UNIVERSITY OF ROCHESTER
CSC 249, MACHINE VISION
SPRING 2017

Homework 2

Nathan Conroy

February 8, 2017

1 CREATING A BINARY MAP

For the first section of this assignment, we were required to convert a grayscale image ("Shapes-blurred.png") containing an array of many different pixel values into a binary mapping with just two different pixel values with the goal of separating the background from the foreground. In order to do this I had to find an ideal pixel value threshold, I chose 60, and then iterate through every pixel in the image, setting values above the threshold to 255 and values at or below the threshold to 0. The resulting image was then saved as "binary_map.png".

Figure 1.1: Shapes-blurred.png

Figure 1.2: binary_map.png

As you can see, the binary map now displays what looks like 10 individual components, surrounded by the background. However, because all of the components have the same pixel value of 255, they are not yet distinguishable by the program. In order to make this happen, I implemented the two-pass connected component algorithm as described in class. The purpose of the first pass is to assign labels to pixels to represent connectivity, while the second pass is used to "clean up" the labels so that each component is uniform.

The first step of the first pass of my implementation of the algorithm involved creating an empty 2D matrix called "labels" with the same dimensions as the original image. The purpose of this matrix was to keep track of the label of each individual pixel. Pixels with the same label represent a connected component. I automatically assigned any pixel representing the background a label of 0.

In order to assign labels to the remaining pixels, I first had to cycle through all of the pixels in the binary map and find the neighboring labels of the pixel in focus. For this assignment, I used an 8-connected neighborhood, meaning that the 8 pixels to the left, right, up, down, and 4 diagonals are all recognized as neighbors. I created an array called "neighbors" to store all of the non-background, relevant neighboring labels for the pixel in focus. I use the term "relevant" because even though there are 8 neighbors, I only had to check 4 at a time. This is because as I was cycling from the top to bottom and left to right, I only needed to check the values of the four neighboring pixels that had already been cycled through. That is the pixel to the immediate left, top left, top middle, and top right. See figure 2.1.

After the neighbors array has been populated, the pixel in focus is assigned the label matching the minimum value in the neighbors array. If the neighbors array is empty (none of the relevant neighbors have the same pixel value in the original image), then a unique label is assigned to the pixel in focus.

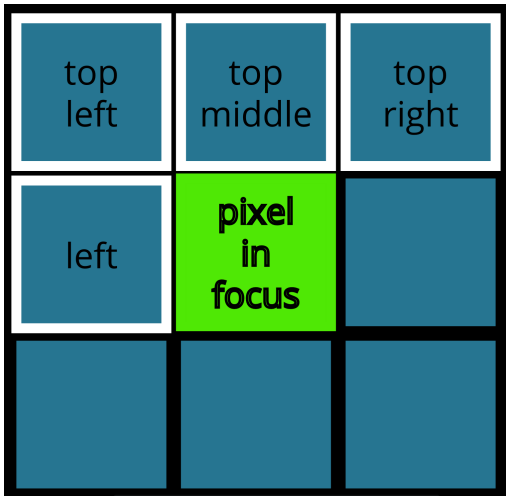

Figure 2.1: relevant neighboring pixels

The other crucial step of the first pass is marking the equivalences of labels that make up the same component. In order to do this, I created a Java HashTable, with the keys representing each of the label numbers, and the values array representing all labels that are equivalent. When the neighbors array is not empty, all of the elements of the neighbors array are added to the values of every key represented in the neighbors array. The equivalence HashTable for this example is shown in figure 2.2.

Key	Value
1	[1, 3]
2	[2, 4]
3	[1, 3]
4	[2, 4]
5	[5]
6	[6]
7	[7, 8, 9, 10]
8	[7, 8, 9, 10]

9	[7, 8, 9, 10]
10	[7, 8, 9, 10]
11	[11]
12	[12]
13	[13]
14	[14]
15	[15]

Figure 2.2: example equivalence HashTable

Figure 2.3 displays the labeled components after the first pass is complete. I used the imagesc function to display the figure using the full range of colors so the separate components are easier to distinguish. The resulting image contains 15 components, which are labeled with their corresponding pixel value. The figure is saved as "first_pass.png".

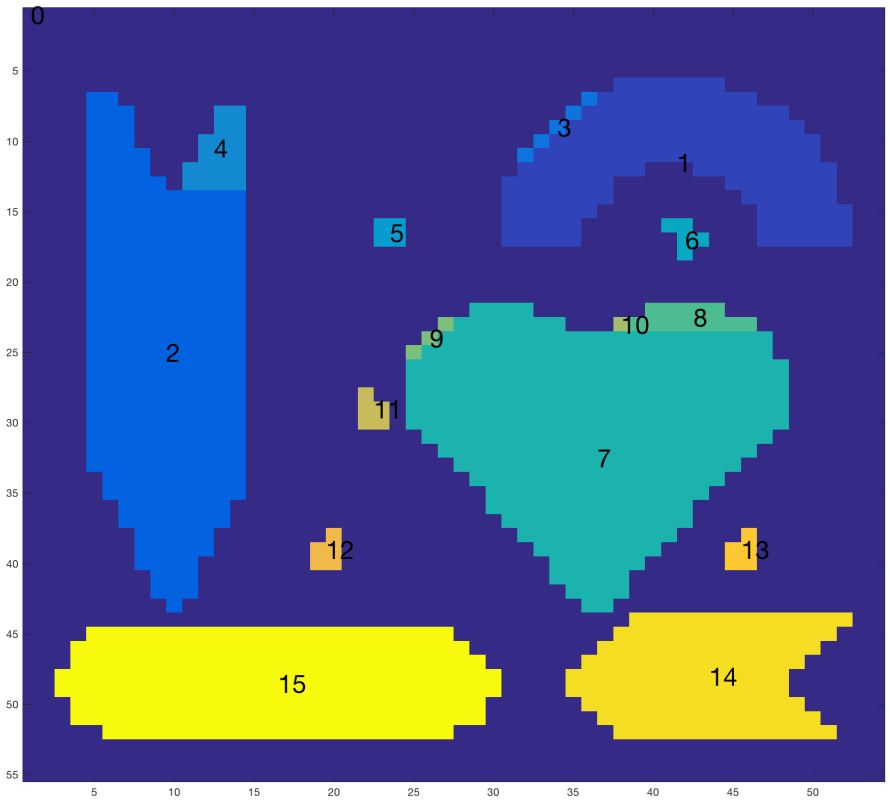

Figure 2.3: first_pass.png

The second pass of the algorithm begins by iterating through all of the non-background pixels of the image and reassigning the label to be the minimum label number within the array stored in the HashTable at the corresponding key. This makes it so that now all of the pixels within the component now have the same label. However, the labels are still not necessarily in consecutive numeric order.

To handle this, I created a function called "makeLabelsConsecutive" which once again cycles through the pixels in the image and reassigns the labels in the order that they are seen from 1 to n, where n is the number of foreground components.

The result of the second pass can be seen in figure 2.4, and is saved as "second_pass.png". The new image has only 10 components.

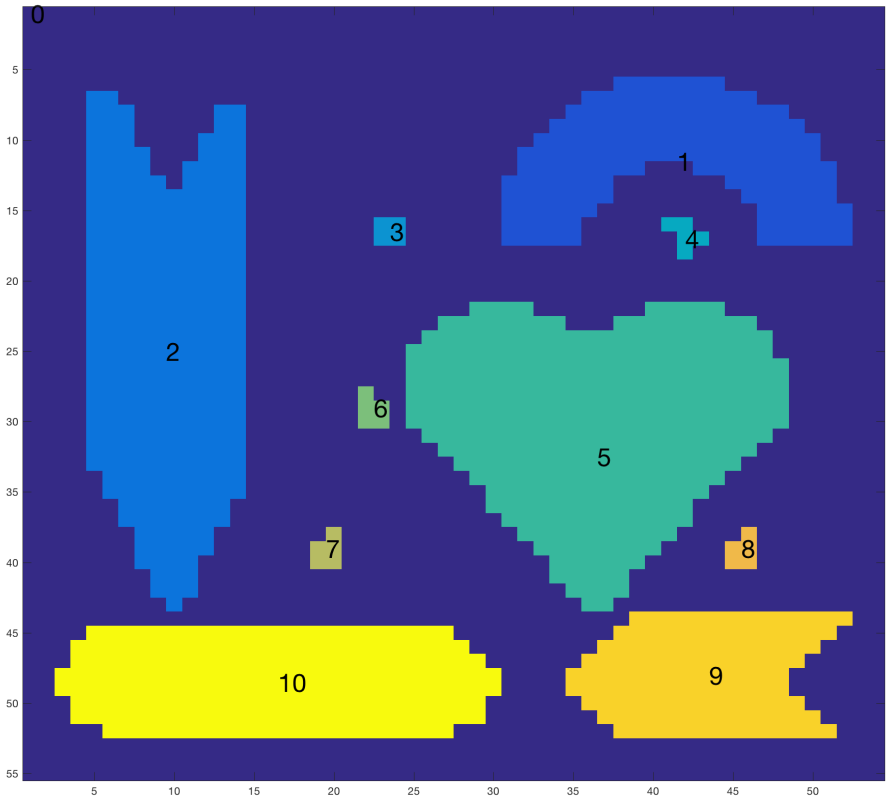

Figure 2.4: second_pass.png

3 Region Selection and Removal

Part 3 of the homework prompt asked us to remove all regions of the image with areas smaller than 10 pixels. In order to do this, I created another HashTable, with the component labels as the keys and the pixel counts as the values. I iterated through all of the pixels in the image twice, once to populate the HashTable (as seen in figure 3.1), and the second to change any pixel value with a count less than 10 to value 0 (background). I then called my "makeLabelsConsecutive" method to reassign all of the remaining components numerically consecutive labels. The resulting image can be seen in figure 3.2, and is saved as "remove_regions1.png".

Key	Value
1	163
2	295
3	4
4	5
5	341
6	5
7	5
8	5
9	126
10	204

Figure 3.1: pixel area HashTable

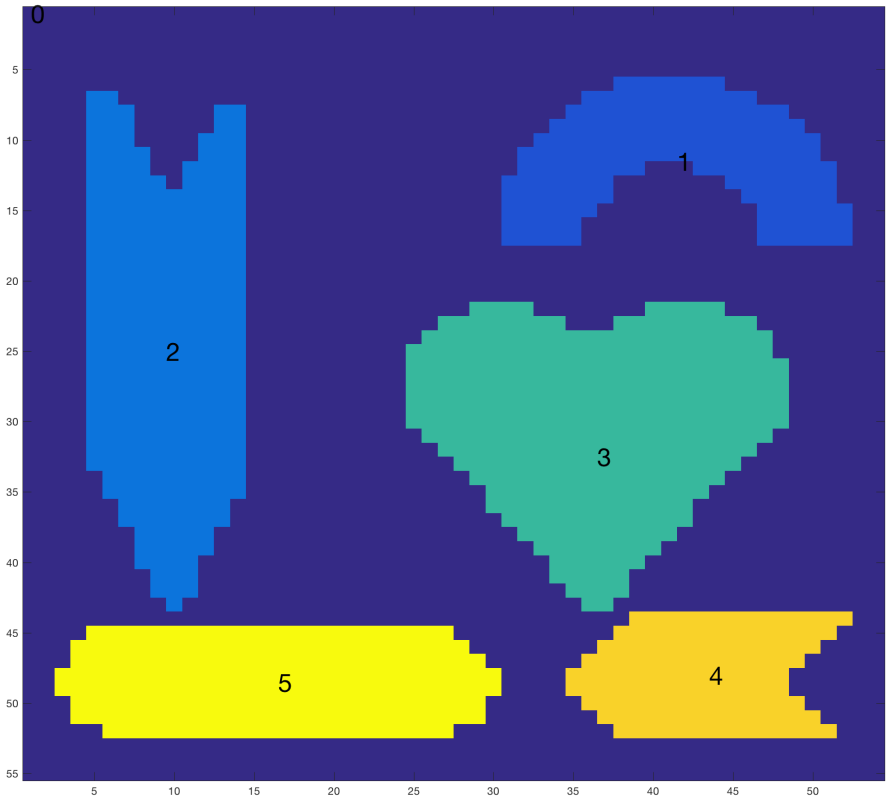

Figure 3.2: remove_regions1.png

We were also asked to remove all of the "skinny" regions from the image, as in all components with aspect ratios greater than 3:1. I did this within my "removeSkinnyRegions" method

which first loops through all of the foreground components and calls my "getBoundingBox" method which returns an array of size 4 containing the component's min X, max X, min Y, and max Y coordinates. With that information, I am easily able to calculate the height and width of the components' bounding boxes. I then iterate through all of the pixels, and set any pixel to 0 if its component's width is greater than the height multiplied by 3. I then make a call to my "makeLabelsConsecutive" method once again to correctly order the labels. The resulting image, which now only has 4 components, can be seen in figure 3.3, and is saved as "remove_regions2.png".

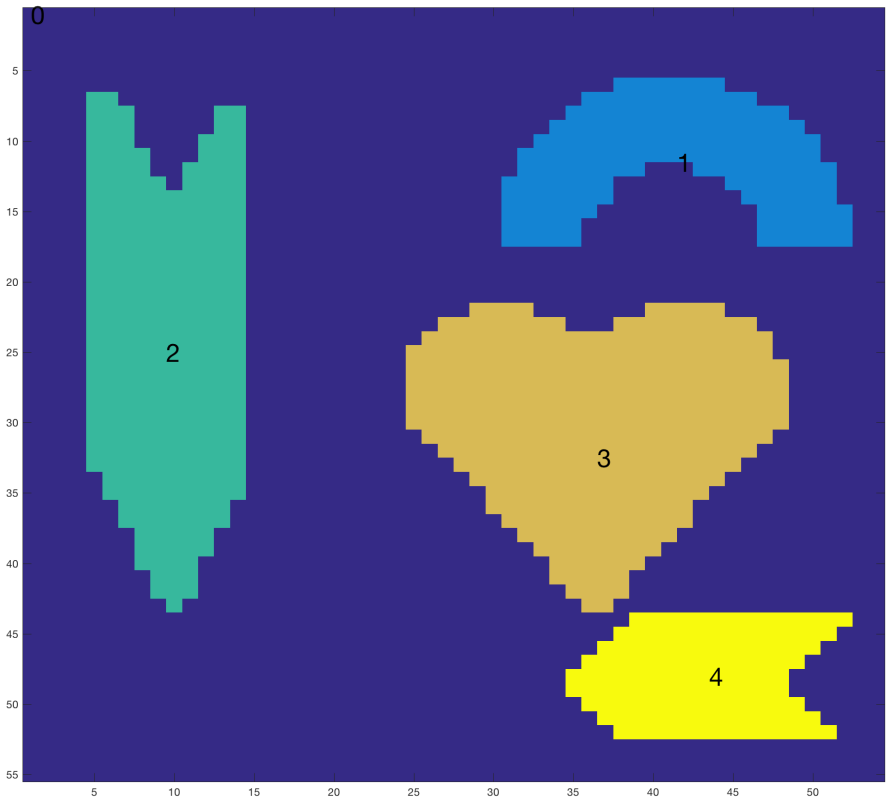

Figure 3.3: remove_regions2.png

4 Extra Credit: Finding the Heart

For the extra credit portion of this assignment, we were prompted to find the heart in the image. I decided to do so by checking the following 3 criteria:

- The lowest point (biggest y coordinate) is within 10% of the middle x coordinate of the component
- The shape has "humps" (the top row of pixels in the component are not continuous)
- The shape is relatively as tall as it is wide (within 25%)

In order to check the first criterion, I iterated through bottom row of the bounding box until I found the x coordinate of the first pixel that matched the proper label. I then checked to see if that X coordinate was greater than the component's middle X coordinate multiplied by .9, and smaller than the component's middle X coordinate multiplied by 1.1. If it passed both of those checks, the component moved on to the next criterion.

I checked the next criterion by iterating through the top row of the bounding box, adding the X coordinates to an array every time I saw a pixel in the row matching the proper label. I then cycled through the array, and checked that there were at least 2 "humps" by making sure that the elements of the array were not in continuous order. For example, if the array contained [1, 2, 3, 4] it would not pass, but if it contained [1, 2, 4, 5] it would pass.

For the last criterion, I simply used the coordinates of the bounding box to check that the width was greater than the height multiplied by .75, and less than the height multiplied by 1.25.

If the component passed all of the criteria, a label was added to the figure with the text "heart". In the case of this example, there was only one heart, and it was correctly identified by the program. The result is displayed in figure 4.1, and is saved as "extra_credit.png".

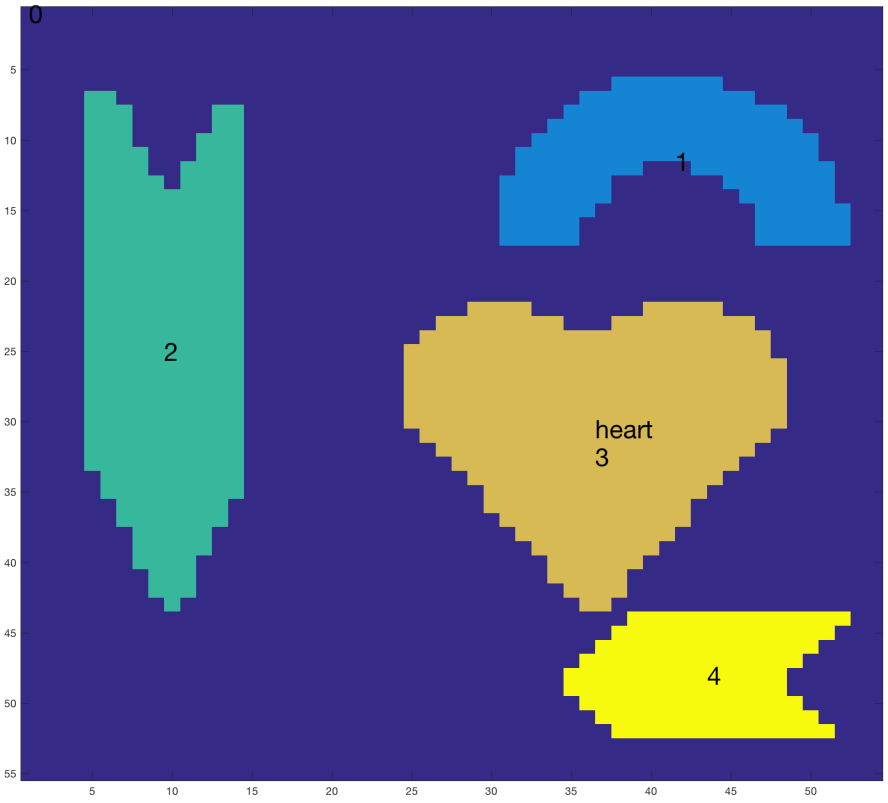

Figure 4.1: extra_credit.png