, No. 1, 1987, pp. 75-90.

11. R. M. Haralick, "A Measure of Circularity of Digital Figures," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-4, 1974, pp. 394-396.

12. N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man and Cybernetics,* Vol. SMC-9, 1979, pp. 62–66.

13. J. Kittler and J. Illingworth, "Minimum Error Thresholding," *Pattern Recognition,* Vol. 19, 1986, pp. 41–47.

14. S. Cho, R.M. Haralick, and S. Yi, "Improvement of Kittler and Illingworth's Minimum Error Thresholding," *Pattern Recognition,* Vol. 22, 1989, pp. 609–617.

15. P. K. Sahoo, *et al.,* "A Survey of Thresholding Techniques," *Computer Vision, Graphics, and Image Processing,* Vol. 41, 1988, pp. 233–260.