SEGMENTATION AND EXTRACTION OF INDIVIDUAL LEAVES FROM PLANT IMAGES FOR SPECIES CLASSIFICATION

A Thesis
by
DALE GARRETT HENRIES

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

August 2011
Major Department: Computer Science

SEGMENTATION AND EXTRACTION OF INDIVIDUAL LEAVES FROM PLANT IMAGES FOR SPECIES CLASSIFICATION

A Thesis
by
DALE GARRETT HENRIES
August 2011

APPROVED BY:

_____

Rahman Tashakkori
Chairperson, Thesis Committee

_____

Cindy A. Norris
Member, Thesis Committee

_____

James T. Wilkes
Member, Thesis Committee
Chairperson, Department of Computer Science

_____

Edelma D. Huntley
Dean, Research and Graduate Studies

ABSTRACT

SEGMENTATION AND EXTRACTION OF INDIVIDUAL LEAVES FROM PLANT IMAGES FOR SPECIES CLASSIFICATION

Dale Garrett Henries

M.S., Appalachian State University

Thesis Chairperson: Rahman Tashakkori

Plant species classification through the examination of images of plant leaves requires as input an image of a single leaf with no stems or other non-leaf objects. Images of plants, however, usually include more than one leaf, stems, branches, flowers, and other non-leaf objects. For such images each individual leaf needs to be extracted into a unique sub-image, and these sub-images must be cleaned to remove all non-leaf objects. A target leaf could then be selected from the group of sub-images to be provided as the input to the plant species classification program. As a part of the research on this thesis, an algorithm was developed to automate the tasks of detecting and extracting leaf sub-images from plant images and to clean the leaf sub-images by removing all non-leaf objects. To implement the algorithm, software was developed in Java. The proposed algorithm produced at least one perfect leaf result in 18 of the 21 (86%) plant images used in this research, while the remaining three (14%) plant images produced acceptable leaves.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my advisor, Dr. Rahman Tashakkori, for his continuous guidance, encouragement, and friendship throughout my graduate studies. I am also grateful to Dr. Cindy Norris and Dr. James Wilkes for their advice and encouragement. Thanks to the faculty and staff at the Department of Computer Science, and my fellow graduate students who made this venture much more enjoyable.

I would also like to thank the Office of Research and Graduate Studies and the NSF STEP Program for providing funding throughout my studies.

Finally, I would like to thank my family for their never ending support. They have believed in me in times when I did not believe in myself, and without them I could not have made it this far.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

A significant amount of research has been devoted to plant species classification through the examination of images of leaves. The classification process relies heavily upon being able to extract shape related features and measurements of the leaf itself. Photographs of plants, however, almost always contain multiple leaves, stems, branches, and background objects that interfere with the examination process and must be removed from the image prior to species classification. Removal of these interfering parts is most often accomplished by researchers manually editing the image. In order for species classification through the examination of plant photographs to be more practical, the process of cleaning the leaf image should be automated. This would allow applications to be developed where an end-user provides a photograph of a plant, and the software determines the species of the plant.

Complete automation of selecting and extracting a target leaf from a photograph requires several steps. First, the plant must be separated from the background objects in the image such as the ground and sky. Next, all leaves within the image should be detected and separated into individual sub-images. For each of the leaf sub-images, the objects that are not leaves, such as stems and branches, must be removed. Then, any leaves that are partially occluded should be discarded as possible targets. Finally, if multiple non-occluded leaves are found, one must be selected as the best possible candidate for classification.

Although a complete automated solution is not out of reach for the near future, this research focuses on several of the issues presented while leaving others for future research.

Segmentation of the plant from background objects is a challenging task due to the fact that background objects often closely resemble the plant in color. Disregarding partially occluded leaves is also a challenging problem, because leaves that are from the same plant are often of the same color. It can even be a challenge for the human eye to detect where one leaf ends and another begins. For these reasons, this thesis focuses on images which contain no partially occluded leaves and where the plant is already segmented from the background. Selection of the most suitable target leaf for species classification will require an algorithm that provides a quantitative way to rank leaves in order of suitability for analysis. Developing such an algorithm is challenging because it must work for many different types of leaves and cannot use species specific information in its calculations. For this reason, this thesis will provide all of the extracted leaves and will leave the selection of most suitable leaf for future research.

Several approaches for automatic leaf extraction from images have been proposed [1, 2, 3, 4]; however, all of these techniques make assumptions that severely limit their effectiveness in certain situations. For example, some of the proposed approaches [1, 2, 4] assume prior knowledge of the shape of the target leaf. Leaves, however, vary drastically in shape as different species may have smooth or serrated edges, single or multiple lobes, or various other shape differences.

Another common assumption is that leaves are green [1, 3]. While it is true that most leaves are green, many plant species have leaves that are other colors, are multiple colors, or even change colors in direct sunlight. Leaves of some species also change color at different times of the year. Assuming that leaves are green eliminates unhealthy or dead leaves that often become dark or brown.

The method in which the target leaf is selected presents a challenge with some of the proposed approaches [3, 4]. These studies make an assumption that the target leaf is the largest foreground region in the image. The largest leaf, however, may not be the most suitable leaf for species classification. For example, if the largest leaf in an image is torn and the image contains a smaller complete leaf, the smaller leaf would be better suited for classification. In addition to this problem, the algorithm provided by Tang et al. [3] makes the assumption that the target leaf is centered in the original photograph. In photographs with multiple leaves, a leaf could be present in any part of the image. Requiring the target leaf to be in a certain location within the image drastically hinders the potential of the algorithm.

The existing approaches for automated leaf extraction are not satisfactory for an application that allows end-users to provide a plant image. For such an application to be feasible, the automated leaf extraction algorithm should handle leaves of various colors, shapes, sizes, and locations within the image. In this thesis, an algorithm will be proposed to automate the process of extracting the possible target leaves from a plant image.

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of image processing techniques related to this work. Chapter 3 outlines the overall algorithm proposed and the algorithms for target leaf extraction, to detect and extract individual leaf sub-images, and individual leaf image cleaning, to remove stems and other interfering objects. Chapter 4 provides an overview of the software tool that was created in Java to implement the proposed algorithms. Chapter 5 presents the results and effectiveness of the proposed algorithms. Chapter 6 discusses the outcomes of this study, and provides possible future work in this area.

CHAPTER 2: IMAGE PROCESSING TECHNIQUES

2.1 Introduction

Digital image processing refers to the processing of digital images through the use of a computer [5]. While digital image processing has many applications, this study focuses on the automated extraction of leaves from a digital image. Digital images of plants are processed to locate and extract sub-images of individual leaves. Each individual sub-leaf image is then cleaned to remove background objects. Various image processing techniques were used throughout the proposed algorithms to accomplish these tasks. Sections 2.2 through 2.6 provide details and background information for these techniques.

2.2 Morphological Operations

Morphological operators are tools that can be used to extract image components [5]. These tools are used on binary images to trim, expand, isolate, or connect regions of foreground, or white pixels, within an image. Morphological operations in digital image processing are based on the concepts of set theory. For the purpose of the morphological operations used in this study, the pixels within a binary image are considered to be in a set A. A smaller binary image known as a structuring element is also created, and the pixels within this smaller image are considered to be in a set B. Structuring elements are generally created at the beginning of a morphological operation and can be a variety of shapes and sizes. For the purpose of this thesis, it can be assumed that a circular structuring element was used unless otherwise noted. Each morphological operation is therefore performed by the

convolution of set A and set B with some given set operation. Sections 2.2.1 through 2.2.4 describe the morphological operations used in this thesis.

2.2.1 Erosion

Erosion is the morphological operation used to trim away the edges of a foreground region from a binary image. Using set theory, erosion can be defined as the intersection of sets A and B. Figure 2.1 displays an example of erosion on a binary image. The dark green portion illustrates the foreground region prior to the erosion. The light green portion represents the foreground region after the erosion, and the circles represent the structuring element used in the erosion. If all of the pixels beneath the structuring element are foreground in the original image, the pixel where the structuring element is centered remains foreground in the eroded image. If any pixel beneath the structuring element is background in the original image, the pixel where the structuring element is centered is set to background in the eroded image. As can be seen, the result of erosion is that the foreground region has been trimmed around the edges.

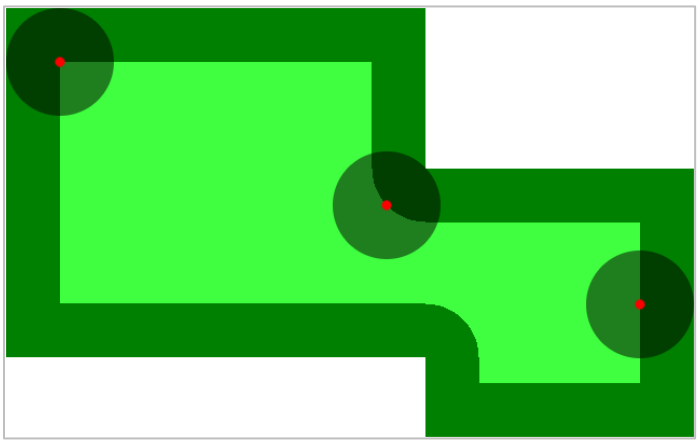

Figure 2.1: Illustration of Erosion

2.2.2 Dilation

The morphological operation of dilation has the effect of expanding or growing the edges of the foreground region of a binary image. In set theory, dilation can be defined as the union of sets A and B. Figure 2.2 depicts an example of dilation on a binary image. The dark green illustrates the foreground region prior to the dilation. The light green represents the foreground region that was added by the dilation, and the circles represent the structuring element for the operation. If any pixel under the structuring element is foreground in the original image, then the pixel where the structuring element is centered is set to foreground in the dilated image. If no pixels under the structuring element are foreground in the original image, the pixel where the structuring element is centered is set to background in the dilated image. As Figure 2.2 depicts, the result of dilation, which includes both the dark and light green areas, is an expansion of the foreground region around the edges.

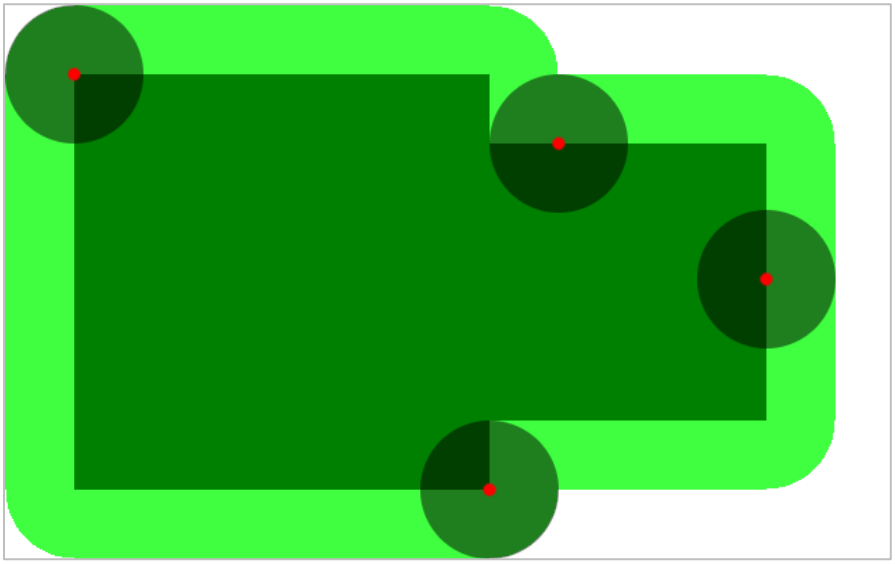

Figure 2.2: Illustration of Dilation

2.2.3 Opening

The morphological operation of opening is the erosion followed by the dilation using the same structuring element. The effect of the opening operation, shown in Figure 2.3, is that foreground regions smaller than the structuring element are removed. Foreground regions larger than the structuring element are disconnected by eliminating corners and any thin sections connecting the larger foreground regions. The dark green areas in Figure 2.3 illustrate the foreground region prior to the opening process while the light green areas represent the resulting foreground after the opening process. As can be seen, the small dark green rectangle is removed by the erosion process so that there is nothing to expand during the dilation process. The two larger dark green squares, however, would result in smaller squares from the erosion process, and then are expanded by the dilation process to contain all of their original shape except for the corners.

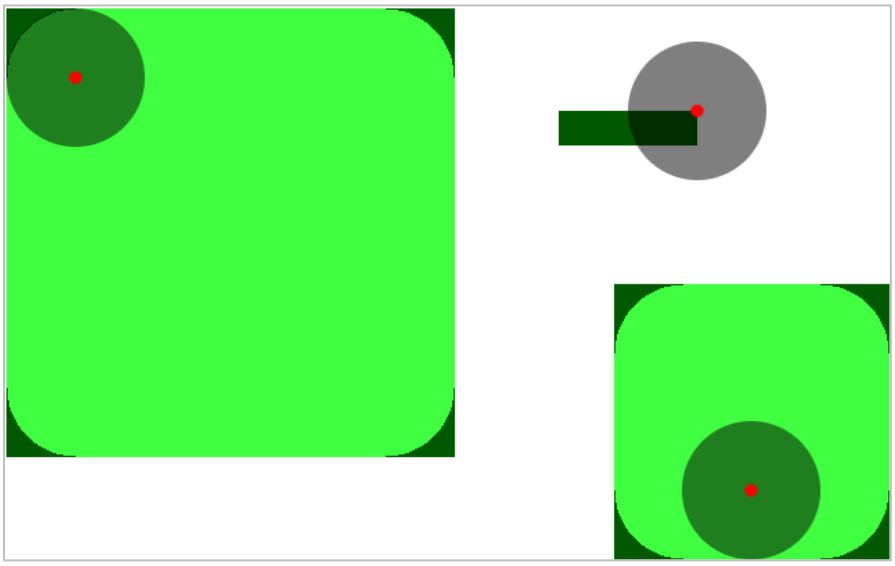

Figure 2.3: Illustration of Opening

2.2.4 Closing

The morphological operation of closing is the dilation followed by the erosion of the image using the same structuring element. The effect of the closing operation, illustrated in Figure 2.4, is that small holes in foreground regions or small gaps between foreground regions are changed to foreground. The dark green areas in Figure 2.4 illustrate the foreground regions prior to the closing operation. The result of the closing operation is the combination of the dark green regions and light green regions. The result of the dilation would be an expanded foreground region. The erosion that takes place on the result of the dilation trims the foreground region back to approximately the original shape except for regions that were close together or close to the edge of the image in the original binary image.

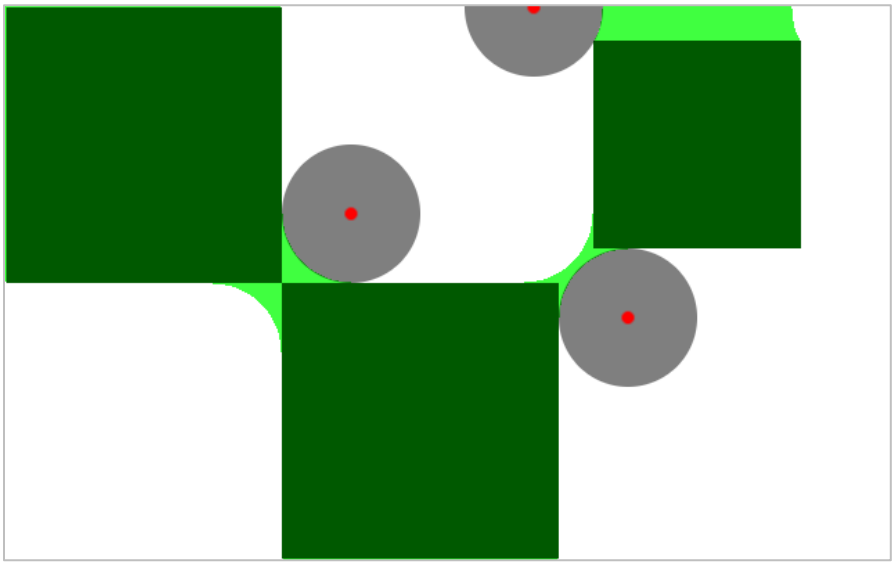

Figure 2.4: Illustration of Closing

2.3 Gradient Images

Gradient images are grayscale images that reflect the change in intensity for each pixel in the image through analyzing the intensity values of neighboring pixels. Several algorithms exist for the production of gradient images; however, for the purposes of this study, the Sobel edge detector was used [5]. As can be seen in Figure 2.5, the result of the Sobel edge detector is lighter or higher pixel values in the gradient image where the pixel values change from dark to light or light to dark rapidly in the grayscale image. Where values remain relatively similar in the grayscale image, however, the result of the Sobel edge detector is darker or lower pixel intensity values in the gradient image.

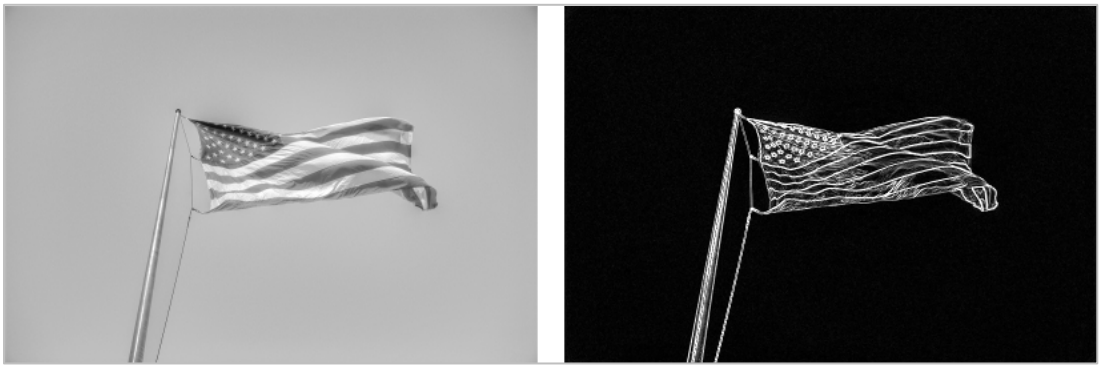

Figure 2.5: Left - Grayscale Image, Right - Gradient Image

2.4 Conversion of RGB to Grayscale

The conversion to a grayscale image from an RGB image involves calculating a single grayscale value from the three values found in an RGB image. This value can either be calculated by averaging the three values from the RGB image or by using a weighted average to give each of the three values a priority. Giving priority to certain values in an RGB image, however, implies that some values are more important than others. Because leaves can be

found in many different colors, this research uses an unweighted average for grayscale conversions.

2.5 Otsu Thresholding

To convert a grayscale image into a binary image, a threshold value is selected in which all values less than the threshold become a zero, or black, in the binary image and all values greater than or equal to the threshold value become a one, or white, in the binary image; therefore, the outcome of the binary image is largely dependent upon the value selected for the threshold. If the threshold value selected is low, background regions of the image will be included as foreground elements in the binary image and if the threshold value selected is too high, elements that should be included as foreground objects in the binary image will be eliminated as background regions.

For the purpose of this study, Otsu's thresholding algorithm [6] was used to determine the optimal threshold value for converting grayscale images into binary images. Otsu's algorithm analyzes the histogram of the grayscale image to determine the value which maximizes the between-class variance [5]. The between-class variance is a measure of spread for the pixel values both above and below a given threshold value. By maximizing the between-class variance, a threshold value is selected that optimizes the separation of foreground and background regions within the image.

2.6 Marker Controlled Watershed Segmentation

Image segmentation is the process of subdividing an image into its constituent regions or objects [5]. Watershed segmentation is a method which produces a segmented image that includes connected regions, and as a result, isolates individual objects within an image.

Watershed segmentation is based on the concept of visualizing a gradient image in three-dimensions like a topographical map. The first two dimensions are the coordinates of each pixel (x, y), and the third dimension is represented by the value of each pixel. Pixels with high values would therefore be mountain tops while pixels with low values would be valleys on a topographical map. The watershed segmentation algorithm works by symbolically flooding the terrain created by the gradient image with rain. As the rain drops fall, they run down the mountains until pools are formed in the valleys. As the pools rise, they begin to run into one another. In cases where any two pools meet, the location is marked as a boundary between objects. This process continues until all of the terrain is flooded.

The main problem with watershed segmentation is that it has a tendency to over segment an image. To overcome this problem, markers can be set prior to flooding that control the amount of segmentation. Basically, markers pre-flood the region image with region identifiers. Each region identifier corresponds to a single object within the image. If two pools meet during flooding that have the same marker, the pools are allowed to combine and no object segmentation occurs. If two pools meet during flooding that have different markers, however, the location is marked as a boundary between objects.

CHAPTER 3: METHODOLOGY

3.1 Introduction

The database of plant images in the Irvin Watson Carpenter, Jr. Herbarium, located in the Department of Biology at Appalachian State University, was the primary source of plant images for this research. These images were manually cropped and edited to remove non-plant objects and areas with overlapping leaves. Other than removing overlapping leaves, however, the plant images were not altered and therefore contain branches, stems, stalks, seeds, flowers, and other normal plant parts. Figure 3.2 illustrates an example specimen from the I. W. Carpenter, Jr. Herbarium and the manually edited version used in this thesis. The three steps, illustrated in Figure 3.1, were performed by the algorithm that extracts possible target leaves:

- Image Preprocessing
- Detection and Extraction of Individual Leaves
- Individual Leaf Image Cleaning

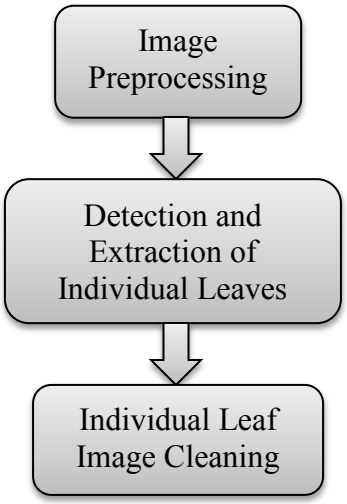

Figure 3.1: Three Steps of Methodology

Figure 3.2: Left - Original Specimen from I. W. Carpenter, Jr. Herbarium, Right - Manually Edited Version

For the actual implementation of the leaf sub-image extraction and leaf image cleaning algorithms proposed in this research, a software tool was developed in Java. The remainder of this chapter will present the theoretical implementation of the algorithms, while the following chapter will address the actual implementation of the software.

3.2 Image Preprocessing

The first step in an automatic leaf extraction algorithm is to locate the individual leaves within the plant image. Since digital images vary drastically in size, this could lead to unexpected results for different sized images using the same algorithm. In order to unify the approach, before attempting to locate leaves within an image, an image is scaled so that the largest of either its width or height is exactly 500 pixels. The image's original ratio of width to height was not altered in the scaling process. By scaling images to a standard size, more accurate results can be obtained by the algorithm designed to detect and extract individual leaves.

3.3 Detection and Extraction of Individual Leaves

As illustrated by Figure 3.3, the detection and extraction of individual leaves requires three steps: segmentation of the plant, segmentation of the leaves, and creation of individual leaf images. The first step, segmentation of the plant, is required so that all non-plant background objects are disregarded as insignificant. Once the plant is identified, segmentation of the leaves is required to identify which portions of the plant are known to be leaves. Finally, individual sub-images are created for each of the areas identified as leaves.

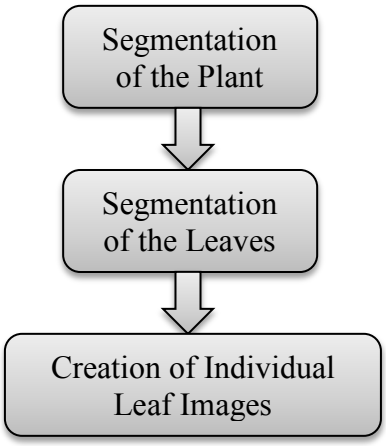

Figure 3.3: Three Steps of Detection and Extraction of Individual Leaves

3.3.1 Segmentation of the Plant

The logical first step in locating individual leaves within a plant image is to separate the plant from the rest of the image. To accomplish this, the RGB image is converted into a grayscale image. This grayscale image is then slightly blurred with a Gaussian filter to minimize the effects of small holes or irregularities in the leaves. A binary image is then created using the grayscale image and Otsu's algorithm [6] to determine the optimal threshold value. Figure 3.4 shows the RGB, grayscale, and binary versions of a sample image. As can be seen in the binary image, the plant becomes the foreground of the image. At this point the background regions in the binary image are deemed to be insignificant and are therefore removed, or set to white, in the original RGB image.

Figure 3.4: Left - RGB Image, Center - Grayscale Image, Right - Binary Image

3.3.2 Segmentation of Leaves

With the background and all non-plant objects removed, the task of segmenting leaves within the image would seem to be as simple as removing all non-leaf parts. The process of categorizing a leaf by species, however, is highly dependent upon the shape of the leaf. The more common methods, such as the morphological operations generally used in such a situation, would not only remove the stems and branches of the image but would also alter the shape of the leaves themselves. For a leaf extraction algorithm to be effective for the purpose of species classification, the branches, stems, and other non-leaf plant parts have to be identified and removed without altering the shape of the leaves.

A marker controlled watershed segmentation is used to determine the location of all possible leaf objects. The internal markers, which represent regions considered to be possible leaves, are created by eroding the original binary image with a structuring element of 30

pixels in both width and height that is shaped like a geometric cross. The external markers, which represent background regions or objects known to not be leaves, are created by inverting the original binary image. Prior to flooding the gradient image, the stems and branches are dammed up to prevent them from being considered as part of the leaves.

To dam up the stems and branches, a binary image is created from the gradient image using Otsu's method to determine an appropriate threshold. The morphological operation of closing is used on this new binary image to fill the insides of the stems and branches. The foreground region of the new binary image is then copied back to the original gradient image so that all stems and branches are filled with the maximum value. Due to the fact that the stems and branches are now filled with the highest possible mountain peaks, the flooding can take place without fear of stems and branches being included as parts of a leaf. Figure 3.5 illustrates the steps used in filling the stems and branches in the gradient image.

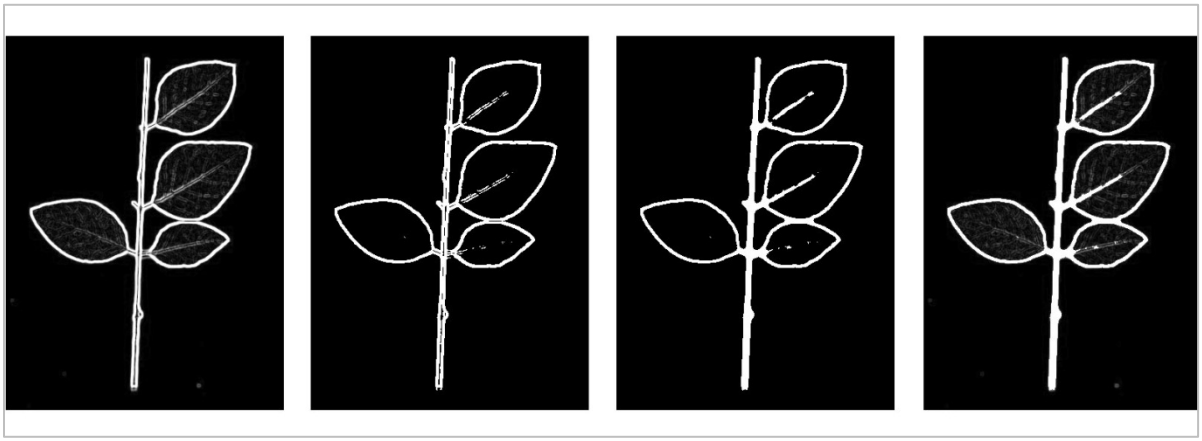

Figure 3.5: Left to Right - Original Gradient Image, Binary Image Created from Gradient, Binary Image After Closing Operation, Resulting Gradient Image After Stems and Branches Filled

Figure 3.6 illustrates the state of the image when it is ready to be flooded. The color red in Figure 3.6 represents external markers or regions known to not be leaves, and the color blue represents internal markers or regions known to be leaf like objects. The color green in Figure 3.6 represents mountain peaks created by filling the stems and branches, and white and green areas represent the portions of the image that will be flooded. Figure 3.7 shows the results of the watershed segmentation as compared to the original binary image.

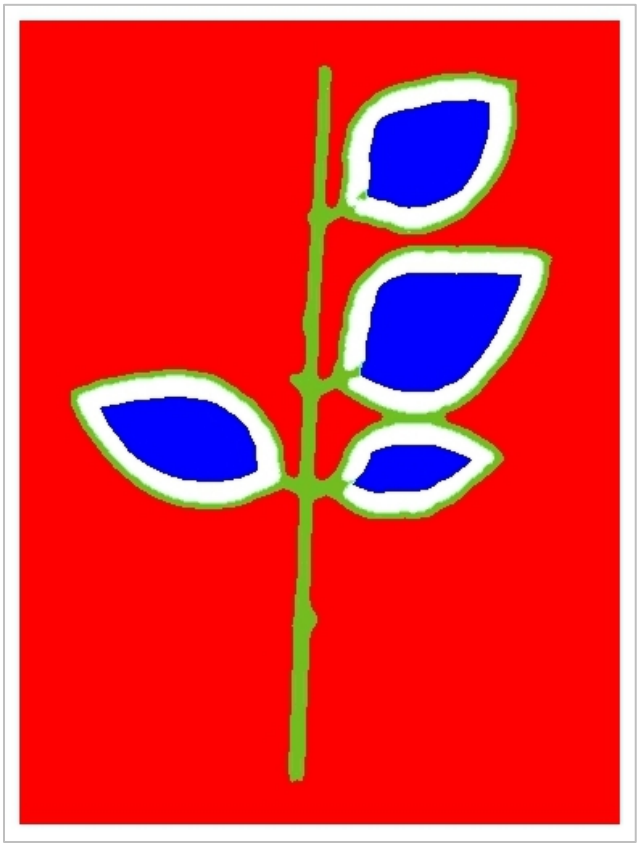

Figure 3.6: Gradient Image Prior to Flooding

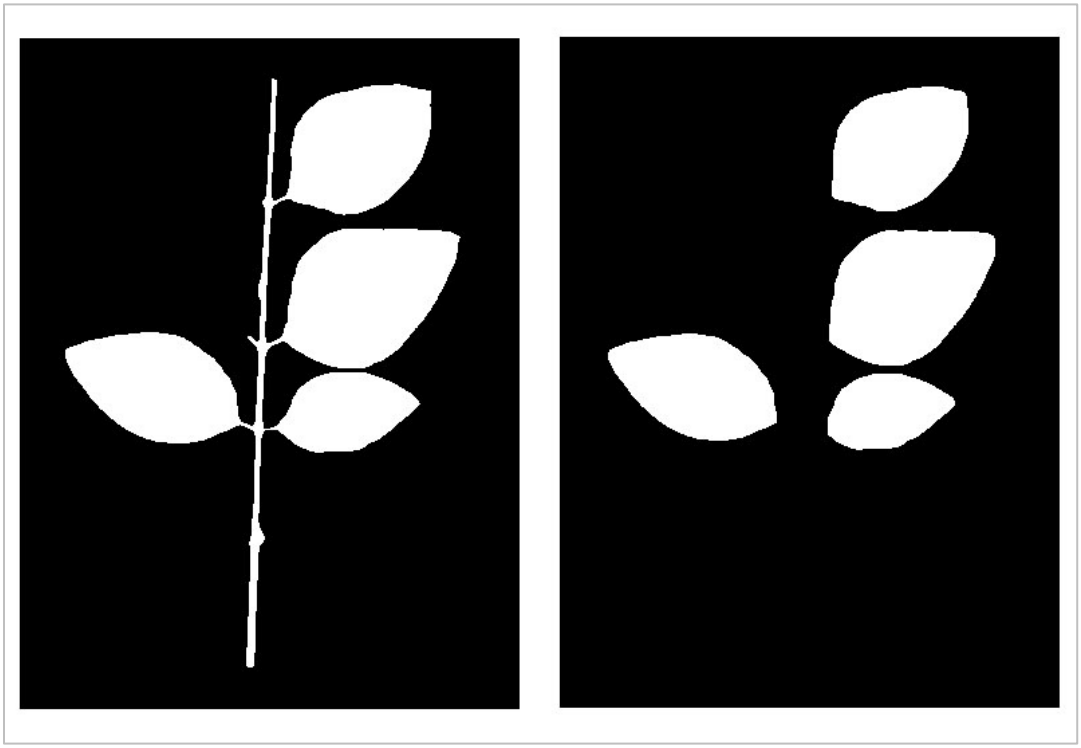

Figure 3.7: Left - Original Binary Image, Right - Result of Watershed Segmentation

3.3.3 Creation of Individual Sub-Leaf Images

Once the watershed flooding was complete, all internal regions are labeled with a unique region identifier. The locations and dimensions for each region are then calculated, and regions that had less than 961 pixels, i.e., 31 pixels by 31 pixels, are discarded as insignificant and not likely the target leaf. Prior to extracting the sub-leaf images, the dimensions of each region are padded by ten percent to guarantee that the entire leaf would be contained in the resulting image. A binary image is created from each sub-image, and those which have ten percent or less foreground pixels are discarded as non-leaf objects. Due to the fact that the leaf segmentation algorithm does not maintain the original shape of the leaf, each sub-leaf image is created from an exact copy of the pixels from the original RGB

image. While this process produces images with leaves unaltered, it also copies stems, branches, and parts of other closely located leaves as well. Therefore, each individual sub-leaf image must be cleaned to remove other partial leaves and non-leaf objects.

3.4 Individual Leaf Image Cleaning

When each leaf from the original plant image is extracted into its own sub-image, each sub-image must be cleaned to remove stems, branches, other partial leaves, and non-leaf objects. The original shape of the leaf, however, must be maintained as well as possible for accurate species classification. As depicted in Figure 3.8, the proposed algorithm for cleaning an individual leaf image requires the following steps:

- Determination of Core Leaf Body
- Examination of Border Regions
- Removal of Selected Regions

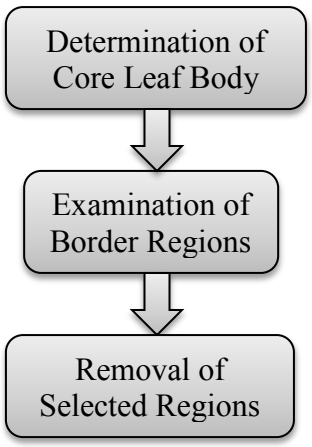

Figure 3.8: Three Steps of Individual Leaf Image Cleaning

3.4.1 Determination of Core Leaf Body

The initial objective of cleaning a leaf image is to determine the portion known to be part of the leaf. To accomplish this, a binary image of the original RGB leaf image is created. The morphological operation of opening is performed on the binary image to remove all small border regions, such as leaf tips and stems, while retaining as much of the leaf's original shape as possible. To determine appropriate size for the structuring element used in the opening process, the largest of either the width or height of the original image is determined, and one-eighth of this size was selected. An additional binary image was then created to contain the foreground regions removed by the opening operation. This second binary image contains all border regions that are not part of the core leaf body and might need to be removed during the cleaning process. Figure 3.9 illustrates an example of an original binary image, the binary image containing the core leaf body, and the binary image containing all border regions that might need to be removed.

Figure 3.9: Left to Right - Original Binary Image, Core Leaf Body, Possible Regions for Removal

3.4.2 Examination of Border Regions

To determine which of the border regions are not parts of the leaf, each border region is examined to determine its level of connectivity. Connectivity is a ratio of the number of pixels in a region that are neighbors with a pixel in the core leaf body over the total number

of pixels in the region. Regions with a high level of connectivity are likely to be part of the leaf while regions with low connectivity, such as stems, are not likely to be part of the leaf. Through experimentation, it was determined that regions with greater than 16% connectivity are most likely to be part of the leaf. The regions determined to be part of the leaf are then erased from the binary image containing possible regions to remove. This results in a binary image with only those regions that should be removed. Figure 3.10 illustrates the results of this procedure. It should be noted that leaves with numerous long tips did not perform well with this algorithm. To accommodate for such leaves, if more than three regions are selected to be removed, only the largest of these regions is removed.

Figure 3.10: Left to Right - Original Binary Image, Possible Regions to Remove, Regions Selected for Removal

3.4.3 Removal of Selected Regions

To remove the selected regions from the original RGB image, all pixels that are foreground in the binary image containing possible regions to remove are set to white in the original RGB image. The result is a cleaned RGB leaf image that is suitable for species classification.

CHAPTER 4: IMPLEMENTATION

As a part of this thesis, a software tool was developed to implement the proposed algorithms discussed in the methodology. The software requires a digital image of a plant with a mostly solid light colored background and no overlapping leaves as input. This image is first sent through the leaf extraction algorithm which returns a collection of individual sub-images for each leaf found in the original input image. Each individual leaf image is then sent through the leaf cleaning algorithm to remove all non-leaf objects such as stems, flowers, and other leaves. The resulting clean leaf images are then saved to the same directory as the original input image.

The programming language chosen for implementation was Java due to its portability on a wide range of operating systems; Java, however, does not provide many built in image processing tools. Therefore, many of the most basic image processing techniques, such as converting an image into a grayscale or binary image, were implemented as part of this thesis.

The remainder of this chapter discusses the implementation of the software and is organized as follows. Section 4.1 addresses the GrayscaleImage and BinaryImage wrapper classes used extensively throughout the program. Section 4.2 addresses other helper classes created to perform specific tasks. Section 4.3 provides an overview of the LeafExtractor class which implements the detection and extraction of individual leaf sub-images algorithm. Section 4.3 provides an overview of the LeafCleaner class which implements the individual

leaf cleaning algorithm. Samples of code are provided throughout this chapter; in addition, the entire code is provided on an enclosed CD.

4.1 GrayscaleImage and BinaryImage Wrapper Classes

Access to images in Java is provided by the java.awt.image.BufferedImage (BufferedImage) class which provides useful methods for working with RGB images. Grayscale and binary images, however, have unique value ranges and image processing operations. To deal with these requirements, two wrapper classes, GrayscaleImage and BinaryImage, were created as a part of this thesis that take as input an RGB BufferedImage and provide access to the RGB BufferedImage as if it were a grayscale or binary image.

4.1.1 Accessing and Setting Pixel Values

Java's BufferedImage provides useful methods for working with RGB images such as *getRGB(int x, int y)* and *setRGB(int x, int y, int val)* which respectively return or set the value of the pixel located at the *x* and *y* coordinates provided. The problem with these methods is that they return or require as input a single thirty-two bit value where each byte represents, from most significant to least significant, the pixel's: alpha value, red value, green value, and blue value. A significant amount of bit manipulation is required to access or set the actual four values from the single value returned or required by these methods.

When working with grayscale or binary images, each of the red, green, and blue values are equal, and the alpha value never changes. GrayscaleImage and BinaryImage, which store the original RGB BufferedImage *image*, provide methods *getPixel(int x, int y)* and *setPixel(int x, int y, int val)* which perform all of the required bit manipulation and return or accept as input an acceptable grayscale or binary value.

Figure 4.1 illustrates the code segment for the GrayscaleImage's *setPixel* method. As can be seen, a single value is provided as input, and the bit manipulation required to store it as a single ARGB value is performed inside the call to BufferedImage's *setRGB* method.

```
/**
 * Set the value for a pixel
 * @param x
 * @param y
 * @param val - grayscale value
 */
public void setPixel(int x, int y, int val){
    image.setRGB(x, y, 0xff000000 | (val << 16) | (val << 8) | val);
}
```

Figure 4.1: GrayscaleImage's *setPixel* Method

For a binary image, only two values are acceptable: one for white and zero for black. The *getPixel* method in Figure 4.2 uses a helper method *RGBToBinary* to convert the single thirty-two bit value provided by BufferedImage's *getRGB* into an acceptable one or zero.

```
/**
 * Obtains the value of the pixel from the BufferedImage and returns the
 * binary representation of the value
 * @param x
 * @param y
 * @return 0 = black, 1 = white
 */
public int getPixel(int x, int y){
    return RGBToBinary(image.getRGB(x, y));
}

/**
 * Converts a 4BYTE_ARGB hex value to a binary value
 * @param val
 * @return
 */
private int RGBToBinary(int val){
    if(val == -1) // 0xffffffff
        return 1;
    else
        return 0;
}
```

Figure 4.2: BinaryImage's *getPixel* Method

4.1.2 GrayscaleImage and BinaryImage Construction

A GrayscaleImage can be constructed from an ARGB BufferedImage. Figure 4.3 illustrates the code required to construct a GrayscaleImage. In the constructor method the input BufferedImage is stored as a field *image* along with the image's height, width, and raster. The method *convertFromRGB* parses each pixel and uses the methods *rgbToGrayscale*, *getRed*, *getGreen*, and *getBlue* to perform the required bit manipulation and calculation of the grayscale value. This value is stored in the BufferedImage *image* as an ARGB value, which then allows the previously mentioned *getPixel* and *setPixel* methods to respectively return and accept as input simple one byte unsigned grayscale values.

Similar to GrayscaleImages, BinaryImages store an ARGB BufferedImage in a field *image*. The previously mentioned *getPixel* and *setPixel* methods provide access to the ARGB BufferedImage using simple one bit binary values. BinaryImages are constructed using a GrayscaleImage and a threshold value. All pixels with grayscale values less than the given threshold value are set to zero or black, and all pixels with grayscale values greater than or equal to the threshold value are set to one or white. The construction of BinaryImages is similar to that of GrayscaleImages, presented in Figure 4.3, and differs only in that the methods are altered to convert grayscale images into binary images.

```java
/**
 * Create a GrayscaleImage from a ARGB BufferedImage
 * @param img
 */
public GrayscaleImage(BufferedImage img){
    height = img.getHeight();
    width = img.getWidth();
    if(img.getType() == BufferedImage.TYPE_4BYTE_ABGR){
        image = convertFromRGB(img);
        raster = image.getRaster();
    }
}

private BufferedImage convertFromRGB(BufferedImage src){
    BufferedImage grey = new BufferedImage(width, height,
                                           BufferedImage.TYPE_4BYTE_ABGR);
    for(int y=0; y < height; y++){
        for(int x=0; x < width; x++){
            grey.setRGB(x, y, rgbToGrayscale(src.getRGB(x, y)));
        }
    }
    return grey;
}

private int rgbToGrayscale(int rgb){
    double red = getRed(rgb);
    double green = getGreen(rgb);
    double blue = getBlue(rgb);
    int val = (int)((red + green + blue)/(double)3);
    return 0xff000000 | (val << 16) | (val << 8) | val;
}

private int getRed(int rgb){
    return (rgb << 8) >>> 24;
}

private int getGreen(int rgb){
    return (rgb << 16) >>> 24;
}

private int getBlue(int rgb){
    return (rgb << 24) >>> 24;
}
```

Figure 4.3: GrayscaleImage Construction

4.1.3 Grayscale Image Specific Operations

Operations specific to grayscale images are also included as methods in the GrayscaleImage class. These methods include *fillStems*, which performs the essential step of damming up stems and branches prior to the flooding in the leaf detection and extraction algorithm, and *sobel*, which returns a new GrayscaleImage that is a gradient image created using the sobel edge detection algorithm [5]. The method *sobel* can be seen in Figure 4.4. To apply the sobel edge detection, first a new empty GrayscaleImage is created with the same width and height of the original GrayscaleImage. Following this step, the sobel value for each pixel is determined using the methods *getSobelXVal* and *getSobelYVal*. The value is then tested to make sure it is not greater than the maximum one byte unsigned grayscale value, and if so, the value is set to the maximum. This value is then used as the pixel value for the GrayscaleImage sobel. Finally, after all pixels have been processed the new GrayscaleImage sobel is returned.

```
/**
 * Create a gradient image using sobel edge detection algorithm
 * @return GrayscaleImage sobel
 */
public GrayscaleImage sobel(){
    // Create new empty GrayscaleImage sobel
    GrayscaleImage sobel = new GrayscaleImage(width, height);
    for(int y=0; y < height; y++){
        for(int x=0; x < width; x++){ // For each pixel
            int val = Math.abs(getSobelXVal(x,y))+
                      Math.abs(getSobelYVal(x,y)); // calc sobel value
            if(val > 255)
                val = 255;
            sobel.setPixel(x, y, val);
        }
    }
    return sobel;
}
```

Figure 4.4: GrayscaleImage's *sobel* Method

4.1.4 Binary Image Specific Operations

Operations specific to binary images are included as methods of the BinaryImage class. These methods include morphological operations, regional operations, global operations, and comparative operations. The following sections provide examples and code for each of the operations that are unique to binary images.

4.1.3.1 Morphological Operations

The BinaryImage class contains methods for morphological operations such as *erode*, *dilate*, *open*, and *close*. Figure 4.5 provides the code for the methods *erode* and *erodeTest*. The *erode* method returns a new BinaryImage that has been eroded with a circular structuring element with the width and height of *size*. Initially, a new empty BinaryImage is created with the same width and height of the original BinaryImage. Each pixel of the original BinaryImage is tested with the boolean method *erodeTest*. The pixels that return true are set to zero (black) in the eroded image while those that return false are set to one (white) in the eroded image. The other three morphological operations are implemented in similar fashion. An *erodePlusSign* method also exists to erode a binary image with a structuring element shaped like a geometric cross and is used to create the internal markers necessary during the marker controlled watershed segmentation.

4.1.3.2 Regional Operations

A region within a binary image refers to a unique area of foreground that is not connected to any other area of foreground. The BinaryImage class contains methods for regional operations such as *countRegions*, which returns the number of unique regions within the

image; *onlyLargestRegion*, which returns a BinaryImage containing only the largest unique region; and *removeRegion*, which removes a unique region from a BinaryImage.

Figure 4.6 shows the code segment for *removeRegion*, which takes as parameters a BinaryImage, and the x, y coordinates of a foreground pixel in the region. The algorithm changes the value of the foreground pixel to background and then recurses on any neighboring pixel that is determined to be foreground. The result is that all foreground pixels connected to the initial pixel are removed from the BinaryImage.

```
/**...*/
public BinaryImage erode(int size){
    BinaryImage eroded = new BinaryImage(width, height);
    for(int y=0; y<height; y++){
        for(int x=0; x<width; x++){
            if(!erodeTest(x,y,size))
                eroded.setPixel(x,y,1);
        }
    }
    return eroded;
}

/**...*/
public boolean erodeTest(int x, int y, int size){
    int half = size >> 1; // divided by 2
    int startX = x - half;
    int startY = y - half;
    if(startX < 0) startX = 0;
    if(startY < 0) startY = 0;

    int endX = x + half;
    int endY = y + half;
    if(endX >= width) endX = width -1;
    if(endY >= height) endY = height -1;

    for(int j = startY; j <= endY; j++){
        for(int i = startX; i <= endX; i++){
            double distance = Math.sqrt(((x-i)*(x-i)) + ((y-j)*(y-j)));

            // circular structuring element
            if(distance <= half){
                int val = getPixel(i,j);
                if(val != 1)
                    return true;
            }
        }
    }
    return false;
}
```

Figure 4.5: BinaryImage's *erode* Method

```
/**
 * Removes a foreground region from a binary image
 * @param bin
 * @param x
 * @param y
 * @return
 */
public BinaryImage removeRegion(BinaryImage bin, int x, int y){
    bin.setPixel(x,y,0);
    int startX = x - 1;
    int startY = y - 1;
    if(startX < 0) startX = 0;
    if(startY < 0) startY = 0;

    int endX = x + 1;
    int endY = y + 1;
    if(endX >= width) endX = width -1;
    if(endY >= height) endY = height -1;

    for(int j = startY; j <= endY; j++){
        for(int i = startX; i <= endX; i++){
            if(x == i && y == j){
                // skip the center cell
            }
            else {
                if(bin.getPixel(i, j) == 1)
                    bin = removeRegion(bin, i, j);
            }
        }
    }
    return bin;
}
```

Figure 4.6: BinaryImage's *removeRegion* Method

Figure 4.7 shows the code for BinaryImage's *countRegions* method which returns the number of unique regions within the image. First, a copy of the original BinaryImage is created so that the original image will remain unaltered, and the variable *regions*, which holds the current region count, is initialized to zero. The algorithm then iterates through each of the image's pixels until finding a foreground pixel. Upon finding a foreground pixel the *regions* variable is incremented and the region is removed from the image so that it will not be counted twice. Once all of the pixels are iterated through, the count of regions is returned.

```
/**
 * Returns the number of unique foreground regions in an image
 * @return
 */
public int countRegions(){
    int regions = 0;
    BinaryImage bin = copy();
    for(int y=0; y<height; y++){
        for(int x=0; x<width; x++){
            if(bin.getPixel(x, y) == 1){
                regions++;
                bin = removeRegion(bin,x,y);
            }
        }
    }
    return regions;
}
```

Figure 4.7: BinaryImage's *countRegions* Method

4.1.3.3 Global Operations

BinaryImage's global operations include methods such as *inverse* and *countWhitePixels*. The method *inverse* changes all foreground pixels to background pixels and all the background pixels to foreground pixels, therefore creating the inverse of the original image. As can be seen in Figure 4.8, the *inverse* method's code inverts the value of each pixel.

```
/**
 * Converts the inverse of this binary image, therefore converting all
 * white pixels to black and all black pixels to white.
 */
public void inverse(){
    for(int y=0; y<height; y++){
        for(int x=0; x<width; x++){
            if(getPixel(x,y) == 1)
                setPixel(x,y,0);
            else
                setPixel(x,y,1);
        }
    }
}
```

Figure 4.8: BinaryImage's *inverse* Method

4.1.3.4 Comparative Operations

Comparative operations in the BinaryImage class perform comparisons between two BinaryImage. The *diffImage* method, illustrated in Figure 4.9, compares the pixels of the original BinaryImage to the pixels of another BinaryImage and returns a new BinaryImage containing foreground pixels everywhere the two BinaryImages have different values.

```
/**
 * Create a binary image containing white pixels everywhere that a pixel of
 * 'other' is not equal to the corresponding pixel in 'this' binary image.
 * @param BinaryImage other
 * @return BinaryImage diff
 */
public BinaryImage diffImage(BinaryImage other){
    BinaryImage diff = new BinaryImage(width, height);
    for(int y = 0; y < height; y++){
        for(int x = 0; x < width; x++){
            if(getPixel(x,y) != other.getPixel(x,y))
                diff.setPixel(x,y,1);
        }
    }
    return diff;
}
```

Figure 4.9: BinaryImage's *diffImage* Method

BinaryImage's *minus* method, shown in Figure 4.10, subtracts the foreground pixels of another BinaryImage from the foreground of the original BinaryImage. The result is a new BinaryImage with foreground pixels only where the original BinaryImage is foreground and the other BinaryImage is background.

```
/**
 * Subtracts 'other' from 'this' BinaryImage, therefore leaving only
 * foreground pixels that are not foreground pixels in 'other'.
 * @param other
 * @return
 */
public BinaryImage minus(BinaryImage other){
    BinaryImage minus = copy();
    for(int y=0; y<height; y++){
        for(int x=0; x<width; x++){
            if(other.getPixel(x, y) == 1 && getPixel(x,y) == 1)
                minus.setPixel(x, y, 0);
        }
    }
    return minus;
}
```

Figure 4.10: BinaryImage's *minus* Method

4.2 Other Classes

While the GrayscaleImage and BinaryImage classes contain a large amount of functionality, several other helper classes had to be created to perform specific tasks. These include classes such as OtsuThreshold, which determines an appropriate threshold value for converting grayscale images into binary images, and the Watershed class, which performs the marker controlled watershed segmentation. The following sections provide details on the implementation of these helper classes.

4.2.1 OtsuThreshold

The OtsuThreshold class is used to determine an adequate threshold value during the process of converting a grayscale image into a binary image. The constructor of OtsuThreshold, seen in Figure 4.11, takes an instance of a GrayscaleImage as its only parameter. The constructor stores the GrayscaleImage instance as a field *gray* along with other fields for *width*, *height*, and number of pixels (*pixels*). A histogram for the

GrayscaleImage is then calculated in the field *histData* using the *calculateHistogram* method.

```
public OtsuThreshold(GrayscaleImage src){
    gray = src;
    width = gray.getWidth();
    height = gray.getHeight();
    pixels = width * height;
    calculateHistogram();
}

private void calculateHistogram(){
    for(int y = 0; y < height; y++){
        for(int x = 0; x < width; x++){
            histData[gray.getPixel(x, y)]++;
        }
    }
}
```

Figure 4.11: OtsuThreshold Contruction

Once an instance of OtsuThreshold is created, the *determineThreshold* method calculates and returns the selected threshold value. Figure 4.12 illustrates the code for *determineThreshold*. Each valid unsigned single byte grayscale value is visited to calculate the between class variance (*varBetween*). Along the way the maximum between class variance is stored in *varMax*, and the index of the maximum is stored in *threshold*. When all values are checked, *threshold* is returned.

```
public int determineThreshold(){
    float sum = 0;
    for(int i=0; i<256; i++)
        sum += i * histData[i];

    float sumB = 0;
    int wB = 0;
    int wF = 0;
    float varMax = 0;
    int threshold = 0;

    for(int i=0; i < 256; i++){
        wB += histData[i];               // Weight Background
        if(wB == 0) continue;

        wF = pixels - wB;                // Weight Foreground
        if(wF == 0) break;

        sumB += (float)(i * histData[i]);

        float mB = sumB / wB;            // Mean Background
        float mF = (sum - sumB) / wF;    // Mean Foreground

        // Calculate Between Class Variance
        float varBetween = (float)wB * (float)wF * (mB - mF) * (mB - mF);

        // Check if new maximum found
        if(varBetween > varMax){
            varMax = varBetween;
            threshold = i;
        }
    }
    return threshold;
}
```

Figure 4.12: OtsuThreshold's *determineThreshold* Method

4.2.2 Watershed

The Watershed class performs the marker controlled watershed segmentation that is used in the algorithm for the detection and extraction of individual leaves. During construction of a Watershed instance, as can be seen in Figure 4.13, two GrayscaleImage instances are required as input. The first GrayscaleImage, which is stored in the field *gray*, is a gradient

image that is interpreted as three-dimensional terrain during the flooding stage of the watershed segmentation. The second GrayscaleImage, *regionImage*, has its pixel values set as region values that will be used as the markers for the flooding. Once the necessary fields have been set, the *flood* method is called to start the segmentation.

```
public Watershed(GrayscaleImage sobel, GrayscaleImage regionImage){
    gray = sobel;
    this.regionImage = regionImage;
    width = sobel.getWidth();
    height = sobel.getHeight();
    flood();
}
```

Figure 4.13: Watershed Construction

Figure 4.14 provides the implementation of the *flood* method that performs the segmentation algorithm. The water level starts at the lowest possible value of zero. Every pixel is then visited, and those that have the value of zero, which represents unlabeled, and have a value less than the water level are tested by the method *labeledNeighbors*. The method *labeledNeighbors* returns the value of a pixel's neighbors if at least one of the pixel's neighbors is not labeled zero and all of the pixel's non-zero labeled neighbors share the same label. Otherwise, *labeledNeighbors* returns zero. If *labeledNeighbors* returns a non-zero value, the pixel being tested gets labeled with this value. The number of pixels added at the current water level is then incremented. The water level stays the same until no pixel is added during an iteration. Following this step, the water level is incremented by one and the process continues until the water level goes above the acceptable range.

```
private void flood(){
    int pixelsAdded = 0;
    int waterLevel = 0; // current water level
    while(waterLevel <= 255){ // until water level reaches the top
        for(int y=0; y<height; y++){
            for(int x=0; x<width; x++){
                if(regionImage.getPixel(x, y) == 0 &&
                                    gray.getPixel(x, y) <= waterLevel){
                    int label = labeledNeighbors(x, y);
                    if(label != 0){
                        regionImage.setPixel(x, y, label);
                        pixelsAdded++;
                    }
                }
            }
        }
        if(pixelsAdded > 0){ // go through at this level again
            pixelsAdded = 0;
        }
        else {              // increment water level
            waterLevel++;
        }
    }
}
```

Figure 4.14: Watershed's *flood* Method

4.3 The LeafExtractor Class

The LeafExtractor class implements the detection and extraction of individual leaf sub-images algorithm discussed in the methodology. Figure 4.15 illustrates the code responsible for construction of a LeafExtractor instance and outlines the previously mentioned algorithm. The input required to create a LeafExtractor instance is the original RGB BufferedImage containing the plant photograph. Initially, a GrayscaleImage, gradient image, and BinaryImage are created, which are used throughout the rest of the process. The *removeBackground* method converts all pixels that are determined as background in the BinaryImage to white in the original RGB BufferedImage. The stems and branches are then

filled which is a required step prior to the watershed segmentation. The watershed segmentation is then performed which detects all leaves or large objects within the image. The LeafRegion class stores the top, bottom, left, and right bounds of a possible leaf or large object, and the call to *flood.identifyRegions()* returns a collection of LeafRegion instances for all objects detected. The *createSubLeafImages* method analyzes each object detected by the program, removes those that it determines not to be leaves, and creates a sub-image for all probable leaves. Finally, each leaf image is padded by five pixels on each side to ensure that the leaf is not located on the border of the image. A call to LeafExtractor's *getLeafImages* method will return a collection of all individual leaf sub-images.

```
public LeafExtractor(BufferedImage image){
    this.image = image;
    GrayscaleImage gray = new GrayscaleImage(image);
    gray = toolbox.blurImage(gray);

    GrayscaleImage sobel = gray.sobel();            // Sobel Creation
    BinaryImage bin = createOtsuBinary(gray, 15);   // Binary Creation
    removeBackground(bin);

    // Fill Stems and Branches
    GrayscaleImage sobelFilled = fillStemsAndBranches(sobel);
    // Perform watershed to detect leaves or large objects
    Watershed flood = performWatershed(bin, sobelFilled);
    // Create a list of all leaves or large objects detected
    LeafRegion[] probableLeaves = flood.identifyRegions();
    GrayscaleImage regionImage = flood.getRegionImage();
    // Create sub-images for each leaf and rule out large objects that
    // are not leaves
    leafImages = createSubLeafImages(regionImage, probableLeaves);

    // Pad each leaf image on all sides by 5 pixels
    leafImages = padImages(leafImages, 5);
}
```

Figure 4.15: LeafExtractor Construction

4.4 The LeafCleaner Class

The LeafCleaner class implements the individual leaf image cleaning algorithm discussed in the methodology which includes the following steps: determination of the core leaf body, examination of border regions, and removal of selected regions. All three of these steps are implemented in the method *cleanLeaf*; however, discussion of each step will be broken down for better clarity. Figure 4.16 provides the code that implements the determination of the core leaf body.

To obtain the core leaf body, a BinaryImage *bin* is created. A copy of this BinaryImage, *binCopy*, is made such that the original is left unaltered, and the copy has the morphological operation of closing performed on it, leaving only the foreground known to be part of the leaf body. Due to the fact that individual leaf sub-images vary in size, the size of the structuring element has to be proportional to the size of the image. Through experimentation, it was determined that a structuring element one-eighth of the largest width or height of the image provided the best results.

```
BinaryImage bin = new BinaryImage(image);
int size;
int height = bin.getHeight();
int width = bin.getWidth();
if (height > width) {
    size = height;
} else {
    size = width;
}

size = size >> 3; // divided by 8

BinaryImage binCopy = bin.copy();
binCopy = binCopy.open(size); // binCopy is now core leaf body
```

Figure 4.16: LeafCleaner's *cleanLeaf* Method Part 1

Figure 4.17 illustrates the examination of border regions step of the leaf cleaning algorithm. To obtain a BinaryImage containing only the border regions, the regions removed during the open operation, BinaryImage's *diffImage* method is used to obtain a new BinaryImage called *diff* that contains the foreground regions present in *bin* but not present in *binCopy*. Small or insignificant regions that contain less than ten pixels are then removed for better efficiency during the examination process. A copy of the border regions BinaryImage is then created that will store the regions selected for removal later. Next, an array of BinaryImages is created that stores a unique BinaryImage instance for each region. Each BinaryImage in this collection contains as foreground only one unique region. Each region is then examined to determine what percentage of pixels in the region border a foreground pixel in the core leaf body. This value is referred to as a region's connectivity. Regions that have a connectivity of greater than 16% are determined to be part of the leaf and are removed from the *toRemove* instance.

```
// Create an image containing all foreground regions removed by the
// open operation.
BinaryImage diff = bin.diffImage(binCopy);
// Discard small (insignificant) regions by removing them.
diff = diff.removeSmallRegions();
// Create an image to store the regions that will eventually be removed
BinaryImage toRemove = diff.copy();
// Get a list of individual regions (These are the sections that may or
// may not need to be removed to clean the image).
BinaryImage[] regionList = diff.individualRegionsList(diff);
// Create a list to store the ratio of border pixels to non-border
// pixels for each image
float[] borderPercentList = new float[regionList.length];
for(int i=0; i < regionList.length; i++){
        // Create a binary image of only this region
    BinaryImage region = bin.minus(regionList[i]);
    int borderPixels = 0;
    // Count the number of border pixels in region
    for(int y=0; y < height; y++){
        for(int x=0; x < width; x++){
            if(regionList[i].getPixel(x, y) == 1 &&
                                isBorderPixel(x,y,region))
                borderPixels++;
        }
    }
    borderPercentList[i] = (float)borderPixels /
            (float)regionList[i].countWhitePixels(regionList[i]);
    // Remove region from toRemove image if ratio > 16%.
    if(borderPercentList[i] > 0.16)
        toRemove = toRemove.minus(regionList[i]);

}
```

Figure 4.17: LeafCleaner's *cleanLeaf* Method Part 2

Through experimentation it was determined that when more than three regions are selected to be removed, several of the removed regions are in fact part of the leaf. For this reason, when more than three regions are selected to be removed, only the region with the lowest connectivity is removed. Figure 4.18 provides the code for the removal of selected regions step of the leaf cleaning algorithm. First, all of the regions are visited to determine

which region has the lowest connectivity. Then, if the *toRemove* instance has more than three regions, all regions except for the region with the lowest connectivity is removed from *toRemove*. The regions that are selected to be removed are then dilated with a structuring element of size three which has the effect of expanding each foreground region by one pixel on each side. This step is necessary to ensure that all of the selected regions will be removed from the original image. Finally, all of the regions selected for removal are removed from the original RGB image resulting in a cleaned leaf image.

```
float lowestBorderPercent = 1; // region with lowest ratio
int lowestBorderIndex = -1;    // index of region with lowest ratio
for(int i=0; i < regionList.length; i++){
    if(borderPercentList[i] < lowestBorderPercent){
        lowestBorderPercent = borderPercentList[i];
        lowestBorderIndex = i;
    }
}

// if more than 3 regions are selected to be removed, it was determined
// that valid leaf parts are usually removed as well. In this case,
// remove only the region with the lowest ratio as long as it is <= 16%
if(toRemove.countRegions() > 3 && lowestBorderPercent <= 0.16)
    toRemove = regionList[lowestBorderIndex];

// Dilate the regions to remove by 1 pixel in every direction to ensure
// that all non-leaf pixels will be removed
toRemove = toRemove.dilate(3);
// Remove selected regions.
removeFromImage(toRemove);
```

Figure 4.18: LeafCleaner's *cleanLeaf* Method Part 3

CHAPTER 5: RESULTS

Twenty-one plant images obtained from the Irvin Watson Carpenter, Jr. Herbarium were selected to test the algorithms described in chapters 3 and 4. These images contained a total of 84 leaves with an average of 4 leaves per plant image. The minimum number of leaves in a plant image was 1, and the maximum number of leaves in a plant image was 7. The remainder of this chapter will separately analyze the effectiveness of the detection and extraction of individual leaves algorithm and the individual leaf cleaning algorithm. The last section analyzes the effectiveness of the overall approach.

5.1 Detection and Extraction of Individual Leaves

The detection and extraction of individual leaves algorithm accurately extracted the correct leaf sub-images for all 21 plant images. Figure 5.1 provides an example of an input image and the individual leaf sub-images that were produced from this algorithm. The results of the detection and extraction of individual leaves algorithm for all of the 21 plant images used in this thesis are provided in Appendix A.

Figure 5.1: Left - Input Image, Right - Individual Leaf Sub-images Produced by the Detection and Extraction of Individual Leaves Algorithm

The marker controlled watershed segmentation algorithm was successful in isolating only leaf objects in all but one of the plant images. The image in which the watershed algorithm failed to correctly isolate only the leaves is displayed in Figure 5.2. This plant contains branches that are close to being as large as the leaves themselves. The marker-controlled watershed segmentation algorithm detected five possible leaf objects in this image, the three leaves and two large branch sections. The algorithm designed to rule out non-leaf objects, however, successfully removed the two large branch regions that the marker-controlled watershed segmentation algorithm identified as possible leaf objects.

Figure 5.2: Plant Image Where Non-leaf Objects Were Identified as Possible Leaves by Watershed

5.2 Cleaning Leaf Images

To analyze the results of the leaf image cleaning algorithm, the original sub-images, that were produced by the leaf detection and extraction algorithm, were compared to the sub-images that were produced by the leaf cleaning algorithm. During this comparison, leaf images were marked with the following characteristics: non-leaf object(s) removed (OR), non-leaf object(s) not removed (ONR), and leaf part(s) removed (LPR). Figure 5.3 illustrates examples of these characteristics.

Figure 5.3: Top Row – OR, Center Row – ONR, Bottom Row – LPR

Forty-six out of the total 84 leaf images (57.76%) were changed in some way during the cleaning process while 38 (45.24%) remained unaltered. The number of leaf images where a non-leaf object was removed was 43 which accounts for 51.19% of the total 84 leaf images and 93.48% of the 46 images that received alterations during the cleaning process. Twenty-three of the leaf images still contained a non-leaf object after the cleaning process which accounts for 27.38% of the total 84 leaves and 50% of the 46 images altered by the cleaning process. Only 9 or 10.71% of the total 84 leaf images had some leaf part removed which accounts for 19.57% of the 46 leaves that were changed during the cleaning process.

To measure the overall success of the leaf cleaning algorithm, each cleaned leaf image was categorized into one of the following categories:

- The basic shape of the leaf was unaltered and all non-leaf objects were removed. (Perfect)
- The basic leaf shape was slightly altered and/or one or more very small non-leaf objects were not removed. (Acceptable)
- The basic leaf shape was altered significantly and/or one or more significant non-leaf objects were not removed. (Failure)

As Figure 5.4 illustrates, 54 (64.29%) of the total 84 leaf images were categorized as perfect, while the remaining 30 (35.71%) were categorized as acceptable. None of the leaves were categorized as failures. It should be noted that 20 (23.81%) of the total 84 leaf images were not altered in any way during the cleaning process and were still categorized as perfect. The remaining 18 unaltered leaf images where categorized as acceptable. The results of the leaf image cleaning algorithm for all of the 84 leaf images used in this thesis are provided in Appendix B.

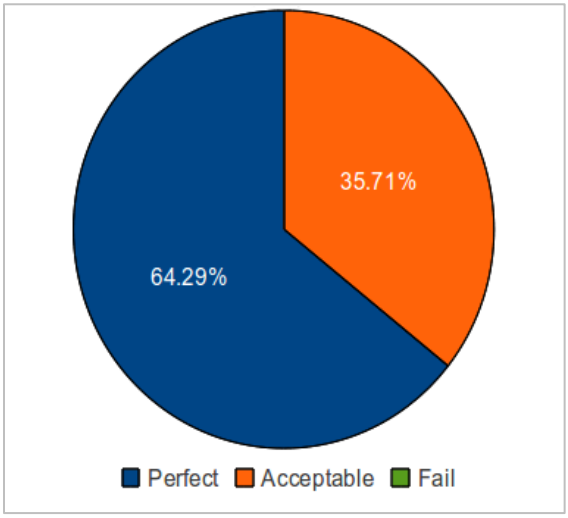

Figure 5.4: Leaf Image Cleaning Algorithm Results

5.3 Overall Algorithm Results

The overall goal for the leaf extraction algorithm was to provide an automated solution that would be capable of producing leaf images that could successfully be used in species classification. For a leaf image to be successfully used in species classification, a target leaf should be provided that retains the unique shape of the individual leaf while all non-leaf objects are removed. To determine how successful the leaf extraction algorithm was at achieving this goal, each of the 21 plant images were divided into two categories, successful or unsuccessful. Successful plant images were those that resulted in at least 1 leaf categorized as perfect by the leaf cleaning algorithm while unsuccessful plant images were those that resulted in no leaves categorized as perfect. Of the total 21 plant images, 18 (86%) were determined to be successful while only 3 (14%) were determined to be unsuccessful. It should also be noted that 8 (38%) of the total 21 plant images resulted in all leaves categorized as perfect by the leaf cleaning algorithm and 16 (76%) had at least half of their leaves categorized as perfect.

CHAPTER 6: CONCLUSIONS

6.1 Outcomes

The algorithm designed to detect and extract individual leaf images successfully detected 100% of the leaves within the 21 plant images and extracted no non-leaf objects. This algorithm also produced leaf images categorized as perfect by the leaf cleaning algorithm for 23.81% of the leaves. This algorithm, however, requires that none of the leaves are overlapping, and if overlapping leaves are present within an image the algorithm generally extracts a single image for the whole section.

The 21 plant images used in this research contained only plants with a near white background. When provided with images containing complicated backgrounds such as grass, the sky, or buildings, the extraction algorithm was far less effective. When working with images containing complicated backgrounds, Otsu's thresholding algorithm produced a binary image where parts of the background with lighter values were determined to be foreground and parts of the plants with darker values were determined to be background.

The algorithm designed to clean the leaf images produced perfect leaves 64.29% of the time, acceptable leaves 35.71% of the time, and no failed leaves. The majority of leaves determined to be acceptable contained only very small stems and would most likely produce satisfactory results if used in species classification. The task of cleaning leaf images provides a challenging problem in that non-leaf objects often contain the same characteristics as leaf tips or points. A threshold must be determined to remove as many of the non-leaf objects as

possible without removing the leaf tips or points. The proposed algorithm resulted in only 10.71% of leaves where tips or points were removed and 51.19% of leaves where some non-leaf object was removed. When compared to only those leaves that received alterations during the cleaning process, these values are 19.57% of leaves where tips or points were removed and 93.48% of leaves where some non-leaf object was removed.

When combining the extraction algorithm and the cleaning algorithm, the overall algorithm produced successful results for 85.71% of the plant images tested. Of the remaining 14.29% that were determined to be unsuccessful due to no perfect leaves, at least one leaf image was acceptable enough to get promising results if used in species classification.

6.2 Future Work

The proposed algorithms do not provide a solution capable of fully automated plant species classification from a plant image. They do, however, provide several successful solutions to some of the tasks necessary for such a system. For a solution capable of fully automated plant species classification to become a reality, portions of the proposed algorithms need to be improved or have functionality added, and the tasks not implemented by the proposed algorithms need to be addressed.

The images used in this research are plants without a background. The leaf detection and extraction algorithm does not provide adequate results for images with complicated backgrounds. For images containing complicated backgrounds, a new algorithm could be designed to separate the plant from the background before it is provided to the leaf detection and extraction algorithm.

When taking photographs of leaves, it is usually very difficult to capture an image where none of the leaves are overlapping. In the case of overlapping leaves, there is usually at least one whole leaf on top with other partially occluded leaves behind it. An new algorithm could be developed to remove the partially occluded leaves and retain the original shape of the whole leaf. Such an algorithm could be added to the cleaning algorithm proposed to provide an adequate solution to the problem.

The cleaning algorithm proposed provides adequate results for species classification by retaining the original shape of most leaves while removing the majority of non-leaf objects. Species classification relies heavily upon the shape of the leaf, but the proposed algorithm is not perfect. More research in this area could provide better solutions to the problem which would directly result in better results from species classification.

The proposed algorithm does not provide a way to automatically detect the best leaf for species classification. For a fully automated system, this is an essential task that needs an adequate solution. One possibility is to try all leaves classified as perfect within an image to see if the recognition algorithm agrees on a species. While a fully automated system for plant species classification is not provided by this research, this research provides sufficient evidence that an adequate solution is possible.

REFERENCES

[1] Camargo Neto, J., Meyer, G.E., & Jones, D.D. (2006). Individual leaf extractions from young canopy images using gustafson-kessel clustering and a genetic algorithm. *Computers and Electronics in Agriculture, 51*, 66-85.

[2] Manh, A.G., Rabatel, G., Assemat, L., & Aldon, M.J. (2001). Weed leaf image segmentation by deformable templates. *Journal of Agricultural Engineering Research, 80*(2), 139-146.

[3] Tang, X., Liu, M., Zhao, H., & Tao, W. (2009). Leaf extraction from complicated background. Proceedings of the 2009 2nd international congress on image and signal processing (pp. 1-5). 10.1109/CISP.2009.5304424

[4] Wang, X.F., Huang, D.S., Du, J.X., Xu, H.X., & Heutte, L. (2008). Classification of plant leaf images with complicated background. *Applied Mathematics and Computation, 205*(2), 916-926.

[5] Gonzalez, R.C., & Woods, R.E. (2008). *Digital image processing*. Upper Saddle River, NJ: Prentice Hall.

[6] Otsu, N. (1979). A threshold selection method from gray-level histograms. IEEE *Transactions on Systems, Man, and Cybernetics, 9*(1), 62-66.

APPENDIX A

Results of Leaf Detection and Extraction Algorithm

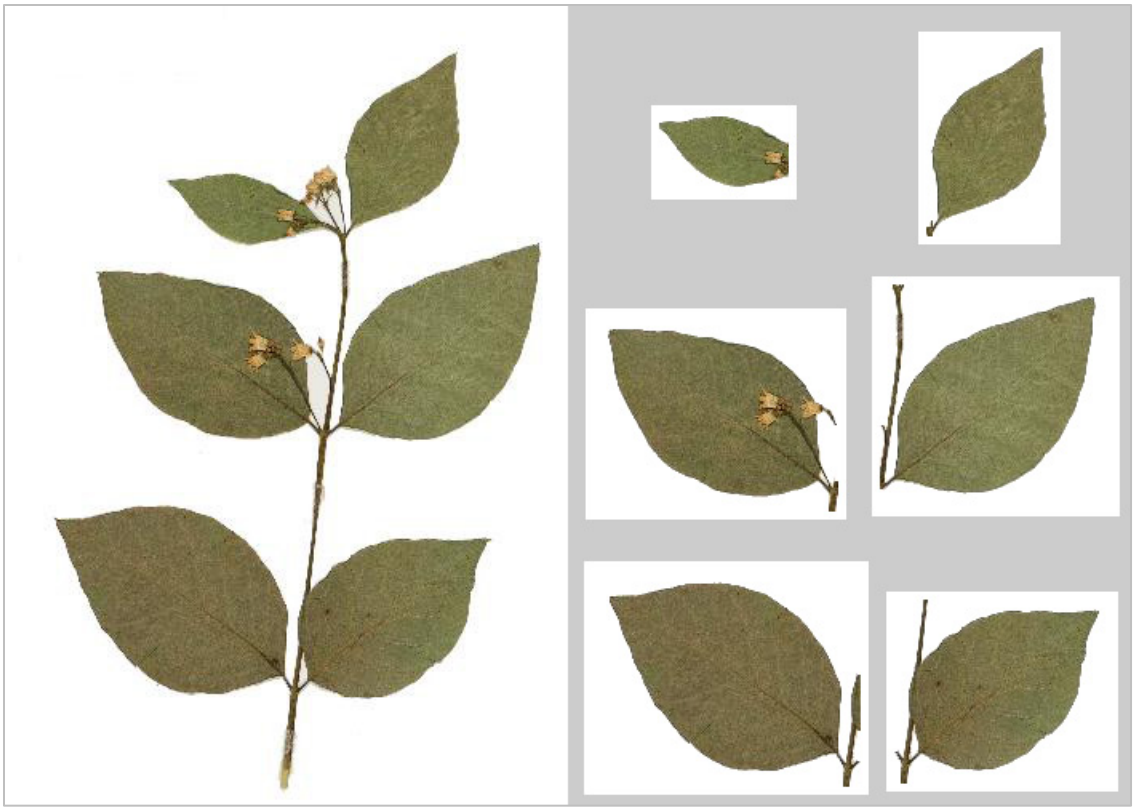

Figure A.1: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.2: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.3: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.4: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.5: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.6: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.7: Left – Original Image, Right – Extracted Leaf Sub-images

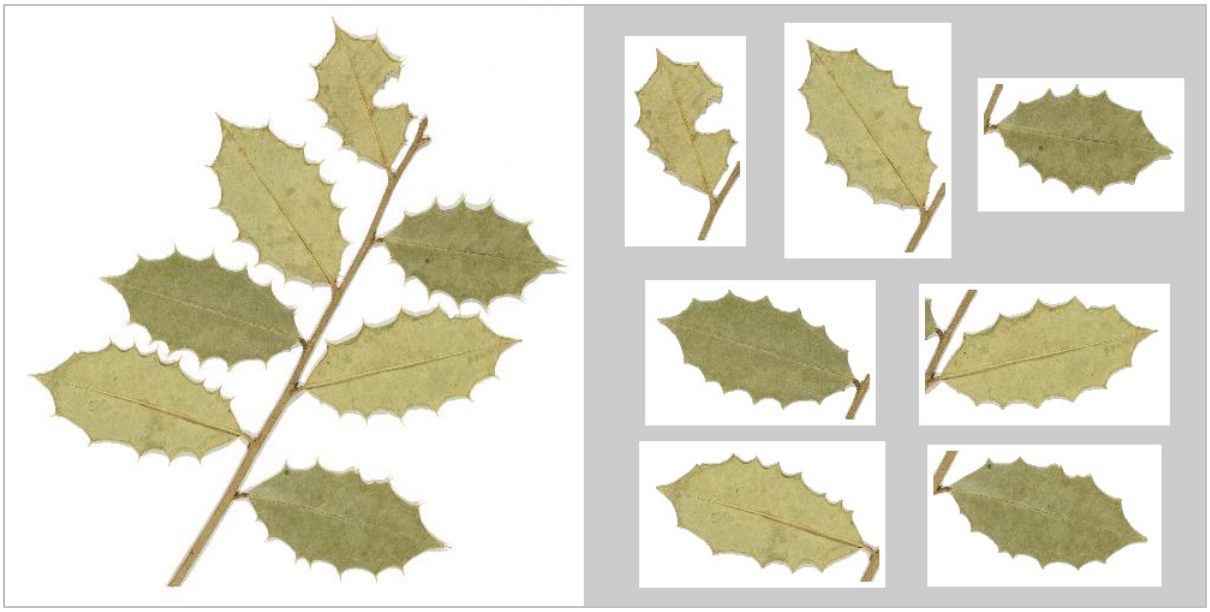

Figure A.8: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.9: Left – Original Image, Right – Extracted Leaf Sub-images

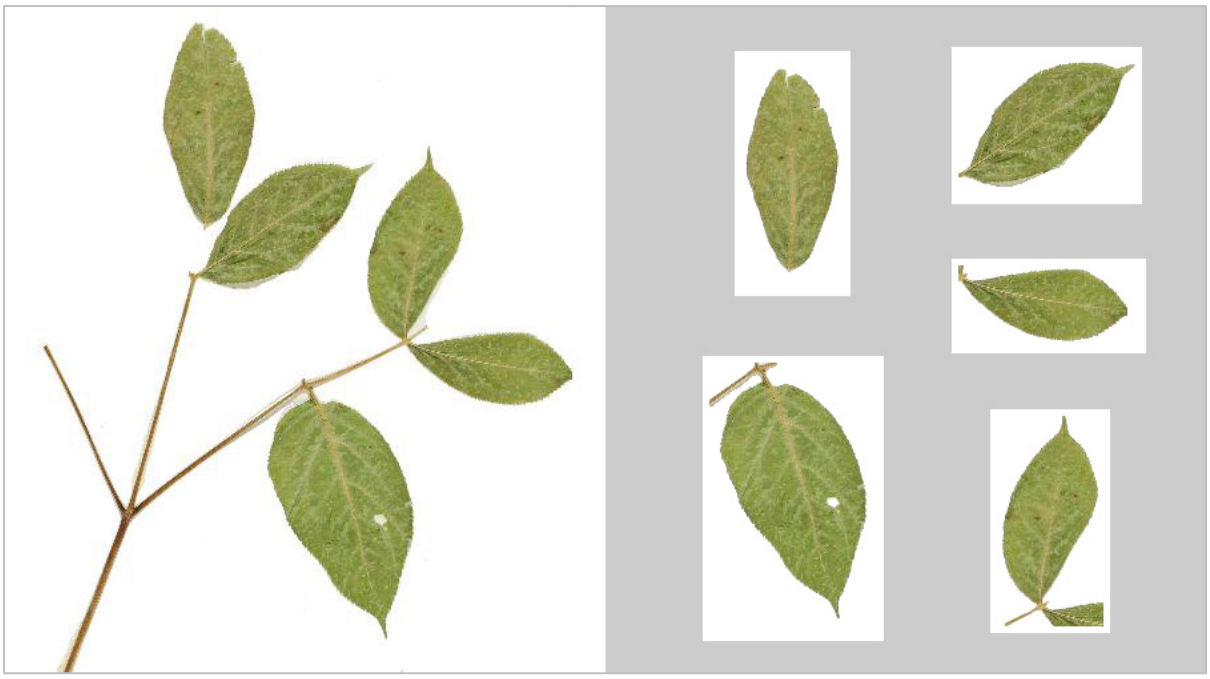

Figure A.10: Left – Original Image, Right – Extracted Leaf Sub-images

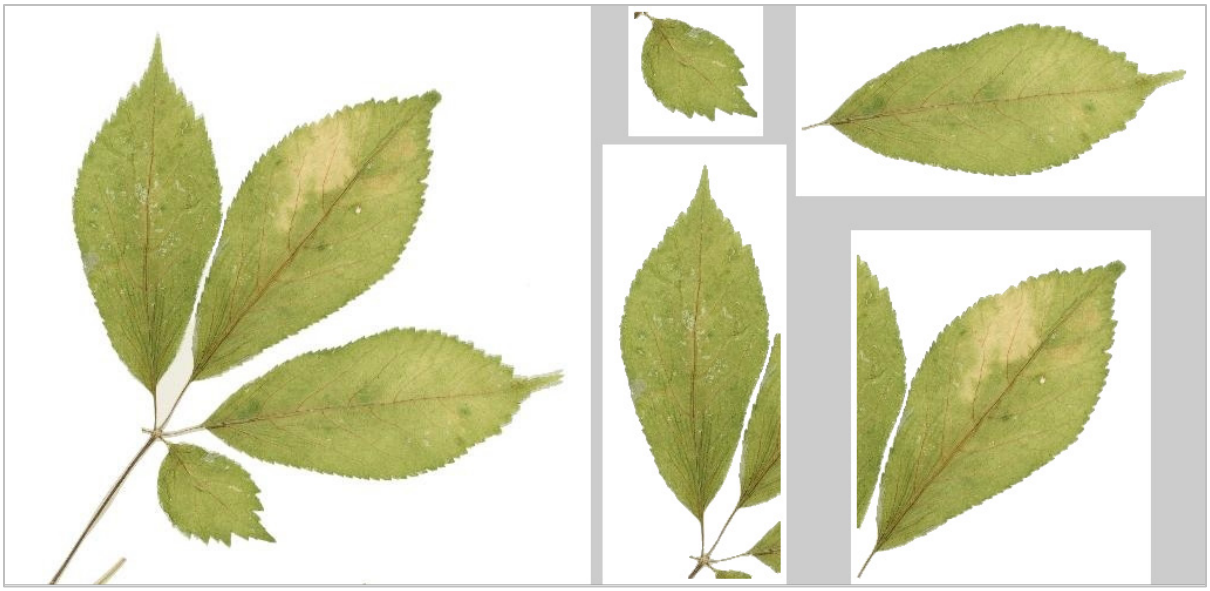

Figure A.11: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.12: Left – Original Image, Right – Extracted Leaf Sub-images

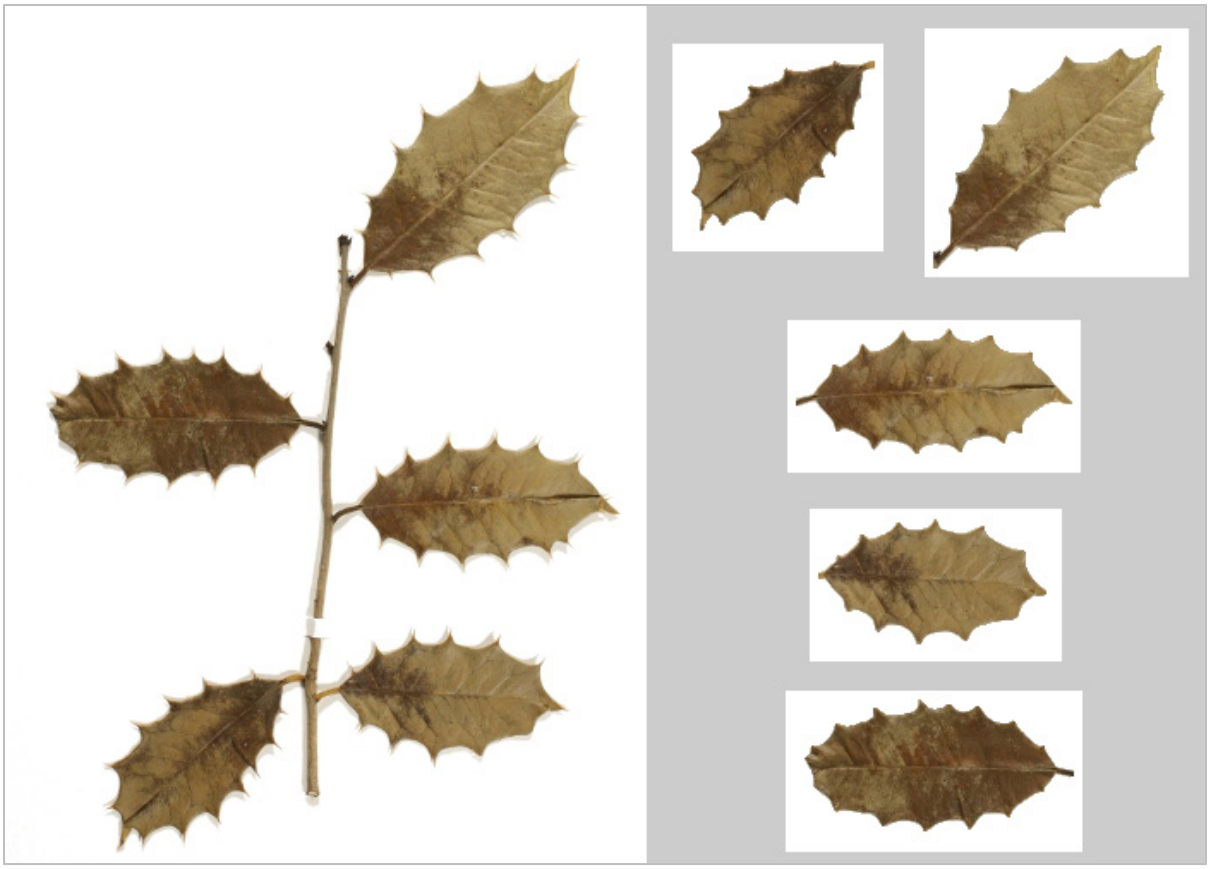

Figure A.13: Left – Original Image, Right – Extracted Leaf Sub-images

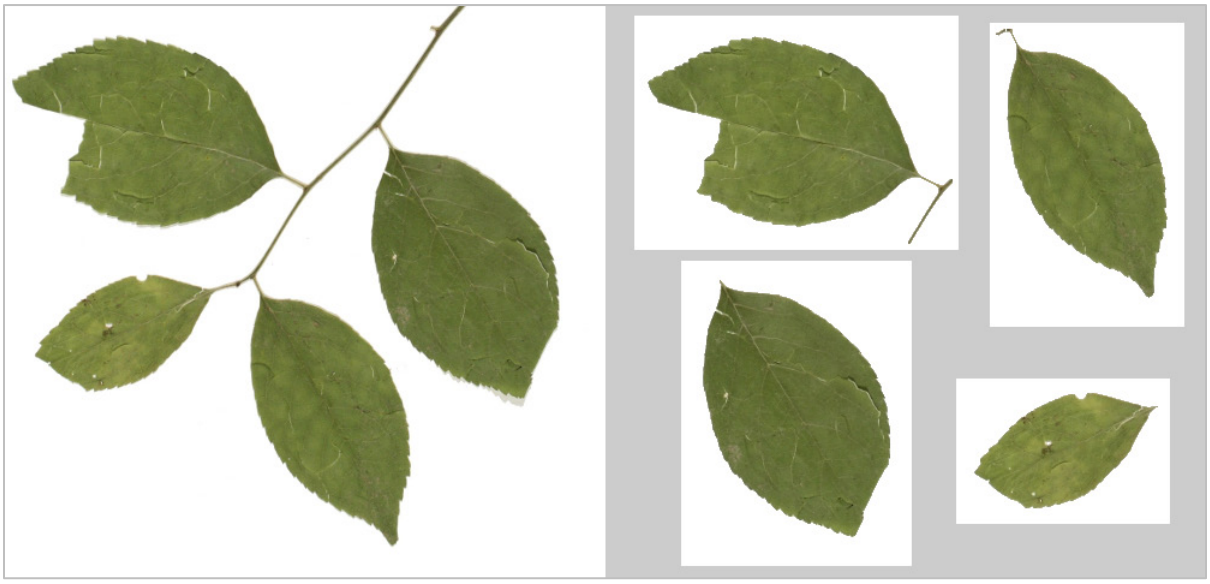

Figure A.14: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.15: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.16: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.17: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.18: Left – Original Image, Right – Extracted Leaf Sub-images

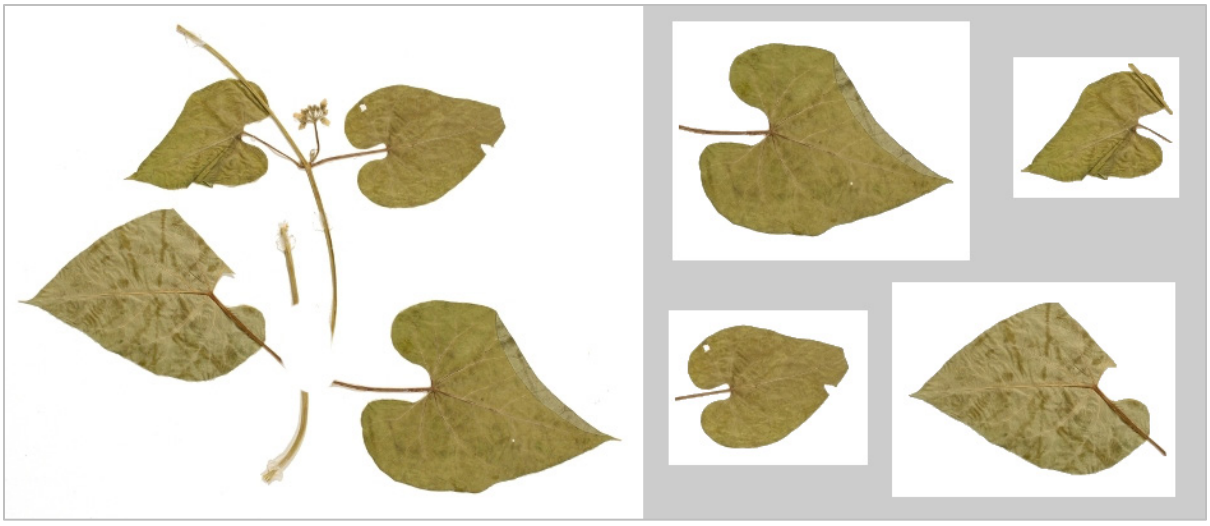

Figure A.19: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.20: Left – Original Image, Right – Extracted Leaf Sub-images

Figure A.21: Left – Original Image, Right – Extracted Leaf Sub-images

APPENDIX B

Results of Leaf Cleaning Algorithm

Figure B.1: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.2: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.3: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.4: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.5: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.6: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.7: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.8: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

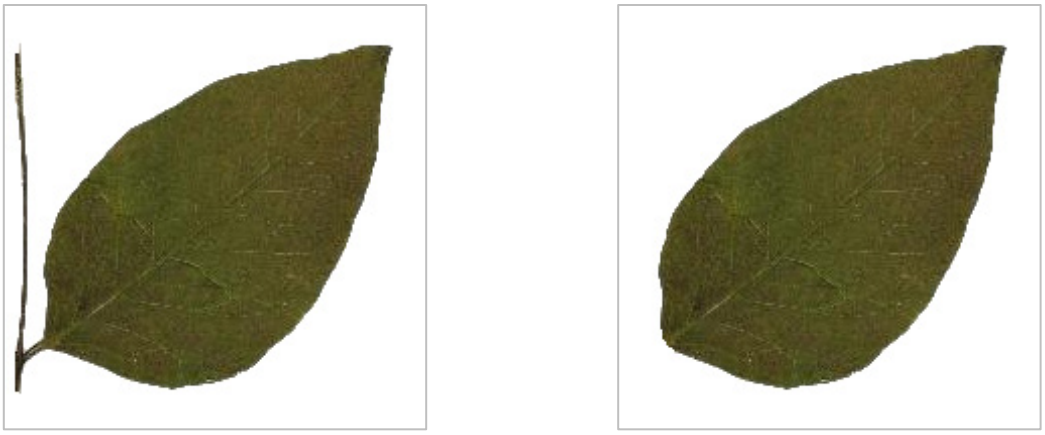

Figure B.9: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.10: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.11: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.12: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.13: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.14: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.15: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.16: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.17: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.18: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.19: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.20: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.21: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.22: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.23: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.24: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.25: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.26: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.27: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.28: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.29: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.30: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.31: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.32: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.33: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.34: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.35: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.36: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.37: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.38: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.39: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.40: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.41: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.42: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

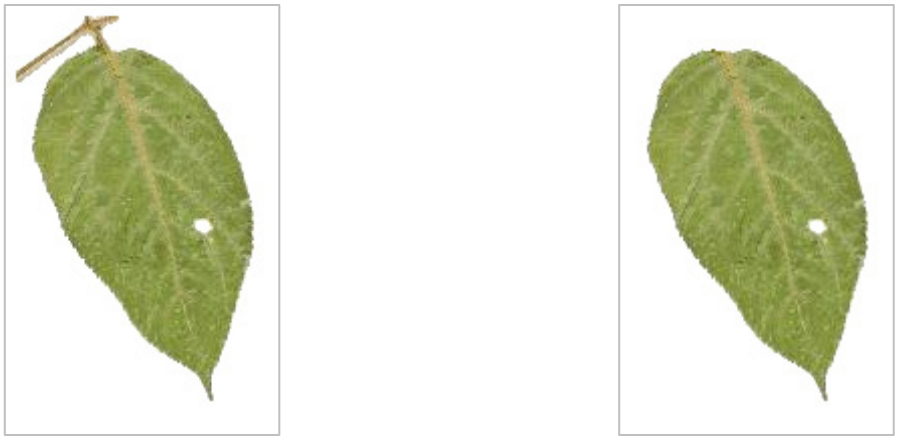

Figure B.43: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.44: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.45: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.46: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.47: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

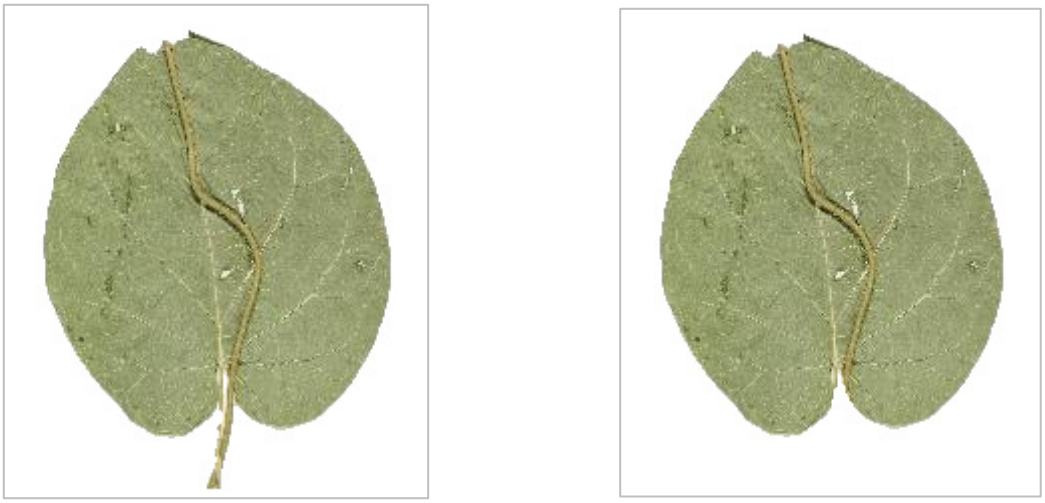

Figure B.48: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.49: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

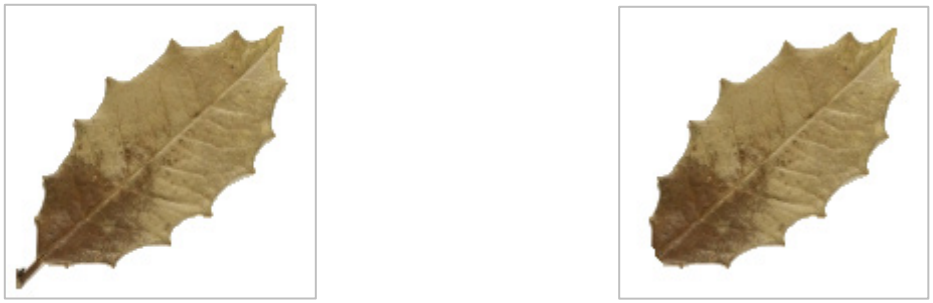

Figure B.50: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.51: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.52: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.53: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.54: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.55: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

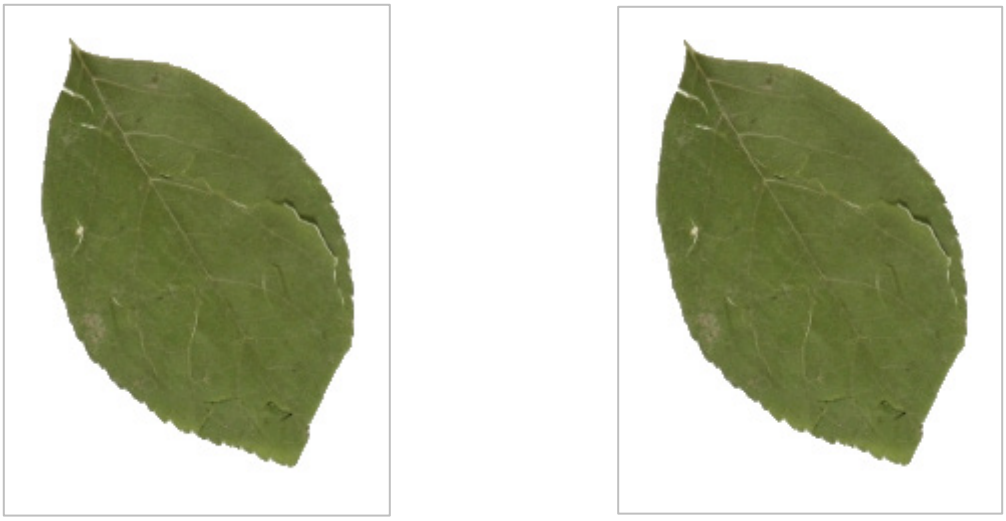

Figure B.56: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.57: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.58: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.59: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.60: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.61: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.62: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.63: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.64: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.65: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.66: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.67: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.68: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.69: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.70: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.71: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

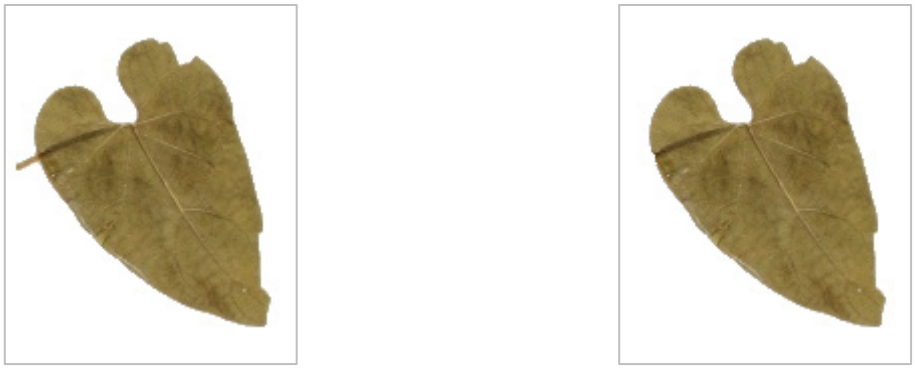

Figure B.72: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.73: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.74: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.75: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.76: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.77: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

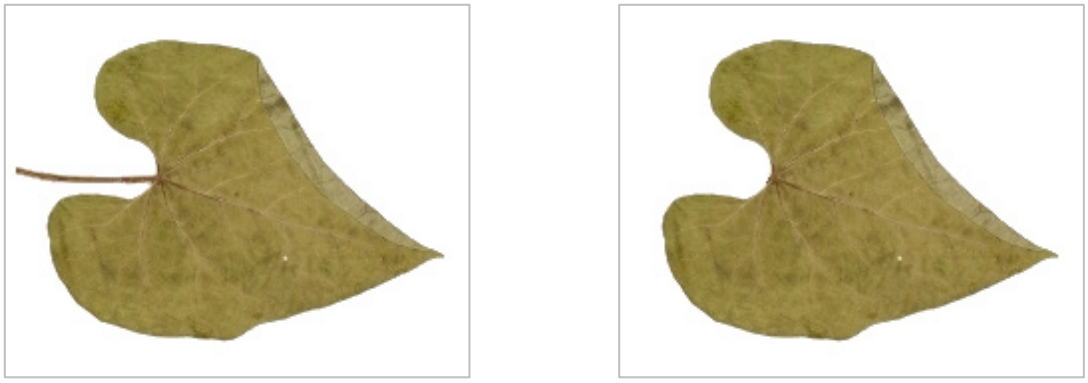

Figure B.78: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

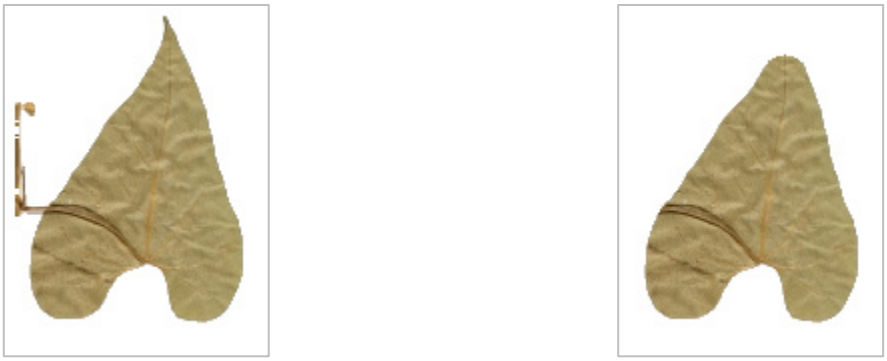

Figure B.79: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.80: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.81: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.82: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.83: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

Figure B.84: Left – Original Leaf Sub-image, Right – Leaf Sub-image Cleaned

VITA

Dale Garrett Henries was born in Boone, North Carolina in 1984. He graduated from Watauga High School in 2003, and began attending Appalachian State University the same year. While taking electives to finishing his undergraduate degree in Criminal Justice, he discovered a passion for Computer Science. In August of 2008 he both graduated with his Bachelor of Science in Criminal Justice and immediately began to pursue a Master of Science in Computer Science. Mr. Henries received the Master of Science in Computer Science degree in August of 2011.