CS 308 Data Structures

Spring 2003 - Dr. George Bebis

Programming Assignment 2

Due date: 4/25/03

In the next two assignments (i.e, 2&3), you will implement a simple system to perform coin recognition from gray-level images. In assignment 2, you will implement the first part of the coin recognition system which involves counting and labeling the coin regions (no recognition to be performed yet). To implement this part, you would need to use *stacks* and *queues*. Figure 1 illustrates the main steps of the subsystem you are supposed to implement in this assignment. The output of your program should be the number of regions and the labeled image (you can assign different colors to the regions for visualization purposes). The objectives of the assignment are the following:

- Improve your understanding of recursion
- Improve your skills with manipulating stacks and queues.
- Illustrate how to convert a recursive algorithm to an iterative one.
- Learn more about image processing.
- Learn to document and describe your programs.

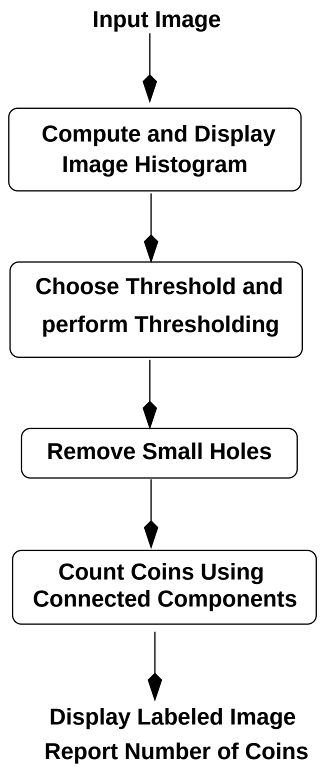

Input Image
Compute and Display Image Histogram
Choose Threshold and perform Thresholding
Remove Small Holes
Count Coins Using Connected Components
Display Labeled Image
Report Number of Coins

Figure 1. The steps for counting the coins in an image.

Thresholding: The goal of thresholding is to produce a binary (black/white) image from a gray-level (or color image). This is a required step for many applications where we need to detect objects of "interest" in an image (e.g., coins, faces).

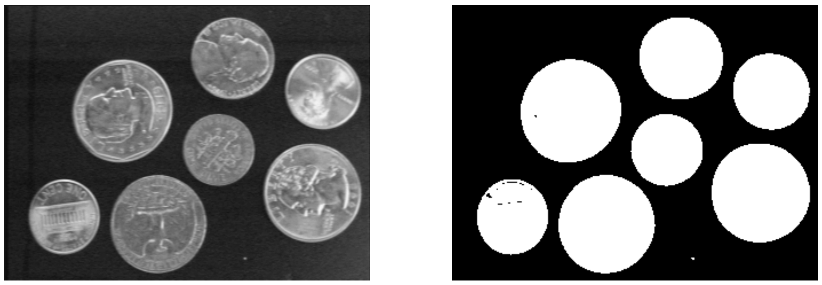


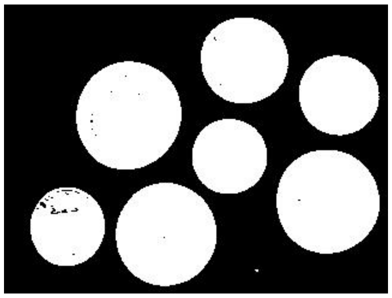
Figure 2. The original coins image (left) and its thresholded counterpart (right).

threshold(image, thresh): Given an input image (only gray-level images will be considered in this assignment), 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.

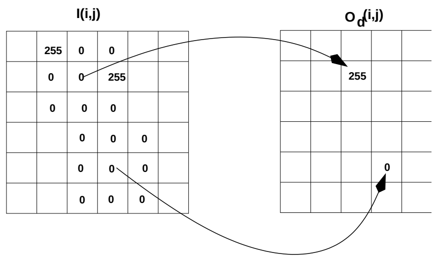
Holes within regions: In most cases when we perform thresholding, the results are not completely perfect (e.g., the regions might contain small holes because the pixel intensity within a region is not uniform). An example is shown below. In such cases, we would like to fill in the holes before extracting important features for recognition. Below, we describe two operators that will be useful for alleviating some of these problems.



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

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

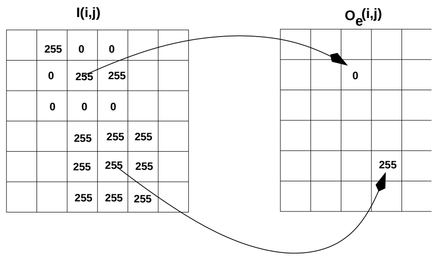
Specifically, to determine the value of pixel (i,j) in the output image O_d, we consider all the neighbors of that pixel in the input image I. If all the neighbors of the (i,j) pixel are "black" (0), then we set $O_d(i,j) = I(i,j)$, otherwise, we set $O_d(i,j) = 255$. The overall effect of dilation is that it expands regions. Implement this function as a client function.



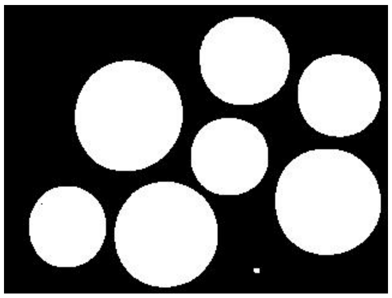
erode(image): Given a *binary* (black/white) image, erosion performs the following operation:

$$O_e(i,j) = \begin{cases} 255 & \text{if } \textit{all neighbors are } 255 \\ I(i,j) & \textit{otherwise} \end{cases}$$

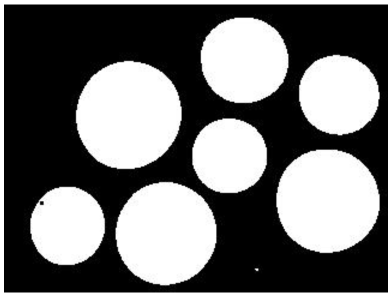
Specifically, to determine the value of pixel (i,j) in the output image O_e, we consider all the neighbors of that pixel in the input image I. If all the neighbors of the (i,j) pixel are "white" (255), then we set $O_e(i,j) = 255$, otherwise, we set $O_e(i,j) = I(i,j)$. The overall effect of erosion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



Using dilation and erosion: To get rid of small holes in a region, first we apply dilation. An example is shown below (note that most holes inside the coin regions have been removed).



Although dilation will eliminate small holes, it will also increase the size of the regions by one layer of pixels. To reverse this effect, we would have to 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. Implement this function as a client function.



displayHistogram(image): this function computes and displays the histogram of an image which is simply a bar graph of the pixel value frequencies (i.e., the number of times each value occurs in the image). Use an array of "counters" to store the pixel frequencies. The figure below illustrates a simple example. The horizontal axis corresponds to the different values in an image (e.g., for PGM images, there are 256 possible values from 0 to 255) and the vertical axis corresponds to the number of times a particular value occurs in the image.

0	0	1	0	2	0
1	0	7	7	7	0
0	7	0	0	7	0
1	0	0	7	2	0
0	0	7	1	0	1
1	0	7	7	7	0

frequencies

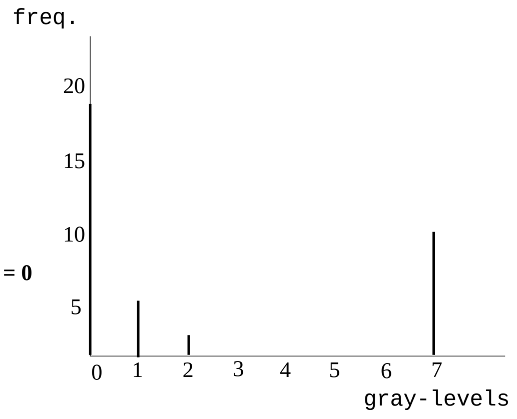
f(0)= 18

f(1)= 6

f(2)= 2

f(3)=f(4)=f(5)=f(6) = 0

f(7) = 10



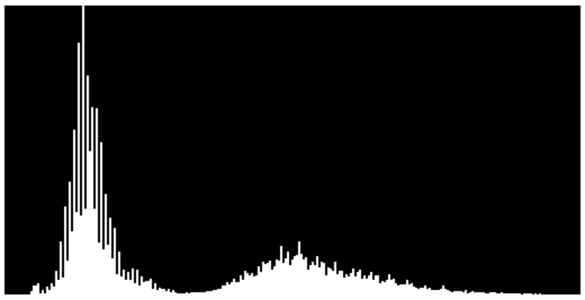


Figure 3. The histogram of the coins image (Fig2, left).

You can display the histogram as an image. First, you need to initialize the histogram image to 0. Then, you need to draw the bars (one pixel wide) one by one using the counters. I would suggest that you normalize the height of the bars before you draw the histogram image (certain counters might have pretty high values if a particular pixel value appears in the image very frequently). You can normalize the counters using the following formula:

$$\bar{c} = \frac{c}{max\_c} 500$$

where c is the original counter value, max_c is the largest counter value, and $\bar{c}$ is the normalized value. The above formula will scale every value in the interval [0, 500]. According to this normalization scheme, the number of rows of the histogram image will be 500 (500 is just an example, you can choose any other number that might be more appropriate) and the number of columns 256 (for PGM images). Implement this function as a "client" function.

Connected Components Labeling: One of the most common operations in machine vision is finding the connected components in an image. The points 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 be white in the thresholded images). 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.

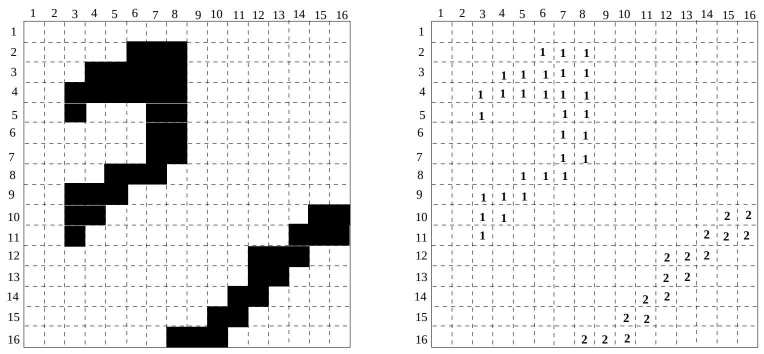


Figure 4. A simple image with two regions (left); the labeled image (right).

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. There are two algorithms for connected components: recursive and sequential. Below is the pseudo-code for the recursive connected components algorithm.

```
    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
```

The neighbors of the pixel (i, j) are simply the closest pixels to it as shown in Figure 5. Below, I have provided for you the framework for implementing the connected components algorithm.

You need to implement two functions: **connectedComponents()** and **findComponent()**. The second function (*findComponent*) is the function that finds each component recursively (as described by the pseudo-code above) while the first function (*connectedComponents*) calls the recursive function. **connectedComponents()** returns a labeled image (i.e., same as the input image but with each component labeled with a different gray-level value) and the number of connected components (regions). I will not give you the description of the parameter list of *findComponent*; it is part of the assignment to decide what you need to pass to *findComponent*.

```
int connectedComponents(inputImage, outputImage)
{
 // ...
 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(parameters); // recursive function

       // non-recursive function (see below)
       // findComponentBFS(inputImage, outputImage, i, j, label);
       // findComponentDFS(inputImage, outputImage, i, j, label);
     }

  return connComp;
}
```

As we have discussed in class, recursive implementations are not efficient when the depth of recursion is very high. Here, you will also implement two iterative versions of the same algorithm. 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.

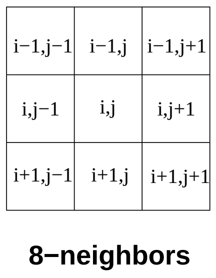


Figure 5. The eight neighbors of pixel (i, j).

findComponentBFS(inpuImage, outputImage, i, j, label): this function finds the connected component of *inputImage* which includes pixel (i,j) (we call this the "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; // label this pixel
   for each neighbor (ni, nj) of (pi, pj) // Enqueue neighbors
     if (inputImage(ni, nj) == inputImage(pi, pj) and outputImage(ni, nj) == 255) {
       outputImage(ni, nj) = -1; // mark this location
       Queue.Enqueue((ni, nj));
     }
 }
}
```

findComponentDFS(inpuImage, outputImage, i, j, label): this function finds the connected component of *inputImage* which includes pixel (i,j) (we call this the "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 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; // label this pixel
   for each neighbor (ni, nj) of (pi, pj) // Stack neighbors
     if (inputImage(ni, nj) == inputImage(pi, pj) and outputImage(ni, nj) == 255) {
       outputImage(ni, nj) = -1; // mark this location
       Stack.Push((ni, nj));
     }
 }
}
```

An illustration of the two approaches, using Figure 4, is shown below:

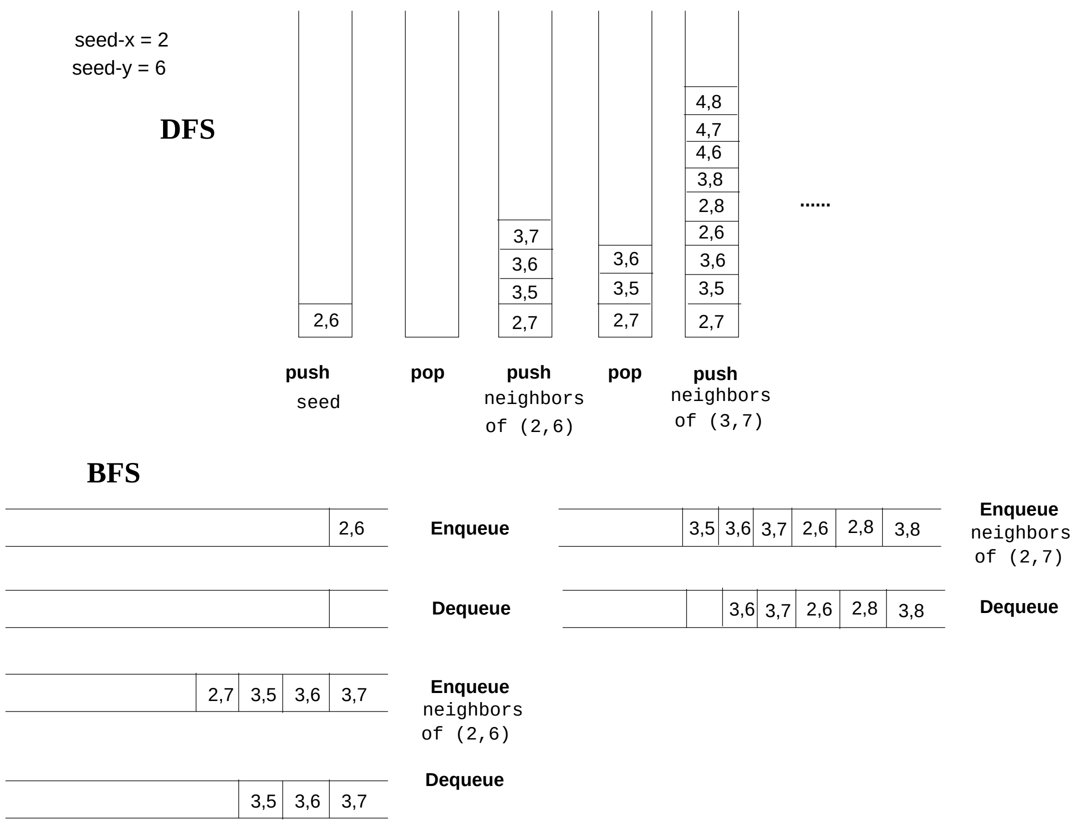


Figure 6. Demonstration of BFS and DFS using the simple image shown in Figure 4.

Instructions

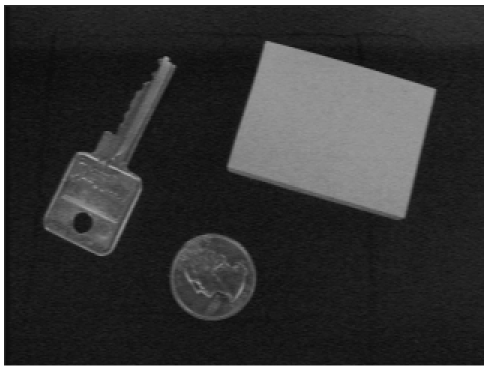
Implement the above functions as client functions. Each function should be discussed in a separate section with the name of the section being the same as the name of the function. Functions you have implemented in the previous assignments do not need to be described again here (only if you have made significant changes). The sections should be clearly separated from each other. In this assignment, your program should output the labeled image and the number of regions (coins) found. You will not do any coin-identification in this assignment (that would be the goal of the next assignment).

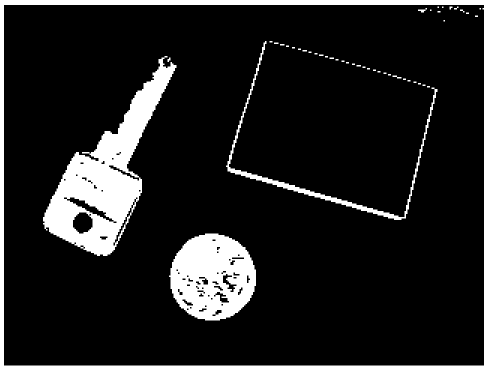
Questions

Answering the following questions does not require that you have prior knowledge of image processing. Just spend some time thinking about them and give us your best possible answers along with some justification. You do not have to do any extra coding regarding these questions but you are encouraged to do so. Interesting ideas which are implemented and demonstrated will get **extra credit** !! Make sure that you document in your report any extensions you might have made. In addition, make sure that you mention this to the TA during the demo.

1. (2pts extra) Although it is a simple matter to convert an image to a binary image using thresholding, it is much harder to do it in such a way that important information in the image is preserved. The problem is choosing the threshold value appropriately. If we choose a very high threshold, then we might miss important information. On the other hand, a very low threshold will not segment the objects properly. In general, choosing a good threshold automatically is a difficult problem, in certain cases, however, the histogram of the image can provide us with good hints for selecting good thresholds. Can you think how the image histogram can help us in this regard? You can illustrate your ideas using the coin or character images I have made available from the course's webpage.

2. (2pts extra) There are cases in image processing where, given an image containing various objects, we want to segment specific objects only. For example, consider the image shown below. Suppose that we just want to segment the key and and coin. Is there a way to segement these objects only using the thresholding function you implemented above? (try various threshold values to convince yourself). If your answer is yes, provide in your report the threshold value that does that. If your answer is no, discuss why this is not possible. Can you think of other ways to solve this problem?





3. (2pts extra) The erosion/dilation functions you implemented can only eliminate small holes in a region as it is obvious from the example I provided above. Can you suggest ways to improve these functions so that they can handle larger holes? Discuss your ideas as well as possible disadvantages.

4. (2pts extra) It is very common when we use thresholding to find some objects of interest that some regions in an image are due to noise (e.g., look carefully at the thresholded image shown in Figure 2; there is a very small region at the bottom-right of the image). Can you suggest how connected components can help us to eliminate such regions?

5. (2pts extra) The connected components algorithm you implemented in this assignment operates on binary images (i.e., thresholded) only. Do you think it is possible to generalize the connected components algorithm to gray-scale or even color images? Describe how you would do it. What would be the major difficulty in this case?

6. (2pts extra) Although you will not be doing any coin recognition in this assignment, it is a good idea to start thinking about it. How would you approach this problem? In other words, how would you recognize the coins present in an image? What features would you use for recognition? Discuss your ideas as well as possible problems (i.e., identify cases where your solution might fail).