Anthony Christe
ICS 683
9/24/13

# Software Report

## Compiling and Running

Please see the readme.txt file in the source directory root.

## Runtime Environment

The software is written in Java (specifically version >= 1.6) with no additional libraries. The BufferedImage class is used for displaying and manipulating images.

When the program starts, you can view the different image processing steps by selecting the options on the left hand side with either the mouse or the arrow keys. Each option produces a separate image.

Any textual information will be displayed in the lower left of the application. Information such as threshold, structuring element size, or the number of connected components can be found here.

## Layout of Source Code

- All source code can be found under the directory src/edu/achriste/*.
- Source code related to image processing is in the directory src/edu/achriste/processing.
- Source code for the user interface is found at src/edu/achriste/ui.
- Variious image utilities can be found under the directory src/edu/achriste/utils.

## Performing Otsu's Method

Running this algorithm over balls.gif yields a threshold value of 126.

All of the methods related to Otsu's methods can be found in src/edu/achriste/processing/Otsu.java.

First the histogram is computed by passing a BufferedImage to the getHistogram method. getHistogram stores the total count of each pixel's grayscale at the appropriate index in an array. Then each count is divided by the total number of pixels to find the probabilities.

I found it easier to split up Otsu's Method into several methods that were logically separated. This greatly aided in my understanding of the algorithm.

Using the histogram, otsusMethod is called next, which for each possible threshold, subsequently calls getQ1, getQ2, getU1, getU2, and getWihinVariance. The maximum within variance is recorded along with the index location of that maximum variance and then returned.

The getQ* methods compute the group probabilities for below and and above the current threshold by summing over the two halves respectively.

The getU* methods compute the group variance by multiplying the index and the value at that index and then dividing it by the group probability. These values are summed to form to complete the computation.

Finally, getWinthinVariance uses the previous obtained group probabilities and group variances to compute the within variance at the current threshold.

**Creating a Binary Image**
After the threshold value of 126 is found, we call the makeBinary method in src/edu/achriste/utils/ImageUtils.java. The method simply checks the grayscale value at each pixel, and if the pixel is less than the threshold, changes it to foreground (black), otherwise it will change it to the background (white).

**Erosion with Disk SE**
I found that the disk SE with a radius of 5 produced the best results using my particular implementation of the algorithm. The disk SE with a radius of 4 still contains too many connected balls, while the disk SE with a radius of 6 removes many of the smaller balls completely. The disk SE with a radius of 5 provides a nice compromise of splitting enough balls into separate components, and not completely washing out the small balls.

The class DiskSE at src/edu/achriste/processing/DiskSE.java represents a disk structuring element. The disk SE is represented by a matrix where foreground pixels are represented by a 1 and background pixels are represented by a 0.

The disk is formed using the midpoint circle algorithm adapted from [https://en.wikipedia.org/wiki/Midpoint_circle_algorithm](https://en.wikipedia.org/wiki/Midpoint_circle_algorithm). Since this algorithm only creates the outline of the circle, a subsequent method fills in the diameter of the circle with foreground pixels.

My interpretation of the radius is the number of pixels to the edge from and including the center pixel. Hence the following center line through a circle would be considered radius 3, *****.

The image is eroded by applying the DiskSE object within the Erosion class at src/edu/achriste/processing/Erosion.java.

The erode method takes a BufferedImage and a DiskSE and for each pixel in the image, calls canOrAll which tests if every foreground pixel in the DiskSE matches a foreground pixel in underlying image. If all of the foreground pixels match, then a foreground pixel is ORed into the image at the upper-left (0,0) coordinate of the DiskSE.

Since the DiskSE origin was not specified by the assignment, I chose (0,0) as the origin for it's ease of use. Choosing (0,0) as the origin also means that padding only has to occur on two sides (right and bottom). However, I did not have enough time to add in proper padding, which is why in my eroded images, you can tell that balls on the very right and very bottom did not get separated properly.

**Connected Component Labeling**
The following amount of connected components were found for each disk SE.

| **Radius of Disk SE** | **r = 4** | **r = 5** | **r = 6** |
|---|---|---|---|
| **# of Components** | 1662 | 1668 | 1606 |

The number of components increases from radius 4 to radius 5 due to the fact that more erosion means that more connections between balls become broken. There is a sudden drop off from radius 5 to radius 6 due to the fact that many of the smallest ball components are completely eroded away.

The ConnectedComponents class can be found at src/edu/achriste/processing/ConnectedComponents.java.

Instead of the approach outlined in the lecture notes, I chose to implement 4-neighbor connected component labeling recursively. The recursive method is much easier to reason about and plenty efficient on components of such small size.

First, a matrix is produced from the original image in which all foreground pixels are set to -1 and background pixels remain 0. Then each pixel is scanned row-by-row and non-labeled foreground pixels cause a recursive method, setNeighbors, to be called which will recursively scan and label the North, West, South, and East neighbors that are also non-labeled foreground pixels.

Finally, each component is assigned a unique random color and a BufferedImage is returned with the updated colors.