CS302 Data Structures

Spring 2012 – Dr. George Bebis

Programming Assignment 2

Due Date: 3/13/2012

In this programming assignment, you will implement a simple system to automatically count the number of regions in an image. You will apply this system to automatically count the number of galaxies in images taken by NASA's Hubble telescope (http://hubble.nasa.gov/). The output of your program should be the number of galaxies along with a labeled image (i.e., an image where each galaxy region has been assigned a different gray-value for visualization purposes). S*tacks* and *queues* will be the main data structures to be used in this assignment. Figure 1 illustrates the main steps. Specifically, the objectives of this assignment are the following:

- Improve your skills with manipulating stacks and queues (use array-based stacks/queues).
- Illustrate how to convert a recursive algorithm to an iterative one.
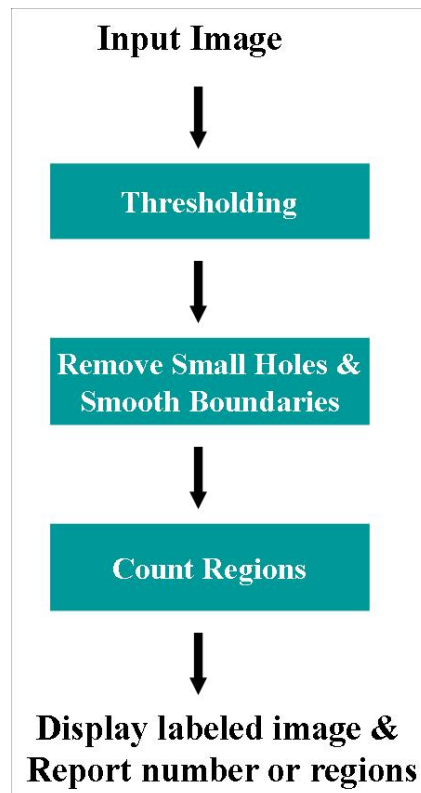- Learn more about image processing.

Figure 1. Main steps of counting the galaxies in an astronomical image.

Thresholding: The goal of thresholding is to produce a binary (black/white). This is a required step for many applications where we need to detect objects of "interest" in an image.
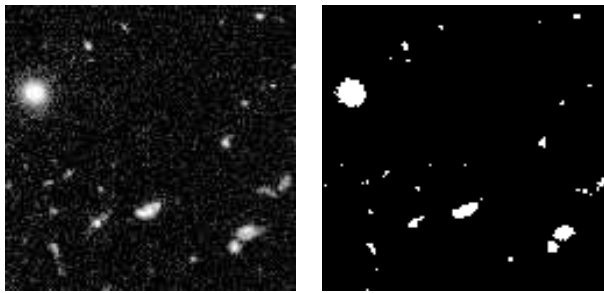
Figure 2. Original image (left) and thresholded image (right).

threshold(T) (member function): Given an input image, this function should compute a binary image of the input. This involves looking at each pixel in the input image and deciding whether to make the corresponding pixel in the output image white (255) or black (0). The decision is generally made by comparing the numeric pixel value(s) against a fixed number called a threshold. If the pixel value is less than the threshold, the pixel is set to zero; otherwise, it becomes 255. We illustrate thresholding below assuming a gray-level image:

$$O(i, j) = \begin{cases} 255 & \text{if } I(i, j) > T \\ 0 & \text{if } I(i, j) <= T \end{cases}$$

where I is the input image, O is the output (binary) image, and T is the threshold. Choosing a good threshold is important as shown in Figure 3 below.
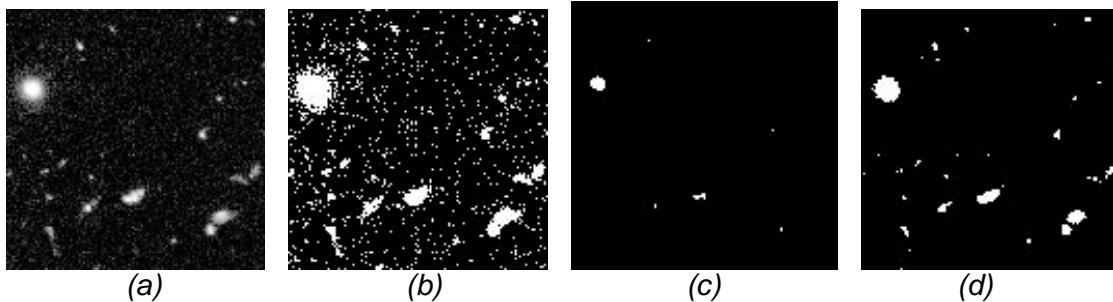

(a)   *(b)*   *(c)*   *(d)*

Figure 3. (a) original image, (b) threshold=10, (c) threshold=230, (d) threshold=128.

Holes within regions: some regions in the thresholded image might contain small holes because the pixel intensity within a region is not uniform. In such cases, we would like to fill in the holes before further processing the regions. We describe below two operators that are useful for removing small holes as well as smoothing region boundaries.

dilate() (member function): Given a binary (black/white) image, dilation performs the following operation:

$$O_d(i, j) = \begin{cases} 255 & \text{if at least one neighbor is 255} \\ I(i, j) & \text{otherwise} \end{cases}$$

Specifically, to determine the value of pixel (i,j) in the output image O_d, we consider all the neighbors of that pixel in the input image I. If all the neighbors of the (i,j) pixel are "black" (0), then we set $O_d(i,j)=I(i,j)$, otherwise, we set $O_d(i,j)=255$. The overall effect of dilation is that it expands regions. Figure 3 shows an example.
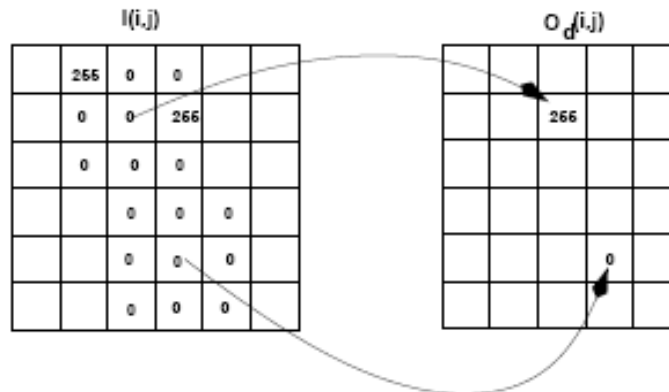
(a)   (b)   (c)

Figure 3. (a) original image, (b) thresholded image, (c) dilated image.

erode() (member function): Given a binary (black/white) image, erosion performs the following operation:

$$O_e(i, j) = \begin{cases} 0 & \text{if at lest one neighbor is 0} \\ I(i, j) & \text{otherwise} \end{cases}$$

Specifically, to determine the value of pixel (i,j) in the output image O_e, we consider all the neighbors of that pixel in the input image I. If all the neighbors of the (i,j) pixel are "white" (255), then we set $O_e(i,j)=255$, otherwise, we set $O_e(i,j)=I(i,j)$. The overall effect of erosion is that it shrinks regions. Figure 4 shows an example.
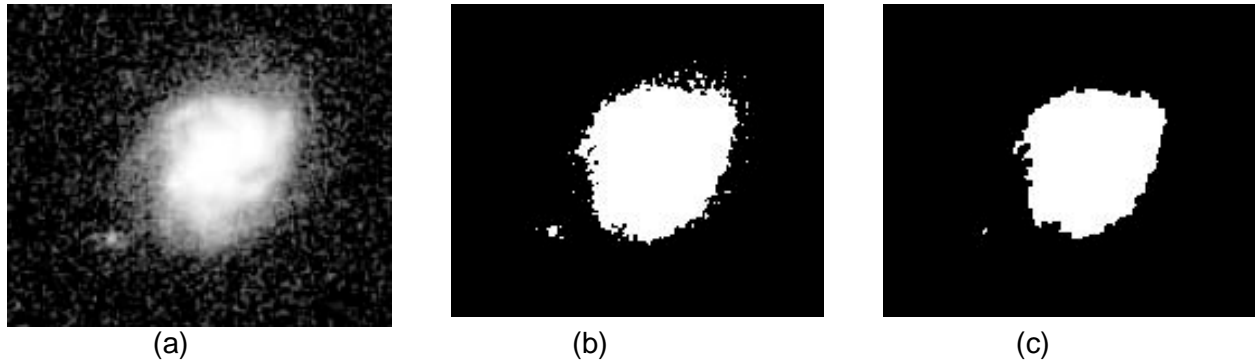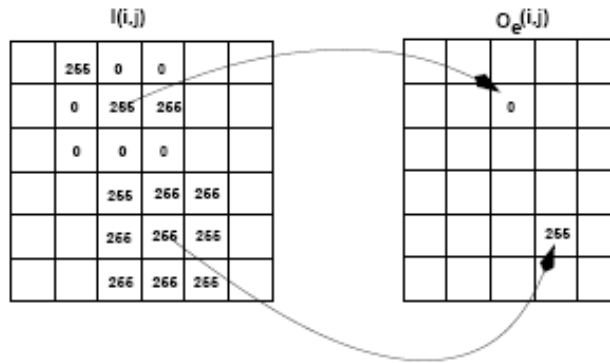
I(i,j)

255	0	0		
0	255	266		
0	0	0		
	255	266	266	
	266	266	266	
	266	266	255	

$O_e(i,j)$

(a)   (b)   (c)

Figure 4. (a) original image, (b) thresholded image, (c) eroded image.

Removing small holes and smoothing boundaries: To get rid of small holes in a region and smooth its boundary, first we apply dilation. Although dilation will eliminate small holes, it will also increase the size of a region by one layer of pixels. To reverse this effect, we apply erosion on the result of dilation. An example is show below. Please note that erosion cannot bring back the holes that were eliminated by dilation, however, larger holes that were not completely eliminated by dilation will still be present after erosion. Figure 5 shows an example.
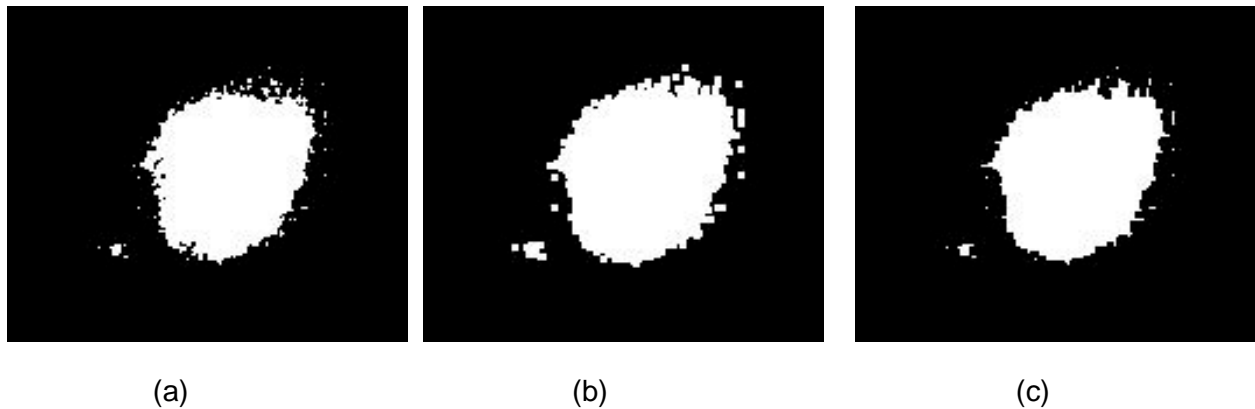

(a)   (b)   (c)

Figure 5. (a) original image, (b) dilated image, (c) eroding the dilated image.

Extracting galaxy regions (i.e., connected components labeling): The pixels in a connected component form a candidate region for representing an object. Let us consider, for example, the 16 by 16 image shown below. There are two connected components (shown in black - note that the components will actually be white in a thresholded image). You will mark these pixels in a new blank image with the component number. Thus for the image on the left, your output will be

another image with the pixels in the top-left connected component marked with a value of 1 and the pixels in the bottom-right marked with a value of 2.
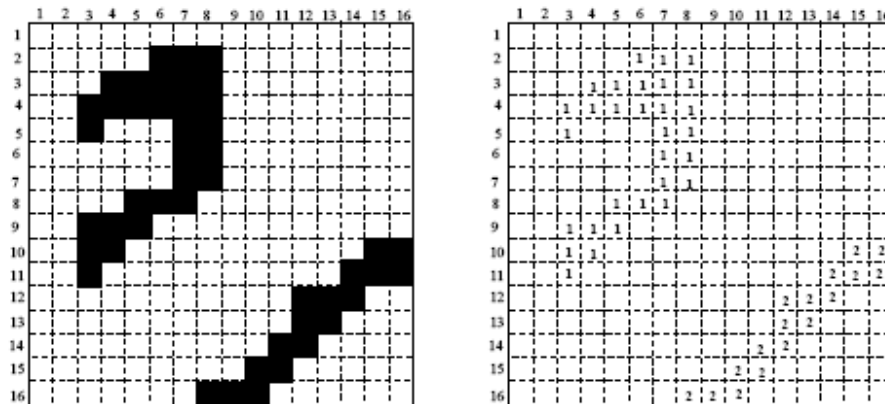

Figure 6. A simple image with two regions (left); the labeled image (right).

A connected component labeling algorithm finds all connected components in an image and assigns a unique label to all points in the same component. In many applications, it is desirable to compute characteristics (such as size, position etc.) of the components. Below is the pseudo-code of connected components algorithm.

1. Scan the thresholded image to find an unlabeled white (255) pixel and assign it a new label L.
2. Recursively assign the label L to all of its 255 neighbors.
3. Stop if there are no more unlabeled 255 pixels.
4. Go to step 1

$i-1,j-1$	$i-1,j$	$i-1,j+1$
$i,j-1$	i,j	$i,j+1$
$i+1,j-1$	$i+1,j$	$i+1,j+1$
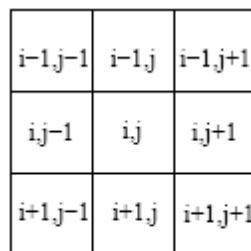
8-neighbors

Figure 7. The eight neighbors of pixel (I,j).

The neighbors of the pixel (i,j) are simply the closest pixels to it as shown in Figure 7. Below, I am providing the framework for implementing the connected components algorithm. You will need to implement two functions: **computeComponents** and **findComponent**. The function *findComponent* finds each component recursively (as described in the pseudo-code above) while the function *computeComponents* calls *findComponent* to start the recursion. *computeComponents* returns a labeled image (i.e., same as the input image but with each component labeled with a different gray-level value for visualization purposes) and the number of connected components (i.e., galaxies).

```
int computeComponents(inputImage, outputImage) // "client" function
{
 // ...
 set outputImage to white (255) // unlabeled
 connComp = 0;

 for(i=0; i<N; i++)
   for(j=0; j<M; j++)
     if(inputImage[i][j] == 255 && outputImage[i][j] == 255) {
       ++connComp;
       label = connComp; // new label

        findComponent // implement recursively ...

        // or ... implement iteratively
        // findComponentBFS(inputImage, outputImage, i, j, label);
        // findComponentDFS(inputImage, outputImage, i, j, label);
      }

  return connComp;
}
```

As we have discussed in class, recursive algorithms are not efficient when the depth of recursion is very high. Here, you will implement two iterative versions of *findComponents*. The first version is called the *Breadth-First-Search (BFS)* algorithm and uses a queue as its main data structure while the second algorithm is called the *Depth-First-Search (DFS)* and uses a stack.

findComponentBFS(inpuImage, outputImage, i, j, label) <u>(client function):</u> this function finds the connected component of *inputImage* which includes pixel (i,j) (we call it a "seed" pixel) and labels all the pixels in the same component using "label". *inputImage* needs to be a binary image (i.e., already thresholded with small holes removed using erosion and dilation).

The main data structure used by BFS is the queue. Basically, BFS uses a queue to "remember" the neighbors of a pixel (i,j) that need to be labeled in future iterations. Because of the FIFO property of queues, the pixels that will be labeled first, given (i,j), will be the closest neighbors of (i,j).In other words, BFS will first label all pixels at distance 1 from (i,j), then pixels at distance 2, distance 3, etc. The pseudo-code for BFS is given below:

```
findComponentBFS(inpuImage, outputImage, i, j, label)
{
 Queue.MakeEmpty();

 Queue.Enqueue((i,j)); //initialize Queue

 while (!Queue.IsEmpty()) {
   Queue.Dequeue((pi, pj));
   outputImage(pi,pj) = label * 10; // label this pixel
   for each neighbor (ni, nj) of (pi, pj) // Enqueue neighbors
     if (inputImage(ni, nj) == 255 and outputImage(ni, nj) == 255) {
       outputImage(ni, nj) = -1; // mark this location
       Queue.Enqueue((ni, nj));
     }
 }
}
```

findComponentDFS(inpuImage, outputImage, i, j, label) (client function): this function finds the connected component of *inputImage* which includes pixel (i,j) and labels all the pixels in the same component using "label". *inputImage* needs to be a binary image.

The main data structure used by DFS is the stack. Basically, DFS uses a stack to "remember" the neighbors of a pixel (i,j) that need to be labeled in future iterations. Because of the LIFO property of stacks, the pixels to be labeled first, given (i,j), are the most recently visited pixels of (i,j) (i.e., not necessarily the closest neighbors of (i,j)). Essentially, DFS labels pixels by following a path as deep as possible in the image. When the path ends, DFS backtracks to the most recently visited pixel. The pseudo-code for DFS is given below:

```
findComponentDFS(inpuImage, outputImage, i, j, label)
{
 Stack.MakeEmpty();

 Stack.Push((i,j)); //initialize Stack

 while (!Stack.IsEmpty()) {
   Stack.Pop((pi, pj));
   outputImage(pi,pj) = label * 10; // label this pixel
   for each neighbor (ni, nj) of (pi, pj) // Stack neighbors
     if (inputImage(ni, nj) == 255 and outputImage(ni, nj) == 255) {
       outputImage(ni, nj) = -1; // mark this location
       Stack.Push((ni, nj));
     }
 }
}
```

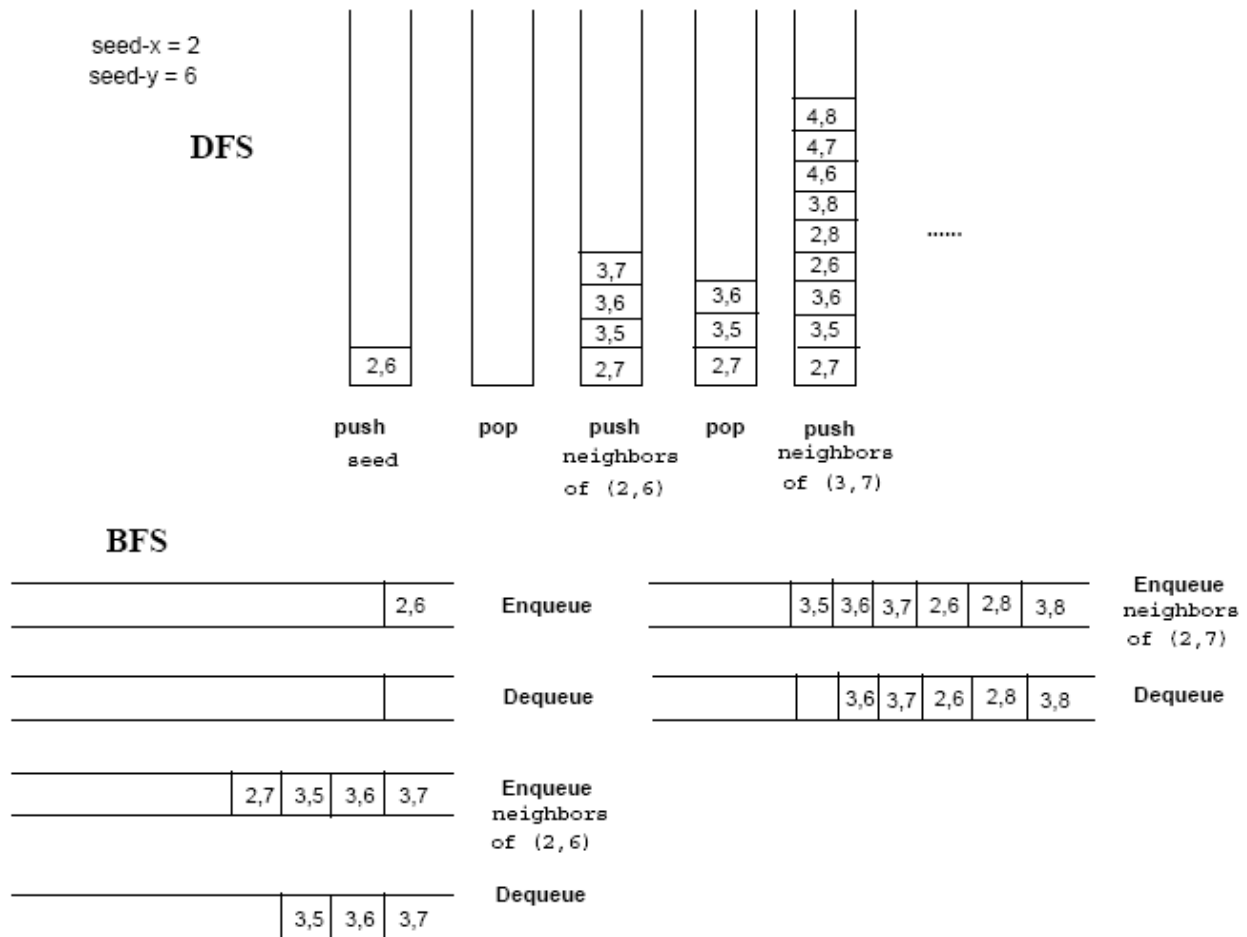An illustration of the two approaches, using Figure 8, is shown below:

Figure 8. Demonstration of BFS and DFS using the simple image shown in Figure 6.

Instructions

Each function should be discussed in a separate section with the name of the section being the same as the name of the function. Functions that you have implemented in previous assignments do not need to be described again (only if you have made significant modifications). The sections should be clearly separated from each other.

Extra Credit

(5 points) Implement "StackType" and "QueueType" as templates (i.e., manipulate objects of type "PixelType").

(5 points) Implement **findComponent** recursively. Explain your implementation in the report.