## Binary (Morphological) Image Processing

For the ring of pixels on the left below, it is intuitive to say that all of the black pixels are connected, and they divide the white pixels into those *interior* to the ring, and those *exterior* to the ring. All of the white interior pixels are connected to each other. Also, all of the white exterior pixels are connected to each other.

What about the ambiguous ring on the right?
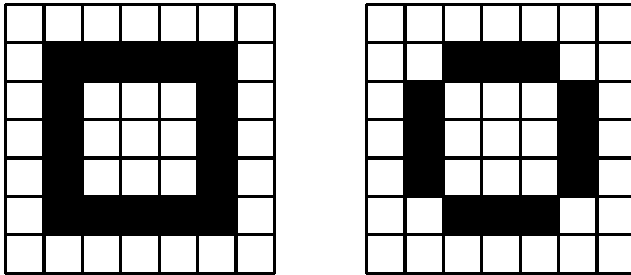
PATH: A *path* from the pixel at $[i_0, j_0]$ to the pixel at $[i_n, j_n]$ is a sequence of pixel indices $[i_0, j_0]$, $[i_1, j_1]$, ..., $[i_n, j_n]$ such that the pixel at $[i_k, j_k]$ is a neighbor of the pixel at $[i_{k+1}, j_{k+1}]$. If the neighbor relation uses 4-connection, then the path is a 4-path. For 8-connection, the path is an 8-path.

FOREGROUND: The set of all 1 pixels in an image is called the *foreground*, and is denoted S.

CONNECTED: A pixel p in the foreground is said to be *connected* to a pixel q in the foreground if there exists a path from p to q consisting entirely of pixels in the foreground.

Note that connectivity is an equivalence relation. For any 3 pixels p, q, r in S, we have the following properties:

1. Pixel p is connected to p (reflexivity)

2. If p is connected to q, then q is connected to p (commutativity)

3. If p is connected to q, and q is connected to r, then p is connected to r (transitivity)

CONNECTED COMPONENT: A set of pixels in which each pixel is connected to all other pixels is called a *connected component*.

BACKGROUND: The set of all 0 pixels can be called the background; more precisely, the background is the set of connected components of 0 pixels that include border pixels. Connected components of 0 pixels that do not include border pixels are called *holes*.

Labeling/extraction of connected components:

The machine knows the value of all pixels, but does not know which pixels belong to the same blob. To extract and label all the components, we can use a recursive or sequential algorithm.

Recursive algorithm:

1. Raster scan the image to find an unlabeled object pixel (value=1).

2. If there are no more unlabeled object pixels, then stop. Otherwise, assign it a new label L.

3. Recursively assign a label L to all its 1 neighbors.

4. Go to step 1.

The recursive algorithm is slow on a sequential processor.

Sequential algorithm:

1. Proceeding in raster scan order, find the next 1 pixel.

    (a) If only one of its upper and left neighbors has a label, then copy the label.
    (b) If both have the same label, then copy the label.
    (c) If they have different labels, then copy the upper's label, and enter the labels in the equivalence table as equivalent labels.
    (d) Otherwise, assign a new label to this pixel

2. If there are more pixels to consider, go to step 1

3. Find the lowest label for each equivalent set in the equivalence table.

4. Scan the picture, replacing each label by the lowest label in its equivalent set.

5. Renumber the equivalence table to remove gaps in the labels.

6. Rescan the image and assign the renumbered labels.

The sequential algorithm generally requires 2 passes through the image.

Hit or Miss Transformation

A small odd-sized mask (typically 3x3) is scanned over an image. If it matches the state of the pixels underneath (hit) then the output pixel in spatial correspondence with the center pixel of the mask is set to some desired state (1). If there is pattern mismatch (miss) then it is set to the opposite state (0).

The final desired output image might directly be the "hit array" showing the location of the hits, or else the hit array might be used to modify the input image.

Isolated pixel noise removal in binary images can be stated as: Search for an isolated 1 pixel surrounded by zeroes. If this is encountered, set the isolated 1 to 0. Otherwise, leave things alone.

This can be thought of in terms of the hit or miss transformation. First find the pattern. Then:
output image = (input image) AND NOT (hit array)

Matlab implementation of simple binary noise cleaning

Method 1(a): Brute force, with no padding of image boundaries

```
outim = im;
for i = 2:(N-1),
   for j = 2:(N-1),
       if im(i,j) == 1 if im(i-1,j) == 0 etc.
       outim(i,j) = 0;
       end end etc.
   end
end
```

Method 1(b): Brute force, with padding

```
[a,b] = size(im);
outim = zeros(a+2,b+2);
outim(2:(a+1),2:(b+1)) = im;
for i = 2:(a+1),
   for j = 2:(b+1),
        if im(i,j) == 1 if im(i-1,j) == 0 etc
        outim(i,j) = 0;
        end end etc.
   end
end
finalim = outim(2:(a+1),2:(b+1));
```

Method 2: Filtering. This sort of operation is often implemented by pixel stacking.

```
function out = ipclean(in)
fil = [ 1 8  64;
        2 16 128;
        4 32 256];
im2 = filter2(fil,in);
hitarray = (im2 == 16);
out = in .* ~hitarray;
```

The last two lines could have been combined into one: `out = in .* (im2 ~= 16);`

filter2 performs 2-dimensional filtering. As used here, it produces output values between 0 and 511. There is a one-to-one correspondence between the 512 possible input patterns and the 512 output numbers.

Method 3: Use a built-in command. The image processing toolbox in Matlab provides the command bwmorph, which performs a number of different operations on binary images, including isolated pixel cleaning.

```
out = bwmorph(in,'clean');
```

Method 4: Create a look-up table, and use applylut. Applylut uses the following pixel stacking

```
1  8  64                     lut = zeros(512,1);
2  16 128                    lut(17) = 1;
4  32 256                    hitarray = applylut(in,lut);
                             out = in .* ~hitarray;
```

Suppose we wanted to implement a hit-or-miss transformation for matching the pattern below, where x signifies "don't care." If the x is a zero, we need $16 + 4 + 32 + 256 = 308$, and add 1, so 309. If the x is a one, we're at 311.

3

```
0 0 0                              lut = zeros(512,1);
x 1 0                              lut(309) = 1;
1 1 1                              lut(311) = 1;
                                   hitarray = applylut(im,lut);
```

We can use the command makelut to automatically figure out the positions in the LUT that correspond to the desired pattern.

Suppose we wanted to implement a hit-or-miss transformation for matching the pattern below:

```
x x x
x 1 1
x x x
```

There are too many qualifying patterns to make the LUT by hand. So we can do:

```
f = inline('x(2,2) == 1 & x(2,3) == 1');
lut = makelut(f,3);
out = applylut(im,lut);
```

What effect does this particular operation have?