Project 2: Connected-Component Labeling and Set Operations

Austin Huyett, Kadri Nizam, Hayden Weber

September 27, 2022

Objective

- To familiarize ourselves with the algorithm of connected component labeling
- To refresh our memories on logical set operations and familiarize with its implementation

Methods

Connected-Component Labeling

The main goal of this section is to perform a connected-component labeling of the image in Figure 1. Before that, we must first convert this image, f, into a binary-valued image. This is done by defining pixels to be in the foreground (set to 1) if a pixel $p(x, y)$ is in the set $V = \{f > f_{\text{threshold}}\}$ and vice versa. The assignment gave us the prerogative to arbitrarily pick

Figure 1: File "`lenna.gif`"

a threshold value, $f_{\text{threshold}}$. However, simply choosing a number seemed too unstructured, thus we opted to instead filter based on a percentage, α, of the maximal pixel value in the image

$$f_{\text{threshold}} = \alpha \cdot \max_{\forall(x,\, y)} \{f(p(x,\, y))\}$$

where $f(p(x, y))$ is a pixel value in the image at (x, y). In our code, $\alpha = 0.6$. Doing so results in a binary-valued image as presented in Figure 2. Next we perform connected-component labeling on the converted binary image. Conveniently, one can simply use MATLAB's built-in

Figure 2: Binary-valued version of "`lenna.gif`"

`bwlabel` command to do this for us. This command (along with `label2rgb` which we use later) is only available if the "Digital Image Processing" toolbox license is owned by the user. For our own curiosity (and a fun challenge), we decided to implement this function ourselves.

Custom Implementation By default, we will use 8-connectivity in order to label the image though the method is generic and works for 4-connectivity as well. The goal is to identify connected regions in the image. Consider a pixel $p(x, y)$ from a binary image f. Trivially, we skip any processing steps if $f(p(x, y) = 0$ as the pixel is not in the foreground.

If the pixel is in the foreground, we check its neighbours for an existing label. A new label is assigned to this pixel when there are no unique labels in the neighbourhood. If only a single unique label exists, then the pixel is classified with the same label. When multiple unique labels exists, one is chosen to label the new pixel and used as a unique key that map to the others in a dictionary. We will use this dictionary later to clean up multi-labeled-connected regions. In our implementation, we picked the smallest valued label as the key. Repeat this process until all foreground pixel is assigned a label.

At this point, we are left with an image that is labeled, though some regions will have multiple labels. A second pass is then made mapping all values from the dictionary to their respective keys. This ensures that a connected-component is associated with a single label. We implemented this step so that the final labels are ordinal and increasing in value. Our implementation can be found in the `my_bwlabel.m` file.

With the regions now labeled, we visualize them into an RGB coloured image using MATLAB's built-in `label2rgb` function, mapping differet labels to different hues, saturation, and values. The result is demonstrated in Figure 3.

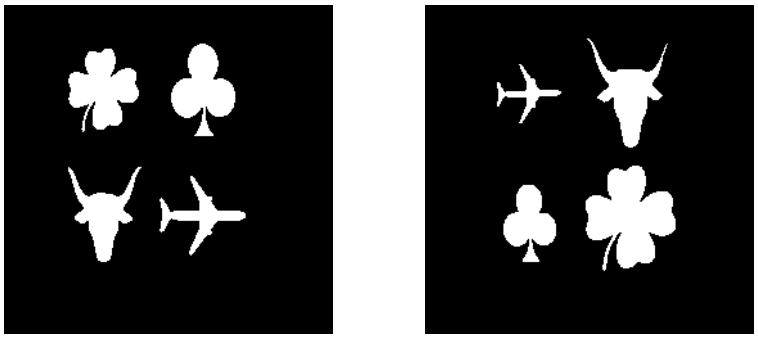
Extracting the Top Four Largest Regions

To extract the four largest connected regions, we compared the area covered by each labeled connected-component in the image. Let $\mathcal{L}_k$ be the set containing pixels that have been labeled k and k ranges from $[1, N]$ where N is the total number of connected-components. The cardinality of the set, $\text{card}(\mathcal{L}_k)$, returns the number of elements in the set. This happens to be the number of pixels covering the image i.e the area. Thus, the largest four regions are four distinct sets $\mathcal{L}_k$ that have the four largest cardinality amongst all possible labels k.

Figure 3: Color labeled connected components in "`lenna.gif`"

Logical Set Operations

Consider two binary images $\mathbf{A}$ and $\mathbf{B}$, and arbitrary pixels $p_{\mathbf{A}}(x, y)$ and $p_{\mathbf{B}}(x, y)$ from each image respectively. For this section, we will be considering the images from Figure 4. We outline here the approach we took to implement each required logical set operator.



(a) File "`match1.gif`" (b) File "`match2.gif`"

Figure 4: The original files used for testing the custom-written logical set operators

`NOT` Operator The unary `NOT` operator negates a logical value i.e

$$\texttt{NOT}(p_{\mathbf{A}}(x,y)) = \begin{cases} 0, & p_{\mathbf{A}}(x,y) = 1 \\ 1, & p_{\mathbf{A}}(x,y) = 0 \end{cases}$$

In MATLAB, one can simply use the "$\sim$" `NOT` operator on a pixel value to achieve this behaviour. Because that feels like cheating, another method is to perform the comparison against the value 0 i.e

$$p_{\mathbf{A}}(x,\, y) \stackrel{?}{=} 0$$

which effectively performs the same thing since any value other than 0 gets mapped to 0 (`False`) while the value 0 gets mapped to 1 (`True`).

`AND` Operator The binary operator `AND` returns 1 iff both inputs are 1. For every pixel $p_{\mathbf{A}}(x,\, y)$ and $p_{\mathbf{B}}(x,\, y)$, the `AND` operation can simply be obtained by multiplying the value of

the two pixels

$$p_{\mathbf{A}}(x, y) \cdot p_{\mathbf{B}}(x, y).$$

Only when both pixels are high, i.e. 1, will the output value be high, emulating the `AND` operator.

`OR` Operator The binary operator `OR` returns 1 if **at least one** of its input is 1. Rather than multiplying, we now check if the sum of the two pixels are non-zero i.e.

$$p_{\mathbf{A}}(x, y) + p_{\mathbf{B}}(x, y) \overset{?}{>} 0.$$

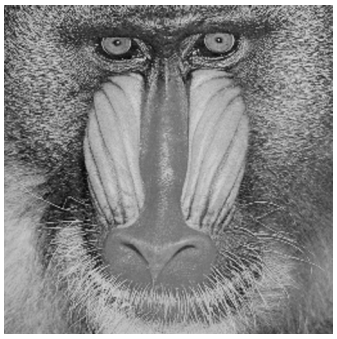
This comparison returns high whenever the sum is non-zero emulating the `OR` operator.

`XOR` Operator The binary operator `XOR` returns 1 iff one of its two inputs is high. Another way to approach this is to realize that `XOR` returns 1 if both inputs are different. Thus we simply test if both pixels are not equal to each other

$$p_{\mathbf{A}}(x, y) \overset{?}{\neq} p_{\mathbf{B}}(x, y).$$

The comparison only returns 1 when both inputs are not equal to each other, implying that only one input is high, successfully emulating the `XOR` operation.

The `MIN` Operator



(a) File "`mandrill_grey.tif`"

(b) File "`cameraman.tif`"

Figure 5: The original files used for testing the custom-written `MIN` operator

Code Structure and Execution Detail

All source code and relevant input files are located in the root directory of the zipped folder. The main source code file is labeled `main.m` and should be the starting point to exploring our code base. For general overview of our project, use

```
help main
```

in the MATLAB IDE console. Similarly, we've written docstrings for all functions, thus getting help on usage of any function can be found by simply typing

```
help function_name
```

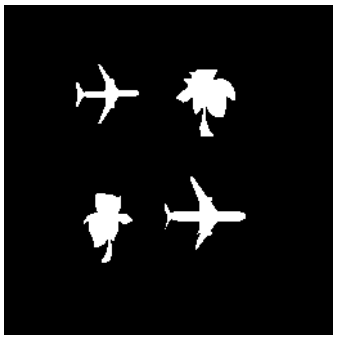
Executing our project can be done by typing `main` in the MATLAB IDE console or simply hitting the "Run" button on the IDE toolbar with `main.m` open and active. All output files can be found in the `output` folder.

Results

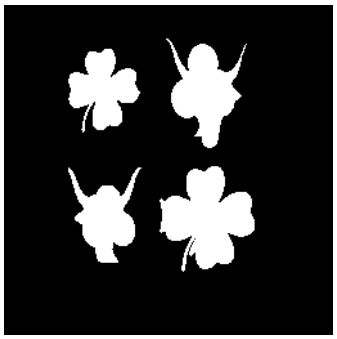
Connected-Component Labeling

Figure 6: 4 Largest connected components in "`lenna.gif`"

Logical Operation Output

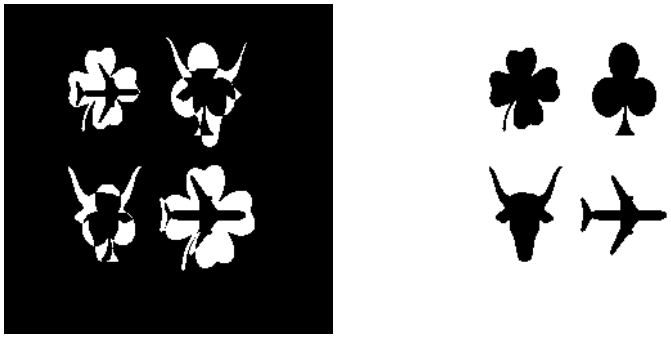


(a) `AND`



(b) `OR`

Figure 7: Output of AND and OR binary image operators



(a) `XOR` (b) `NOT`

Figure 8: Output of XOR and NOT binary image operators

Figure 9: Minimum Operator Intersection of "`mandrill grey.tif`" and "`cameraman.tif`"

It feels intuitive that this operation is the analog of a set intersection for grayscale images as we ended up with a composite of both images.

Conclusions

We were able to utilize our new understanding of digital image processing concepts such as neighbours, connected-components, and set operations in order to implement all of the required algorithms for this project. We continue to learn how to use MATLAB and LaTeXwell. We were able to create multiple connected components from the gray-scale image (and even implement our own algorithm!), and then label them in color in order to clearly see the components that we had just created. In the second portion of the project we we're able to create logical binary image operators for AND, OR, XOR, and NOT operations. This allowed us to create an output that clearly detailed set union, intersection, and complement, through the use of those operators on images A and B. And finally, we coded a minimum operator that produced a combination image by selecting the minimum gray-value pixel for every matching pixel between 2 images.