Chapter 3

Binary Image Analysis

3.1 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 3.1a) 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 3.1b) shows the connected components labeling of the binary image of Figure 3.1a).

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

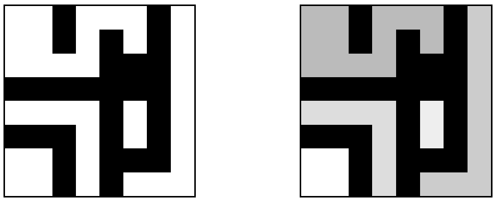
Suppose that B is a binary image with $MaxRow + 1$ rows and $MaxCol + 1$ columns. We wish to find the connected components of the 1-pixels and produce a labeled output image

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

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 3.2. 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 3.2: Scan-line order for returning the neighbors of a pixel.

Figure 3.3 illustrates the application of the recursive connected components 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 1: Recursive Connected Components to the first (top leftmost) component of the binary image of Figure 3.1.

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 3.3: The first five steps of the recursive labeling algorithm applied to the first component of the binary image of Figure 3.1. 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 3.2, the first unprocessed neighhbor of the boldface pixel whose value is -1 is selected at each step as the next pixel to be processed.

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.

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

PARENT

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

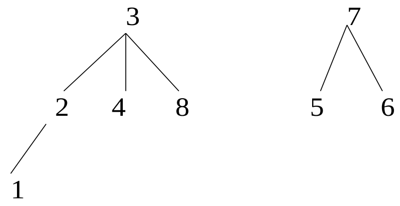


Figure 3.4: The union-find data structure for two sets of labels. The first set contains the labels { 1,2,3,4,8 } , and the second set contains labels { 5,6,7 } . For each integer label i, the value of $PARENT[i]$ is the label of the parent of i or zero if i is a root node and has no parent.

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 3.4 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.

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 2: Find

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

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 3: Union

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 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 3.5 illustrates the application of the classical algorithm with union-find to the binary image of Figure 3.1. Figure 3.5a) shows the labels for each pixel after the first pass. Figure 3.5b) shows the union-find data structure indicating that the equivalence classes

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 4: Initialization for Classical Connected Components

determined in the first pass are $\{\{1,2\},\{3,7\},4,5,6\}$. Figure 3.5c) 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 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 3.6 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.

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 5: Classical Connected Components with Union-Find

Exercise 1 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 2 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 3.5: The application of the classical algorithm with the union-find data structure to the binary image of Figure 3.1:

Exercise 3 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.

	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 3.6: 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.