Mobile High-Throughput Phenotyping using Watershed Segmentation Algorithm

by

Shravan Dammannagari Gangadhara

B. Tech., CVR College of Engineering, Jawaharlal Nehru Technological University, India, 2015

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2017

Approved by:

Major Professor
Dr. Mitchell L. Neilsen

Copyright

Abstract

This research is a part of BREAD PHENO, a PhenoApps BREAD project at K-State which combines contemporary advances in image processing and machine vision to deliver transformative mobile applications through established breeder networks. In this platform, novel image analysis segmentation algorithms are being developed to model and extract plant phenotypes. As a part of this research, the traditional Watershed segmentation algorithm has been extended and the primary goal is to accurately count and characterize the seeds in an image. The new approach can be used to characterize a wide variety of crops. Further, this algorithm is migrated into Android making use of the Android APIs and the first ever user-friendly Android application implementing the extended Watershed algorithm has been developed for Mobile field-based high-throughput phenotyping (HTP).

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my sincere gratitude to my Advisor, Mentor, Instructor and Major Professor, Dr. Mitchell L. Neilsen for his constant support and for trusting my abilities to complete this project. I owe it all to him. Many Thanks!

I am also grateful to my committee members Dr. William Hsu and Dr. Torben Amtoft for their encouragement and taking their time to serve on my committee.

My special gratitude goes out to my cousin Sri Mithra Vemula for believing in me and guiding me all the way in this research. I will miss our long lasting late night discussions over phone calls. I will miss our screams of joy whenever we cracked something. Hope we continue like this for the rest of our lives.

I would also like to acknowledge the support of National Science Foundation by providing funding for this work. Further, this material is based upon work supported by the National Science Foundation under Grant No. 1543958. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

I would like to thank my family for their immense love and belief. And finally, last but by no means least, also to my friends Anamika Nupur Choudhary, Nikitha Vootla and Chandravyas Annakula for their love and support. It is great having them in my life. I will always be grateful to them.

Chapter 1 - Introduction

Nutritional security is a big challenge in the coming decades. The demand for food is increasing with the rapidly increasing population. To handle this, advancements in Plant breeding to improve plant varieties is needed. Plant breeding is known as the manipulation of plant species in order to change the characteristics of plants to create new seed stock with desirable properties. Genotypes of plant species can easily be gathered with advanced technology but phenotyping is still in its infancy. A Genotype is a genetic constitution of a cell and a Phenotype refers to an organism's physical characteristics or properties. A Phenotype [2] results from the Gene expression of an organism's genetic sequence, its genotype and the environmental influence. Basically, different kinds of phenotypes can exist in same kind of species. Over the past 10 years, the genomic data has been vastly increasing whereas the methods to determine the corresponding phenotypes have been minimal. While the genomic data is wildly available, the corresponding phenotypes which help to study the functions of the genomes for selection of breeding programs have remained passive, notably for the phenotypes gathered from field trials in breeding programs [1]. There are hardly any advancements in the models to collect corresponding phenotypes. This has resulted in a major imbalance in the data sets linking genotypes and phenotypes. The need for advancement in the field of phenotype determination in Plant Breeding programs has been escalating. So, this project seeks to converge contemporary advances in image processing and machine vision to efficiently collect precision phenotypic data to decipher plant genomes and advance plant breeding [1]. This research is a part of BreadPheno, a project under Basic Research to Enable Agricultural Development (BREAD) by National Science Foundation (NSF).

The main aim of this project is to characterize and count the number of seeds present in an input image. This project strives to deliver transformative mobile applications through several breeder

networks [1]. To determine the number of seeds in a Wheat head, we usually rub the wheat heads with our hands and count them manually. This can lead to an accurate result as few seeds may be missed while counting as they are very small. Also, there is a highly likely chance for losing the seeds in the process of rubbing. So, all the seeds in a Wheat head are dropped onto a background and an image is clicked and is processed using this application. This application provides an accurate number of seeds which will be used further to estimate the yield of a crop. There are many such applications being developed in this research and this project strives to move up in the field of 3D graphics, modelling, data mining and deep learning through assimilation of concurrent ground truth phenotypic measurements and visualization with mobile technology. As a part of this research in this project, one of the novel image analysis algorithms known as Watershed segmentation algorithm has been analyzed. Though it has been very effective in image segmentation, the traditional Watershed algorithm has not been so effective in accurately extracting plant phenotypes [1]. So, this traditional algorithm has been extended to achieve field-based high-throughput phenotyping (HTP). Further, this extended Watershed algorithm is incorporated in an interactive mobile application and can be widely used for mobile HTP.

Watershed segmentation algorithm is used to separate the touching objects in an image by a process called flooding. This algorithm has several implementations and is widely used for extracting plant phenotypes. However, there is no mobile application currently which implements Watershed segmentation algorithm. Further, there are no image processing mobile applications which can efficiently collect and characterize phenotypic data. To meet this purpose, with an aim to advance plant breeding in this project, the first ever mobile application which can achieve High Throughput Phenotyping has been developed in this project. This user-friendly mobile application implements the extended Watershed segmentation algorithm to rapidly count and characterize

phenotypes of various seeds in a captured image. This application can rapidly process a captured image dynamically and characterizes the seeds. Hence, by focusing on the deployment of such advanced algorithms with the help of mobile applications, innovating phenotyping tools can be promptly delivered for broad practice. Thus, various breeders around the world can be equipped with which help them to rapidly collect, analyze and process several phenotypes and ultimately, we can lay the platform for advancements in productivity of food and nutrition security.

Chapter 2 - Background

Image Processing is a process of applying certain mathematical operations over an image to generate an output image which displays certain corresponding characteristics. Digital Image Processing (DIP) deals with the manipulation of images over a computer and Computer Vision [3] is a field which deals with training computers to understand and process digital images and videos efficiently. Image Processing overlaps significantly with Computer Vision. Computer Vision is a way to automate human vision with the means of a computer. Human vision cannot reconstruct or clearly interpret a digital image or a particular frame in a rapidly moving video. Hence, Computer Vision is used to reconstruct and interpret an image or a 3D video based on its 2D images in terms of properties corresponding to that image or video. Computer Vision involves tasks such as processing and analysis for obtaining high-level data and information related to the processed image. The hierarchy of Computer Vision consists of Low-Level vision, Middle-level vision and High-level vision. Low-level vision deals with feature extraction such as edge detection in an image. Middle-level vision uses the properties obtained from low-level vision to deal with object recognition, analysis of motion and 3D reconstruction. Further, High-level vision combines the properties gathered from low-level and middle-level visions and interprets the evolving information from the image or a video. High-level vision assists with behavioral description of a sequence of images or a scene. Computer Vision is also applicable when the image is irregular of noisy. Computer Vison is closely related to Pattern Recognition or Machine Learning. Machine Learning is a field of study which employs several statistical mathematical techniques for pattern classification. However, the input data for Machine Learning can be any kind of data. As Computer Vison deals with the same as Pattern Recognition, the former employs several techniques of latter. Open Source Computer Vision, most prominently known as OpenCV [6], is an Image Processing

computer vision library consisting of several programming functions to meet the requirements of real-time Computer Vision. OpenCV is open-source and is initially created by Intel and is currently maintained by Itseez [5]. There are numerous functions available in the OpenCV library which perform corresponding operations for image processing. OpenCV provides functionalities such as face detection, shape detection, text recognition, modification of image quality and type and many more. The OpenCV library is written in C and C++ languages and it can be run under Windows, Mac OS X and Linux environments. It can take wide advantage of multicore processors as it is written in C. It provides over 500 functions that cover many areas of computer vision and hence, it is perceived as a medium to make computer vision universally accessible.

The basic structures provided by Open CV are [7]:

- Point, Point2f – For a 2D point.
- Size – 2-Dimensional size structure
- Rect – 2-Dimensional rectangle object
- RotatedRect – Rectangle object (Rect) with an angle
- Mat – Image object.

Mat is the primary data structure in OpenCV [7]. It is used to store the image and it's components. An image is stored as a two dimensional matrix consisting of intensity values of the corresponding pixel points. Prior to Mat, in C, IPIImage data structure is used to store the image data. Manual memory management being its biggest disadvantage, after the evolution of C++, Mat object came into the lime light which requires no manual allocation of memory and also provides scope for memory reusability. A Mat object consists of two parts, namely matrix header and a pointer. The header consists of information such as matrix size, the storing method used and the address at which the matrix is stored. The pointer of a matrix contains the pixel values. Mat improves the

speed of the system by handling copy operations efficiently as they can eat up the processor time if the image to be copied is very large. To handle such cases, OpenCV implements a reference counting system. When a Mat object is copied from another Mat object, their headers vary but the pointers they point to will remain the same saving both the time and space. The main features provided by this data structure [7] are:

- rows, cols – These represent the height and width of an image (Mat).
- Channels – 1 channel represents a grayscale image and 3 channel image represents a three-channeled object consisting of Blue Green and Red layers.
- Depth – It is designed by CV_<depth> or C<number chan>

There are three different types of images. They are Binary image, Grayscale image and Colored image. Each pixel in a binary image can only have the intensity value of 0 or 1. 0 corresponds to Black color and 1 corresponds to White color. So, a binary image comprises of only black and white colors. As each pixel can take any of the two colors, the number of channels required for a binary image is 1 and the depth of the image will be 1 bit. Similarly, Grayscale and Color images differ accordingly. To handle these, OpenCV provides different storing methods [8]. Each method has a way of storing the pixels. We can select desired color space and the data type. A color space refers to a combination of color components to form a given color. Simplest color scale is the gray scale. Gray scale images contain colors which are combination of black and white pixel intensity values thus, forming an image consisting of several shades of gray. Other methods include RGB and HSV. In an RGB image, each pixel intensity value is a result of combining the three Red, Green and Blue pixel intensity values corresponding to that pixel. However, in OpenCV, the default ordering is BGR. The HSV method decomposed pixel colors into their corresponding natural Hue, Saturation, Value components. Hue Saturation Value (HSV) color space separates the

image intensity values from the color components [24]. The HSV color space gives the detail information related to the image whereas the RGB color space data is much noisy. In detail, the hue (H) of an image pixel represents the pure color it resembles [25]. The hue will be same for all the shades of a color. That is, it doesn't depend on the shade of the color based on certain amount and White and Black which dilute the concentration of the color. The Saturation (S) represents the whiteness of present in a shade a color of a pixel [25]. It gives the concentration of White shade in a given color of a pixel. The Value (V) represents the lightness or the dark shade of a color of the pixel [25]. The Value for Black is 0, if we are representing a lighter color, it is moving away from Black which increases its lightness. Its dark value depends on this lightness. The HSV representation can be visualized as follows

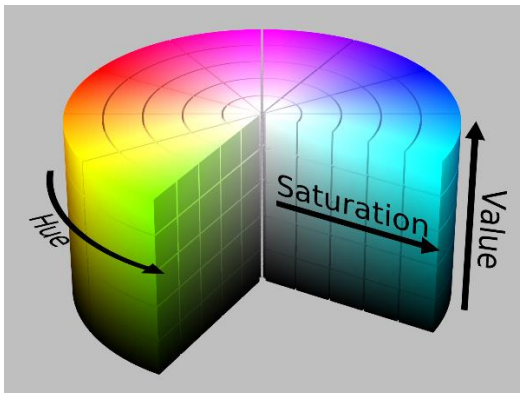

Figure 2-1 Hue, Saturation and Value plotted on a Cylinder [26]

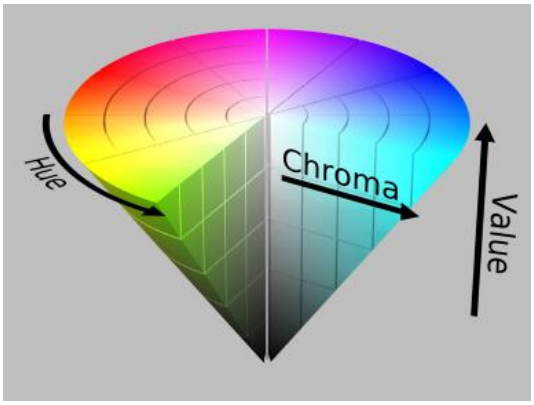

Figure 2-2 Hue and Value plotted on a Cone [26]

Other less popular methods include YCrCb and CIE L*a*b* [8]. Each of the color components of the corresponding storing methods have belong to a particular domain. So, while creating a Mat object, we need to specify the corresponding datatype for storing the pixels. In addition, we also need to specify the number of channels for our Mat object. So, the basic structure of an image TYPE is: CV_{U|S|F}C(<number_of_channels>). The {U|S|F} part represents the depth/data type of the Mat object. U represents Unsigned, S represents Signed and F represents Floating point. A much broader representation would be:

CV_<number_of_bits><U|S|F><type_prefix>C<number_of_channels>.

If the image is a Gray scale image, we just require a single channel matrix to represent that image as the pixels of a gray scale image are shades of black and white. For instance, we can use CV_8UC1, an 8-bit unsigned representation with a single channel as a gray scale image will have 2^8 that is, 256 different shades of gray which are combinations of Black and White. If the input image is a color image, then we need a 3 channeled matrix to represent this image. For instance, CV_8UC3 represents an 8-bit unsigned integer matrix consisting of 3 channels. Here, we are using 8 bits (depth) to represent a pixel intensity value ranging from 0 to 255. Here, we can use up to 8-bits for storage per channel (BGR). So, we can use up to 24 bits per each pixel of the matrix. Increasing the number of bits per channel will give more scope for larger shades of colors. Other possible depths are:

- CV_8U: 8-bit unsigned integer
- CV_8S: 8-bit signed integer
- CV_16U: 16-bit unsigned integer
- CV_16S: 16-bit signed integer
- CV_32S: 32-bit signed integer

- CV_32F: 32-bit floating point number
- CV_64F: 64-bit floating point number

Further, we can make use of a Scalar to pass the pixel values and to represent the color of the pixels. A Scalar [9] is a 4-element vector and has the form Scalar (a, b ,c) where a represents Blue (B), b represents Green (G) and c represents Red (R) if we use BGR as a storing method. For instance, Scalar (0, 0, 0) is a combination of BGR values which represent Black color. Scalar (255, 255, 255) represents White, Scalar (0, 0, 255) represents Red color, Scalar (0, 255, 255) represents Yellow color and so on.

Storage of Mat object in the memory:

The storage of the image which is represented by the Mat object depends on the storage type and the number of channels used [10]. For example, if we are representing a gray scale image using a Mat object, it is stored like something which looks like:

	Column 0	Column 1	Column ...	Column m
Row 0	0,0	0,1	...	0, m
Row 1	1,0	1,1	...	1, m
Row ...	...,0	...,1	...	..., m
Row n	n,0	n,1	n,...	n, m

Figure 2-3 Storage of a Grayscale image

If we are representing a color image using a Mat object with multiple channels, each column has as many sub columns as the number of channels defined. If we are talking about the BGR color space, it is stored in the memory as:

	Column 0			Column 1			Column ...			Column m		
Row 0	0,0	0,0	0,0	0,1	0,1	0,1	...	...	...	0, m	0, m	0, m
Row 1	1,0	1,0	1,0	1,1	1,1	1,1	...	...	...	1, m	1, m	1, m
Row ...	...,0	...,0	...,0	...,1	...,1	...,1	...	...	...	..., m	..., m	..., m
Row n	n,0	n,0	n,0	n,1	n,1	n,1	n,...	n,...	n,...	n, m	n, m	n, m

Figure 2-4 Color image storage

2.1 Image Segmentation

Image Segmentation is a part of Image Processing. It is a way of partitioning an image into a set of regions that it comprises of. A broader goal of Image segmentation is to partition an image into a collection of set of pixels which are connected. Segmentation is used to decompose the image into several parts which can be used for further analysis and then performing a transformation on the image for a meaningful representation of the pattern or connection. Thresholding is one of the simplest methods of Image segmentation. Here, the change is performed based on the user specified threshold. OpenCV provides cvThreshold(<args>) function to perform Thresholding on a given image. One of the primary argument is the threshold type which determines the type of transformation we are applying on the image. The different threshold types [11] provided by OpenCV are:

- CV_THRESH_BINARY: Assigns maximum value for a pixel if it's intensity value is more than the specified threshold. Assigns 0 otherwise.
- CV_THRESH_BINARY_INV: Assigns 0 for a pixel if it's intensity value is more than the specified threshold. Assigns the maximum value otherwise.
- CV_THRESH_TRUNC: Assigns the threshold value for a pixel if it's intensity value is more than the specified threshold. The pixel value is unchanged otherwise.
- CV_THRESH_TOZERO: The pixel value is unchanged if it's intensity value is more than the specified threshold. The pixel value is changed to 0 otherwise.
- CV_THRESH_TOZERO_INV: The pixel value is changed to 0 if it's intensity value is more than the specified threshold. The pixel value is unchanged otherwise.

There are several image segmentation methods [12] to partition an image based on different criteria. Some of them include: Region segmentation using clustering algorithms and Histogram-

based methods, Region growing, Edge detection, Hough Transform, flood fill, pyramid segmentation and some such. In this project, we performed Image Segmentation in OpenCV using Watershed segmentation algorithm.

Chapter 3 - Literature Survey

3.1 Watershed Segmentation algorithms

Watershed segmentation is a region-based segmentation method. As the name suggests, the term ‘Watershed’ defines an area or a ridge of land which separates waters flowing to different basins, rivers or seas. The input image is initially converted to gray scale before performing the watershed segmentation. Every gray scale image can be visualized topographically with peaks and valleys. High intensity pixels form the peaks and hills where as low intensity pixels form the valleys in the topography. Then we start flooding this topographical surface until all the peaks submerge in water and dams are built in between them to create partitions. There are two approaches for performing watershed segmentation., standard approach and marker based Watershed segmentation. In the standard approach, as stated earlier, the original image is converted into a gray scale image and each pixel represents the height of the surface in the image. Hence, all the pixels represent a geographical topography. The group of pixels with low intensity values become the valleys or the catchment basins and the pixels with higher intensities become the peaks [13].

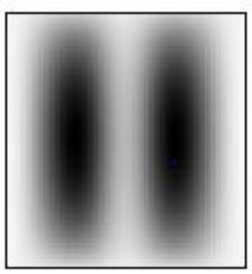

Figure 3-1 Grayscale image

In the above gray scale image, if we consider black pixels as low intensity values pixels and white pixels as high intensity values pixels, then the valleys (catchment basins) and peaks can be visualized as follows:

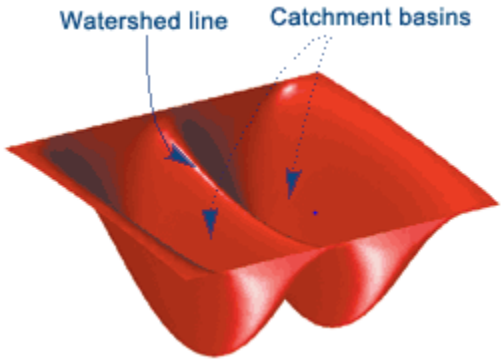

Figure 3-2 Corresponding topographic representation

Then, we gradually flood, starting from the low points (valleys). When waters from two different catchment basins merge, we build a dam to divide these two basins. These dams are called as watershed lines. Finally, we partition the image into catchment basins and corresponding watershed lines. To understand this approach, consider an image below

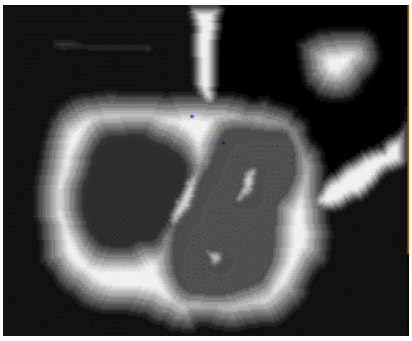

Figure 3-3 Initial Grayscale image

This is an input gray scale image. This can be visualized [4] as a topography as shown below

Figure 3-4 Corresponding topographic representation

Now, we start flooding this topographic surface with water starting from its minima regions (valleys) as follows:

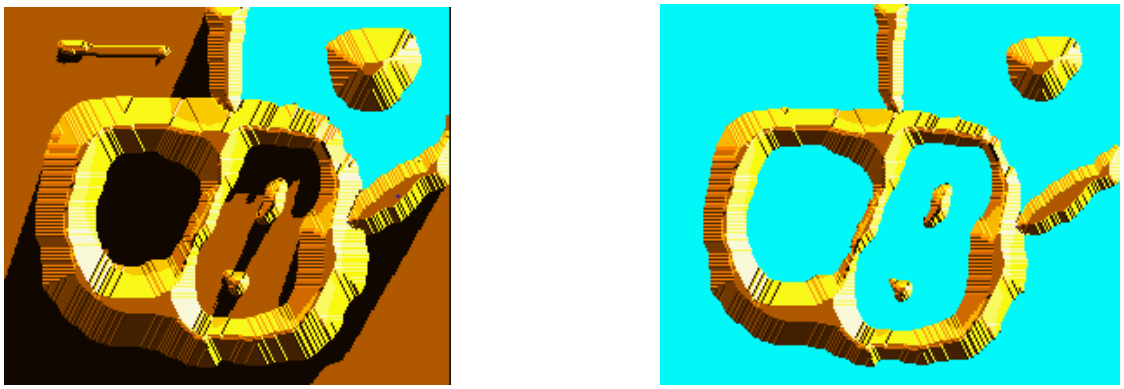

Figure 3-5 Initial flooding [4]

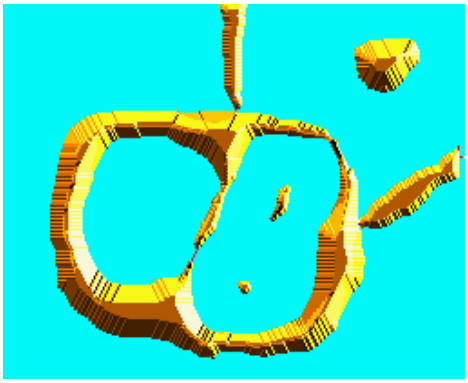

Figure 3-6 Flood until peaks submerge

When the water from two different catchment basins (valleys) begin to merge, we build a partition or dam to separate them as shown in the below figure.

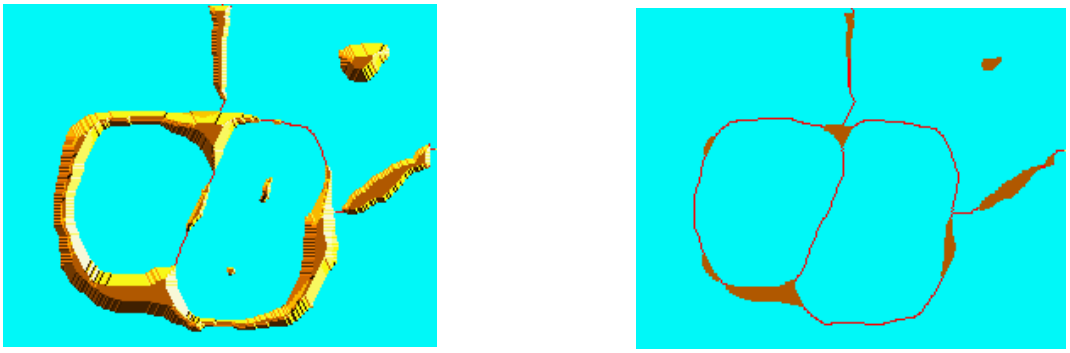

Figure 3-7 Build dams to avoid merging

Finally, when the highest peak is submerged in water, we stop the flooding process and we will have the watershed lines as shown in the below figure

Figure 3-8 Watershed lines

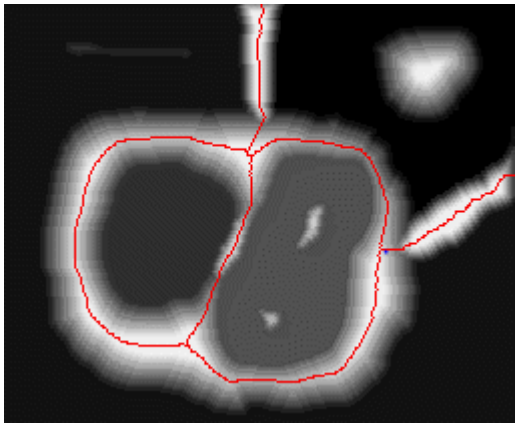

Figure 3-9 Watershed lines

However, this standard approach of watershed segmentation results in over segmentation of the image which may be due to the noise present in the image. The enhance for the standard approach is the marker-based segmentation approach [14]. In this approach, the segmentation is based on the markers which are specific locations in the image which are marked either by the user or by a user specified algorithm. These markers signify which points are to be considered as valleys and which points are not to be considered as valleys before the flooding starts. WE define these markers by defining the parts of the image which are surely a part of the background and parts of the image

which surely belong to the foreground. The main motive of this approach is to let the algorithm know that these points belong together during the process of flooding. So, the algorithm treats the marked areas together while flooding. These belong the primary entities of the valleys and they are merged into different connected segments resulting in successful partitioning. For instance, consider the below image. The marked portions represent the user defined markers.

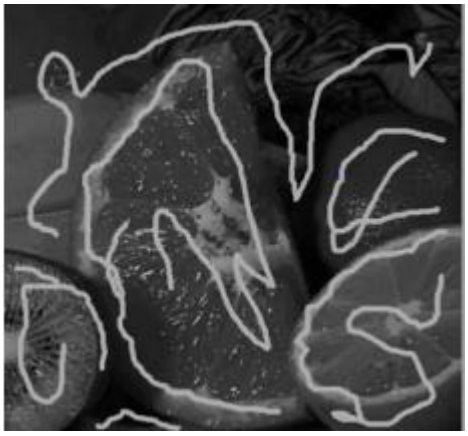

Figure 3-10 User specified markers [14]

By marking such portions, the user suggests the algorithm that these portions go together. So, we need to mark the lemon with a value 1, lime with 2 and orange with a value 3. During the segmentation process, the algorithm merges the marked portions into corresponding segments as follows:

Figure 3-11 Segmented image

The working of marker based watershed segmentation [4] can be visualized as follows

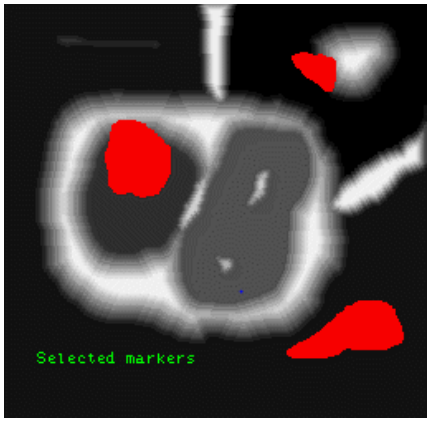

Figure 3-12 Custom markers

Then we flood from the markers as follows:

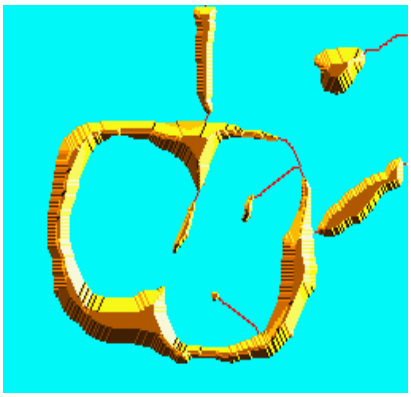

Figure 3-13 Flooding process [4]

Finally, our resulting watershed lines will be as follows:

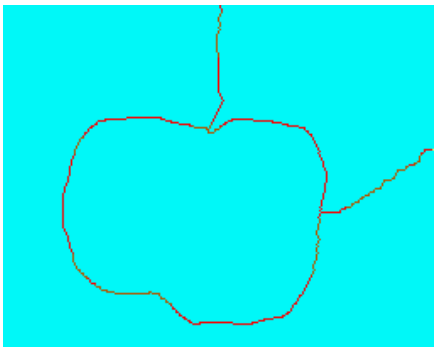

Figure 3-14 Watershed lines

Further, for separating the closely touching objects, the marker based watershed segmentation with Python works efficiently. The process of marking the foreground avoids over segmentation. So initially, we need to convert the original image into binary using Otsu's binarization. Then, we need to determine which part if sure foreground and which part is sure background so that we can define a marker for every object in the image. For instance, the original image and its corresponding binary image [15] are given below:

Figure 3-15 Input image with touching objects and its corresponding binary image

To determine the sure foreground of the objects, we need to calculate the distance transform. For every pixel, the distance transform gives the distance from nearest 0 pixel. The distance transform can be computed with the help of cv2.distanceTransoform() function [15]. The distance transform for the above binary image will be:

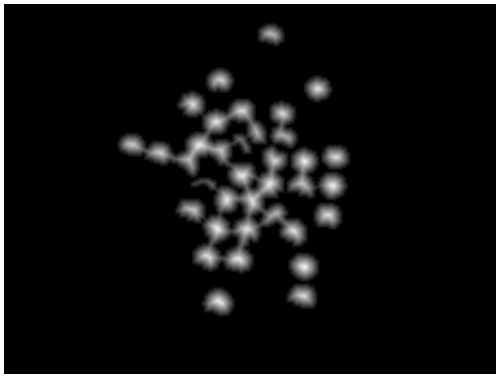

Figure 3-16 Distance transform

It is mandatory to have a marker for every object to flood accordingly and separate them.

We further need to determine sure background. To be on a safer side, we can consider all the pixels very far from our sure foreground can be considered as sure background. This can be done by dilation as it widens the boundaries of the objects. We can perform dilation by applying the function cv2.dilate(). Now, we need a marker. A marker is a 32-bit signed image same as the size of the original image but with just one channel. The corresponding sure foreground and background areas must be labelled inside this marker. We can create such marker by means of cv2.connectedComponents() method which assigns a positive value for all the sure foreground pixels. Then we can mark the sure background with any value (say 0) and also the region other than sure foreground and background with any other value.

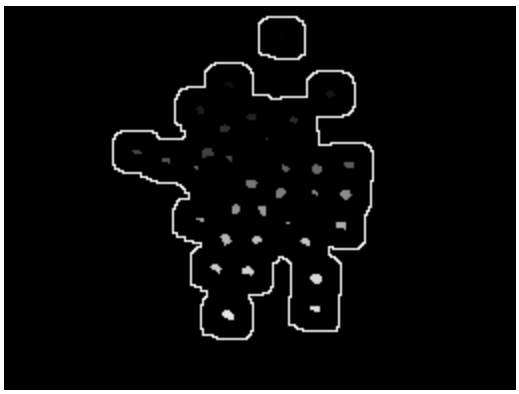

Figure 3-17 Sure background and sure foreground

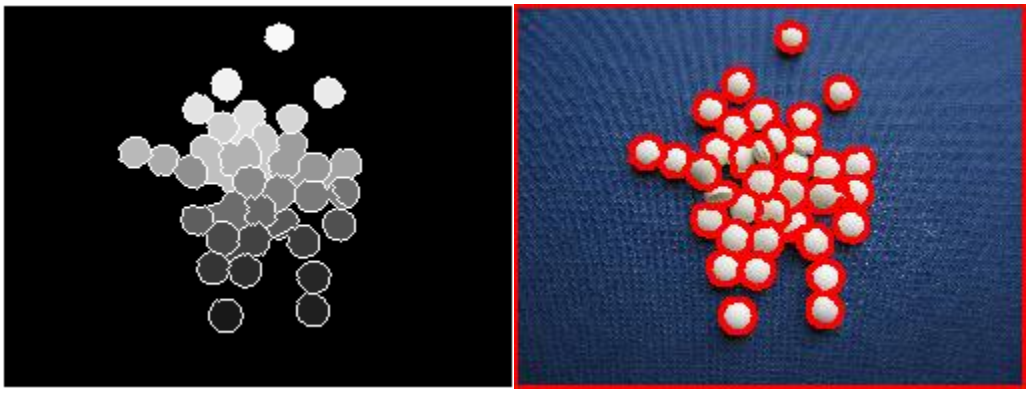

Figure 3-18 Final contours

Here, the part of the image outside the outlines region will be the sure background (value=0) and the markers are the white dots present inside the perimeter (positive values). The region between the sure background and the sure foreground is the unsure region which maps the boundaries of the touching objects. Now, we can start flooding by consdiering these markers and as stated above, the dams will be constructed separating the merging water from different markers. The watershed segmentation can be performed by means of the function cv2.watershed(image, markers). The resultant marker image is shown above. This segmentation process works efficiently for classifying images with touching objects. However, if the image has too many overlapping objects which are much smaller in size, the distance transform is not accurate enough to generate sufficient markers for each of the minute objects. This results in most of the smaller objects segmented as a single one. As a result of this, we need a further optimized implementation of watershed segmentation algorithm. Hence, as a part of this research we moved forward with the Watershed segmentation provided by ImageJ.

3.2 ImageJ

ImageJ is an open source application based on Java used for Image Processing. It provides several plugins to perform extensive analysis and processing. ImageJ provides a user-friendly graphical user interface with several functionalities which enable the users to efficiently view, edit, process, transform and save images of varying sizes. As ImageJ is completely written in Java, it is platform independent and can run on any operating system. The ImageJ software appears as follows:

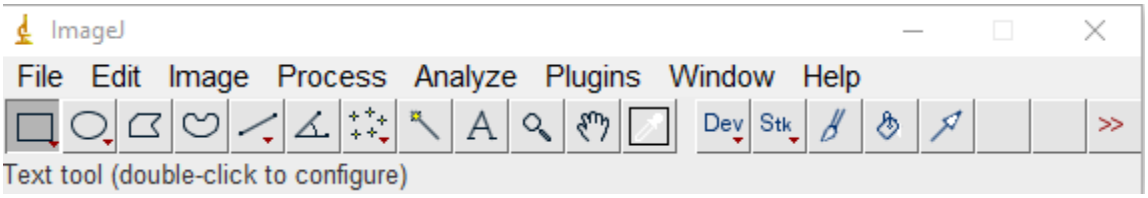

Figure 3-19 ImageJ layout

Some of the primary features provided by ImageJ are:

- Thresholding
- Color thresholding
- Smoothing and Sharpening
- Edge detection
- Maxima determination
- Histogram calculation
- Binarization
- Eroding
- Dilation
- Distance map
- Determination of Ultimate Eroded Points
- Watershed
- Voronoi and many more

While supporting varied file formats, ImageJ allows 8-bit grayscale, 16-bit unsigned integer, 32-bit floating point and RGB color data types. The primary reason for considering the watershed segmentation provided by ImageJ as it is the world's fastest image processing application written in pure Java. The rate at which ImageJ works is 40,000,000 pixels per second. At this rate, it can process an entire 2048*2048 image in $1/10^{th}$ of a second. Further, it is open source and we can use

its resources without the need for any license. The following chapters include the installation and setup of OpenCV in Java and Android. The further chapters include various steps involved in the process of Watershed segmentation, their corresponding conversion to Android using native Android APIs.

Chapter 4 - Project Setup

4.1 Setting up OpenCV and ImageJ for Java

4.1.1 ImageJ

ImageJ is an open source image processing web application purely written in Java. It is used for rapidly iterating and manipulating pixels of an image. The ImageJ API provides classes such as ImageProcessor, ImagePlus, ByteProcessor, ColorThresholder, FloatProcessor, Coordinates, Histogram, IJ and many more. All the classes provided by ImageJ API can be obtained from https://imagej.nih.gov/ij/developer/api/allclasses-noframe.html. In order to use any of these classes, we have to import the ImageJ Java Archive file into our Eclipse workspace.

Download

The latest version of ImageJ for respective platforms can be downloaded from https://imagej.nih.gov/ij/download.html.

Setting up ImageJ

After downloading and extracting the ImageJ latest version, we need to add the ImageJ Java Archive (ij.jar), to your current eclipse java project [16], open the project properties and click on the Java Build Path tab. Then click on the “Add External JARs” and select the ImageJ JAR file “ij.jar” located in the “ij150-win-jre6 > ImageJ > ij.jar”. Click on Apply and rebuild the project. Now, we have ImageJ setup in our Eclipse project and we can use all the classes provided by ImageJ.

4.1.2 OpenCV

OpenCV is an Image Processing computer vision library consisting of several programming functions to meet the requirements of real-time Computer Vision. In order to use OpenCV, we need to download and set it up in our project. There are several versions of OpenCV

with numerous enhancements. The latest version is OpenCV 3.2, released on 23rd December, 2016. However, the most prominent version is OpenCV 2.4.13 which is used predominantly.

Download

Any version of OpenCV can be downloaded from http://opencv.org/downloads.html. It is available for Windows, Linux/MAC, Android and iOS operating systems.

Setting up OpenCV

After downloading the executable of any version of OpenCV, we need to extract it and add the jar file into our Eclipse workspace. Similar to ij.jar, we need to add the opencv-<version_number>.jar present in "opencv > build > java > opencv-<version_number>.jar". Then, we need to set the Native Library Location. To set this, we need go to the project properties, and navigate to Java Build Path section. There we need to click on the corresponding OpenCV library listed in the Libraries section. Now, we need to set the Native library location of the opencv.jar to the path "opencv > Build > java > <x86 or x64>". Now, we have OpenCV setup in our workspace.

Further, along with the classes provided by ImageJ and OpenCV, we also use the services by Java Image class. Java Abstract Window Toolkit (AWT) is an interface to develop interactive graphical user interface based applications in Java. Further, all the image classes which represent graphical images are the sub classes of the Image class which is present in the java.awt.Image package.

4.2 Setting up OpenCV and ImageJ in Android

4.2.1 Android Studio

Android Studio is to Android; what Eclipse is to Java. It is the official IDE for Android development. Android Studio is freely available public usage. Prior to this, Android Development has been done in Eclipse using Eclipse Android Development Tools (ADT). We do not need an

Android device while using Android Studio as it provides an emulator which simulates a real time device of our choice with the help of the AVD manager and displays it on our personal computer. Further, the emulator makes use of the multi core processors available in your computer which makes it even more faster and efficient when compared to a connected hand held hardware Android device. All the applications we develop will be automatically installed on this emulator and will be displayed when we build and run the code. It also provides a Gradle-based support and one of its primary advantages include its user-friendly layout editor which enabled the feature of dragging and dropping the UI components. The latest version is the Android Studio 2.3. The benefits of Android Studio come from its inbuilt intelligent code editor with advanced code analysis and refactoring. Compared to the previous versions, the emulator is must faster and efficient.

System requirements for Android Studio version 2.x

Operating System: Windows 7/8/10 (32-bit or 64-bit), Mac OS X 10.8.5 or higher up to 10.10.5, Linux Fedora or KDE or GNOME or GNU/Linux Debian or Unity desktop on Ubuntu.

RAM: 3GB or higher

Java version: JDK 8

Disk space for Android Studio: 500 MB

Space for Android SDK: 1.5 GB or higher

All the build files related to the project will be available under the Gradle Scripts section. Each android project contains multiple modules such as app modules, library modules et cetera., which are ultimately integrated. Each of the application modules includes three main sub folders:

- manifests: includes the AndroidManifest.xml file
- java: It includes the source code files related to the project.

- res: It includes the user interface related resources files such as XML files and bitmap drawables.

Further, Android Studio also enables with a feature of Version Control System through which we can push our code onto our GitHub repository on the go by just giving the remote name of our repository. We can review, commit and push whenever changes are made.

Download Android Studio

The latest version of Android Studio can be downloaded from the following link: https://developer.android.com/studio/index.html.

Setting up Android Studio

Before moving on to the setup, we need to set an environment variable pointing to the location of the JDK. We need to do this in order to let the launcher know the exact location of the JDK and to make the JDK compatible with Android Studio. Now, once the download is completed and the JDK is setup, we need to launch the executable file of the Android Studio bundle and follow the setup wizard. Once we are done with the setup, we need to launch Android Studio and we need to configure it. So, we need to go to the Standalone SDK Manager in the System Settings and we need to download additional packages suggested by the SDK manager. After installing the additional packages, we are ready with the setup of Android Studio to start a new project. Now, we need to setup OpenCV library in our project.

4.2.2 OpenCV for Android

Download OpenCV

OpenCV is also available for Android. Corresponding version of OpenCV library can be downloaded from http://opencv.org/downloads.html. The latest version is the "opencv-3.2.0-android".

Setting up OpenCV

After downloading, we need to extract the OpenCV Android SDK. Now, to add this library to our current project in Android Studio, click on File > New > Import Module… > Redirect the source directory location to OpenCV-android-sdk > sdk > java. Now, we need to configure the application's build.gradle file. We need to change the target SDK version and minimum required SDK version so as to support OpenCV. To do this, we need to incorporate the following changes:

- compileSdkVersion: 24
- buildToolsVersion: 24.0.0
- minSdkVersion: 21
- targetSdkVersion: 24

Now, we need to include the dependencies for the corresponding OpenCV version. To do this, go to Module Settings and navigate to app modules and click on the 'Dependencies' tab. Click on the '+' sign and add a Module dependency and add the 'openCVLibrary' module to our app modules. This states that our application module depends upon the OpenCV module. After adding the dependencies, we need to synchronize the gradle to accommodate the recently added changes. Now, we need to add the Java Native Interface (JNI) libraries. These are the shared library files built from the native C/C++ sources. Our native code depends on these library files during compilation. To do this, in our project we need to add a new JNI Folder and modify the location as "src/main/jniLibs/". Now, we need to go to our downloaded OpenCV android SDK folder and navigate to sdk > native > libs and copy all the folders which include arm64-v8a, armeabi, armeabi-v7a, mips, mips64, x86, x86_64 and paste them in the recently created jniLibs folder. By this way we have all the native libraries setup in our project. Now, we are done with all the steps and OpenCV is successfully setup in our project. Further, when we try to build and run any sample

OpenCV test code and the compiler gives a “compileDebugNdk” error, it means that the integration of the Native Development Kit is deprecated in the current using plugin. To continue using the current Native Development Kit, we need to set “android.useDeprecatedNdk” to “true” in our project’s gradle properties and then synchronize the gradle. Now, we are all set and OpenCV is successfully installed in our project in the current workspace.

4.3 ImageJ support on Android

The ImageJ library mostly depends on the Java Abstract Window Toolkit package especially on the java.awt.Image package. Most of the classes provided by ImageJ implement the classes provided by java AWT. Android has very minimal support for AWT as well as Swing [17]. Android has its own graphics packages. Out of all the packages in AWT, Android only supports the fonts package. The reason for this is AWT provides high end graphics which requires a suitable hardware. Further, Android defines it’s user interface components in XML resource files which are internally linked and loaded into the class files. Since Android has limitations regarding varied hardware, screen resolutions, screen sizes and unfortunately, java AWT is not written to handle all these limitations as these are very uncommon for Java SE. Due to all these reasons, Android provides very minimal support, rather no support for java.awt packages and libraries relying on these. However, in order to use AWT in Android, we have to port all the native code into Android before using them. Further, if we try to add the java archive file consisting of all the java.awt.* and javax.* packages (java-rt-jar-stubs-1.5.0.jar), these packages overlap with the existing core classes under java and javax packages in the internal Android Java Archive files (android.jar and rt.jar) which are used to build the project [19]. The reason for this conflict is that the added java-rt.jar has the same classes as those in the Android SDK. We cannot force the gradle to compile by

removing the android's rt.jar in order to remove the conflict as it consists of native code and the build process will crash. Also, another reason for the conflict is that, the rt.jar is a runtime library developed for the desktop JRE. To serve the purpose of Android, efforts have been made worldwide to port the native code of AWT to Android.

4.3.1 Porting AWT to Android

Lately, many efforts have been employed to port the AWT to Android platform. One such project is the 'awt-android-compat' project [19] initiated by Google. The project's main goal is to make AWT rendering compatible with the Android API. Unfortunately, after porting some of the 'java.awt.geom', the project has been suspended due to several issues. The primary reason is that the AWT event hierarchy is designed for a windowed system making it difficult to port it and make it compatible with the small screen system. This is the reason why we cannot port AWT packages into Android. Further, as my personal approach, I started decompiling the java-rt-jar-stubs-1.5.0.jar and obtained the source code for the awt packages. Then, I tried adding specific java awt classes manually. But, these classes are internally dependent upon classes such as Rectangle, Plane from the java.awt.geom.* package and when I tried to enter these classes manually, they have overridden the core java and javax native classes in Android. As stated earlier, many classes of ImageJ are internally dependent on the java.awt.Image and java.awt.geom.*. For the same reason, we cannot use ImageJ in Android as it primarily relies on AWT packages.

4.3.2 Porting ImageJ to Android

Currently, a new API called ImageJ2 [20] has been under development whose primary goal is to serve the exact functionalities as provided by ImageJ 1.x but without any dependency on the java AWT. It is being developed by a team led by Curtis Rueden. However, the structures provided by ImageJ2 might not serve the same purpose as that of ImageJ. Also, ImageJ2 cannot implement

the services provided by the ‘imglib2-ij’ component of ImageJ 1.x [21]. This is the reason why no one have tried to implement ImageJ2 on the Android platform. That being said, it is still unclear whether we can compile the base components of ImageJ2 [21]. The current developer of ImageJ2 himself has warned that many issues would rise as a result of this integration [21] as they have not developed this by keeping Android’s support towards the Java standard library in mind.

This leaves us to a conclusion that, before migrating the Watershed segmentation algorithm to Android, we have to rewrite the parts of the algorithm which rely on the java AWT library by using structures supported by Android. The goal of the following chapters is to explain the algorithm in both Java and Android versions simultaneously for a better understanding.

Chapter 5 - Watershed Segmentation

In this BreadPheno project, we are analyzing and improvising novel image segmentation algorithms to determine plant phenotypes. The primary goal of this project is to characterize and count the number of seeds present in an image. As a part of this, we majorly analyze images contains seeds of various plant types. Our study of phenotypes includes Wheat seeds, Cassava roots, Canola seeds, Bean seeds, Potatoes, Silphium seeds and many more. In the case of processing the images containing Wheat seeds or Canola seeds, the segmentation process is extremely complex as the seeds are relative very small when compared to other seeds and there are hundreds of them touching and overlapping randomly in the images. As stated in earlier sections, for such cases, the Watershed Segmentation algorithm works efficiently to segment and separate these seeds. There are multiple implementations of Watershed segmentation. We are implementing the Watershed algorithm proposed by Soille and Vincent. We follow the classic flooding approach here. Further, this project seeks to extend the watershed algorithm in order to improve the process of segmentation in a much more efficient manner. To begin with, the process of Watershed segmentation starts with the Euclidean Distance Map for each of the pixels followed by the determination of Maxima points which are also called as Ultimate Eroded Points (UEPs). Then we start flooding from each of these Maxima points, to segment and separate the seeds from the background and from each other. Finally, we count the number of seeds. The image we process can consist of any number of seeds. The complexity of segmentation depends on the characterizes of the phenotype we are analyzing. Cassava roots and potatoes are relatively large and can easily be segmented. Wheat seeds or Canola seeds are remarkably small. Having them in a greater number may result in overlapping which will affect the characterization. There are several parameters to consider depending on the type, size, color of the seed we are dealing with and the

lighting conditions. The images we process typically look like the figure below. All the seeds are spread out on a green background for us to easily determine the foreground and background objects.

Figure 5-1 Wheat seeds

Figure 5-2 Black Beans

Figure 5-3 Potatoes

Figure 5-4 Soybeans

Figure 5-5 Silphium

Figure 5-6 Cassava

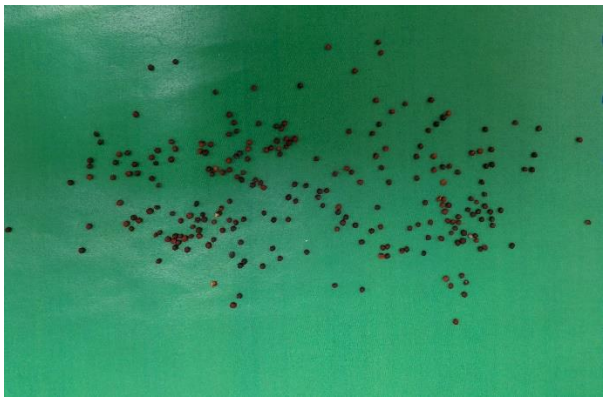

Figure 5-7 Canola seeds

The Blue circles are included as a reference for sizing the seeds. By calculating the size of each of the circles both in pixels and square millimeters, we can determine the number of pixels occupied by each of them which helps in determination of individual seed size. Before we move forward to the implementation, let us see how we access the images in Java and Android in the next section.

Structures used

In java, we use the latest Java Image I/O API which enables simple encoding and decoding of the input images by providing convenience methods for locating ImageReaders and ImageWriters. It is much faster, flexible, efficient and powerful than the previously existing APIs for loading and saving images. The Java Image I/O API also provides several additional plug ins to support varied formats and it consists of packages which include javax.imageio. Also, we use ImagePlus object provided by ImageJ's ij.imageplus package which implements java.awt.Image.ImageObserver. This ImagePlus object contains an ImageProcessor which is used to store and manipulate the pixel data of the image. In addition, we used BufferedImage, a subclass of Image to access the buffer of image data efficiently. And to store Image data, we use DataBufferByte a subclass of DataBuffer which stores the image data in the form of Bytes. As Android doesn't support Java AWT, we cannot use all these structures. However, in Android, we have Bitmap [22] to represent an image.

A Bitmap can be represented by a Drawable say BitmapDrawable [23]. Any image data of bytes can only be stored in a Bitmap and displayed by means of an ImageView. However, Bitmap only supports three file formats for images which are png, jpg and gif (not recommended). But, to support the hardware limitations, Bitmaps come with a feature of compressing the image without any loss of pixels and quality. By doing this, the memory requirement reduces and the build process will be faster than usual. But, java AWT is way more advanced than Bitmaps both in terms of accessing and storing. The Bitmap class is provided by the graphics package. Further sections explain how we perform the segmentation process with the help of all these objects.

Representation

Before we begin, we need to know the representation of White and Black pixels. The Black pixels are considered as background and the White pixels are considered as foreground. In our case, in Java, 0 represents a Black pixel and 255 represents a White pixel. This representation may vary. In Android, the Color class is responsible for representation of pixel intensity values. The pixel intensity is represented using Color int is a 32-bit integer which stores four components which are Alpha, Red, Green and Blue components. It also represents the sign. These components are encoded in the following format:

(A & oxff) << 24 | (R & oxff) << 16 | (G & oxff) << 16 | (B & 0xff)

Thus, Black color is stored as -16777216 which is 0xff000000 in Hex representation [28]. As the range is from 0 to 255. We get 0 if we compute the modulo of -16777216 with 256 which gives us 0. Similarly, White color is represented by -1 (0xffffffff). We can also see that -1%256 gives us 255. These are the color representation we follow. In addition, the Color class also provides individual methods to extract each of the components respectively.

5.1 Pre-processing

As explained in earlier sections, before the process of segmentation, we need to convert the image to a Binary image with just Black and White pixels. OpenCV plays an important role here as it provides several functions to perform thresholding. Thus, we move forward by applying a series of functions provided by OpenCV. Initially, we need to create a Mat object which is used to hold the input Image data. Further, we need to change the image's color space from RGB color space to HSV color space. As explained earlier, Hue Saturation Value (HSV) color space separates the image intensity values from the color components. The HSV color space gives the detail information related to the image whereas the RGB color space data is much noisy. Thus, it gives us more control over the image with this HSV representation and it will be useful for our segmentation. OpenCV provides a function called cvtColor which converts an image from one color space to a different one. We can use Imgproc.COLOR_BGR2HSV as the conversion type. The HSV representation of our input image will be as shown in the figure. As we can see here, all the blue circles have fallen into one color range, brown colored seeds into one range and the green background into another range. This helps us to easily distinguish between each other.

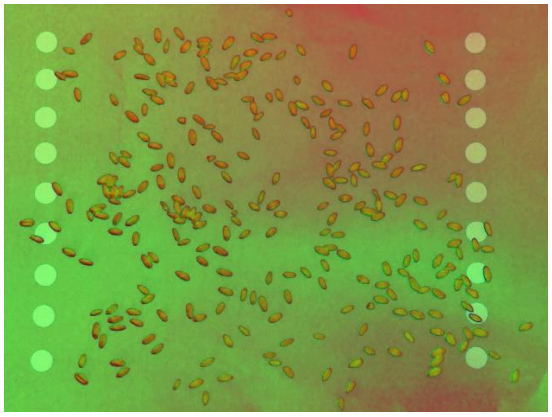

Figure 5-8 HSV representation of an image with Wheat seeds

Further, we need to separate foreground and background. In our case, all the seeds are the foreground and rest of the image is the background. In our input image, all the seeds are in brown color or in shades of brown. Fortunately, we can detect an object based on the pixel intensity values it consists of. As we already know, Hue represents the dominant color. So, all we need is to extract the Seeds from the image based on their Hue value. To do this, we need to perform thresholding on the HSV image for the range of brown color. In HSV, the range for Hue is [0,179], for Saturation and Value its [0,255]. Further, the hue range for our seeds is between [0,50]. As we already know that Scalar is used to pass pixel ranges, we will pass the required corresponding hue range to the function inRange(src, dest, const Scalar(a, ,b, c)). By doing this, we are allowing only those pixels whose hue is in the range of 0 to 50 that is, we are allowing only those pixels which are dominated by brown color (seeds). Thus, all the pixels which fall in the corresponding range will be assigned with a value of White color (255) and all others will be assigned the value of Black color (0). Our corresponding output image will be:

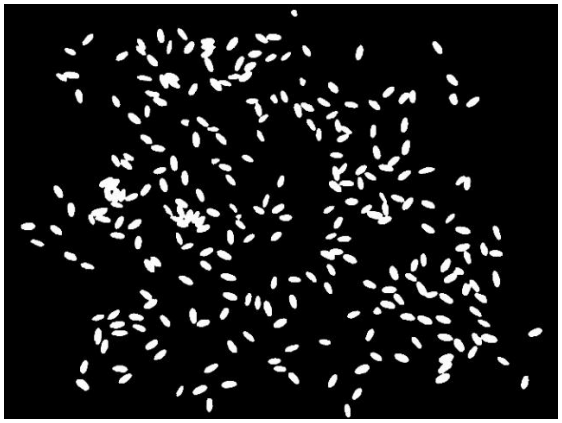

Figure 5-9 Output image after passing the hue range for the seeds

However, if we do not know the hue range for the color we need to extract from the image, we can just pass the B, G, R values of the pixel to the cvtColor() function and giving COLOR_BGR2HSV as the type [27]. Then the exact hue for the corresponding color pixel will be obtained. In order to

extract all the pixels of that color, we just need the pass the [hue-10, hue+10] as a range to the infringe function. Further, we make the binary representation of this image by performing thresholding operation. All the pixels above the threshold will be assigned 255 and all the pixels below the threshold will be assigned 0. Now that we have the binary representation of the input image, we can move towards the next step in the segmentation process that is, determination of the Euclidean Distance Map.

5.2 Euclidean Distance Map

Euclidean Distance is the distance of each pixel from the nearest background pixel in the neighborhood. In our binary image, Black pixels are considered as background and White pixels are considered as foreground. For a black pixel (0), the corresponding Euclidean distance will be 0 and for a white pixel (255), the Euclidean distance is the distance to the nearest black pixel. So, the Euclidean Distance Map (EDM) gives us the floating point distance for each pixel from the nearest Black pixel. We need to consider all the eight directions for a pixel while calculating its EDM. The eight directions are as follows:

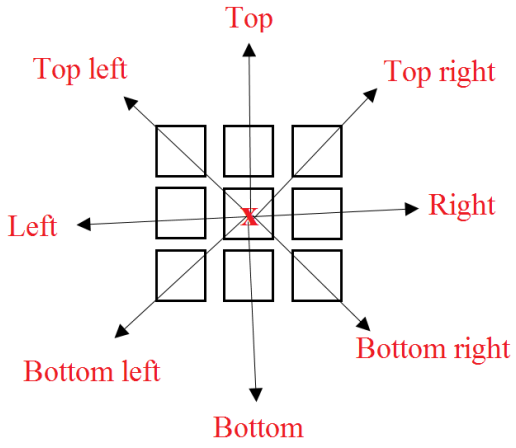

Figure 5-10 Eight directions of a pixel

For each of these directions, we need to compute the distance from the nearest black pixel and then we need to find the minimum of distances from all the directions which gives us the EDM of a pixel. This process will be explained in detail further. As we are traversing all the directions for each and every pixel, the processing time will be relatively high for very large pixels. We need efficient structures for storing, accessing and manipulating purposes. As discussed earlier, we use an ImagePlus object to store the pixel data. Further, we create an ImageProcessor object and using the getProcessor() method, we set this as a reference to the ImagePlus object. Any manipulation performed on the ImageProcessor object will be reflected in the ImagePlus object as well. This rapidly speeds up the process of image access and processing. Unfortunately, as Android doesn't support AWT, we cannot utilize ImagePlus here. The only way to store and access an image is via Bitmap [22] in Android. This is a huge barrier between the Java and Android versions. Though they work similarly, the speed of processing using an ImageProcessor reference for an ImagePlus object is exponentially high when compared to processing through a Bitmap. In addition, all the OpenCV conversions are done on Mat objects and accessing is done on Bitmaps. So, frequent conversion of Mat to Bitmap and Bitmap to Mat also consumes time which is a drawback of the Android platform. However, despite of Android's screen size, storage and processor limitations, Bitmaps work relatively fast. Further, to determine the Euclidean Distance Map of the input image, the makeFloatEdm() function takes the binary byte image data and returns the EDM of all the pixels.

5.2.1 Determination of Euclidean distances

For the makeFloatEdm() function in Java, we send the ImageProcessor object (ip) and a corresponding FloatProcessor (fp), a 32-bit floating point array representing the floating point distances will be returned. For efficiency, we consider a byte array which has the all the pixel data

taken from the ImageProcessor (ip). This is done by just calling the function ip.getPixels() as follows

```
byte[] bPixels = (byte[]) ip.getPixels()
```

This method traverses through the image from left to right., top to bottom and copies the pixel values into a single dimensional byte array. This byte array is called bPixels (say). Further, the getPixels() method returns a reference to the image's pixel array. So, the image data present in the ImageProcessor (ip) will be automatically updated whenever we update its reference bPixels. This provides efficiency and speed. Similarly, we have a float array as a reference to the FloatProcessor (fp) object as well. This is named as fPixels. Thus, All the EDM distances which are gradually updated in fPixels will directly be updated in the fp. However, in Android, we don't have any means to update a double dimensional array automatically by keeping a single dimensional array as its reference. So, to replicate the FloatProcessor, we take a two-dimensional floating point array with rows and cols as image height and width. Initially, we need to set all the EDM distances for White pixel indexes with the maximum possible floating point value (3.4028235E38) in the fPixels array. Then we update these values with the nearest distance from the background for each of them.

Replicating access in Android

We need to replicate the working of a single dimensional array (fPixels in java) with a double dimensional array (fPixels in Android). For instance, while instantiating all white pixel indexes with Float.MAX_VALUE, in java we just need to instantiate the fPixels in the corresponding index we find a pixel with an intensity value 255 in the bPixels array. Automatically, the two dimensional array fp will be updated based on the single dimensional array fPixels. But in Android, we have to traverse through the single dimensional array bPixels, and when we encounter a white pixel, we

need to update the index in the two-dimensional array fPixels. So, we need to convert the single dimensional index into corresponding two dimensional one. We can make use of the image width to know the current row and column we are in. For instance, our image width is 6 pixels and height is 4 rows. The binary representation of our image in a two-dimensional matrix would be:

$$\begin{bmatrix} 0 & 0 & 255 & 255 & 0 & 0 \\ 0 & 255 & 255 & 255 & 255 & 0 \\ 0 & 255 & 255 & 255 & 255 & 0 \\ 0 & 0 & 255 & 255 & 0 & 0 \end{bmatrix}$$

Table 5-1 Representation of the binary image in a two-dimensional matrix

The equivalent single dimensional representation of our image would be:

[0 0 255 255 0 0 0 255 255 255 255 0 0 255 255 255 255 0 0 0 255 255 0 0]

Table 5-2 Representation of binary image in a single dimensional array

So, if we consider the pixel at index 8 in our single dimensional array, it is correspondent to the pixel at 1st row, 3rd column (1,2) in the two-dimensional array. Mathematically, this can be computed by just making use of image width to locate the two-dimensional index of the pixel. So, diving the index by width (index/width) gives us the current row and performing modulo by width (index % width) gives the column. So, for index 8 the corresponding coordinate in the two-dimensional coordinate is (8/6, 8%6) i.e., (1,2). In this manner, we can change a single dimensional representation of an index into two-dimensional index representation.

After setting all the corresponding white pixel indexes with maximum value in the fPixels array, we move forward into the next step. We move from top to bottom initially and then come from bottom to top. Thus, while calculating the EDM, we initially cover left, right, top, top left and top right directions. While moving from bottom to top, we cover the directions left, right, bottom left, bottom and bottom right. Left and right directions will be checked in both the passes. We will

update the distances in the fPixels array with the minimum of all the 8 directions. While moving from top to bottom, we maintain two buffers each of the size of width of the image. These buffers are used to store the coordinated of the nearest black pixel for each pixel in a row. Each of these buffers are updated from time to time for every row. These buffers are instantiated with -1 initially and are updated gradually. These two buffers are declared as a two-dimensional array pointBufs, with 2 rows and width number of columns (pointsBuf [2] [width]). Its significance will be explained in detail in the next step. Now, we start moving from top to bottom, each row at a time and then traverse each row, column wise calculating the distance for each of the pixels. For each row, we call the function edmLine() which calculates the nearest distance for each pixel in that row. It also stores the coordinate of the nearest black pixel in the buffer at the corresponding index for future reference. While traversing from top to bottom, for each row we make a call to the function edmLine(). The primary parameters to the function include bPixels (contains the pixel data of our image), fPixels (stores the EDM values for the pixels), pointBufs (the buffer we use to store the coordinates of the nearest black pixels for each of the pixels), width (the width of the input image), row_number*width (to maintain an offset for two dimensional access), row_number (iteration number) and backgroundValue (0 in our case).

5.2.2 Calculation of EDM for each row

In the function edmLine(), initially we will cover left, right, top, top left and top right directions as we are moving from top to bottom. For a row, we initially move from left to write by calculating the distance for all the pixels from the left, top and top left directions. Then we come from right to left to calculate the distances from right, top and top right directions. Finally, we update the distances for each of the pixels with the minimum distance of the left pass and right pass operations.

5.2.2.1 Top to Bottom

Initially, we move from top to bottom by traversing each row. In each iteration we have a buffer as stated. The reason for maintaining a buffer (pointsBuf) is that, while we are moving from left to right, for every pixel, we calculate the distance from the black pixel and store the distance in the fPixels array. Also, we store the coordinate of that black pixel in the buffer at the related index. We then send this buffer as a parameter for the next iteration (for next row). We can then make use of this buffer for every pixel in the next row to keep track of the coordinate of Black pixel which is nearest to the corresponding top pixel. By making use of this buffer we can avoid traversing all the way to top and top left directions to find a black pixel as it already has the coordinates of the black pixel corresponding to these three directions. This is a dynamic programming approach which will be explained further. Now, for each pixel in the current row, we calculate the minimum distance by comparing black pixel distance from left, top and top left (using buffer) directions and we update the coordinate in that particular index so that it will be used in the next iteration for the next row (for the pixel below it). So, we keep on updating this buffer so as to send it to the next iteration. When we come from right to left pass of the current row, we need the buffer to keep track of the coordinates from the upper two directions (top and top right). But, this buffer doesn't have the coordinates of previous iterations. Instead, it has the coordinates of Black pixels corresponding to the current iteration which is not useful for now. Hence, in order to avoid this, we maintain two buffers one for the left to right pass and one for the right to left pass.

Left to right pass

From the two dimensional array pointsBuf, We consider pointsBuf[0] and store in a single dimensional array called points as the buffer for the left to right pass for tracking top and top left directions. As explained earlier, we make of dynamic programming yet again. Moving left to right, we also keep track of the coordinates of the Black pixel of the current iteration. So, when we are in the next iteration (the pixel to the right of previous pixel), we can use that pixel's coordinate to know the nearest black pixel in the left direction. So, we use pPrev to keep of the coordinates of nearest black pixel to the current pixel (current iteration). As we already have the buffer which has all black pixels coordinates of previous row (top row), we use pDiag to store the coordinates of nearest black pixel for the left top pixel. The left top pixel is the left pixel to the top pixel. We have pNextDiag as a temp variable to update pDiag. These are represented in the right to left pass as well. This can be visualized as shown in the below figure:

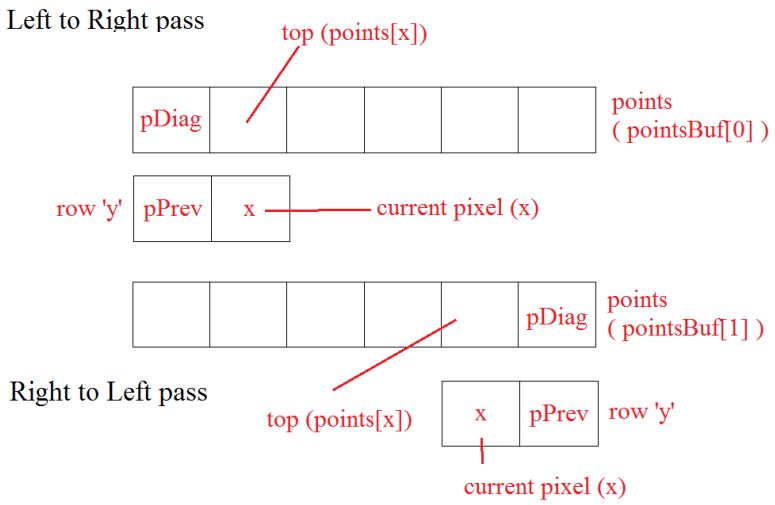

Figure 5-11 Buffers to store coordinates

Further, at the start of each iteration (each row) from left to right (y) and for each pixel x, before we calculate the minimum distance from a black pixel, we need to store the top pixel's value

(points[x]) into pNextDiag. We will see the reason for this in a while. Now, if the current pixel (x) in the current row (y) is a black pixel, then we update the buffer points[i] with the coordinates (x, y) of the current pixel. The column is nothing but the value of x and the row is nothing but the iteration number (y). This is encoded into the buffer at corresponding column location, x (points[x]). So, we just use bitwise operations to store both the x and y coordinates into a single location as follows:

$$points[x] = x \mid y << 16$$

We know that an integer consists of 32 bits. We store the value of y into the most significant 16 bits and the value of x into the least significant 16 bits. Now, if the current pixel is not a black pixel, we make a call to a function named minDist2() which gives the minimum distance from the left, top and top left directions. The primary parameters to this function include points (buffer having the coordinates of nearest black pixels for top row, y-1^{th} row), pPrev (coordinates of nearest black pixel to the left pixel, the x-1^{th} pixel), pDiag (coordinates of nearest black pixel to the top left pixel, the x-1^{th} pixel in the y-1^{th} row. i.e, points[x-1]), current column (current iteration from left to right, x), current row (the current iteration from top to bottom, y). The minimum distance returned by this function is stored in an integer distSqr which is used to update the fPixels later.

Finding the minimum distance

In the function minDist2, for the current pixel x, we store the value the coordinates of nearest black pixel from top direction (points[x]) into variable p0. To keep track of the coordinates nearest black pixel to the current pixel, we use variable named nearestPoint. We initialize it with p0 and move further. Now, we need to calculate the distance from the current pixel to the nearest black pixel from left, top and left top directions. Then we need to return the minimum of these three. To calculate distance from top, p0 has the coordinates of nearest black pixel of previous row

(points[x]). These x and y coordinates are encoded into a single location using bitwise operations. In order to calculate the distance, we need the x and y coordinates individually. We use bitwise operations again to extract them. As x coordinate is stored in the least significant 16 bits, we can get them by performing a bitwise and operation with 0xfff. As the y coordinate is stored in the most significant 16 bits, we can get them by shifting p0 16 bits towards right using bitwise shift and then performing bitwise and with 0xffff.

$$x0 = p0 \ \& \ 0xffff \text{ and } y0 = (p0 >> 16) \ \& \ 0xffff$$

As we know our current pixel's coordinates (row is y and column is x), we can simply use the Euclidean distance formula to calculate the distance as follows:

$$distSqr = (x-x0)*(x-x0) + (y-y0)*(y-y0)$$

The reason for not calculating the square root will be explained after this section. Similarly, we can calculate the distance from coordinates of nearest black pixel from left direction (pPrev) and from top left direction (pDiag) and update distSqr with minimum of these respectively. We also update the nearestPoint with the coordinate from which we acquired the minimum distance. Finally, we update the current location in the buffer (points[x]) with the nearest black pixel coordinates from the current pixel (nearestPoint). As we already know, this is used as pPrev for the next iteration (pixel next to current pixel). Finally, the EDM distance for the current pixel (distSqr) is returned back to the edmLine function.

As we are back in our edmLine function, we now got the EDM distance for current pixel. We check this value with the value in the corresponding index of the fPixels array and we update it if this distance is less than the existing distance value. As fPixels is a reference to the FloatProcessor object fp, fp will be automatically updated. Further, we update the pPrev with points[x] as stated earlier. We already know that the initial value of points[x] is stored into pNextDiag, we now update

the value of pDiag with pNextDiag. This pDiag serves as left top for the next iteration (next pixel). We repeat this process for each pixel while move from left to right until we reach the last pixel (image width). However in Android, as we have the fp as a two dimensional array, we need to convert the single dimensional array into corresponding two dimensional index. For this purpose, we had the offset (row_number*width). We can now set the EDM distance for the current pixel in its index in fp as:

fp[offset%width][offset/width] = dist;

Everything else remains unchanged.

Right to left pass

As we complete iterating one row that is, the left to right pass of a row, we need to comeback from right to left to calculate the EDM from right, top and top right directions. But, the buffer is updated during the left to right pass. But, we already have a backup buffer, pointsBuf[1]. Now, we rewrite our temporary buffer, points with this pointsBuf[1]. Further, we repeat the same process while moving from right to left until we reach the beginning of the row (index 0).

5.2.2.2 Bottom to Top

Now that we have calculated the EDM by checking the directions left, top left, top, top right and right directions, we need to cover the bottom, bottom left and bottom right directions. We do this by iterating from bottom to top. Before doing this, we rewrite the data in the pointsBuf[1][width] with -1s. Then we call edmLine for each row while iterating from the image's last row to the first.

Finally, we have calculated the EDM from all the 8 directions and updated the fp with minimum of all these directions for each of the pixels. As we can see earlier, we did not perform

the square root while calculating the Euclidean distance in have squares of the distances. This is because, the FloatProcessor provides a method called sqrt() which updates the values in the fp object with their accurate square roots. So, finally we can just call fp.sqrt() to get the accurate distances from nearest black pixel for each of the image's pixels. Unfortunately, in Android, as are using a double dimensional array named 'fp' instead of FloatProcessor, we need to traverse through the array fp and calculate the square root for each of the values individually using the Math.sqrt() function. This is one more reason where the algorithm in Java is much faster than that of in Android. To visualize the Euclidean Distance Map of the input image, we can The Some of the EDM values for white pixels are presented here by considering a small part of the input image as shown below

Figure 5-12 Input image with less number of seeds

Its corresponding binary representation is

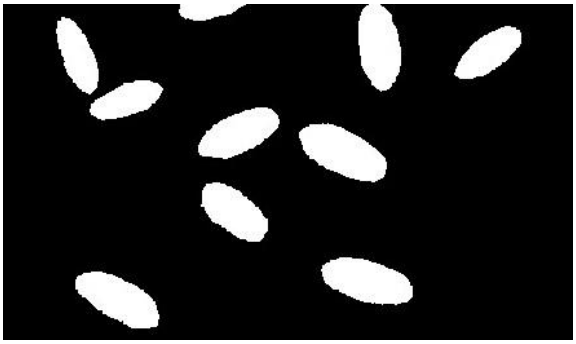

Figure 5-13 Corresponding binary image

This is sent as input to the makeFloatEdm(). It returns a FloatProcessor object fp with size of the image. All the EDM distances of the pixels are stored in their corresponding indexes. The EDM of above binary image will be as follows:

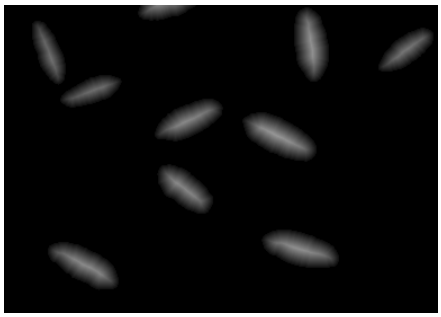

Figure 5-14 EDM of the binary image

Few of the EDM distances for a single seed are listed below

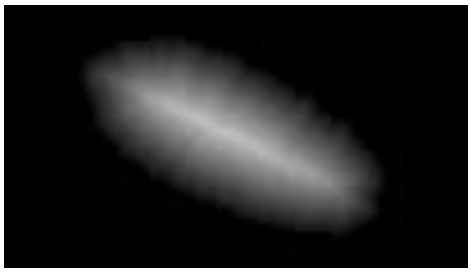

Figure 5-15 EDM of a single seed

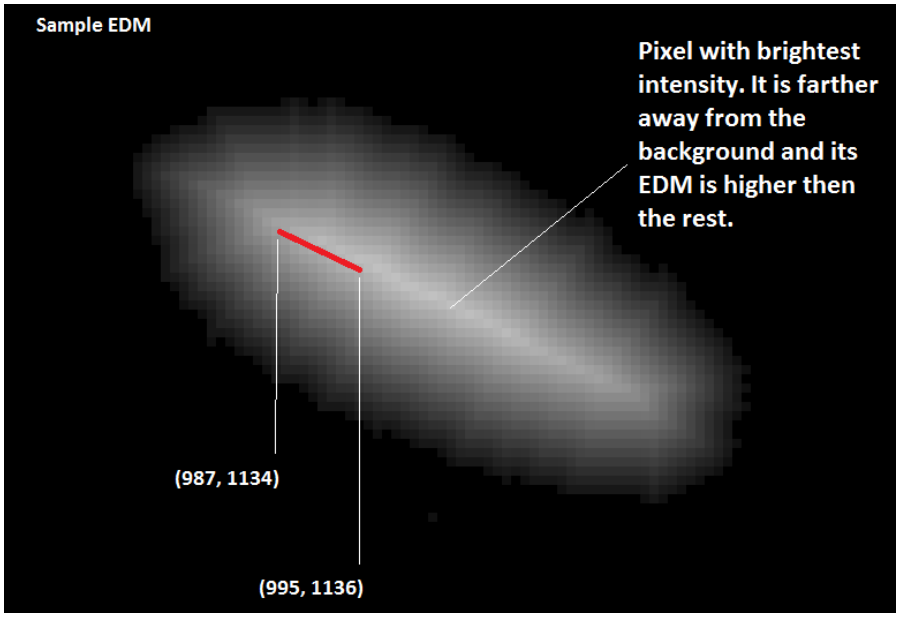

Figure 5-16 EDM values of the highest points

EDM value at pixel (987, 1134) is: 15.033297, EDM value at pixel (988, 1135) is: 15.6205, EDM value at pixel (989, 1134) is: 15.033297, EDM value at pixel (989, 1135) is: 16.03122, EDM value at pixel (989, 1136) is: 15.264338, EDM value at pixel (990, 1135) is: 15.652476, EDM value at pixel (990, 1136) is: 15.811388, EDM value at pixel (991, 1135) is: 15.231546, EDM value at pixel (991, 1136) is: 16.155495, EDM value at pixel (991, 1137) is: 15.6205, EDM value at pixel (992, 1136) is: 15.811388, EDM value at pixel (992, 1137) is: 16.155495, EDM value at pixel (992, 1138) is: 15.231546, EDM value at pixel (993, 1136) is: 15.524175, EDM value at pixel (993, 1137) is: 16.492422, EDM value at pixel (993, 1138) is: 15.652476, EDM value at pixel (994, 1136) is: 15.297058, EDM value at pixel (994, 1137) is: 16.27882, EDM value at pixel (994, 1138) is: 16.124516, EDM value at pixel (994, 1139) is: 15.264338, EDM value at pixel (995, 1136) is: 15.132746.

As seen above, the EDM values will be starting initially from 1 for the pixels which are immediate neighbors for the black pixels and the distance gradually increases as the pixel is located far away from the background. In our case, for the wheat seeds, the maximum EDM value is found to be 24.41. Also, there will be white noise present in the image. So, if a pixel has the EDM value greater than 0 but lesser than 1, it is considered as a noise. We need to replace the EDM values of these pixels with 0. Now, we move to the next step in the segmentation process where me find the local maxima in the image from which we need to start the flooding of water for segmentation. Thus, the function MaximumFinder takes all the EDM distances and determines the maxima points among them.

5.3 Maximum Finder

After returning from the function call of makeFloatEdm, we have a FloatProcessor object floatEdm in java which has the size of the image and has all the EDM values in their corresponding indexes. In Android, we have these values in a corresponding two dimensional array named floatEdm. We further remove the noise in the obtained data. The next is to determine the local minima and maxima so as to determine the markers (local maxima or UEPs) for the segmentation process. If we visualize the seed as a topographic surface, a local maxima is the highest point in the seed and a local minima is the lowest point. So a pixel which is far from background and has maximum EDM distance than the rest of the pixels will be the local maxima. Local minima and local maxima can be visualized as shown in the below figure.

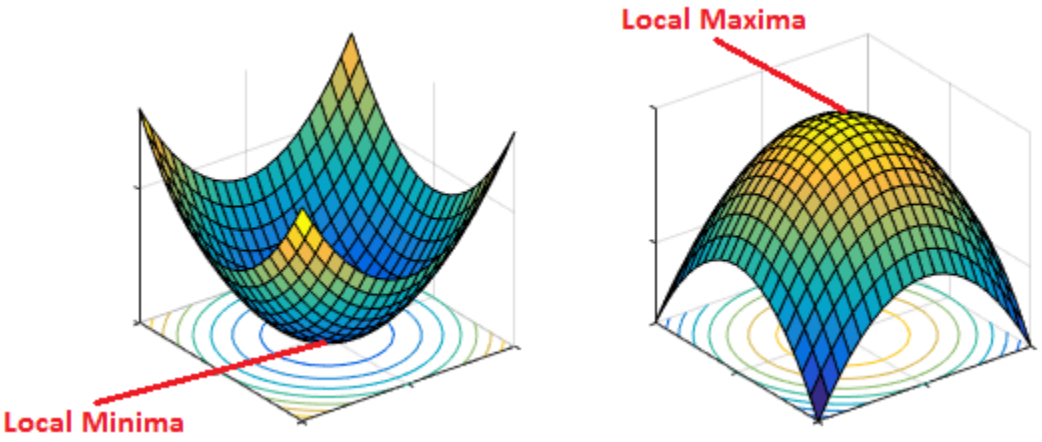

Figure 5-17 Local minima and maxima [31]

The function findMaxima is used to determine the maxima using the EDM values of the pixels. The primary input parameters for this function include the floatEdm (contains all EDM values of the pixels) threshold (an offset used to divide two touching seeds). This function sorts the EDM values and calculates the true height of the maximum points. The based on the true height and the EDM value, it determines several maximum points in the image. Further, it performs the

Watershed segmentation starting from the maximum points by flooding in one direction at a time and segments the image as explained earlier. Finally, it returns a ByteProcessor object which has the segmented particles. While calculating the true height for the pixels, we again need to compare the EDM value of the current pixel with each one of its neighboring pixels in all the 8 directions. In order to easily traverse to all the directions for any pixel, we have a helper function named makeDirectionOffsets.

Reference for maintain all the directions

The function makeDirectionOffsets is one of the most used helper functions in the process of finding the maxima. This function creates an array of offsets for all the 8 directions in clockwise order. This array is globally declared and is named as dirOffset. This dirOffset helps us to access an index of a single dimensional array on the basis of its corresponding two-dimensional array index representation. For instance, consider a two-dimensional array as follows

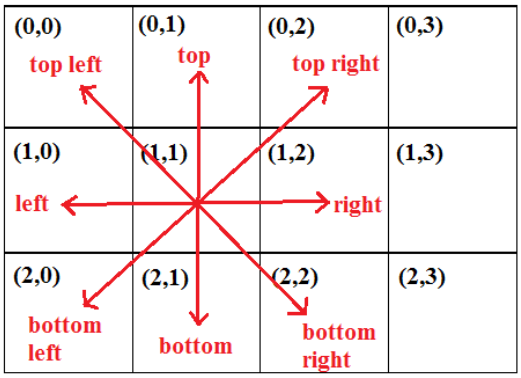

Figure 5-18 Eight directions in a two-dimensional array

As we already know, we will be using the two-dimensional array by making use of a reference which is a single dimensional array. The single dimensional reference for the above array will be as follows:

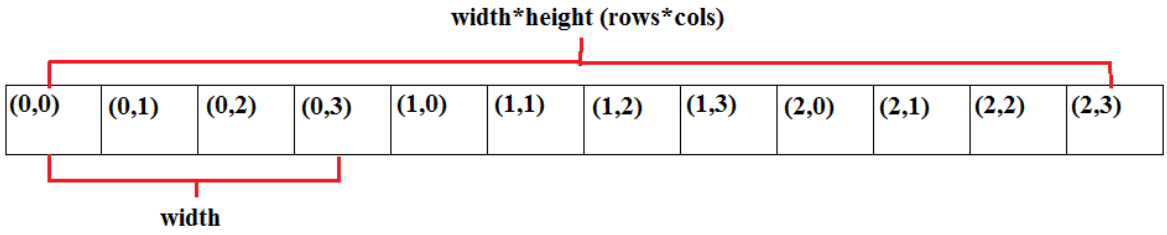

Figure 5-19 Single dimensional representation of the above 2D array

To traverse this single dimensional array which is of the width (rows * cols) by replicating two-dimensional access, we need to make use of the width of an image to traverse the upper row and lower row to reach the top and bottom pixels. This can be visualized as follows:

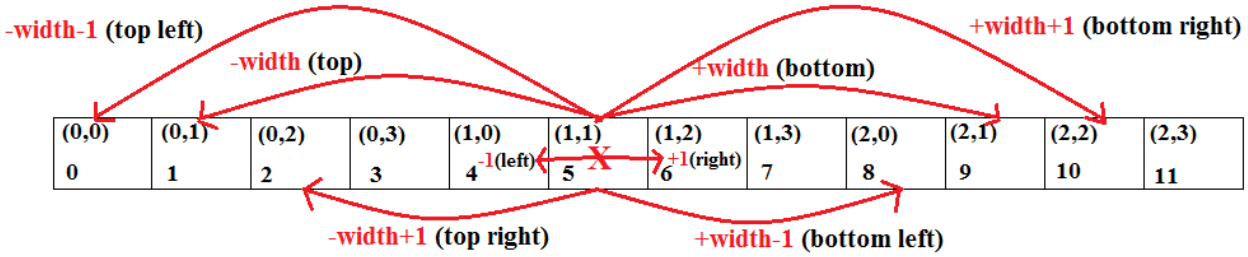

Figure 5-20 Offsets of directions in a single dimensional array

All these operations are statically stored in the dirOffset array as shown below:

dirOffset = new int[] {-width, -width+1, +1, +width+1, +width, +width-1, -1, -width-1 };

By doing this, for any pixel x if we can acquire its top pixel by just calling x+dirOffset[0], top right by calling x+dirOffset[1] and so on. This function also calculates the minimum number of bits required to store the width of the image. For instance, for storing 3 we need at least 2 bits. For storing 6, we need at least 3 bits. So, the storage capacity increases by the power of 2. So, to store the width of the image, we multiply a variable name mult by 2 until and unless it is greater than the width. So, whenever mult is greater than width, we can clearly state that we need minimum of mult-1 bits to store the width. As we have already seen earlier, we encode x coordinate into least

significant 16 bits and the y coordinate in the most significant 16 bits. As we already know, when extracting these x and y coordinates individually, we need to perform bitwise and, bitwise shift operations to get them. So, we also need to know how many bits to shift for getting y coordinates in the most significant 16 bits. To track this, we increment a variable called shift by 1 every time mult is multiplied by 2. This is done as follows:

```
int shift = 0, mult=1;
do {
shift++; mult*=2;
} while (mult < width);
intEncodeXMask = mult-1; //number of bits required for width (x coordinate)
intEncodeYMask = ~intEncodeXMask; //number of bits to store y coordinate (remaining bits)
intEncodeShift = shift; //number of bits to shift
```

This makeDirectionOffsets function is used very frequently in each and every step in the further process.

5.3.1 Finding the Maxima

In the function findMaxima, we take the EDM values into an ImageProcessor object (ip). Further, we also take a new ByteProcessor object named typeP with width and height of that of the image. Initially, we find the lowest and highest of the EDM values and store them in globalMin and globalMax. These act as temporary local minima and maxima respectively. Now, we make a call to the getSortedMaxPoints() function. This function returns all the local maxima based on their EDM value and calculated true height which will be calculated further.

5.3.2 Getting the sorted Maximum points

The function getSortedMaxPoints takes the ip object (EDM values), typeP (new ByteProcessor object), globalMin, and globalMax as primary parameters and returns all the local maxima pixel data in a sorted fashion in the form of a single dimensional array. Further, we take a single dimensional byte array named types which acts as a reference to the double dimensional ByteProcessor object typeP. As we already know, all the changes made on types will automatically be reflected in the typeP object. In the byte image typeP, all the maximum points will be marked as MAXIMUM (which is declared as a flag and is instantiated to (byte) 1). However, in Android, we directly send the double dimensional float array floatEdm having the EDM values into this function. Further, we consider typeP as a double dimensional integer array. As we cannot have a reference for typeP as in Java, we directly update the double dimensional typeP array in this function. For this purpose, we also send the image width and height as parameters to this function for accessing the corresponding two-dimensional array. This function returns a single dimensional long array maxPoints which are sorted by value. In this returned array, each element (long 64-bit integer) consists of the value encoded in the most significant 32-bits and the corresponding pixel coordinates are encoded into the least significant 32-bits.

To begin with, we start by traversing each pixel's index in ip (has EDM stored in corresponding pixel indexes) and get its EDM value as v. We then calculate the true EDM height of this pixel as vTrue. We calculate this by calling the trueEdmHeight function.

5.3.2.1 Calculation of the true EDM height

The function trueEdmHeight() takes the x coordinate, y coordinate and the ip object (EDM) as input parameters and returns the true EDM height for the corresponding pixel as vTrue. While determining the maximum points, we compare each pixel's EDM value with the value of all its

neighbors. If a pixel has an EDM value greater than all its neighbors, then it is considered as a max point. For this, having EDM values alone is not sufficient. We further need the true height of each pixel. The true EDM height is nothing but the accurate height of a pixel. The makeFloatEDM uses Euclidean distance for computing the distance from the nearest black pixel and it does not give the exact height of every pixel. For instance, in the below image, the white pixels represent the seed and the black pixels represent the background. So, the makeFloatEdm function doesn't take diagonals into account while computing the distance. It just calculates the distance based on the coordinate difference. This can be visualized as follows:

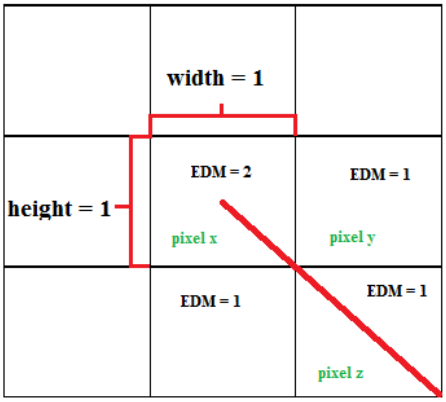

Figure 5-21 Diagonal distance

As we can see, for pixel x in the center, the Euclidean distance will be computed to be same when calculated from right (y) and bottom right (z), directions. But, the accurate distance must be different for the diagonal cases. However, the makeFloatEdm doesn't take them into account which vary the EDM values slightly. By doing this, few points which are not actually maxima will be considered as maxima and few points which are actually maximum, might be missed. The function trueEdmHeight takes the pixel's EDM value as v and then assigns its true height vTrue with the value of v + 0.5 * SQRT(2) as the true EDM height can never cross diagonal distance ($length^2 + width^2 = hypotenuse^2$, length = width = 1 ➔ hypotenuse = 1.414). We need to check

with 8 directions now. Here, we check with the EDM values of the pixels in opposite directions at a time. We can now make use of dirOffset array to get the values of pixels in all the directions. Initially, we take direction d as 0 (top) and d2 as d+4 (0+4 = 4, bottom). We get the EDM values from these directions into v1 and v2 as follows:

v1 = pixels[(x + y *width) +dirOffset[d]];

v2 = pixels[(x + y *width) +dirOffset[d2]];

We use a variable h to store the temporary true height of the current pixel. Now, if our current pixel's value v is greater than both v1 and v2, then we update the value of h with the average of v1 and v2 else, we update h with minimum of v1 and v2. And, if we are checking top, bottom (d=0, d2=4) or right, left (d=2, d2=6) directions currently, we just add 1 to h. But, if we are checking the diagonal directions which are bottom right, left top (d=3, d2=7) or right top, bottom left (d=1, d2=5), then we add 1.414 to the value of h. By doing this, we have the accurate height from the diagonal direction. We repeat until we check for all the four cases (all pairs of opposite directions) and then update our current pixel's trueH with the minimum of all the h values. Consider the following figure in the left with a set of pixels and corresponding EDM values returned by the makeFloatEdm function. Pixel 5 has the highest EDM value (3) in its neighborhood. True height of that pixel determines if it can be accounted as a maximum point or not. So, v is 3. As show in the figure in the right, if we are checking for left and directions we have v1 as 2 v2 as 2 and h as 2. As we are not in the diagonal case, we add 1 to the average and h becomes 3 which is the temporary true height for the central pixel when checked in left and right directions. The calculation is similar for top and bottom directions as the values are same. Now, when d is 3, d2 becomes 7. So, we are checking with the bottom right and top left pixels' EDM values. In this case, v1 is 2, v2 is 2 and h is 2. As we are in the diagonal case, we add 1.414 to h

and h becomes 3.414. This is the temporary true height when checked in diagonal direction. Finally, we assign the value of trueH for the central pixel as minimum of temporary heights h obtained by checking all the 4 direction pairs. Then, trueH becomes 3 (minimum of 3, 3, 3.414, 3.414). This proves that the central pixel indeed is a maximum point. But, we can argue that the initially computed EDM values are true.

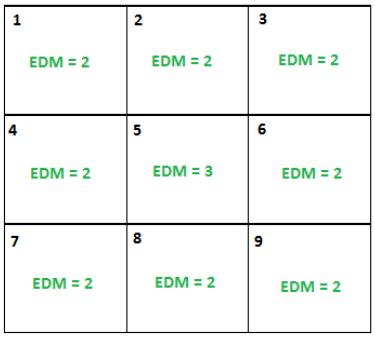

Figure 5-22 Sample EDM values in a neighborhood – case 1

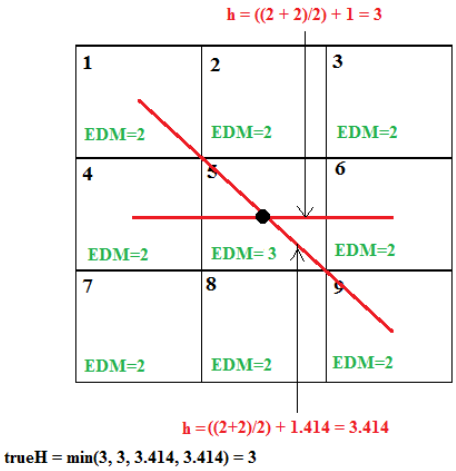

Figure 5-23 Calculation of true EDM height for pixel 5 – case 2

Now, consider a slightly different case. Here pixels 2, 4, 5, 6 and 8 can be accounted as maximum points. Let us compute the trueEdmHeight for the pixel 5. We have the value of v as 2.

1 EDM = 1	2 EDM = 2	3 EDM = 1
4 EDM = 2	5 EDM = 2	6 EDM = 2
7 EDM = 1	8 EDM = 2	9 EDM = 1

Figure 5-24 Sample EDM values in a neighborhood - case 2

As shown in the below figure, if we are checking for left and directions we have v1 as 2 v2 as 2 and h as 2. As we are not in the diagonal case, we add 1 to the average and h becomes 3 which is the temporary true height for the central pixel when checked in left and right directions. The calculation is similar for top and bottom directions as the values are same. Now, when d is 3, d2 becomes 7. So, we are checking with the bottom right and top left pixels' EDM values. In this case, v1 is 1, v2 is 1 and h is 1. As we are in the diagonal case, we add 1.414 to h and h becomes 2.414. This is the temporary true height when checked in diagonal direction. As we can see this value is higher than its corresponding EDM value when calculated from the same direction. Finally, we assign the value of trueH for the central pixel as minimum of temporary heights h obtained by checking all the 4 direction pairs. Then, trueH becomes 2.414 (minimum of 3, 3, 2.414, 2.414). We can clearly see that the true height of this pixel is higher than the previously computed EDM value. Pixel 5 is greater than all values of its neighborhood and it is the maximum point but pixels 2, 4, 6 and 8 cannot be treated as maximum points. This can be seen in the below figure. Hence, by doing this we can narrow down the maximum points based on their true EDM height.

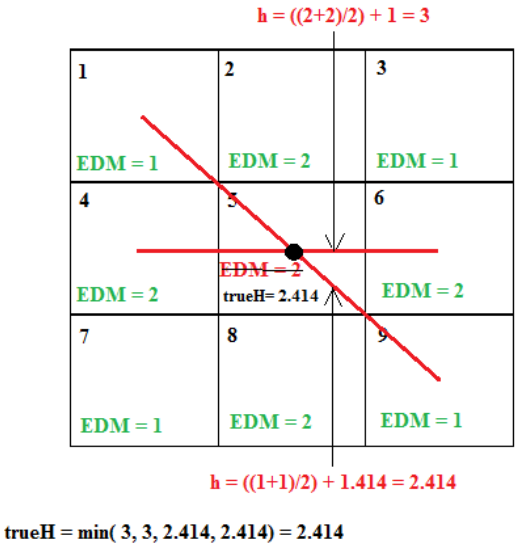

Figure 5-25 Calculation of true EDM height for pixel 5 - case 2

Now, we come back from the trueEdmHeight function call, we have the true height for the current pixel as vTrue. We also have its original EDM value as v. Now, we iterate from d=0 to d=7 to check for each direction at a time. We have a neighboring pixel in each direction. So, for each direction, we take the neighboring pixel's EDM value as vNeighbor and it's true EDM height as vNeighborTrue.

vNeighbor = ip.getPixelValue(x+DIR_X_OFFSET[d], y+DIR_Y_OFFSET[d]);

vNeighborTrue = trueEdmHeight(x+DIR_X_OFFSET[d], y+DIR_Y_OFFSET[d], ip);

In Android, we get the vNeighbor from the two dimensionaol array floatedm itself.

vNeighbor = floated [x+DIR_X_OFFSET[d]] [y+DIR_Y_OFFSET[d]]

So, for each of the directions, we check if the current pixel's EDM value (v) is greater than EDM value of its neighbor (vNeighbor) in that direction and the current pixel's true EDM height (vTrue) is greater than the true EDM height of its neighbor (vNeighborTrue). After checking all the directions, if the current pixel's EDM value and true EDM height are greater than that of its 8 neighbors, we mark it as maximum in the corresponding single dimensional offset (x+y*width) of

this pixel's coordinates (x,y) the previously defined types array. We put MAXIMUM (defined as (byte) 1) in this index to indicate that this pixel is a maximum point. As we already know that this types array is a reference to the ByteProcessor object typeP, typeP will be automatically updated. But, in Android, we directly update the double dimensional integer array typeP (typeP [x] [y]) with MAXIMUM.

After repeating this process for all the pixels, we will have all the pixels which represent the maximum points. These are marked as MAXIMUM at their corresponding indexes in the types array. Further, we have a new long array named maxPoints to represent the maximum points. We again traverse through the entire image (ip). For a pixel (x,y), if its corresponding types[x+y*width] is marked as MAXIMUM, we calculate the trueEdmHeight of this pixel as iValue. We now encode the pixel's true height (iValue) into most significant 32 bits and the pixel's offset in least significant 32-bits in the maxPoints array.

```
maxPoints[i++] = (long) iValue<<32 | (x+y*width);
```

Finally, we have all the maximum points and their true edm heights in maxPoints. We now sort this array based on the true EDM heights and return it. Now, we need to further analyze each these maximum points and mark the final maximum points before the segmentation process. This is done by the analizeAndMarkMaxima function.

5.3.3 Analyzing and marking the maximum points

Now that, we have all the sorted maximum points, we need to analyze each of these maximum points. In this part, we traverse through the neighborhood of each of these maximum points and check if there are any other maximum points greater than them in each of their neighborhood. We need to do this because, while narrowing down the maximum points, we have calculated true EDM height of each pixel and we considered it as a maximum point only if both

of its original EDM value and true EDM value are greater than those of its 8 neighbors. If a pixel's true EDM height is greater than all its neighbors but its EDM value is not greater, then it will not be considered as a maximum point. But, we need to include this for accurate determination of our maxima or markers. Also, we will mark similar maximum points in that neighborhood and group all these maximum points to form well defined local maxima in a seed. That is the reason why we check the corresponding neighborhood of an existing maximum point to find similar maximum points whose true EDM values differ by a very small value than the original max point in that seed. To do this, we have a pre-defined tolerance value for our seeds which is also known as threshold. For each of the originally listed maximum points, we check all other points whose true EDM heights fall below this tolerance level such that they become eligible to be a part of the maximum point in that particular seed. So, we have a tolerance value which describes a radius around the current maximum point being checked. The default tolerance (threshold) value is considered as 0.8. This tolerance plays a very important role in the determination of the UEPs and is relative to the size of the seeds we are dealing with. We do the same for all the originally computed maximum points. The analyzeAndMarkMaxima takes ip (original EDM values), maxPoints (the sorted maximum points), typeP (object in which maximum points marked as MAXIMUM at their corresponding indexes)) and tolerance value. We use a set of flags in this function. They include LISTED which is declared as (byte) 2, EQUAL which is declared as (byte) 16, PROCESSED which is declared as (byte) 4, MAX_AREA which is declared as (byte) 8 and MAX_POINT which is declared as (byte) 32. Also, we store the type of each pixel in the types object. This is a reference to the typeP object. The types object tells us what type each point is that is, a maximum point or listed or processed point. For each of the originally listed maximum point, we add all the offsets of similar maximum points into a single dimensional array named pList. We start iterating from

the largest of the maximum points in maxPoints, and we add that point to the pList array. As we visited it at least once, we mark it as LISTED in types. We mark all the flags using bitwise operations as follows:

types[offset0] |= (EQUAL|LISTED);

typeP[offset0%width] [offset0/width] |= (EQUAL|LISTED); //in Android

Now, we start visiting its neighbors and neighbors of neighbors. If we find any neighbor whose true EDM height is very near to our current max point, we add it to the pList. Also, for each of the neighbors, we check if any point has higher true EDM height than the current listed maximum point. If yes, we replace our current max point with the newly found max point. We reset the pList and push this newly found max point into pList and start visiting its neighbors and add closer points to the pList. But, if any pixel's true height is equal to the current listed maximum point, we mark that pixel's offset as EQUAL in types. Then we find a perfect max point in the middle location of these two points, add this to pList and repeat this process. At the end of one iteration, we will have a set of similar maximum points for the current max point for a seed listed in pList. We will then mark the greatest max point as MAX_POINT all the listed max points as processed and MAX_AREA in the types array. We then repeat this process for all the originally listed maximum points in maxPoints. Finally, we will have all grouped maximas for each and every maximum point found earlier. For each of the seeds, we will have a group of maximum points representing the maxima of that seed.

Process

Let us see how this is being done for a maximum point in maxPoints. We have three important loops in this process. These three loops are nested. In the first loop (say loop1), we will start processing from the greatest of all the sorted maximum points., that is the value at last index in maxPoints. As we already know that in the long array maxPoints, in each of the array elements,

we have encoded the true EDM height and the corresponding single dimensional offset of that pixel. So, now we get the offset of current maximum point by just casting it to integer which gives us the least significant 32 bits. We store this offset into offset0 as follows:

offset0 = (int) maxPoints[iMax]; //iMax = maxPoints.length-1

As we keep on marking already processed maximum points as PROCESSED, we will only move forward if the current maximum point's neighborhood is not already processed. As we set the flags using bitwise OR operation, we can check if the flag is set by making use of bitwise AND operation as follows:

types[offset0] & PROCESSED == 0 //equal to 0 implies that it is not processed

Now, we get the current maximum point's coordinates and store them in x0 and y0 respectively and its true EDM height is stored into v0. For each max point, we repeat the following loop. This is the second loop (say loop2). As stated earlier, that we keep listing the current maximum point's (in loop1) neighboring maximum points into pList. Initially, we store the current maximum point's offset (offset0) into pList[0] and mark current maximum point as LISTED and EQUAL. We maintain a counter to track the number of neighboring points being added to our pList. This counter is listLen and is 1 currently. Another variable listI is used as a reference for current listed element and is 0 initially. Further, we maintain a Boolean flag named sortingError and set it to false. This states that we have not found any new maximum greater than the current in the neighborhood (current seed). Now, for each element listed in our pList, we start checking other maximum points in its neighborhood (seed) and keep listing them.

This is done in the third loop (say loop3). We begin with the currently listed max point here and store its one-dimensional index into a variable named offset. This can be done using the index reference of current listed point listI (offset = pList[listI]). For the originally listed max point, if

any of its neighbors true EDM value (say v2) is within the tolerance of originally listed max point (v0 - tolerance) for this seed, this states that this is a slightly smaller yet similar max point. We then check if it is not already listed and add it to pList next to originally listed max point. Also, if this point's v2 is greater than v0, it states that we have found a new max point. Then we change the flag sortingError to false and replace the original max point with this new one.

```
if (v2 >= v0-(float)tolerance) { //find similar max points
        if (v2 > v0) {  //check if any of the neighbors are bigger then original
            sortingError = true; //new max point found in this seed
            offset0 = offset2; //replace the original max point with new one
            v0 = v2;  x0 = x2; y0 = y2; //replace EDM heights and coordinates
        }
        pList[listLen] = offset2; //push similar max point into the listed points
        listLen++;
        types[offset2] |= LISTED; //mark it as listed
```

One can argue that, for the maximum points in the maxPoints array, we have already verified that their EDM and true EDM values are greater than the values of all their 8 neighbors. So, if we have a point whose value is bigger than the originally listed one, we should have listed it previously in getSortedMaxPoints itself. The answer to this question is that, there we just checked for the immediate neighbors but here, we are checking for the neighbors of neighbors as well. We will check for all those points in the neighborhood who fall into the tolerance. An example of this scenario can be followed in the below picture.

Here v0 is true EDM height of originally listed max point and v2 is one of the points in the neighborhood which is a newly found max point in the neighborhood. We can see that v2 has its

true height bigger than v0. But, v2 is not listed in maxPoints as v2 has its EDM value (24.1) lesser than the EDM values of one of its neighbors (24.6) even though its true EDM height (23.93) is greater than its neighbor's true EDM height (23.91). This is where true EDM height assists us to find the most accurate max points. In other case, if the point inside the tolerance has equal true EDM height as the original max point, then we mark it as EQUAL and add up the offsets to calculate the center of these points in future. We then use that center coordinate to find the perfect nearest maximum point for that particular seed.

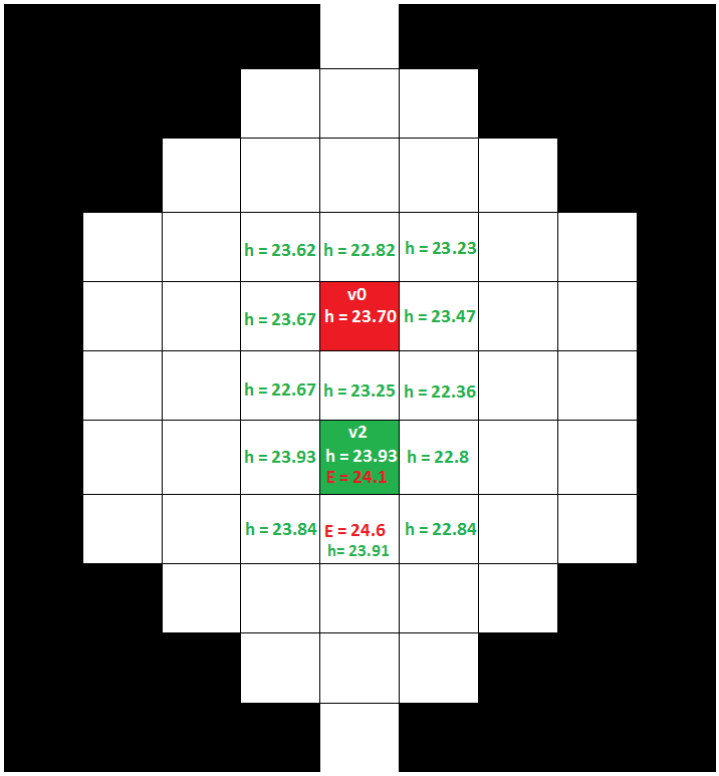

Figure 5-26 Bigger maxima in the neighborhood

We do this as follows:

```
if (v2==v0) {
        types[offset2] |= EQUAL;
        xEqual += x2;    //xEqual has the x coordinate of current max point initially
        yEqual += y2;   //xEqual has the y coordinate of current max point initially
```

```
            nEqual ++;
        }
```

So, by considering all these cases, we repeat this loop (loop3) until we list the offsets of current maximum point along with all similar max points (slightly smaller than the current max point) in pList. In the above process of listing, if we find any new bigger max point than the current one, we reset the entire pList and put this new max point at the beginning of pList and start listing all similar max points into the pList. After this, we have the maximum point in the first index of pList and all similar max points in the rest of the list. We now traverse through this list and mark each of these offsets as PROCESSED. We also mark them with MAX_AREA to state that these points are part of maxima of a particular seed.

```
int offset = pList[listI];
types[offset] |= PROCESSED;
types[offset] |= MAX_AREA;
```

Further, if we have multiple points marked as EQUAL, this means we have multiple maximum points in our seed. This might be due to several reasons. If the seed is distorted, due to its variations the true EDM distances may vary and this may result in several max points. So, to avoid the ambiguity during the flooding process, we calculate the average of the coordinates of these equal max points and then find max point which is listed in pList which is nearest to this average coordinate point.

```
minDist2 = 1e20;
if ((types[offset]&EQUAL)!=0) {
        double dist2 = (xEqual-x)*(double)(xEqual-x) + (yEqual-y)*(double)(yEqual-y);
        if (dist2 < minDist2) {
```

```
                minDist2 = dist2;  //this could be the best "single maximum" point
            }
        }
```

In Android, we check this as follows:

```
(typeP[offset % width][offset / width] & EQUAL) != 0
```

Consider the following example where a seed is normal but has two equal maximum points.

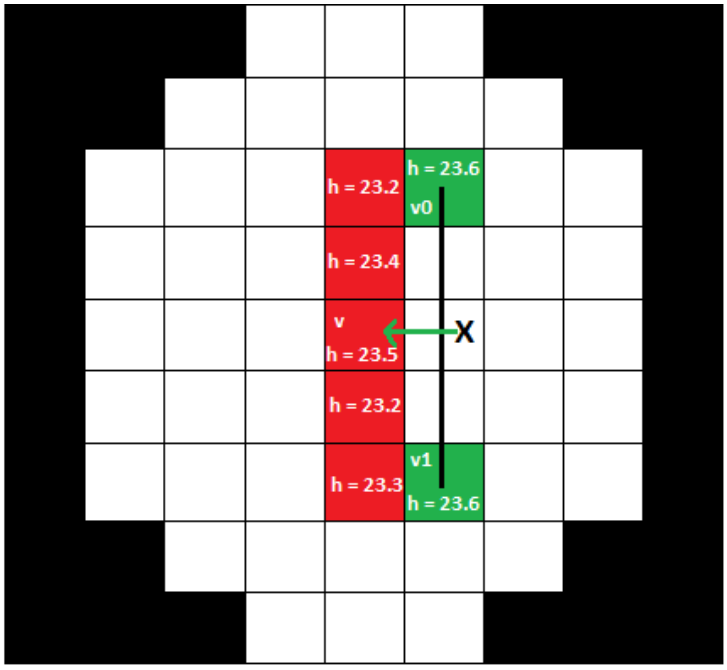

Figure 5-27 Perfect maxima between two equal maximum points

Here, two points v0 and v2 have their true EDM height as 23.6. They are represented in green and all the similar maximum points within the tolerance which are listed in pList are represented in red. We now find the center point of these two equal maxima and then find a maximum point which is nearest to the center point of the two equal maxima. The center point is marked as x in the figure and the nearest listed maximum point is v. Even though this maximum point doesn't have the highest true EDM height in its neighborhood, it forms the perfect maxima for this neighborhood (seed). Consider one more case where there is a possibility of multiple maximum

points in a single neighborhood. The seed might be distorted as shown in the below figure. The two equal maximum points are v0 and v2. Then we find the center point between these two pixels and that pixel itself is listed in pList and it can considered as a perfect maxima for this seed.

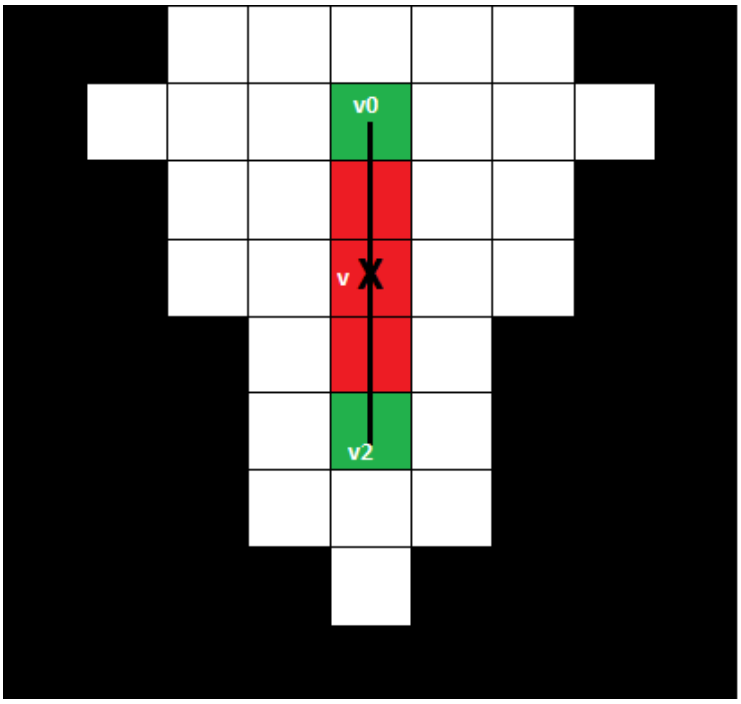

Figure 5-28 Perfect maxima between two maxima points - case 2

There is a new case called as a formation of Saddle points in our image. Seeds are of different shapes and we cannot expect each and every seed to be of the same shape and size. Few seeds may be slightly distorted as shown above. In such cases, even though there are multiple maximum points in the seed, all points which fall between them will have closer true EDM values as them which make them a part of maximum point. But few seeds may be extremely distorted which leads to the formation of valleys. For such seeds, there are two maxima and the points between them are very low (closer to local minima), they form a slope which looks like a valley. These valleys between maxima prominently called as Saddle points [30]. The term Saddle can be well understood

from the seat which is fastened at the back of a horse for riding. It is raised at the front and at the end. It is low in the middle to enable the rider to seat comfortably. It is shown in the below figure

Figure 5-29 Horse saddle [29]

Saddle point can be visualized as follows

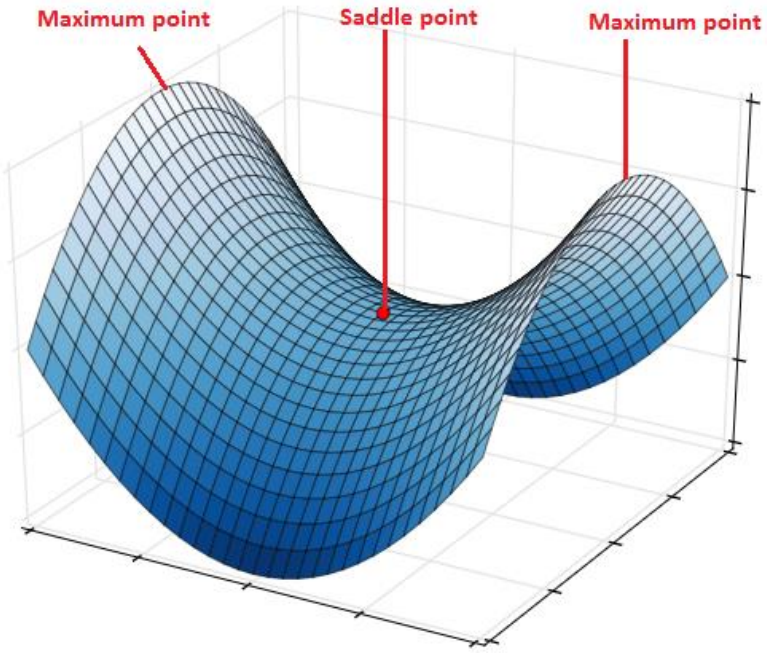

Figure 5-30 Saddle point between two maximum points [30]

If there is a Saddle point in a seed, then there will be two set of maximum points in either side of the saddle point. For instance, Consider the distorted seed as shown below. In this seed, there is a maximum point in the top which has the true EDM height as 23.6. All points closer to this value form a part of this maximum point and are marked in red.

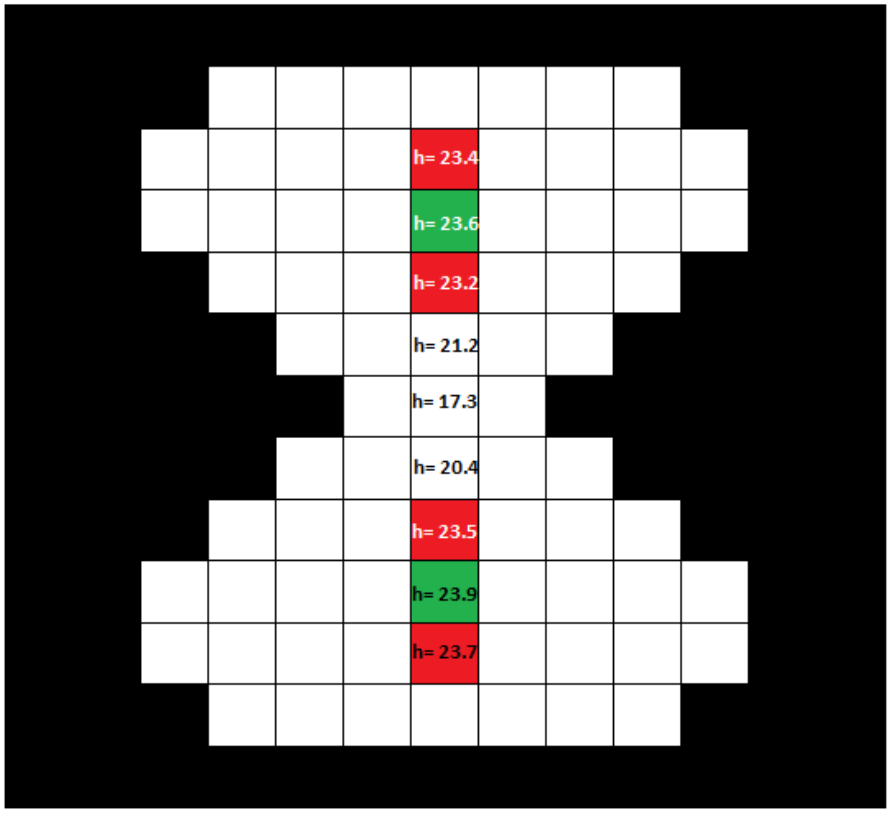

Figure 5-31 Saddle point in a seed

There is another maximum point below with the true EDM height as 23.9 and all the points which are closer to this value form a part of this maximum point. The points in the middle will not be considered as a part of either of these two maximum points as their true EDM values are smaller than the tolerance value and form a saddle point in the middle. In such cases, we will have two maximas in that neighborhood. In this way, we determine the perfect maximum points for different cases. After this step, we will be having a perfect maximum point in this iteration.

```
offset = pList[0];
types[offset] |= MAX_POINT; //Java
typeP[offset % width][offset / width] |= MAX_POINT; //Android
```

Finally, we mark the offset of the biggest maximum point (pList[0]) as MAX_POINT.

We are back to loop1 and we continue the same process until we check and group all similar max points for all the originally determined maximum points in maxPoints. To visualize this, consider the following image. The marked point is far from the background when compared to other pixels and it has the highest true EDM height. It is the maximum point in this neighborhood. Starting from this point, we need to consider all the points which are slightly around a tolerance value from the maximum point. In our case, the default tolerance (threshold) value is considered to be 0.8 and the true EDM height of this maximum point is 23.72. So, all the points whose true EDM heights are around the values of 22.92 and 24.52 will be listed in pList. If any of these listed points has greater true EDM height than the original maximum point, then our list will be reset and the new maximum point is considered as the current maximum point.

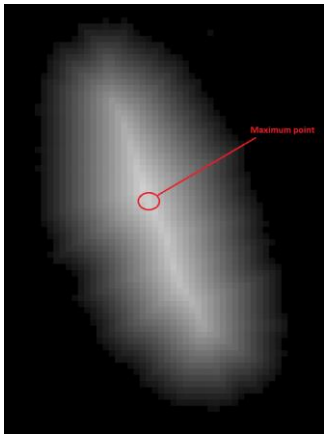

Figure 5-32 Maximum point in an EDM of a seed

If its true height is 24.2 (say), we then consider all the points which have their true EDM height between the values 23.4 and 25.0 will be listed in pList. In this seed, there was no such point which

was greater than the original maximum point. So, the points around the tolerance of the original maximum point will be marked as MAX_AREA in types and the biggest maximum point will be marked as MAX_POINT. In the below figure, the red line shows all the points which fall under the tolerance values and are marked as MAX_AREA. They represent a single maxima together for this particular seed.

Figure 5-33 Ultimate Eroded Points in a neighborhood

Thus, by the end of this analyzing process, we have the ByteProcessor object typeP which has the information regarding the maxima in our image. For all the pixels, which represent the maxima in our image, their corresponding index in typeP is marked with MAX_AREA and the perfect maximum points are marked with MAX_POINT. Similarly, in Android we have all this data in

the two-dimensional integer array typeP. Now, we need to create an 8-bit image by setting the corresponding pixels representing the MAX_AREA as 255.

5.3.4 Making an 8-bit image representing the Maxima

The primary goal in this step is to update all the maximum points and the points surrounding them as 255 (White). We already know that these points are marked by MAX_AREA in the ByteProcessor object typeP. We make this as an 8-bit image using the function make8bit function. This 8-bit image is used for the segmentation process. The primary parameters included in this function are the ip object (has all the EDM distances), typeP (has pixel type marked as MAX_AREA and MAX_POINT) and a threshold value. This function returns an 8-bit ByteProcessor image outIp. In Android, we have the double dimensional integer array typeP instead of the ByteProcessor object as in Java. Also, in Android we return a similar double dimensional integer array outIp representing an 8-bit image. Initially, we get the reference for typeP into types. We further define a threshold value to accurately define the background pixels. The threshold in our case is 0.5. So any pixel with its EDM distance lesser than this threshold value will be considered as a background pixel. As stated earlier, we have a new ByteProcessor object outIp which represents the output 8-bit image with 8-bit integer values representing the pixel intensities. In Android, we have a two-dimensional integer array representing the 8-bit image. Further, for faster accessing, we have a byte array named pixels which is a reference to the outIp. Any manipulations to the pixels array will be directly reflected in the outIp. As this is not supported in Android, we have to make a two-dimensional accessing and update outIp directly.

In this process, for every pixel, if it belongs to any of the maxima, then we mark it as white in outIp. To do this, we check whether the pixel's corresponding index in types is marked with MAX_AREA. If yes, then we mark this index in pixels as 255.

```
if ( (types[i] & MAX_AREA)!=0 ) { pixels[i] = (byte)255; }
```

In Android, we have to do it as follows:

```
if ( ( typeP[i%width] [i/width] &  MAX_AREA ) != 0 ) {
        outIp [i%width][ i/width] = (byte) 255;     }
```

Also, if any pixel's EDM value is lesser than the threshold value (0.5), then we set this pixel's index in outIp with 0. Further, if any pixel doesn't belong to any of the maximas, we round their EDM value to the nearest value using an offset and a factor and then assign this integer value in its corresponding index in outIp.

```
offset = globalMin - (globalMax- globalMin)*(1./253/2-1e-6);
factor = 253/(globalMax-minValue);
v = 1 + Math.round((edmValue-offset)*factor);
if (v<=254) pixels[i] = (byte) (v & 255);
```

The offset value is 0.9537524432826061 and the factor is 10.805911392331847. So, using these two we round the EDM values to their nearest values and also convert them to 8-bit. Here are some examples of EDM values and their corresponding values after conversion.

Before conversion the EDM value is: 1.4142135. After conversion, the value is: 1

Before conversion the EDM value is: 2.236068, After conversion, the value is: 1

Before conversion the EDM value is: 2.828427. After conversion, the value is: 3

Before conversion the EDM value is: 3.1622777. After conversion, the value is: 3

Before conversion the EDM value is: 3.6055512. After conversion, the value is: 4

Before conversion the EDM value is: 3.0. After conversion, the value is: 3

Before conversion the EDM value is: 6.3245554. After conversion, the value is: 6

Before conversion the EDM value is: 9.219544. After conversion, the value is: 9

In this way, we assign 255 to all the pixels belonging to the maximum areas, 0 to the background and their corresponding rounded EDM values to other pixels. So, after this process our input image will be

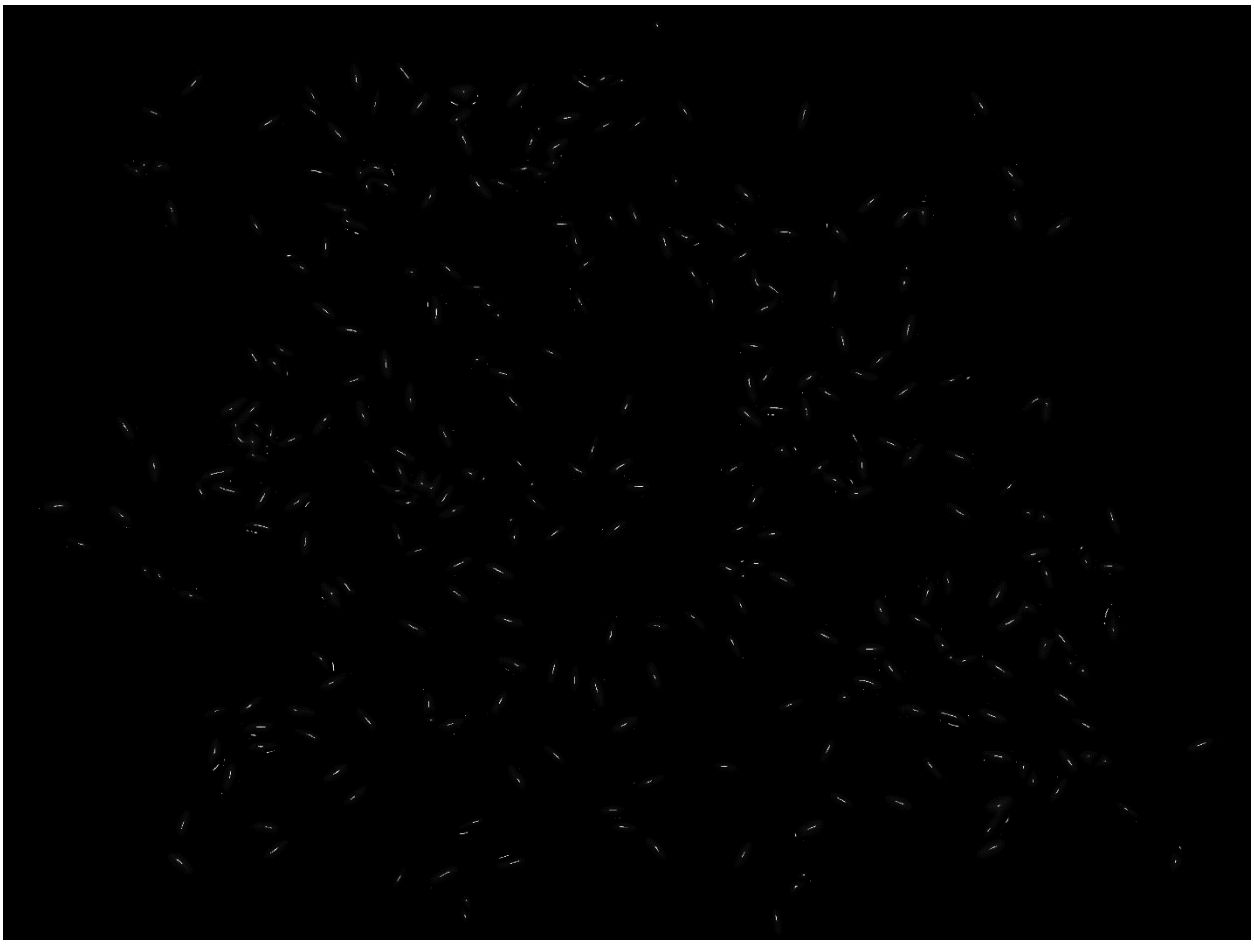

Figure 5-34 UEPs in the input image

We have determined maxima for each and every seed. If we look closely, for each seed the group of points which belong to maxima are marked as White and the points which belong to the rest of the seed are assigned their corresponding EDM values and the rest of the image is the background. For the seed with EDM distances shown in the left figure, the representation of maxima will be:

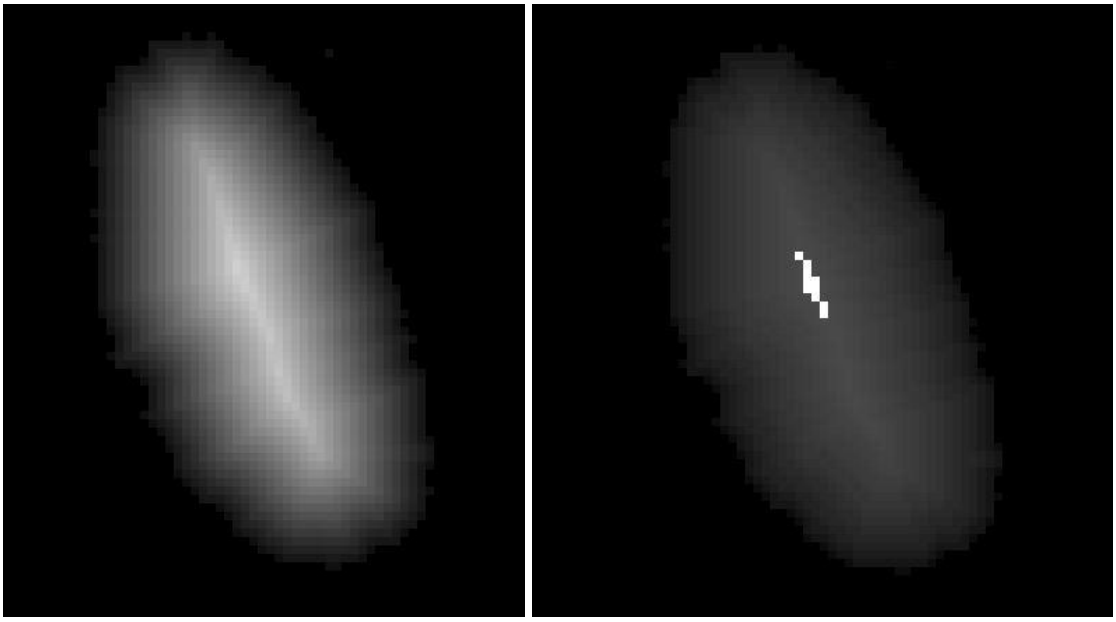

Figure 5-35 UEPs in the given EDM of a seed

So, all the points near the maximum point are set to 255 and the remaining remain with their EDM values. Now, we must clean up these maxima by eliminating the unwanted and unmarked ones.

5.3.5 Cleaning up the Maxima

The function cleanupMaxima is used to eliminate unmarked Maxima before the process of segmentation. While analyzing and marking the maxima we have checked the neighborhood of each of the originally listed maximum points and included corresponding similar points into maximum area of that particular seed by setting them as MAX_AREA. But, in some cases the originally listed maximum point may not be included in the maximum area. For example, we have already seen that, if we have two equal maximum points in a seed, we have considered the nearest similar point in the neighborhood to the average of these two maximum points as the perfect maximum point and then added points similar to this point into the maximum area. Here, the originally listed maximum points in this area which are equal may not be listed in this maximum area. They will not be marked with MAX_AREA. We may no longer need such unmarked maxima

and there is a need to eliminate them from the list of maximum points so as to remove ambiguity during the process of segmentation. This is done by the cleanupMaxima function. We iterate through each of the originally listed maximum points and check if it is included in the maximum area or not. If not, we then start checking its neighbors and neighbors until we find a pixel which is included in the maximum area. While moving along, we add all the neighbors whose EDM value is bigger than the current maximum point, are updated with the level of this point. We keep checking and updating its neighbors until we find a neighbor which is in the maximum area of that neighborhood. By doing so, we will move from a low level to the highest level and reset all the points in between with lowest level. So, we will change all the unmarked pixels who have higher EDM values set to the value of saddle point making them ineligible to be considered as maximum points. The primary parameters for this function include the outIp (8-bit image object with points in maximum area marked as 255), typeP (determines which pixel is marked with MAX_AREA) and maxPoints (originally listed maximum points). Initially, we have a reference to outIp, named pixels and types as a reference to typeP for single dimensional access. We modify the two dimensional integer arrays outIp and typeP directly in Android. We start with the maximum of the originally listed maximum points and check whether it is in maximum area or not. If it is included, we move to the next listed maximum point.

```
offset0 = (int)maxPoints[iMax];
if ((types[offset0]&(MAX_AREA|ELIMINATED))!=0) continue;
if ((typeP[offset0 % width][offset0/width]&(MAX_AREA|ELIMINATED))!=0)
continue; //Android
```

If it is not in max area, we move forward. We store its EDM value in a variable named loLevel and push this max point into pList. We start checking its neighbors and neighbors of neighbor

until some point is included in max area. While doing this, we add all the points to the pList who are at a higher level to the current pixel's level which is relatively at a lower level. When we find any point in the neighborhood to be in max area, we stop this process. Let offset2 be the offset of each neighboring pixel at a time.

```
  types[offset2]&MAX_AREA)!=0 => saddleFound = true; break;
typeP[offset0 % width][offset0/width] ]&MAX_AREA)!=0 => saddleFound =
true; break;  //Double dimensional access in Android
if ((pixels[offset2]&255)>=loLevel { pList[listLen++] = offset2; }
```

Then, we reset all the elements in the pList with the lowest level value (value of current max point).

```
for (listI=0; listI<lastLen; listI++) { //for all points higher than the level of the saddle point
    int offset = pList[listI];
    pixels[offset] = (byte)loLevel;     //set pixel value to the level of the saddle point
    types[offset] |= ELIMINATED;  //mark as processed
}
```

After this process, we can surely say that, there will be no unmarked maximum points in a seed which already has few points as local maxima (MAX_AREA). Now, we can start the segmentation process.

5.3.6 Watershed Segmentation

Now that we have all maxima defined clearly for each and every seed, we can start the process of Watershed segmentation. As explained earlier, we start flooding from the maxima points and proceed until we reach the background or water from another maxima. In our case, we start from these maxima and start moving in one direction at a time and set the pixels to white. By the end of this initial segmentation process, we will have all the seeds flooded with white pixels. When we reach a background pixel, we will stop flooding in that direction which marks the end

of the seed. Further, if we are moving in a direction in which a point is already reached and set from another maxima, this means that, the water from two maxima has collided and is about to merge. We do not flood into this point. By doing so, two touching seeds will be separated. By the end of this segmentation process, we will have all the seeds in the image flooded with white and the pixels between two touching seeds will not be flooded and they will be separated. We decide the direction of flooding based on the fate of each pixel. During the flooding process, if a pixel is set, then we flood in the opposite direction. This fate is predefined by a table named Fate table. This fate table determines which direction to flood when corresponding neighboring are already set.

5.3.6.1 Determining the fate of flooding

The function makeFateTable generates a look up table which is used during the watershed segmentation to check for a direction(s) to flood when the corresponding neighboring pixels are set. Each entry in the table represents a case in the 3*3 neighborhood of a pixel. As we have 8 directions for a pixel, we will have a total of 255 entries in the table covering all the possible cases which can be formulated as ${}^8C_0+{}^8C_1+{}^8C_2+{}^8C_3+{}^8C_4+{}^8C_5+{}^8C_6+{}^8C_7+{}^8C_8$ (= 254, 0 to 254 will result in 255 entries). The table labels the directions from 0 to 7 as shown in the below figure. As seen in the picture, we start representing from the top direction and move in the clockwise direction. So, 0 represents top, 1 represents top right and so on. The table has an array isSet with the length 8. This array is used to check if the neighboring pixels of the current pixel are set or not. If any of them is set, then its corresponding index in the isSet array will be true. So, the fate table just checks all the cases and generates corresponding values. For an unset pixel, we use the fate table's values

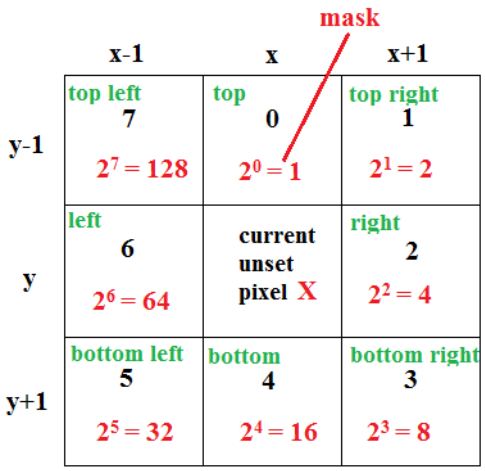

Figure 5-36 Determination of fate of X using neighboring pixels

to determine which direction to flood from there on based on the status of its neighboring pixels. The values generated by the fate table give a mask which is set in the corresponding bit of the pixel and each case is treated as an item in the fate table. The representation of this bits and items will be explained later during the process of flooding. Let us just see how the fate table is directing us towards the flooding direction here. For instance, for an unset pixel x, if only its top pixel is set (0^{th} bit is set), then the value of item is 1 (2^0) and fate table returns 16 which is 2^4 where 4 represents the bottom direction. If a pixel's both top and top right pixels are set (0^{th} and 1^{st} bits are set), then the value of item is (2^0+ 2^1=3), our table returns the value 48 which is 16(=2^4) +32(=2^5) where 4 and 5 represent bottom and bottom left directions. This can be visualized as shown in the below figure. Similarly, some other cases include, if the item is 181 that is, if the current pixel's top, right, top left, bottom, bottom left are set, then the fate table returns 91 which indicates us to flood in left, top, top right, bottom, bottom right directions. Also, fate table plays a major role in separating multiple touching seeds. If the pixel's neighbors are set by different maxima, we should not flood in that direction as further flooding merges them together. So, the fate table returns 0 if the current

pixel's neighbors contain more than one region where 0 represents that we should not flood into any direction from the current pixel.

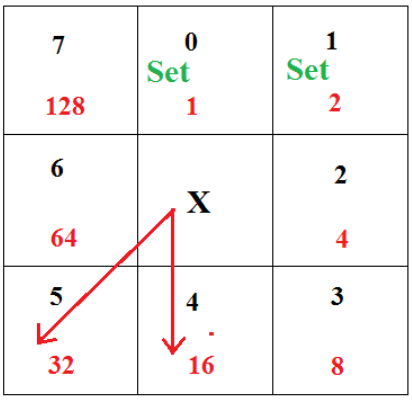

Figure 5-37 Fate returned by Fate table

For instance, if only the corners are set that is bits 1, 3, 5 ,7 (item = 2+8+32+128 = 170), we do not flow into any direction (FateTable[170]=0). If all the corners are set, this means that, they are set from different maxima and we should stop the flooding the current pixel. This can be visualized as follows:

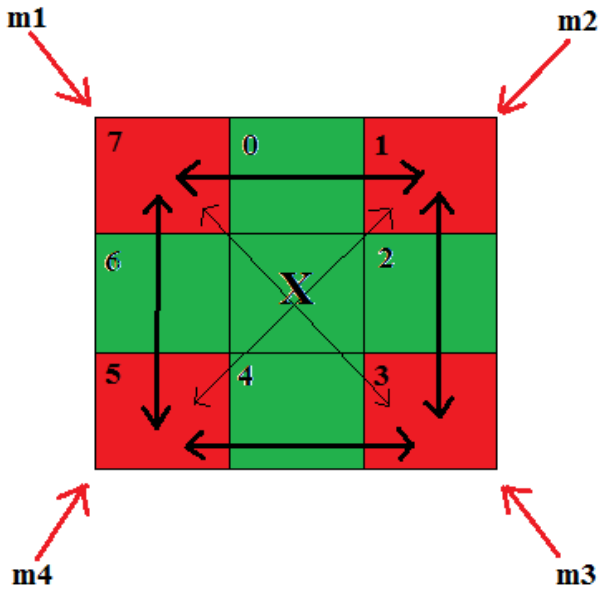

Figure 5-38 Fate of X when corner pixels are set

Here, the current unset pixel is X and all of its corner pixels are set. The set pixels are represented in red color as shown. They will be coming from be either 2 or 4 different maxima. Let us say, each of them are coming from a different maxima, If we flood in right direction, right pixel will be set (in green) and upper right and lower right seeds will be merged. If we flood in left direction, left pixel will be set (in green) and upper left and lower left seeds will be merged as shown. If we set the current pixel itself, then all the four seeds will be merged. So, we should not flood in any direction from this pixel for this case. For such cases, fate table returns 0 to indicate not to flood as to separate different seeds. Hence, our fate table decides the fate of each pixel by the status of its neighboring pixels. We can define a standard fate table with 0 to 255 scenarios and their corresponding values. We can also write a simple function to check the bits and generate a corresponding value by multiplying a predefined mask by 2 as follows

```
if (isSet[(i+4)%8]) table[item] |= mask;
mask*= 2;
```

Here, i is current direction and (i+4)%8 gives us the opposite direction. i changes from 0 to 8 representing all the directions. In each iteration, we set the corresponding bit based on the direction. If i is 0 which represents the top direction and if the top pixel is set then 16^{th} bit is set in table[item] using the bitwise operation above. Finally, after checking all the directions for case 'item', corresponding bits are set in table[item]. The final bits set in the value of table[item] state the directions for flooding.

Generation of Histogram

Before staring the flooding process, we generate a histogram for our outIp object which has all points in max areas set to 255 and other points with their EDM values. A histogram [32] is a plot or representation which consists the data related to the frequency distribution in the input image

or object. Histograms help us to scrutinize the data for its underlying properties of data distribution such as outliers, skewness etc. Histograms are basically represented as graphs and an example of histogram is given below. Consider the following data which represents the age of students: Shravan 22, Anamika 22, Nikitha 22, Chandra 21, Aruna 21, Navya 18, Sagar 19, Ashwin 19, Mahesh 18, Pruthvi 18, Alex 18, James 18, John 22, Robert 19, Ray 20, White 19, Sam 22, Andrea 20, Lisa 19, Haydon 19, Matt 22, Jeff 22, Monica 20, Greg 20.

If we want to group them based on their ages, we can generate a histogram based on their ages. This histogram groups the data based on the age parameter and stored them accordingly. The corresponding graphical representation of this histogram would be:

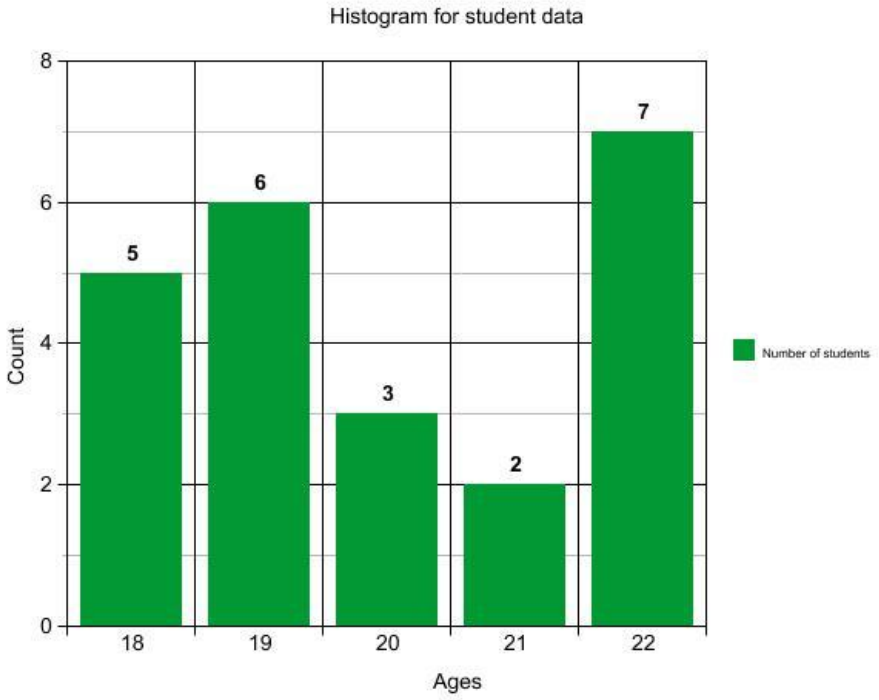

Figure 5-39 Histogram representing students different ages

The generation of histogram in our case, speeds up the watershed segmentation routine. So, we generate a histogram for our outIp object. The ImageProcessor class of ImageJ provides an inbuilt

method named getHistogram() which generates a histogram based on the pixel intensity values in the given object. So, we create a new single dimensional array named histogram and store the generate histogram into this array.

```
int[] histogram = ip.getHistogram();
```

However, in Android, we cannot create a histogram directly like this as we do not have support for the ImageProcesser object. Also, Android doesn't provide any pre-defined methods for generating histograms. So, we have to generate our own histogram based on the pixel intensity values in our corresponding two-dimensional integer array ip (represents outIp). So, we just traverse through each pixel in the array and if the current pixel is a maximum (255), we then increment the value at the index 255. Otherwise, we just update index which corresponds to the current index in the ip array. We can thus generate the histogram as follows:

```
int[] myarray = new int[256];
        for(int i = 0; i < height; i++){
                for(int j = 0; j < width; j++){
                        if(ip[j][i] == (byte) 255 { myarray[255]++; }
                        else { myarray[ip[j][i]]++; }}}
```

Now, we have the array myarray in Android which exactly replicates the histogram generated in Java. We start the flooding the process at highest levels and move towards the lower levels. This is a step by step process. So we move to the lower level only with all the pixels in the current level are processed. This takes up a lot of time as all the pixels are distributed in the image and at each level we have to iterate through the whole image to find the corresponding pixels. For this purpose, we maintain an array with all the offsets of pixels of one intensity grouped together. We need not have to process the background pixels during our segmentation process. We also need not process

the local maxima as they are already set. So, we have all the pixels with intensities greater than 0 and lesser than 255. For our seeds, as we know that the highest EDM value is around 23, we will only have pixels ranging from 1 to 22. (pixels with 23 will be maxima and are already processed). So, we need to group and store the offsets of all pixels from 1 to 22 in the coordinates array. For this we must know the number of pixels falling under this range. As explained earlier, the histogram serves this purpose. As stated earlier, we now create a new array named coordinates with the size of width*height – histogram[0] – histogram[255] (width*height gives us total number of pixels and 0, 255 are not needed). Now, in the coordinates array, we need to have all coordinates of pixels with intensities 1 starting in the beginning of coordinates array. Then we will have all the coordinates of pixels with intensity 2. These are followed by pixels with intensity 3 and so on. T do this, we need to know where each of the pixels with corresponding intensities start in the coordinates array. This is because, we start from the top and encounter pixels with different intensities at different positions as they are distributed. Consider the following image matrix where pixels with different intensities are distributed randomly depending on the image.

(0,0) 1	(0,1) 4	(0,2) 2	(0,3) 1
(1,0) 4	(1,1) 1	(1,2) 2	(1,3) 3
(2,0) 3	(2,1) 2	(2,2) 4	(2,3) 1
(3,0) 3	(3,1) 4	(3,2) 3	(3,3) 1

Figure 5-40 Sample input object with different intensities

The corresponding coordinates array with grouped coordinates will be

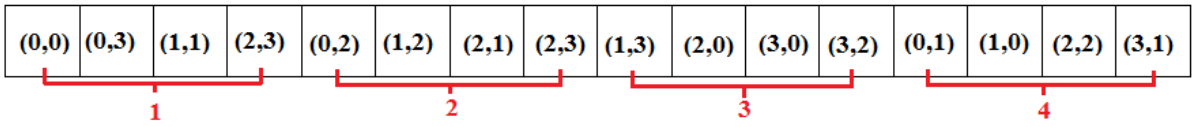

Figure 5-41 Coordinates of pixels with same intensities grouped together

Whenever we come across a pixel with a different intensity, we need to know the beginning offset of this pixel to store in the coordinates array. This beginning offset is the ending of the lower level pixel which can be determined by the sum of histogram values of all the lower level pixels. For example, in the above matrix, the beginning offset for pixel with intensity 4 is 12. This offset falls after all the lower level pixels 1,2 and 3. So, there are 4 pixels at level 1 (histogram[1]), 4 at level 2 (histogram[2]), 4 at level 3 (histogram[3]). Summing up these three gives us 12 which is the starting offset for pixels at level 4. We calculate the beginning offset for each of the levels (1 to 22) by storing them into their corresponding indexes in an array named levelStart. We do this as follows

```
int[] levelStart = new int[256];
for (int v=1; v<255; v++) {
        levelStart[v] = offset;
        offset += histogram[v]; }
```

This levelStart array has the starting offsets for each level. Also, we maintain a variable named highestValue to store the highest level. In our case, as stated earlier, the highestLevel will be 22. Further, for each level we also need to maintain the offset for the next occurring pixel in that level. For example, in the above figure, when we are at pixel with intensity 4 at coordinate (1,0), we know that the beginning offset for level 4 is 12 but we further need to know the index of the previous pixel at the same level so that we can insert the coordinate of this pixel next to it. For this

we have a new array named levelOffset with the size of highestValue. This array has the current offset related to each level in its corresponding index.

```
int[] levelOffset= new int[highestValue + 1];
```

Now, we traverse through our outIp object and for all the pixels in the levels 1 and 254, we store their coordinates in the encoded manner into the coordinates manner. levelOffset[level] gives the count of number of coordinates we previous entered in this level. So, we can get the offset to store the current pixel's coordinate by summing up the value of beginning offset of the current level (levelStart[level]) and number of elements entered in the current level levelOffset[level]). Then we encode the x and y coordinates into this offset as follows

```
for (int y=0, i=0; y<height; y++) {
    for (int x=0; x<width; x++, i++) {
    int v = pixels[i]&255; //
        if (v>0 && v<255) {
         offset = levelStart[v] + levelOffset[v];  //get the offset to store current
         coordinates[offset] = x | y<<intEncodeShift; //encode the coordinates
        levelOffset[v] ++;   //increment the offset to store next pixel in this level 'v'
    } } }
```

Now, we have all the coordinates of pixels grouped together based on their levels. We now call the makeFateTable() and store the fate table into an array named table. We now start the segmentation process starting from the highest level and gradually moving down to the lower levels. At each level, we flood the particles by setting them to White (255). For this process, we make use of the coordinates array to find the coordinates of pixels at each level and the fate table to determine the fate of the flooding process and to prevent merging multiple touching seeds.

5.3.6.2 The process of flooding

As we already know that, we start flooding from local maxima and process until we reach the background or water from other maxima. Flooding is nothing but setting the pixels to White (255). We already have set all the maxima pixels with white and background with black. So, we start with the points at the next highest level and start flooding in each direction. We then repeat this process by moving down to lower levels until we reach and flood the last level. In the below figure, we have a small part of the image representing the EDM distances of two seeds. We can see the local maxima already set to White. The highest level from which we start flooding will be located just next to our local maxima and are represented with Green color. At each level, we flood them by setting these pixels to White color. Then we move to the next lower level. These are represented

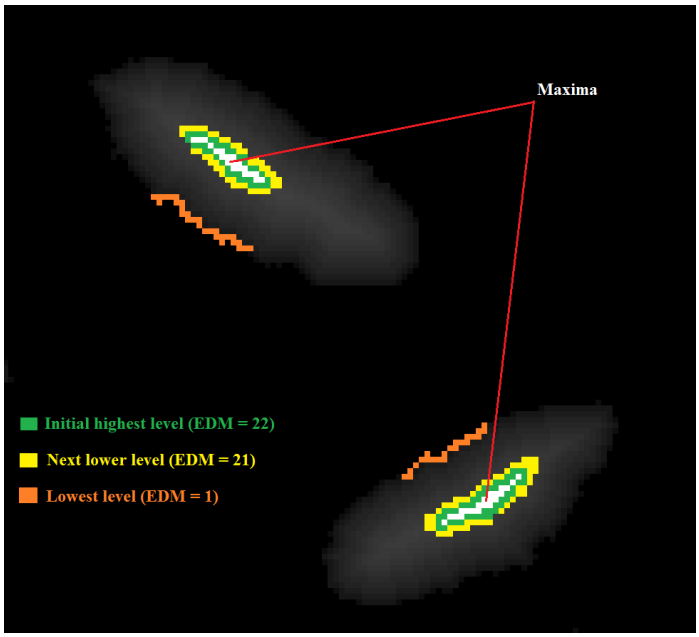

Figure 5-42 Representation of different levels

with Yellow color. We repeat this process until we process the lowest level pixels, that is the border pixels which are adjacent to the background. These are represented with Orange color. For a

current level, we simultaneously check for all the pixels in this level whether we can flood in a particular direction or not. So, for every pixel in this level, we make use of the fate table to determine the possibility of flooding in a particular direction based on the status of its neighboring pixels as explained earlier. If yes, then we set the current pixel to White (255). We do this for all the pixels in this level. The flooding of next level can be seen in the below figure.

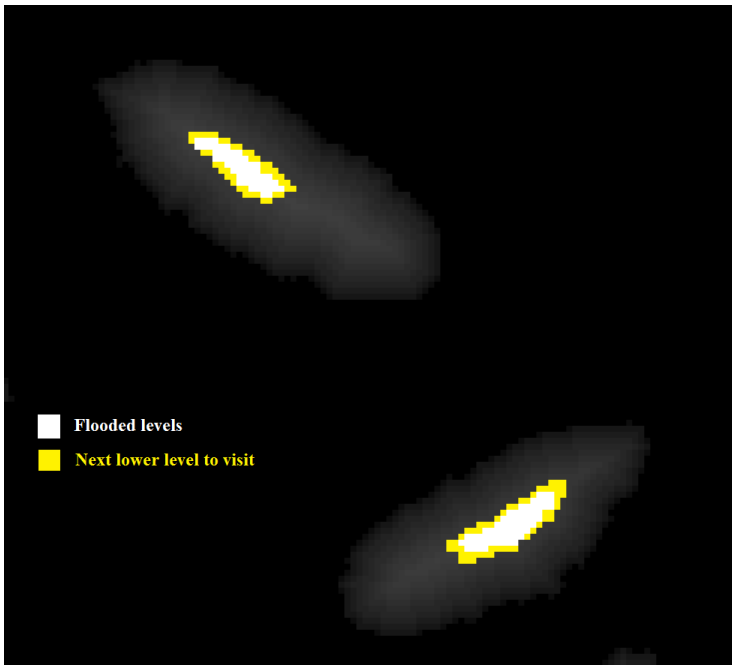

Figure 5-43 Image after flooding highest level

After processing all the pixels in this level, few of them may be set to White and few may not be set due to several reasons. This case might arise when we are in the lower levels. Among these unset pixels in the current level, we then add all the pixels which have at least one neighbor as a black pixel, to the next level. We do this because, they belong to the border of the seed and they require additional processing in the lowest level and if we miss them it results in the erosion of the outline of the seed. Then while processing the next level pixels, we further process these pixels again based on their fate which will be decided by the fate table. Also, we will never flood into a pixel, if its corner neighbors are set or only the diagonal neighbors are set or such cases which

mean that the neighboring pixels are set while coming from multiple maxima which belong to different seeds. As we have already generated the fate for such cases as no flow, the fate table returns 0 for this pixel and we will never set these pixels as white. They will be flushed out by setting them to Black. Consider two touching seeds as follows:

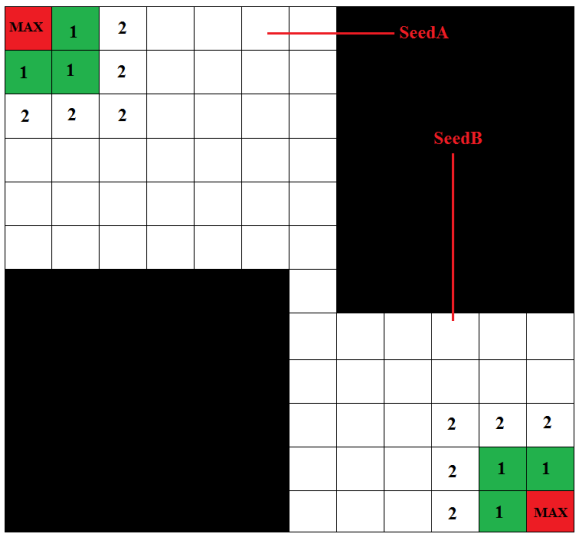

Figure 5-44 Scenario of two touching seeds

Here, we have two seeds A and B with their corresponding local maxima. We start flooding from level 1, flood the pixels and come to lower levels. Finally, we reach a point where the two pixels are about to merge. This can be visualized as shown in the below figure. As we can see, we have flooded to the lowest level and reached a point X, which is the merging point of the two seeds. Here, the Red colored pixels represent the corresponding maxima, Green colored pixels represent already set pixels to White and the Black colored pixels represent the background.

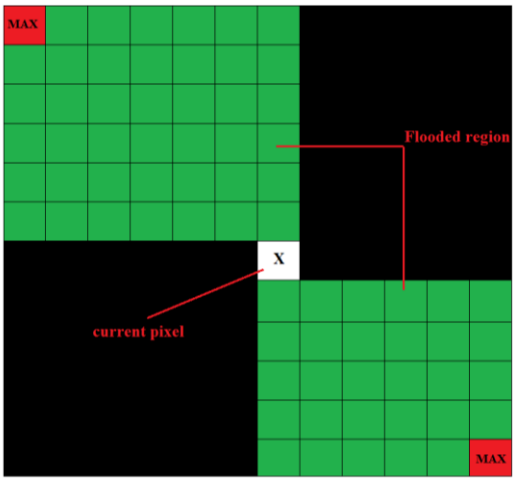

Figure 5-45 Flooding the seed border

Consider the 3*3 neighborhood of this pixel X below.

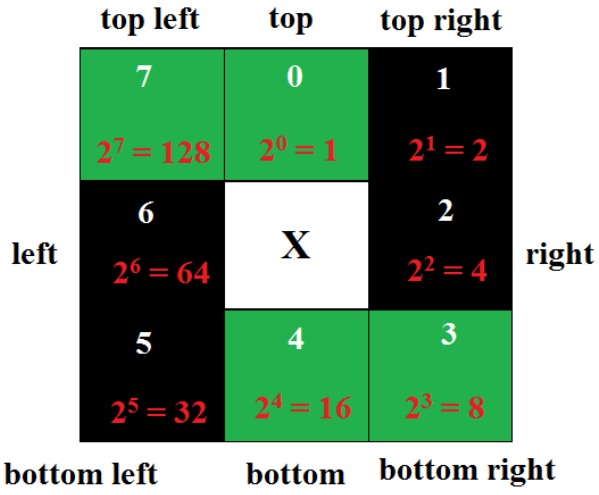

Figure 5-46 Fate of merging point X

Here, we can see that the top left, top, bottom and bottom right pixels are already set for the pixel X. Now, we lookup into the item corresponding to this case in the fate table to determine the direction of the flood. The corresponding item value would be 153 (128+1+16+8). The corresponding value for 153 is 0 (FateTable[153] = 0) which represents no flow. So, this pixel

cannot be set to White. Later all the unset points will be set to Black and the seeds will be separated as follows

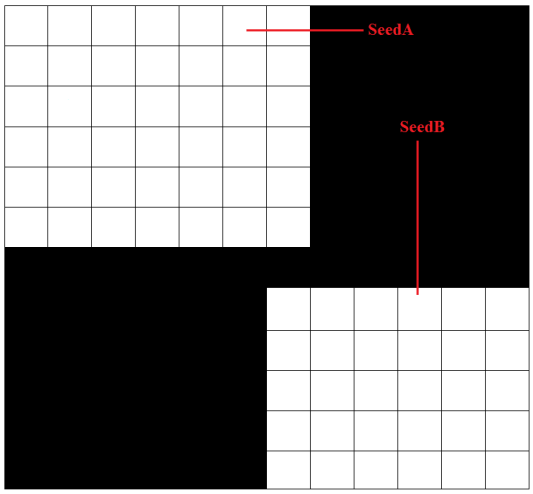

Figure 5-47 Pixel X remains unset

In our case, if the image consists of hundreds of wheat seeds, then multiple seeds will be overlapped and many such cases arise. We can view one such case here

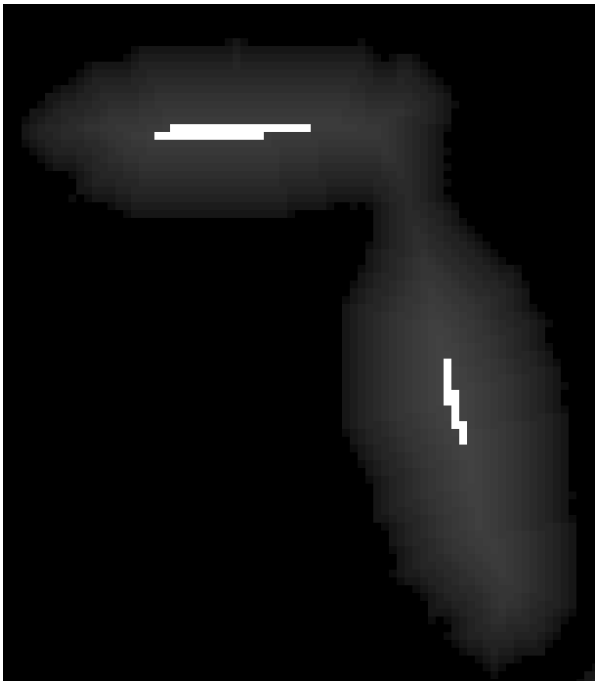

Figure 5-48 UEPs for two touching seeds

This is a small part of the outIp, after computing the local maxima. We can see that these two seeds are touching each other. During the watershed flooding process, the separation of these two seeds can be visualized as shown in the below figure.

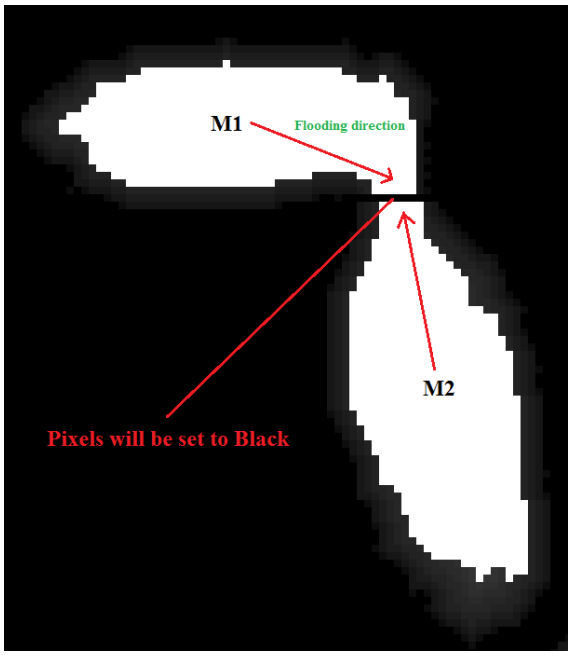

Figure 5-49 Two maximas approaching each other

And final image after segmentation would be:

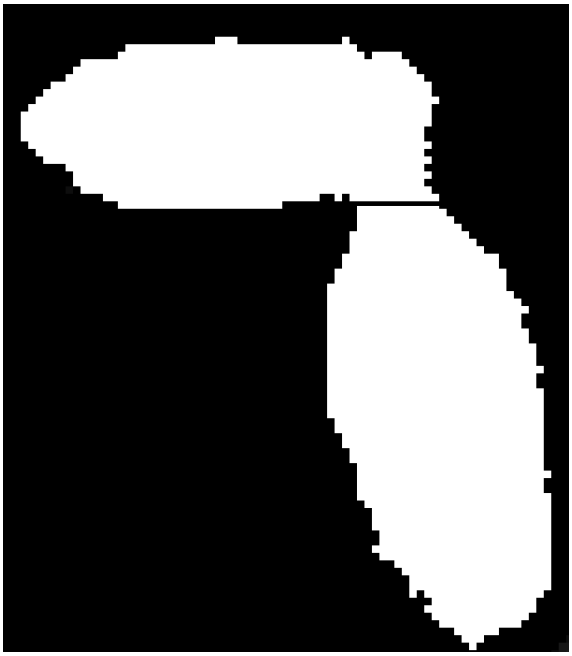

Figure 5-50 Resltant segmentation

This is the process of the segmentation. Coming to the implementation, we have all the grouped pixels according to their levels in the coordinates array. The variable highestValue consists of the next highest value after the maxima. We start from this level and start the flooding process for one level at a time by checking all the directions. We do this all the way until we hit the lowest level (1) adjacent to the background (0).

We have an array called levelStart which has the starting offset of each level in the coordinates array. For a particular level, levelStart[level] has an offset beginning from which all the coordinates of pixels of this intensity level are stored in the coordinates array.

For each level, we repeat the following process to flood

We maintain a variable called remaining which initially has the number of pixels at that level. That is the number of the pixels which have the intensity value corresponding to the current level are stored in remaining. Here we use the histogram to get the count.

remaining = histogram[level]

We also maintain counter to track different directions. We name this variable as idle and a temporary variable dIndex to expand each level in 8 directions. For each level, the order of directions we check for flooding is as follows 7 (top left), 3 (bottom right), 1 (top right), 5 (bottom left), 0 (top), 4 (bottom), 2 (right), 6 (left). Then we make a call to the processLevel() function. We send each one of the directions (pass) in the above order one at a time to this function. Other parameters include outIp (EDM values), fate table (to determine the fate), levelStart (to know the offset of current level in coordinates array), remaining (number of remaining unprocessed pixels in the current level), coordinates array (has all coordinates of pixels grouped together). This function checks whether we can flood in the given direction (pass) based on the status of each pixel's neighboring pixels.

Flooding at each level by calling the processLevel() function

In this function processLevel(), we have a byte array pixels as a reference to the outIp object. In Android, we directly manipulate the outIp object by traversing in two dimensional fashion. We have two local variables nChanged to know how many pixels are changed to White (255) in the current level and nUnchanged to know how many pixels are still unprocessed in the current level. We start checking each pixel in this level, beginning from the offset (named as levelStart) obtained from the levelStart array and do this until we check all the remaining points in this level. For each pixel, we get the individual x and y coordinates from encoded coordinates stored in coordinates[p] where p = levelStart. We initially set the item to 0 which is used later to refer in the fate table. We start checking in each direction and if the current pixel's top pixel is set (255), then we set the 0^{th} bit in item to 1 using bitwise XOR operation or bitwise OR operation.

if(pixels[offset-width] & 255) == 255) {item ^= 1; } //set 0^{th} bit with 1 (offset = x+ y*width)

if(outIp[(offset-width)%width] [(offset-width)/width] & 255 == 255) item ^ =1 ; //In Android

If the pixel in top right direction is set, then we set 1st bit in item to 1 and if right pixel is set, set 2^{nd} bit in item to 1; if bottom right pixel is set, set 3^{rd} bit in item to 1; if bottom pixel is set, set 4^{th} bit in item to 1; if bottom left pixel is set, set 5^{th} bit in item to 1; if left pixel is set, set 6^{th} bit in item to 1; if top left pixel is set, set 7^{th} bit in item to 1. We have all the corresponding bits set in item based on the current pixel's neighbors. We now check if the direction we are checking to flood is suggested by fate table. We do this by making use of left shift operator.

int mask = 1<<pass; // we mark the corresponding bit base don the direction

If we are checking for flooding in top left direction (7), 7^{th} bit is set in mask. We now check if top left is suggested by fate table. We do this by checking if the corresponding bit (7^{th}) is set in the value returned by fate table. If yes, we then we can flood in the current direction. We then add the

current pixel's outIp offset to an array named setPointList to mark it 255 later at once along with all changed pixels in this level. We will also increment nChanged value.

```
if( (fatetable[item]& pass) == mask ) { setPointList[nChanged++] = offset; }
```

If the fate table doesn't suggest flow in this direction, that is if the corresponding direction bit is not set in the value returned by the fate table, we then rewrite its coordinates in the beginning offset of this level in the coordinates array.

```
coordinates[levelStart+(nUnchanged++)] = xy;
```

In this manner, we check all the remaining pixels in the current level for current direction (pass) and keep adding processed pixels' offsets in setPointList and unprocessed pixels in coordinates. Finally, after checking for all pixels in this list, we flood all processed pixels by setting all the offsets in setPointsList to White (255) and return the nChanged value back to the previous function.

We now processed the current level (highest level initially), in one direction. As we have the number of pixels changed in the previous call to processLevel. We now remove this number from the variable remaining to track the number of unprocessed or unset pixels remained in this level. If the nChanged variable is greater than 0 that is, if we have set some pixels to 255 in the previous call, we reset the idle value to 0 as those changes might give a scope to set these pixels in the same old direction. We repeat the call for processLevel for all the directions repeatedly by incrementing idle every time and resetting it we have changes to any pixels in the previous call to processLevel. We repeat this process until there are no remaining pixels or when the idle time reaches for this level. That is if idle value reaches 8, which means that no matter the level has some unprocessed pixels but they cannot be set in any direction as the fate table is not allowing flooding in their case as they will merge two seeds. This case only arises in the lower levels.

Then, we are done flooding the current level. We will have changed most of the pixels to 255 and we might even have some unprocessed pixels. All the unprocessed do not need to be the merging pixels. In some cases, if we are in lower levels we cannot flood few pixels in that particular level. The reason is that they might be falling near the seed border or at the border of the thresholded area and they might have a chance to get set in lower levels as the number of neighboring pixels set increase the fate of the pixel might change in lower levels. So, before we move onto the next level, we find what the next level is to add these pixels to that level before starting the process for that level.

```
if (remaining>0 && level>1) {    // if we have any unprocessed pixels in  the current level
     int nextLevel = level;      // find what level is coming up next
    do
    nextLevel--;    //go to next exisitng level
    while (nextLevel>1 && histogram[nextLevel]==0);
```

Further, we only keep those pixels for further processing who might have a chance to get set in the lower levels. That is, we only add those pixels who have black pixels as neighbors as these have a chance to be disregarded by fate table in second lower levels and their fate might change due to changes in some other pixels in their level. The pixels which are at the border of the sees or at the border of the thresholded area will have black pixels as neighbors and they must be set to white. So, we do this by finding the end of next level and append these pixels in coordinates array. The end of next level can be found by adding the number of elements in the next level to the beginning offset of the next level.

```
newNextLevelEnd = levelStart[nextLevel] + histogram[nextLevel];
```

In the processLevel function, we have overwritten all the coordinates of the unprocessed pixels of the current level in the beginning offset of the current level in coordinates array. So we just start checking the unprocessed pixels at the beginning offset of the current level in coordinates array and add them after the next level.

```
for (int i=0, p=levelStart[level]; i<remaining; i++, p++) {//start checking unprocessed pixels
        if(pixels[pOffset+dirOffset[d]]==0) //neighbor is black
            { coordinates[newNextLevelEnd++] = coordinates[p]; }
```

In Android:

```
    if ( ip[(pOffset+dirOffset[d])%width][(pOffset+dirOffset[d])/width]==0)
            { coordinates[newNextLevelEnd++] = coordinates[p]; }
```

Now, we need to add this number to the number of pixels in the next level. We can do this by incrementing the histogram value of next level.

```
        histogram[nextLevel] = newNextLevelEnd - levelStart[nextLevel];
```

Now, we have finished processing one level and flooded the pixels in this level with white and added the unprocessed pixels to the next level for further processing. In the next iteration, we process the next lower level and repeat flooding as explained. We repeat this until we have processed all the levels.

By the end of this process of Watershed segmentation, we will have all the pixels belonging to the seed as White and all the unprocessed pixels left alone. We now finish up the process by setting all the pixels with values lesser than 255 are reset to Background (0 = Black). This is done by the watershedPostProcess function.

After this process, we completed the initial segmentation over our input image. We followed the traditional approach by calculating the Euclidean Distance Map, determined the local maxima which represent the markers and finally flooded the seed particles and constructed dams (watershed lines) between the touching seeds by not flooding the into merging pixels and ultimately setting them with black. We then count the number of seeds in this segmented binary image. We do this by using a function provided by OpenCV named findContours which retrieves outlines of all the objects present in the image. Each outlined object is known as a contour. We store all these contours into a List and we can get the number of seeds present in the image by determining the size of this list. Further, we can draw each of the detected contours by using another function named drawContours which draws each of the contours from the List of MatOfPoint objects. The traditional algorithm is very efficient and works fine for most of the seeds. But for very small seeds, such as wheat seeds, if the input image consists of several seeds touching each other, the segmentation is not up the mark. Consider the following input image. Its corresponding binary image looks as shown in the right. As we can see, as the seeds are spread out randomly, several seeds find their place near each other and in the binary image, we see them overlapping each other clearly. To separate them, when we perform the initial segmentation. After the Watershed segmentation, most of the seeds have been successfully separated from the background and also from each other.

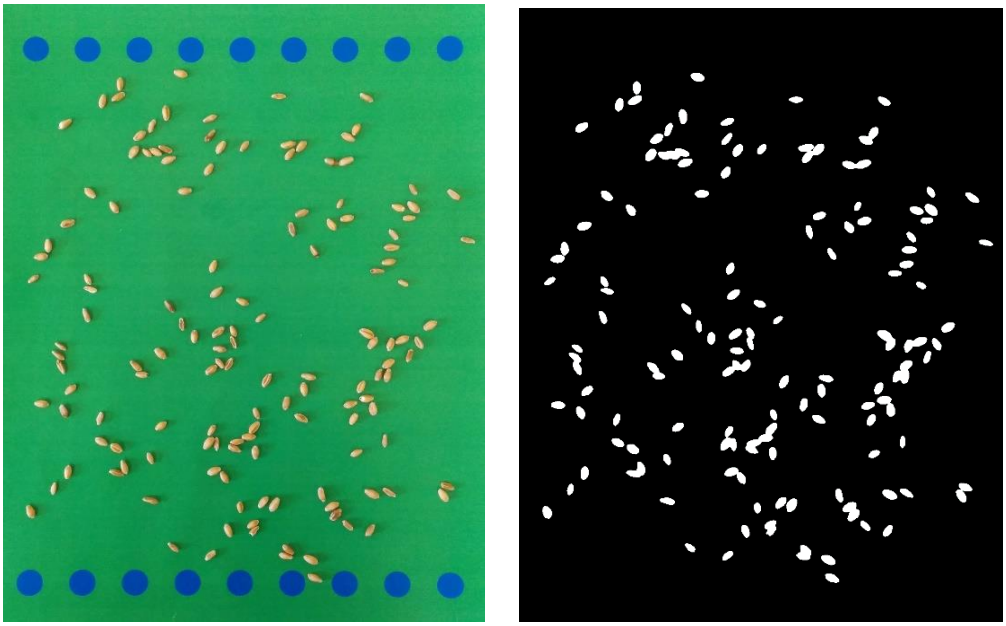

Figure 5-51 Input image and its binary

The segmented image and the corresponding contours are shown below

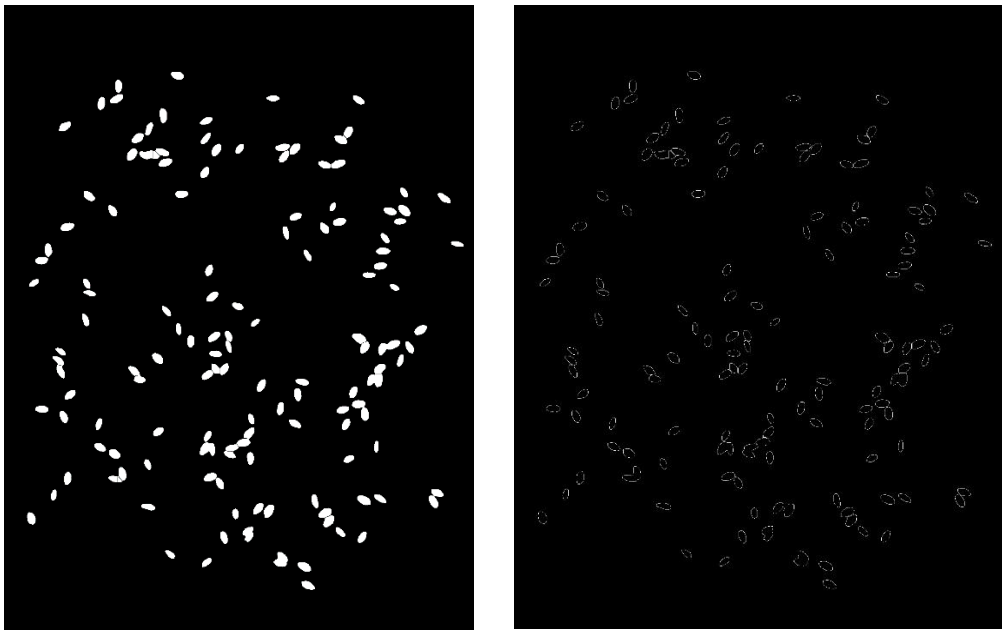

Figure 5-52 Segmented image and the contours present

The actual number of seeds in the image is 140 and the number of contours present in the image are 134. This appears as a very good result and we can state that the algorithm is segmenting

perfectly with an error rate of just 4.2%. However, if we look closer few seeds are not separated in the segmentation process which result in the formation of single contour representing multiple seeds. Further, some seeds are split into half and this is the reason for having the seed count closer to the expected value. The reason behind is the inaccurate determination of UEPs for touching seeds. Further, the traditional algorithm has pre-defined threshold values for marking the maximum areas around the maximum point in the seed. These are the reasons for inaccurate segmentation. Further, if the seeds are much smaller and if the number of seeds in the input image are increase, the distance between the seeds increases which might lead to many such cases where the segmentation might not give the accurate characterization of seeds. Thus, we need to optimize the algorithm based on the contours generated. We have to check all the contours and compare them with the average seed size to know which one represents more than a single seed. We then have to divide all these contours in the initial binary image and then perform Watershed segmentation over this image to accurately characterize and count the number of seeds present. In the next chapter, we deal with the extension of the traditional Watershed segmentation algorithm.

Chapter 6 - Extension to Watershed algorithm

Though the Watershed segmentation is very effective in separating the object in the image, in some cases we will not have the most accurate characterization. One such case is segmenting image with large number of touching objects. When we perform this segmentation on images with seeds such as Beans or Canola roots or Potatoes, the segmentation is accurate. However, smaller seeds such as Wheat and Canola cannot be segmented with 100% accuracy. As we have seen earlier, after the process of segmentation, we had groups of seeds combined and accounted as a single contour. We had two cases where the segmentation was proved to be inefficient. In the first case, a single seed is split into two seeds. In the second case, multiple seeds which are touching each other are combined to form a single seed. Let us have a closer look at these two cases. Consider the below image with some of the contours of the previously analyzed image.

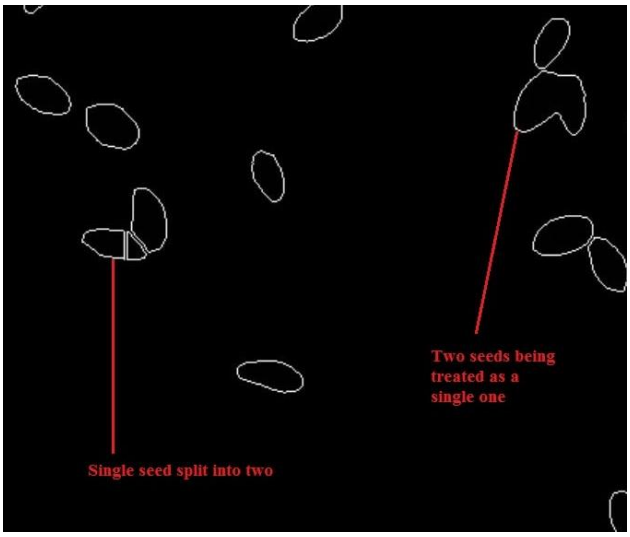

Figure 6-1 A part of image contours with two cases

6.1 Adaptive thresholding to optimize the determination of UEPs

Let us analyze the first case where a single seed is split into two. This is because of the tolerance value we use to compare while adding similar maximum points around a local maxima

to MAX_AREA. In our traditional Watershed algorithm, the threshold is pre-defined and by default it is considered to be around 0.5. In our case, by having a smaller threshold value, a seed can have two initial maxima at two points and during the process of analyzing and marking the maxima, they cannot be merged into a single local maximum area for that seed as there might be few points between the two maxima which might be larger but not large enough to fall under this tolerance level. So, they will not be considered as a part of local maxima which results in having two maximum areas in a single seed. This case can be visualized in the below picture.

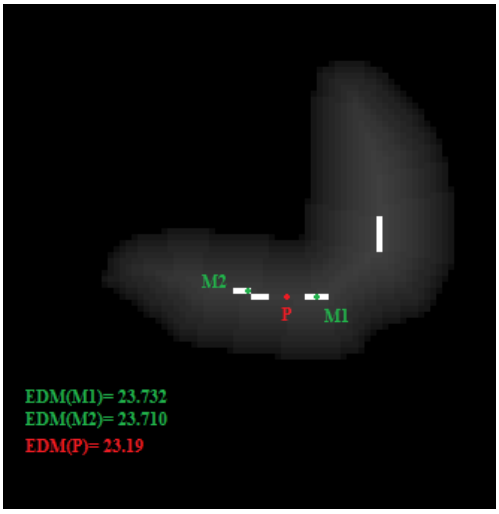

Figure 6-2 Point P outside the tolerance between two local maxima

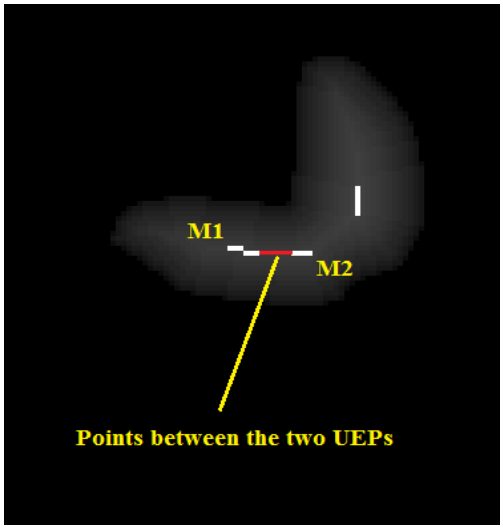

Figure 6-3 All such points between the two maxima

In the bottom seed, all the points marked in red represent all the points between the two maximum points M1 (EDM = 23.732) and M2 (EDM = 23.71). These points should ideally be considered as a part of maximum area in the seed. Even though their EDM values are slightly lesser than the two maximum points, they are not considered as similar maximum points due to the pre-defined threshold value. Consider the point P in the middle which is located in between M1 and M2. Though its EDM is closer to M1 and M2, it is not considered in either of their maximum areas as the threshold is very small (0.5) and its EDM is lesser then M1-threshold and M2- threshold. Only very few points will fall under this tolerance value of the Maxima. For the same reason, all the points marked in red are not considered as a part of UEPs. The resultant segmentation based on these UEPs will split the seed as the points in between two UEPs will not be flooded and all the unset pixels will be set to Black later. The resultant segmentation is shown in the below figure.

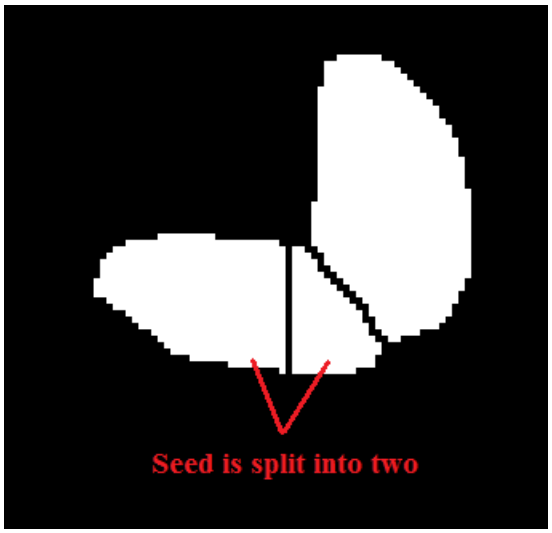

Figure 6-4 Segmented image with two local maxima in a seed

To handle this, we just increase the threshold value to be around 0.8 to 1.2. If the threshold is 0.8, then the EDM value of the P will be greater than the tolerance value (22.91) and will be included into the maximum area and will become a UEP. Further, all points below this tolerance will be

included under the maximum area as show in the below picture. Also, the corresponding segmentation is shown below.

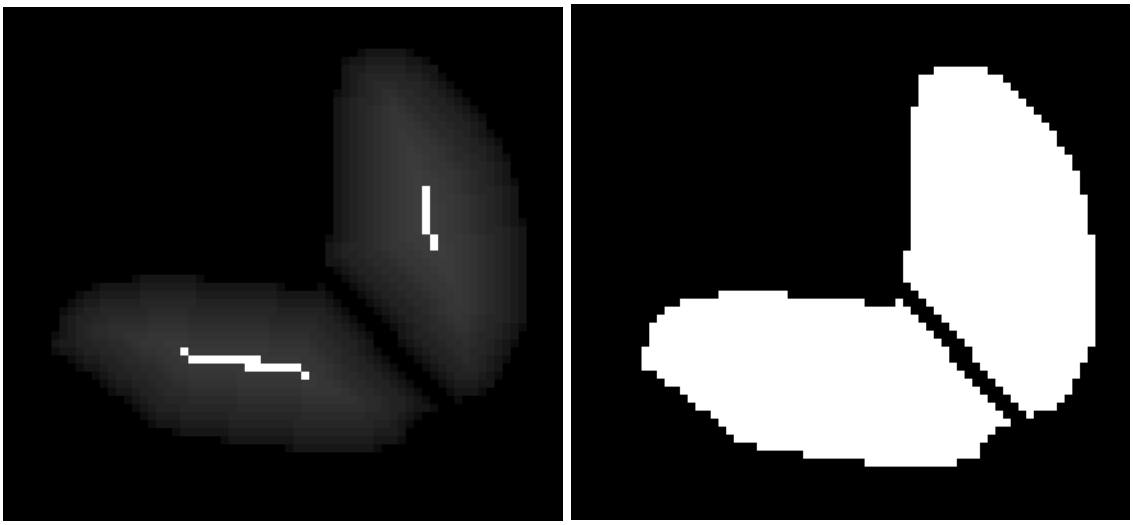

Figure 6-5 Seed with new UEPs and corresponding segmentation

Thus, the augmentation of the threshold makes the determination of UEPs accurate and improvises the segmentation of the traditional algorithm and avoids repeated segmentation. Further, this threshold varies depending upon the type and size of the seeds. For flexibility, the extended algorithm will take the threshold as input from the user based on this requirement. In Android, we have provided an interactive slider which allows the user to select the adaptive threshold within the range of 0.0 to 2.0. Finally, the segmentation will be performed accordingly.

6.2 Dividing large contours and re-segmentation

Let us consider the second case of merging of multiple seeds. In some cases, even if we change the threshold value, few groups of seeds will be combined and are treated as a single contour. This is also because of the incorrectly determined local Maxima which are also known as the Ultimate Eroded Points (UEPs). In the below image, we can see that the contours marked in

green show that the seeds are separated. But, the contours marked in red are actually two seeds but are representing them as one single contour. This is because of the incorrect determination of the local maxima. The process of flooding primarily depends on the position of the local maxima (UEP) of each seed. They form the markers for segmentation. We know that, we will not flood into a pixel, if its opposite pixels are flooded from two different markers. The location of the UEPs depend on the Euclidean Distance Map. In the case of few touching seeds, the points in between the two seeds will have highest the EDM values in that neighborhood and they have a likely chance to be a part of the maximum area and will considered as local maxima for those two seeds. This can be visualized below.

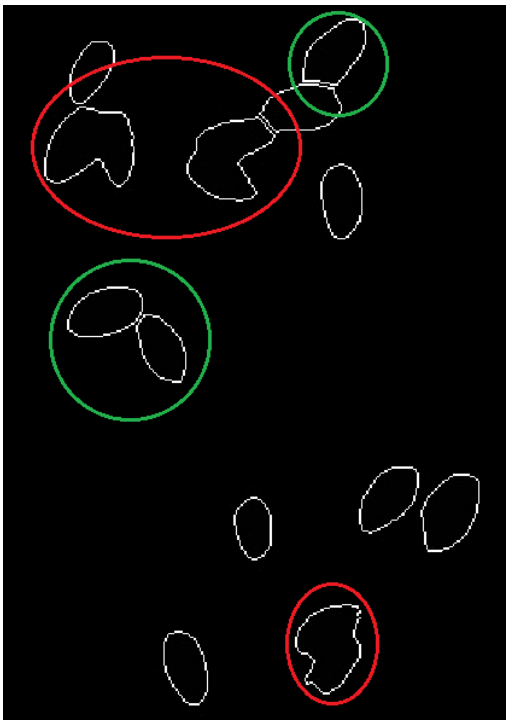

Figure 6-6 Multiple seeds merged together

For instance, the optimum location of UEPS for the above seeds should be as shown in the figure. The points which fall under the marked areas should ideally represent the local maxima or the UEPs for these seeds.

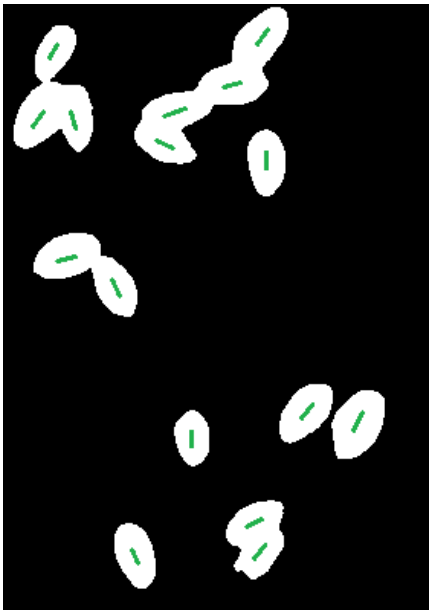

Figure 6-7 Anticipated UEPs

But, as stated above for few cases the UEPs might fall just between the two sides which is what happened in the above case. We can see the UEPs in the below image. In most of the cases, even for touching seeds, the UEPs are determined accurately.

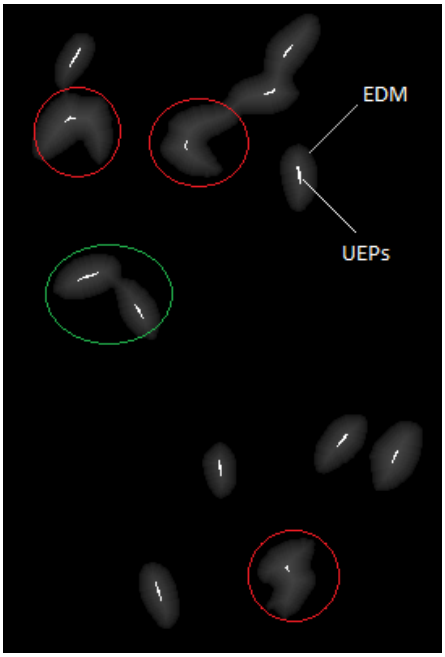

Figure 6-8 Seeds with incorrectly determined UEPs determined

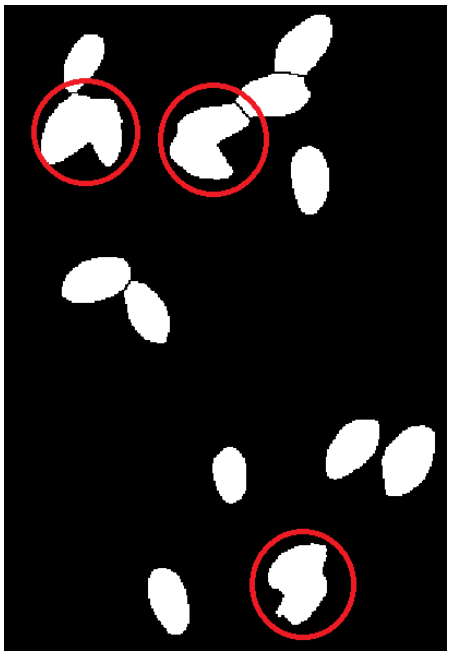

Figure 6-9 Incorrect segmentation

However, in few cases, the points in the middle are selected as UEPs for few neighboring seeds. They are marked in red. In such cases, these points are in maximum area and are set to white even before the process of segmentation. Then we start flooding from this point and flood into both the seeds finally merging them as a single seed. The corresponding segmented image can be seen above. Consider another case where 3 seeds are considered as a single contour. The original picture is shown in the first picture. Their EDM and UEPs are shown in the lower left figure. The segmented image is shown in the lower middle picture and their corresponding contours are shown in the lower right picture.

Figure 6-10 Input image with 4 seeds highlighted

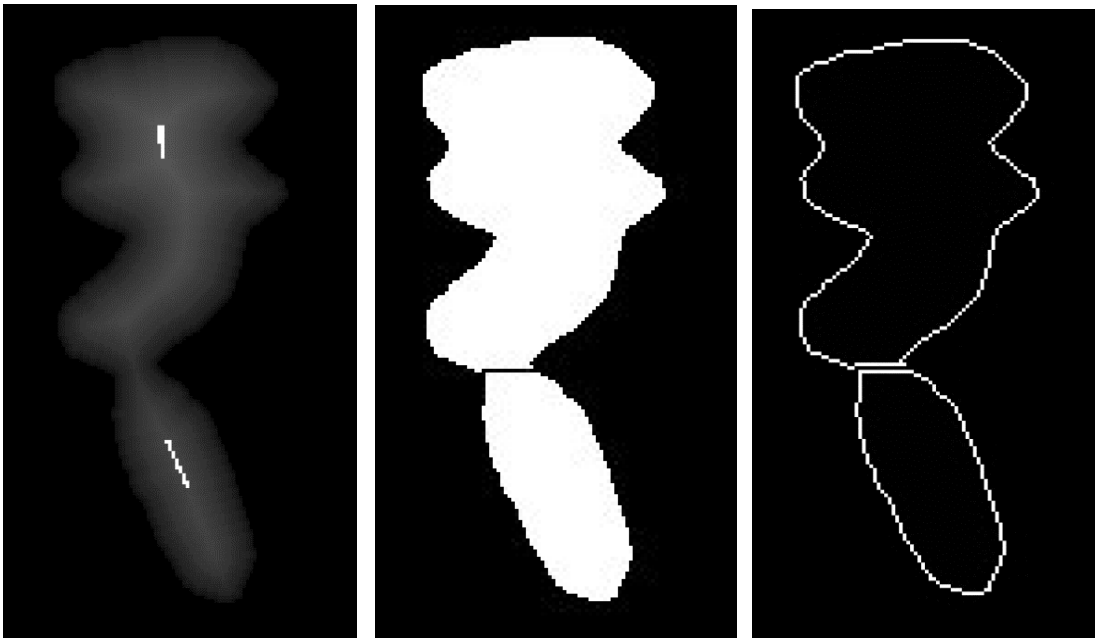

Figure 6-11 Incorrectly determined UPEs, corresponding segmentation and contours

In order to divide each of these large contours as individual seeds, we have to divide them based on their area. We need to compute area of each of the contours and we have to compare the area of each of the contours with a median which represents the average seed size in the image. As we have seen earlier, we can find the contours using Imgproc's findContours() method. Each contour is stored in a MatOfPoint object. Further, each contour is stored as a vector of points and represents the coordinates of all of the boundary points in that particular contour. A MatOfPoint is a Matrix of Points and is used to store Point coordinates in the form of a matrix. So, we need to send a List of MatOfPoint objects the findContours() function and all the border points of each of the contours will be stored in each of these MatOfPoint objects in the List. For each of the contours represented in this list, we determine the area of each contour using the function contourArea(Mat contour) of the Imgproc class. This function uses the Green formula to compute the area occupied by the given contour. Before doing this, we have to change the segmented image into a single channel image by storing it into a single channel Matrix and then change it to Grayscale image. Then, we need to find the contours as follows

```
Imgproc.findContours(mat5gray, contours, hierarchy, Imgproc.RETR_LIST,
Imgproc.CHAIN_APPROX_SIMPLE);
```

Here 'contours' is an ArrayList of MatOfPoint objects, hierarchy is a temporary Matrix, CV_RETR_LIST is used to retrieve all the contours in the image without forming any hierarchical relationships and CV_CHAIN_APPROX_SIMPLE compresses all the vertical, horizontal and diagonal segments and stores only their end points. For instance, consider the following contour shown in the below picture. As stated that the findContours stores all the boundary points of a contour, it compresses the straight lines by just storing their end points but not all the points lying on that line. So here, for the continuous horizontal segment in the seed, it just stores its end points

say (512, 1644) and (543,1644). Thus, while drawing the contours, it will join each of these boundary points. The resultant of this drawing is a straight line between these two end points. That's the reason why most of the contours generated consist of straight lines even though the seed doesn't have the perfect horizontal shape.

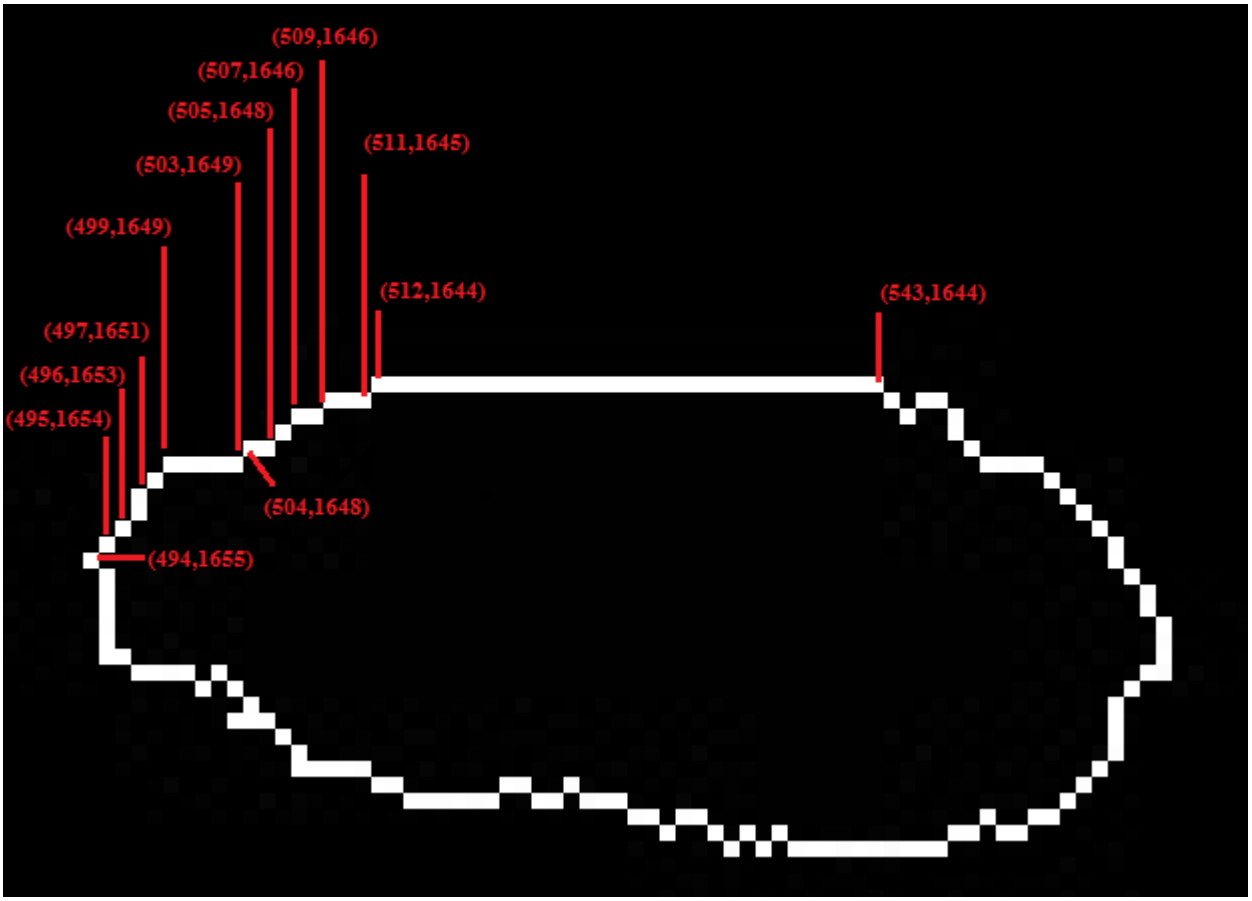

Figure 6-12 Boundary points of a contour

Now that we have all the contours stored in our list named contours, we can now iterate over each of the contours in the list and sum up their areas.

```
double area = Imgproc.contourArea(contours.get(i));
total_area += area;
```

We now find the average seed size in the image by diving the sum of all existing contours by the number of contours (size of the list contours). This average might not represent the accurate seed size and cannot be used as the exact mean of seed sizes. For instance, if there several groups of seeds merged as single units, the average seed size will be relatively larger than normal seed size as we have computed the average size by considering such large contours. For the input image in the left, the corresponding contours are shown in the right.

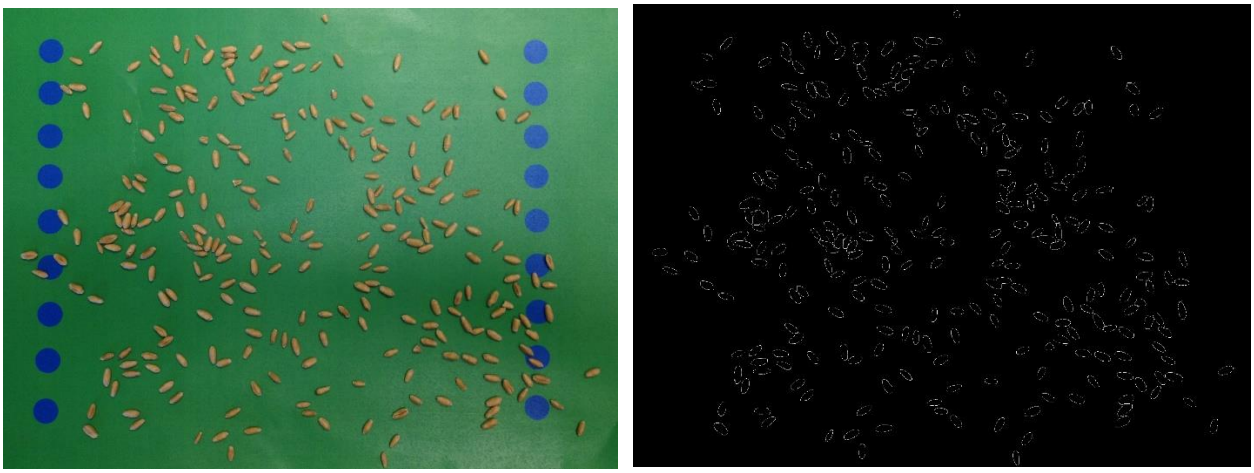

Figure 6-13 Input image and corresponding contours

In this image, the actual number of seeds are 267. The number of contours determined initially are 258. The original average area is computed to be 1360.54. Let us check the areas of some single seeds in the image. The largest contour area is found to be 4106.0 and is represented as follows:
Now, we have to compute total area by considering all the contours which represent single seeds. Any contour which is more than 1.5 times this average can be considered as more than one seed. We filter the single seed contours by checking if their area is lesser than 1.5 times the previously computed average area. For all such contours, we sum up their areas. We also track the number of single contours which are lesser than the value. We can now use this sum and count to get the accurate average seed size. In the above example, the new average area is found to be 1313.2599.

The median value is 1.5 times the average value that is the mean value here is 1969.8898. For our input image, some of the single contours and their corresponding areas are shown below

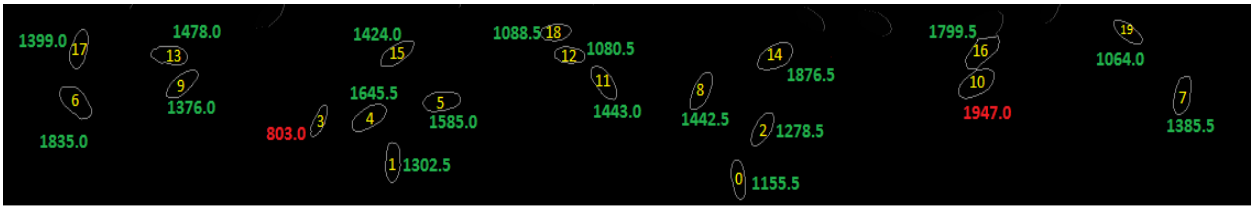

Figure 6-14 Areas of some of the contours representing single seeds

Here, the smallest contour's area is found to be 803.0 and the largest contour's area is 1947.0. Let us consider all the contours which contain more than one seed. Their areas are shown below.

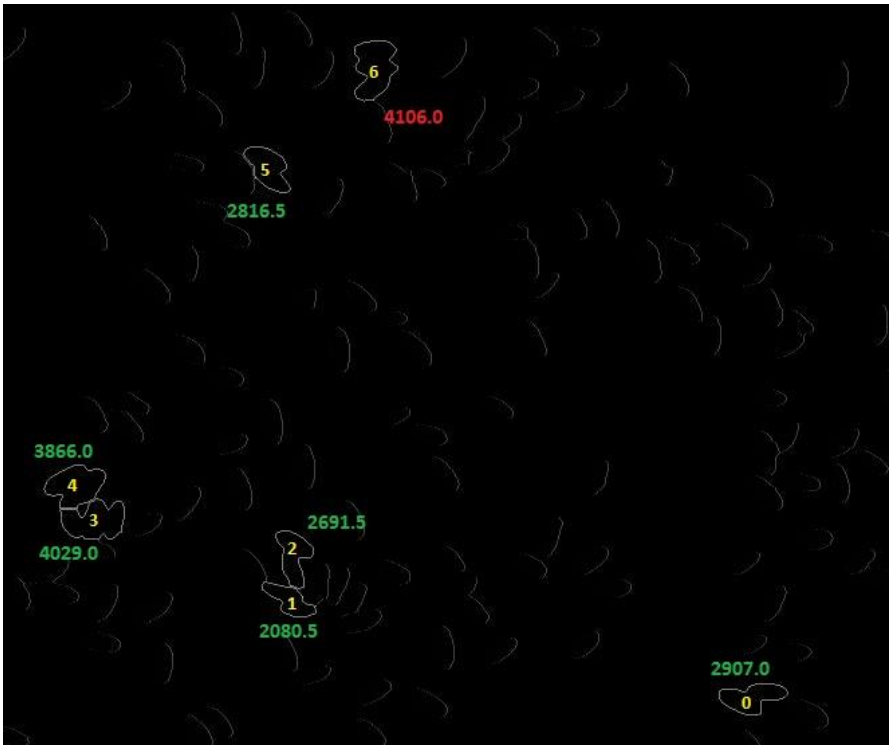

Figure 6-15 Areas of contours representing multiple seeds

Here, the smallest contour has 2 seeds and its area is 2080.5 and the largest contour has 3 seeds and its area is 4106.0. If we have considered the initially computed average (1360.54), the mean value becomes 2040.81. There are cases where two small seeds are merged as a single contour and

its area will be around 1980 to 2030. Then such contours cannot be filtered using the old mean and will be missed. Thus, the new average 1313.26 will give us the mean of 1969.9 filters the contours accurately. Also, the largest contour actually represents 3 seeds. If we divide its rea 4106 by the new average value (1313), we will have 3.127 which is approximately representing 3 average seeds in that area. This mean is named as area_threshold. Further, we can also filter all the contours having more than two seeds by if their area is greater than 2.5 times the average value. This mean is named as area_threshold2. Further, there is a need to eliminate small particles in the image which might affect the count. The smallest seeds have the area around 500 to 600. Thus, we maintain a mean which is $1/10^{th}$ of the average area (371.51) and it is named a s small_size. So, all the contours which are lesser than this small_size will be eliminated. Further, the larger contours must be separated to represent individual seeds and then the segmentation must be performed based on these divided seeds. Let us consider the largest contour and analyze the process of dividing it. The contour is shown below.

Figure 6-16 Process of accurate division

As we can see here, the contour represents 3 different seeds and the seeds must be divided as shown above. As we can see, the lines are passing from the edges of the seeds. So, we need to find the edges of the seeds and then join the corresponding edges. In order to find the edges of the seeds, we find those points where the contour is changing from convex to concave. In simple terms, a convex can be defined as something which is curved outwards and a concave can be defined as something which is curved inwards. This can be visualized as follows.

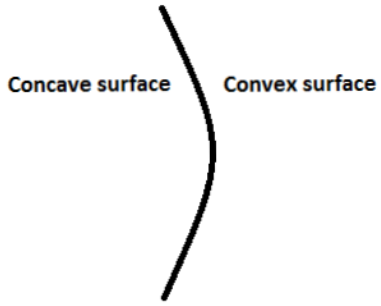

Figure 6-17 Representation of Concave and Convex surfaced

6.2.1 Determination of inflection points

In the given contour, the contour is turning into a concave surface in four regions as represented. We need to identify all the points where the contour is changing from convex to concave. These points are called as inflection points. We then have to join the corresponding edges of the seeds based on these points. To find the inflection points in a contour, for each and every boundary point in the contour, we have to find the vectors corresponding to the pair of edges on each side of the point. Vectors indicate the direction.

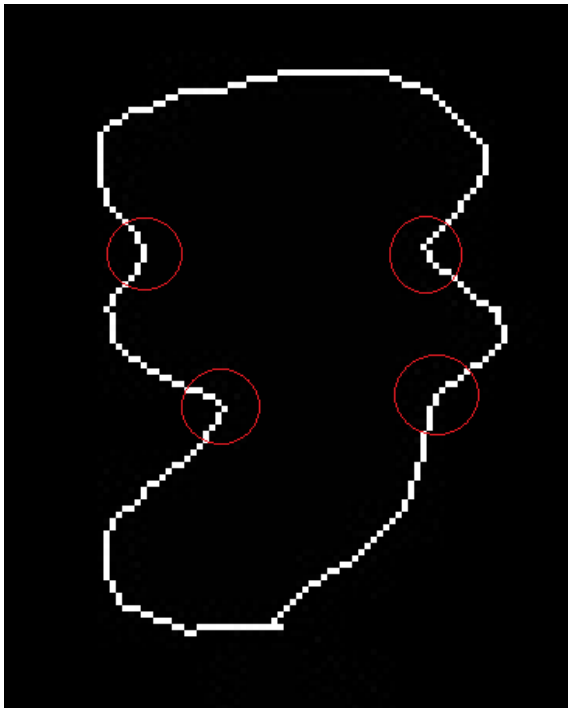

Figure 6-18 Inflection points in the given contour

Consider the following image with three boundary points A, B and C. The corresponding vectors for the edges BA, AC and CB are represented by v1, v2 and v3. Then, we have to compute the cross product of these two vectors.

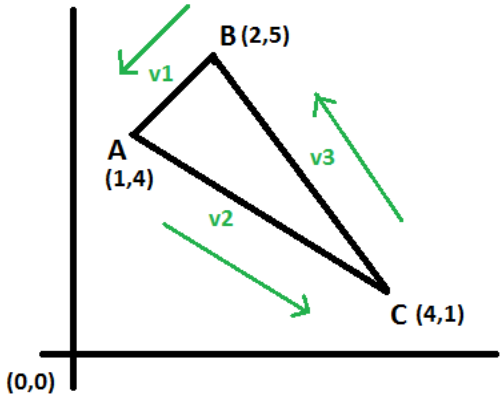

Figure 6-19 Vectors along a convex surface

In this direction, the corresponding vector v1 for the edge BA will be v1 = (1-2, 4-5) = (-1, -1). If the direction is reversed, vector v1 will represent the edge BA and will be (2-1, 5-4) = (1, 1). Thus, the vector of an edge depends on a direction. Cross product of two vectors will determine the

magnitude of the two edges. That is, the cross product of two vectors will determine the difference in slopes of the two lines in that particular direction. As we are calculating cross products for adjacent edges for a boundary point, the cross product of vectors corresponding to these two edges will give the difference in their slopes. For two vector points v1 (a, b) and v2 (c, d), the cross product can be computed using the matrix notation ad – bc. For a boundary point A (x[i], y[i]) which is reached from a point C (x[i-1], y[i-1]) and is going towards the point B (x[i+1], y[i+1]), the cross product can be computed as follows

Vector for the edge CA (Direction is from C to A), V1 = (x[i]-x[i-1], y[i]-y[i-1]

Vector for the edge AB (Direction is from A to B), V2 = (x[i+1]-x[i], y[i+1]-y[i])

The cross product of the two vectors V1 and V2 for the boundary point A (x[i], y[i]) can be written as follows:

cross_prod = ((x[i]-x[i-1]) * (y[i+1]-y[i]) – (y[i]-y[i-1]) * (x[i+1]-x[i]))

For a convex object, all the cross products will either be positive or negative depending on the direction we are checking. In the above picture, the Triangle given is a convex object. The cross products for all the points with the given vectors in the corresponding direction will be:

cross_prod_A= (1-2)*(1-4) – (4-5)*(4-1) = 3+3 = +6

cross_prod_B= (2-(-4)) = +6

cross_prod_C= (12-6) = +6

For a concave object, the cross product at the point of inflection will change in the magnitude. This can be visualized as shown in the below figure. Here, in the above figure, D is an inflection point as the surface is turning into a convex at this point. In the given direction, let us compute the cross products for all the points.

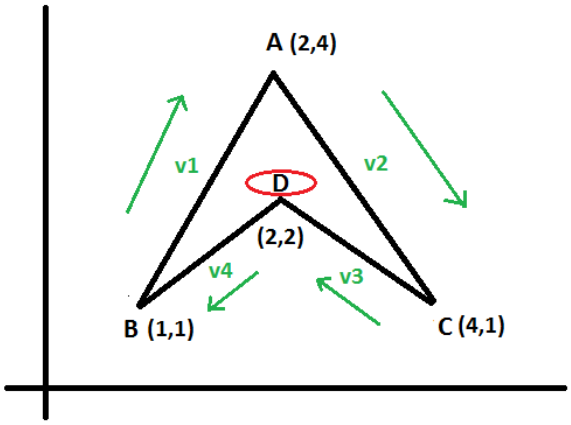

Figure 6-20 Vectors along a concave surface

cross_prod_A = (2-1)*(1-4) – (4-1)*(4-2) = -3-6 = -9

cross_prod_B = (1-2)*(4-1) – (1-2)*(2-1) = -3+1 = -2

cross_prod_C = (4-2)*(2-1) – (1-4)*(2-4) = 2-4 = -4

cross_prod_D = (2-4)*(1-2)-(2-1)*(1-2) = 2+1 = **+3**

As we can see here, at point D, the cross product has changed the magnitude which states that D is an inflection point. In this way, we can compute the cross products to determine the inflection points in our contour. Further, in the case of seeds, the boundary points will not be consistent and we may have a lot of noise in the contour points. To avoid this, for a boundary point in the contour, instead of checking the cross product for adjacent edges, we select points slight farther than the immediate neighbors and compute cross product from these edges. This increased neighboring distance is considered as Δ. So, the cross-product equation with the offset Δ would be:

$$\text{cross\_prod} = ((x[i]-x[i-\Delta]) * (y[i+\Delta]-y[i]) – (y[i]-y[i-\Delta]) * (x[i+\Delta]-x[i])) \quad (\Delta \geq 1)$$

the Δ value depends on the size of the image. Experimental results have stated that a value of 3 or 4 is giving the accurate values for the images of size 2520x1886. We determine the cross products

based on this parameter Δ and all the points with positive cross products will be considered as inflection points which might or might not be the edge points of the touching seeds.

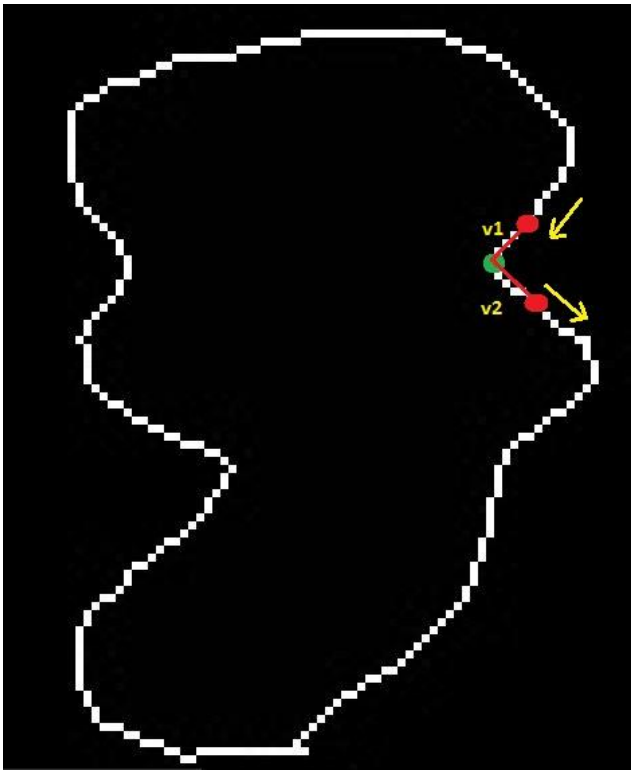

Figure 6-21 Selecting adjacent points using Δ parameter

Based on the Δ factor, the points selected for point marked in green is shown in the above figure. Further, we compute the cross product for edges corresponding to these points. All other contour boundary points will have negative cross products. As we are dealing with seeds, most of them will be irregular in shape and there may be any deviations in the seed which may result in the formation of inflection points. Thus, in order to determine the accurate edge points of the seeds, we need to consider only those inflection points where the cross product of the corresponding adjacent directions is deviating by change in magnitude and also by a significant value. Also, for a contour with two seeds, we need to have at least two inflection points which can be joined as a partition or an edge. To do this, for a contour, we store all the points with cross products greater

than 4.0 in a list and sort the list. We start with the point having maximum cross product in that contour. If this value is greater than 20.0 and the second highest inflection point's cross product is greater than 10.0 then we move forward to divide this contour. If we find more than 3 inflection points in a contour, there is a chance that we are dealing with a contour consists of more than two seeds. We analyze the inflection points and divide the contour accordingly.

6.2.2 Joining the inflection points

Now that we have a set of inflection points for a large contour, we have to join them to divide the contour into individual seeds. By joining, we mean to say that, we will join them by means of a black line over the original binary image of the input image and perform watershed segmentation again. We will explain this part in further sections. So, we have to select the optimal inflection points to join. For instance, consider the largest contour in the above example which is the most complex case. The determined inflection points and their corresponding cross products are shown in the below figure

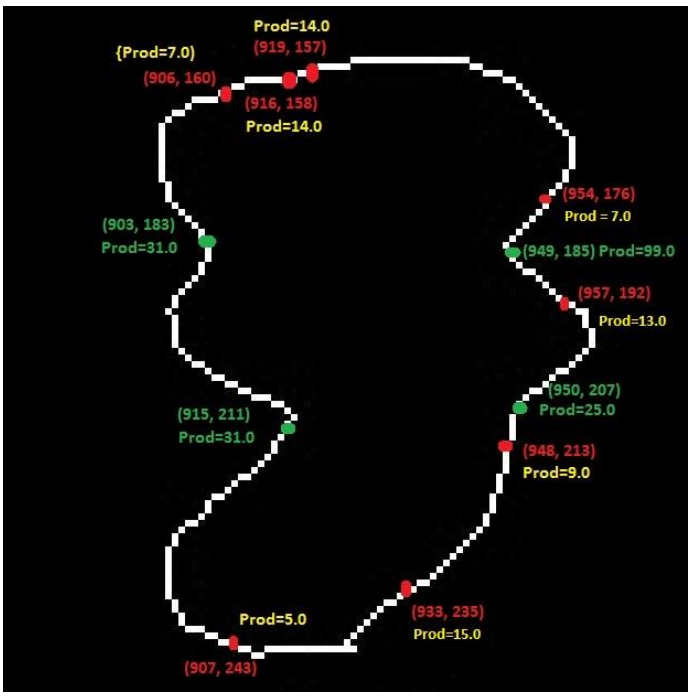

Figure 6-22 Inflection points along the surface

As we can see, all the marked points are the list of inflected points among the boundary points in this contour. We can clearly say that the points marked in green must be the primary inflection points. There are various other inflection points formed due to the irregularity in the contour's boundary and also due to the Δ factor. If we sort based on their corresponding cross products in non-increasing order, we will have these four points as the points with maximum cross products. Even after determining the maximum inflection points, we do not know which two points to join. The possible cases are as follows. The first two cases are incorrect as we are splitting the seed in an invalid manner. The third case is the valid one as the three seeds are separated accordingly. In the initial extension to the traditional algorithm, we have joined the inflection points with top two cross products initially and join the rest of the two later.

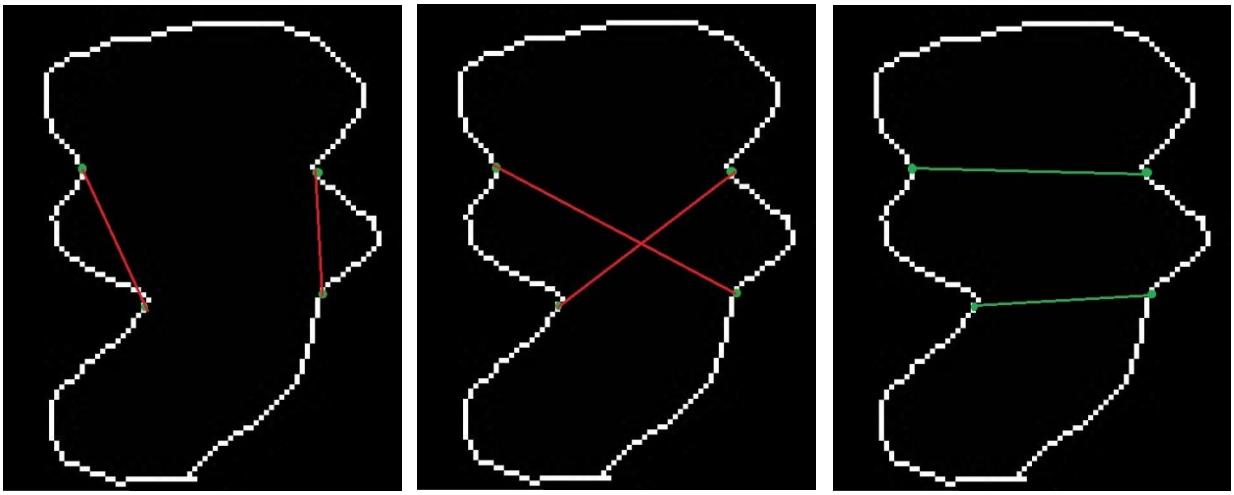

Figure 6-23 Incorrect and correct method of splitting the contour

However, this method cannot be used to divide the seed accurately in all the cases. It might be true in this case but it is not necessary for the inflection point opposite to the inflection point with maximum cross product to have the second highest cross product. We have created a custom test case to test this approach. Consider the original image given below and its corresponding contours after initial segmentation.

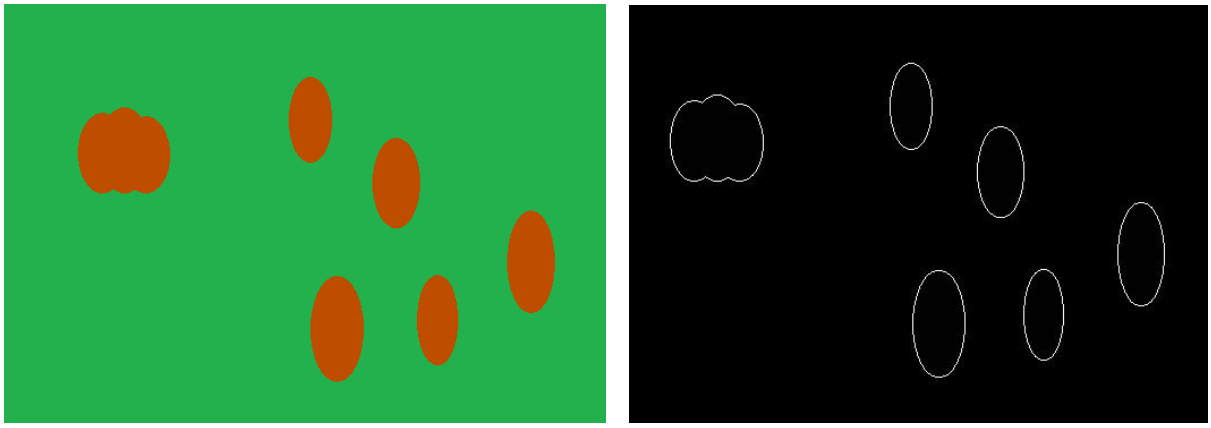

Figure 6-24 Sample image with a large contour

The largest contour has the area of 5151.0 and the inflection points are shown in the below figure. We can see that point (393, 146) has the highest cross product value and point (364, 144) has the next highest cross product. By followig the method proposed in the initial extension, we will end up joining incorrect edge points as shown in the below figure.

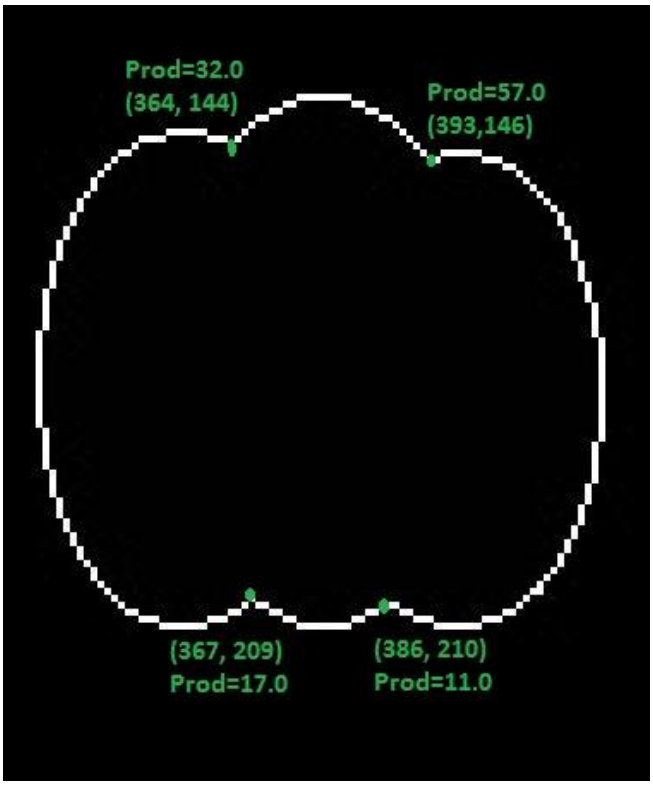

Figure 6-25 Determined inflection points in the sample image

So, these points are joined with black line in the original binary image and the image is segmented again. After the segmentation, the contours look like this.

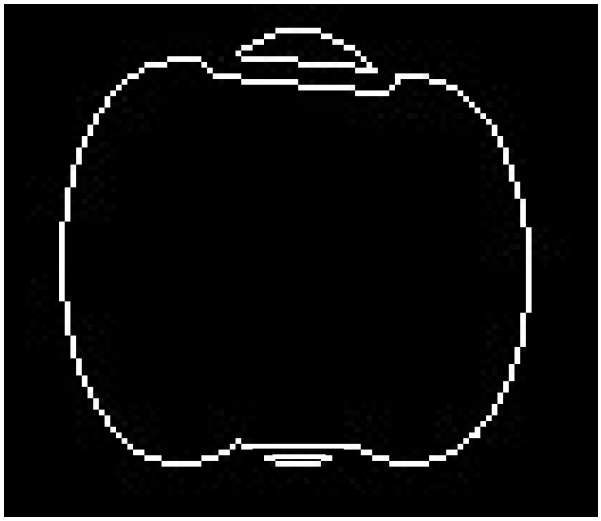

Figure 6-26 Incorrect splitting of the contour

As expected, top two inflection points are joined resulting in an incorrect division of the seeds. Even though this is a rare case, we further need to optimize the algorithm so as to join accurate inflection points to divide the seeds at their edges.

6.2.3 Determination of perpendicular distance

After sorting the determined inflection points on the boundary of the contour, we consider the point with maximum cross product. Then we find an average point of all the adjacent contours of this point. We then compute the slope of the line (c) joining the current point and the point generated by averaging the current point's adjacent points [1]. Then, we project this line and among the remaining inflection points, the point which is closer to this line (y=mx+c) will be the corresponding point of the opposite edge for the current point. Consider the following image. Here, the top four inflection points are marked.

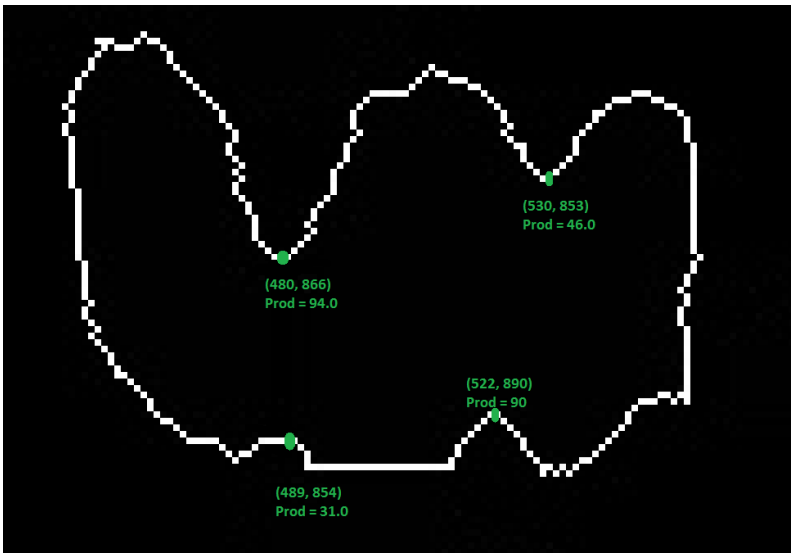

Figure 6-27 Inflection points in another large contour

We initially consider the maximum point P (x0, y0). Then, we find the average of the adjacent contours of P. The adjacent are determined based on the parameter Δ. The average point of adjacent points of P is PA (x, y). We now project the line joining P and PA. The slope of the line will be

$$m = \frac{y0 - y}{x0 - x}$$

The equation of the line will be y0=mx0+c. This is represented in the figure below.

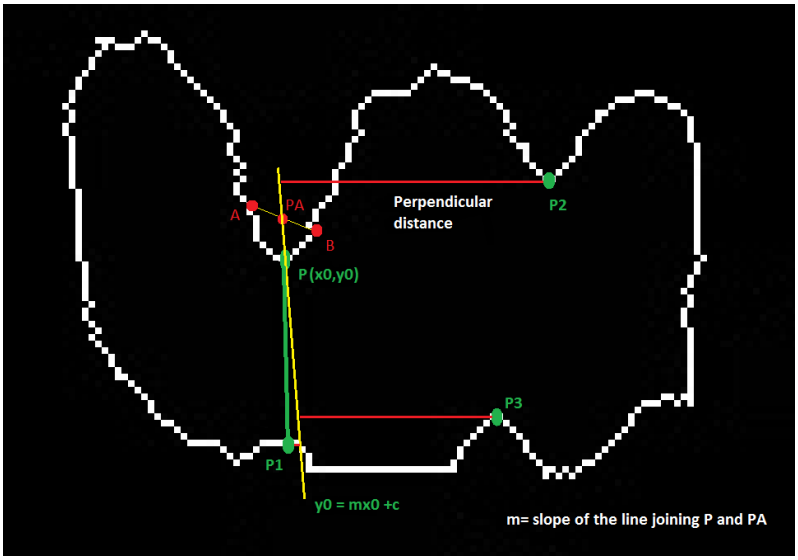

Figure 6-28 Line projection from the maximum inflection point

Now, we determine the perpendicular distance from each of the remaining inflection points to this line. The point which has the minimum distance when compared to rest of them will be considered as corresponding edge point for the current maximum inflection point. Then, these two will be remove from consideration and the same process will be continued for the rest of the inflection points.

6.2.3.1 Derivation of the perpendicular distance

Let us derive the formula for calculating the perpendicular distance from a point to a line whose slope is m and is passing from a point (x, y). Consider the below figure for a clear illustration. Here we are checking the opposite edge point for the current inflection point P (x0, y0). PA (x, y) is the average point of adjacent points of P. P1 (x1, y1) is one of the inflection points to which we need draw a line from P. K (x2, y2) is the intersection point of P1 to the line (y0=mx0+c) projected from P.

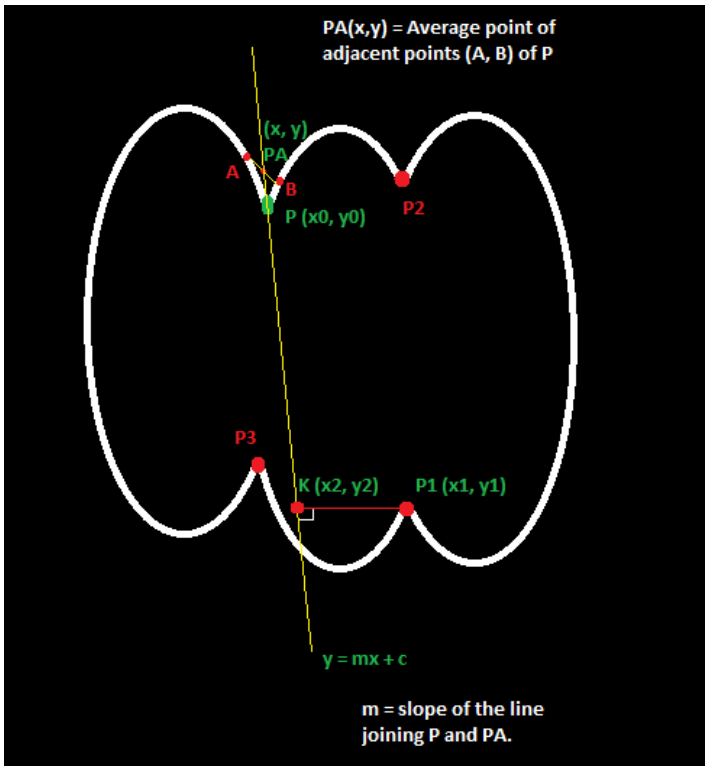

Figure 6-29 Sample image for derivation of perpendicular distance

We know that slope of the line is m. As it is passing from the current point P (x0, y0), we can get the equation

$$y0 = mx0 + c \qquad \text{- A}$$

As point K (x2, y2) is the intersection point of some inflection point P1 (x1, y1) onto the line A, we can get the following equation

$$y2 = mx2 + c \qquad \text{- B}$$

Also, as we are calculating the perpendicular distance from P1 to line A, the line represented by P1K will be perpendicular to the line projected from the point P. We know that the slope of line projected from P is m. As line P1K is perpendicular lo this line, we get:

Slope of the line P1K (say m1) = -1/m

That is, the product of slopes of two perpendicular lines will always be -1.

$$m1 * m = -1 \qquad \text{- C}$$

The slope of line joining P1 (x1, y1) and K (x2, y2) can be computed as follows:

$$m1 = \frac{y2 - y1}{x2 - x1}$$

Substituting the value of m1 in equation C, we get

$$\left(\frac{y2-y1}{x2-x1}\right) * m = -1$$

$$==> \quad y2 - y1 = \frac{x1-x2}{m}$$

$$==> \quad mx2 + c - y1 = \frac{x1-x2}{m} \qquad \text{(from eqn. B)}$$

$$==> \quad x2 = \frac{x1+my1-mc}{m^2+1}$$

$$==> \quad x2 = \frac{x1+my1-m(y0-mx0)}{m^2+1} \qquad \text{(from eqn. A)}$$

$$==> \quad x2 = \frac{x1+m^2x0+m(y1-y0)}{m^2+1}$$

We know from eqn. B that y2 = mx2 +c. Computing y2, we get

$$==> \quad y2 = m\left(\frac{x1+my1-mc}{m^2+1}\right) + c$$

$$==> \quad y2 = \frac{mx1+m^2y1-m^2c}{m^2+1} + c$$

$$==> \quad y2 = \frac{mx1+m^2y1+c}{m^2+1}$$

$$==> \quad y2 = \frac{mx1+m^2y1+(y0-mx0)}{m^2+1} \qquad \text{(from eqn. A)}$$

Now we have the coordinates of the point K (x2, y2). We can now determine the perpendicular distance from an inflection point P1 (x1, y1) to K (x2, y2) as follows:

$$\text{pDist} = \sqrt{(x2-x1)^2 + (y2-y1)^2} \qquad \text{- D}$$

Let us compute x2-x1 and y2-y1 individually.

$$x2 - x1 = \frac{x1+m^2x0+m(y1-y0)}{m^2+1} - x1$$

$$==> \quad x2 - x1 = \frac{m^2(x0-x1)+m(y1-y0)}{m^2+1} \qquad \text{- E}$$

$$y2 - y1 = \frac{mx1+m^2y1+(y0-mx0)}{m^2+1} - y1$$

$$==> \quad y2 - y1 = \frac{m(x1-x0)+(y0-y1)}{m^2+1} \qquad \text{- F}$$

By substituting the values of E and F in equation D, we get:

$$\text{pDist} = \sqrt{\left(\frac{m^2(x0-x1)+m(y1-y0)}{m^2+1}\right)^2 + \left(\frac{m(x1-x0)+(y0-y1)}{m^2+1}\right)^2}$$

$$= \frac{1}{m^2+1}\sqrt{(m^2+1)((m(x1-x0))^2 + (y1-y0)^2 + 2m(x1-x0)(y1-y0)}$$

$$= \frac{1}{\sqrt{m^2+1}}\sqrt{\left(m(x1-x0) + (y1-y0)\right)^2}$$

$$= \frac{1}{\sqrt{m^2+1}}\,(m(x1-x0) + (y1-y0))$$

As, the value of m is constant for all the inflection points, we can ignore the value of $\frac{1}{\sqrt{m^2+1}}$

Finally, we have computed the perpendicular distance in terms of the point P1 (x1, y1) which we are checking and our current point P (x0, y0) to which we need to determine the opposite edge point. So, the perpendicular distance from an inflection point (x1, y1) to the line projected from our current inflection point (x0, y0) can be computed as follows:

$$pDist = (y1-y0)+ m(x1-x0)$$

Further if the slope is infinity, this means that the line projected from our current point is parallel to y axis. In this case, perpendicular distance is nothing but the different between the x-coordinates of the current point and the point which is being checked currently. In this manner, we calculate the perpendicular distance or nearest distance for the current inflection point to every other inflection point. The point which is nearer to the slope of the projected line will be selected as the opposite edge for the current point. Further, these two points will be eliminated from the list and this process will be repeated for the remaining inflection points in the contour.

6.2.4 Implementation

As stated earlier, we have declared a threshold for large contours as 1.5 times the average seed size and stored this threshold into a variable named area_threshold. All the contours which have their area greater than this value will be considered as large contours consisting of at least 2 seeds. Further, we have another threshold called area_threshold2 which is 2.5 times the average seed size. All the contours which have area greater than this value will contain more than 3 seeds. Finally, we have a threshold named small_size which is 0.1 times the average area. All the contours below this value will be eliminated as they represent very small disturbance, outlines or noise. We have all the contours in our image stored in a list of MatOfPoint objects named contours. We now iterate through each of the contours in this list and check whether they are smaller than the small_area or greater than area_threshold. They will be eliminated if they are smaller.

```
if (area < small_size) contours.remove(i);
```

If any contour has its area greater than area_threshold, we need to determine the inflection points and then divide the contour by joining them. Consider the largest contour in the list, having the area of 4106.0 which is show below.

Figure 6-30 Largest contour in the input image

For the determination of the inflection points, we need to calculate the cross products for each of the boundary points on the contour. To do this, we get all the boundary points into a list of Point objects as follows:

```
List<Point> clist = contours.get(i).toList();
```

We start checking each boundary point and calculate its cross product in the relative direction. In each iteration over the boundary points of the current contour, the current boundary point is taken from the list and is stored into a temporary Point object.

```
Point p = clist.get(j);
```

We now need to select points slight farther than the immediate neighbors and compute cross product from these edges. As explained earlier, this neighboring distance is considered as Δ. This is 4 in our case. We get adjacent points in both the directions as follows

```
Point pup = clist.get((j+delta)%n);
```

```
Point pdwn = clist.get((j+n-delta)%n);
```

The x and y coordinates of the average point between these two points are stored into xavg and yavg. For each inflection point, we also store the slope of the line between itself and the average point. This slope will be helpful in the case of maximum inflection point.

```
slope = (p.y - yavg)/(p.x - xavg);
```

If both the point lie on a line which is parallel to Y- axis, the slope will be infinity. To avoid the divide by zero exception and to maintain the track of this slope, we store the slope as 1000000.0 for such points.

```
if (Math.abs(p.x - xavg) > 0.0001)     slope = (p.y - yavg)/(p.x - xavg);
else     slope = 1000000.0;
```

We now compute the cross product for the current boundary point using the current point stored in p and its adjacent points (pup and pdwn). The cross product is computed as follows:

```
crossProd = (p.x - pdwn.x)*(pup.y - p.y) - (p.y - pdwn.y)*(pup.x - p.x);
```

We create a new list of CPoint objects named plist to store all the boundary points who have their cross products greater than 4.0. CPoint is a custom class which is used to store the coordinates of a point along with its cross product and slope. Thus, if the cross product is greater than 4.0, we add the current point, its cross product and is slope into the plist. We repeat the same for all the boundary points on the current contour. Finally, we will sort the plist. The compareTo method of CPoint has been overridden to compare values based om cross products. So, the plist will be sorted based on the cross products of inflection points in decreasing order. The below picture shows some of the listed inflection points on the contour. We also maintain two lists slist and dlist. slist is the source list and is used to store current maximum inflection point we are

checking and dlist is the destination list which has the inflection point which is computed as the opposite edge point of the point in the corresponding index of slist.

```
List<Point> slist = new ArrayList<Point>();
List<Point> dlist = new ArrayList<Point>();
```

We initially start from the inflection point with the maximum cross product. The point at the beginning index of the plist will be the greatest one. We will move forward only if the cross product of the greatest inflection point is at least 20 and the next greatest inflection point's cross product is at least 10.

```
CPoint[] cp = new CPoint[4]; // to get the inflection point from plist
Point[] p = new Point[4]; // to get the coordinates of inflection point from cp
cp[0] = plist.get(0); //get current (largest) inflection point
p[0] = new Point(cp[0].getX(), cp[0].getY()); //get largest inflection point's coordinates
cp[1] = plist.get(1); //get next largest inflection point
p[1] = new Point(cp[1].getX(), cp[1].getY()); //get cp[1]'s coordinates
```

If not, we will not divide the contour. If yes, we further check if the contour is representing at least 3 seeds and also if the plist has more than 3 inflection points. If either of them are false, it means that the current contour represents only two seeds. As we only have two inflection points listed, we add the current point to first index of slist and the next largest inflection point to the first index of dlist. We use these lists later to join these points in the binary image.

```
slist.add(p[0]);     dlist.add(p[1])  //add the two points
```

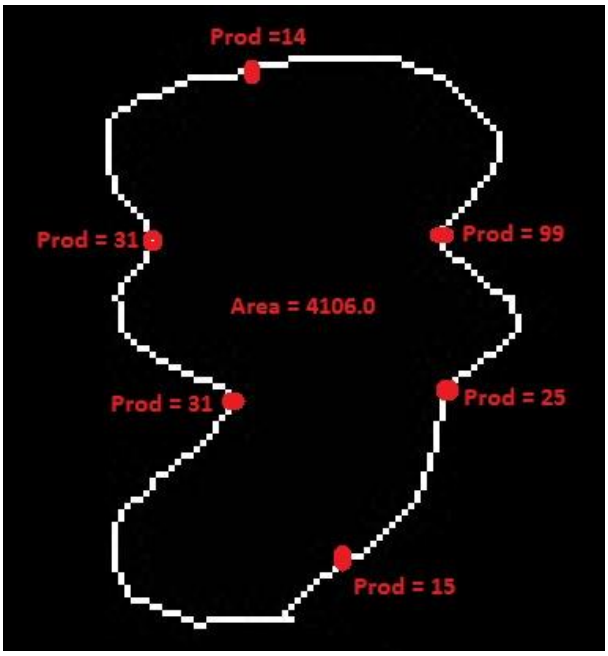

Figure 6-31Determined inflection points on the largest contour

However, if the contour is representing more than 3 seeds, we have to get the rest of the inflection points in the list. The sorted inflection points in the contour based on cross products are given here:

Sorted: 949, 185 -> 99

Sorted: 903, 183 -> 31

Sorted: 915, 211 -> 31

Sorted: 950, 207 -> 25

Sorted: 933, 235 -> 15

Sorted: 919, 157 -> 14

As explained earlier, we start with the largest point and find the slope between the line joining the current point and the average point of its adjacent points. We have already calculated this slope earlier while computing the cross product and each slope corresponding to each of the inflection points have been stored in their corresponding CPoint objects. We also have derived a formula to calculate the perpendicular distance in terms of the current point P (x0, y0) to which we need to

determine the opposite edge point and other point P1 (x1, y1) which we are checking with. It is given by:

$$pDist = (y1-y0)+ m(x1-x0)$$

We need to get each of the rest of inflection points into corresponding CPoint objects from plist and their coordinates into Point objects from Cpoint as seen earlier.

```
cp[2] = plist.get(2);

p[2] = new Point(cp[2].getX(), cp[2].getY());

cp[3] = plist.get(3);

p[3] = new Point(cp[3].getX(), cp[3].getY());
```

The projection of the line from the current inflection point P can be seen in the below image. Here, PA is the average point of all the adjacent points (A, B) for the current point. P1, P2, P3, P4 and P5 are the rest of the inflection points in the plist.

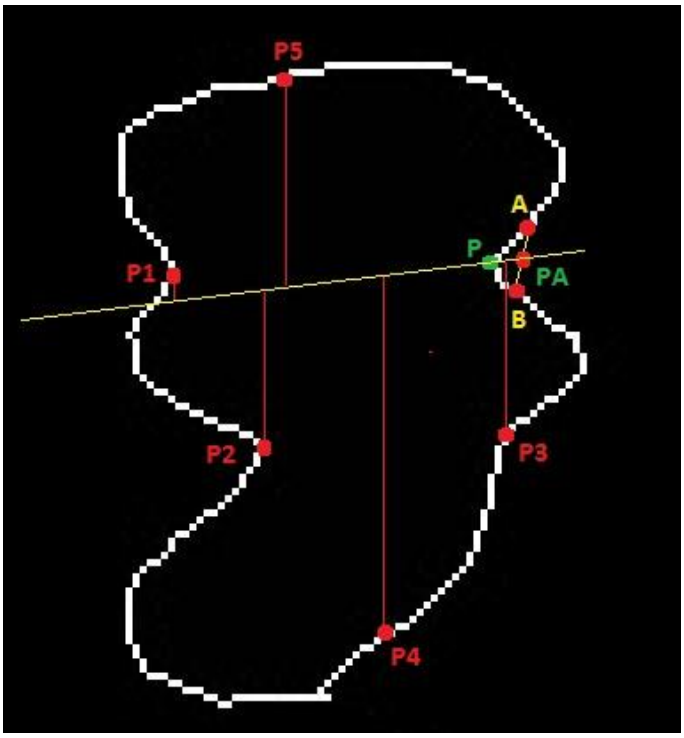

Figure 6-32 Project of line from inflection point P and determination of perpendicular distance from other points

Now, for each of the points, we calculate the perpendicular distance based on the slope value. We know that, if the slope of our current projected line is 1000000.0, we calculate the perpendicular distance from this line to all other points as the difference between the x-coordinates of other point and our current point. But, if the slope is not greater than 1000000.0, we calculate the perpendicular distance by using the above formula.

```
minError = Math.abs(p[1].y - (p[0].y + cp[0].slope*(p[1].x-p[0].x)));
```

We then compare the distances and store the minimum distance and corresponding nearest point to the projected line.

```
if (cp[0].slope >= 1000000.0) {
  minError = Math.abs(p[1].x - p[0].x); //distance between top 2 points
  minError = Math.abs(p[1].y - (p[0].y + cp[0].slope*(p[1].x-p[0].x))); //slope > 1000000
  for (int jj=2; jj<=3; jj++) { //for rest of inflection points
        double error = Math.abs(p[jj].x - p[0].x);
minError = Math.abs(p[1].y - (p[0].y + cp[0].slope*(p[1].x-p[0].x))); //slope > 1000000
            if (error < minError) { //compare the minimum distance
            minError = error; //store the minimum distance
            minIndex = jj;
 } } } //store the closest point
```

We now have the index of the closest point to the line projected from current point. We just add our current point to slist and the nearest point in dlist.

```
slist.add(p[0]); //add current point as source
dlist.add(p[minIndex]); //get the nearest point from p[minIndex]
```

In our contour, the slope of the line projected from our current point P (product=99) is found to be -0.07692307692307693. The next greatest point is P1 with the cross product of 31. Then follow P2 (product=33) and P3 (product=25). We can clearly see that P1 is very near to the projected line and programmatically the perpendicular distance from P1 to the line is found to be 5.53846. The distances from P2 and P3 are found to be 23.3846 and 22.07692 respectively. As the point P1 is nearest to the line, we have successfully found the opposite edge for the current inflection point P. We now add the current point is added to slist and P1 is added to dlist. This can be visualized in the below figure in the left. We are left with two other inflection points. The process is repeated and then P2 is added to slist and P3 is added to dlist. This can be visualized in the below figure in the right.

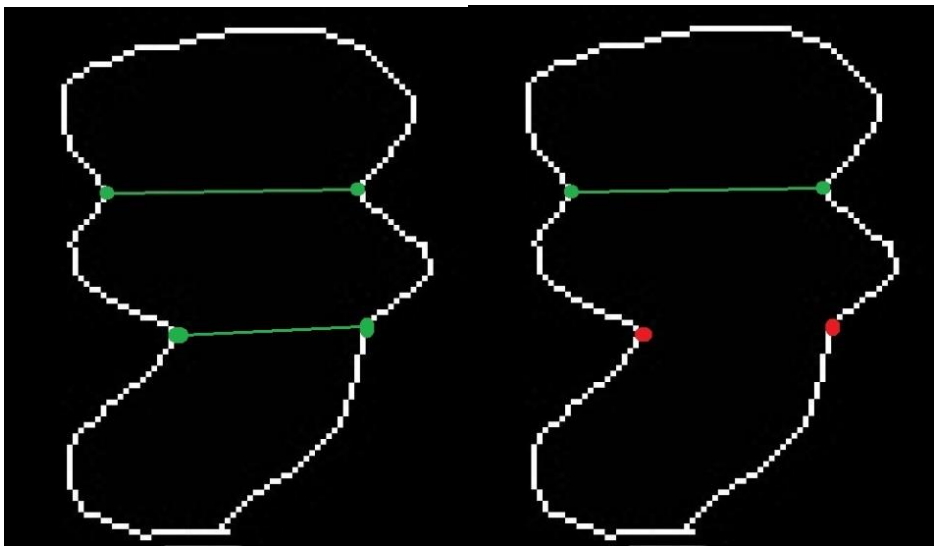

Figure 6-33 Joining correct inflection points

In this way, for all the contours, we will add the points to be joined into slist and dlist. After processing all the contours, we will join them. As explained earlier, we do not join these points in the contours. Instead, we join these points in the original binary image with black lines. This is done by getting the original binary image's file path into an ImageP

lus object and getting a reference of this object into an ImageProcessor object. Then we can draw a black line between the selected points on this image reference using the function drawLine() provided by ImageProcessor object. We already know that, all these changes will be directly updated into the ImagePlus object.

```
ImagePlus ip2 = new ImagePlus(FilePath); //get the original binary image
ImageProcessor imp2 = ip2.getProcessor(); //reference object
ip2.setColor(0) //set the color to Black
```

We then traverse over both the lists slist and dlist and get the coordinates of points and draw lines between the corresponding points.

```
Point src = slist.get(i); //source
Point dst = dlist.get(i); //destination
int x1 = src.x; int y1 = dst.x; int x2 = src.y; int y2 = dst.y; //get the end points
```

We then draw the line between the two coordinates (x1, y1) and (y1, y2)

```
ip2.drawLine(x1, y1, x2, y2); //draw the line
```

As Android doesn't support ImageProcessor object, we directly manipulate the bitmap we use the Canvas class provided by Android's graphics package which allows us to draw lines on a Bitmap object. We select the color using the object of Paint class and draw black lines joining corresponding points.

```
Canvas canvas = new Canvas(threshBitmap1); //original binary image
Paint paint = new Paint();
paint.setColor(Color.BLACK); //set color to black
canvas.drawLine(x1, y1, x2, y2, paint); //to draw the line
```

Then the Watershed segmentation is re-performed on the modified binary image by following the same old steps of determining the Euclidean Distance Map followed by the determination of the Ultimate Eroded points (local maxima) and then flooding the image from these local maxima.

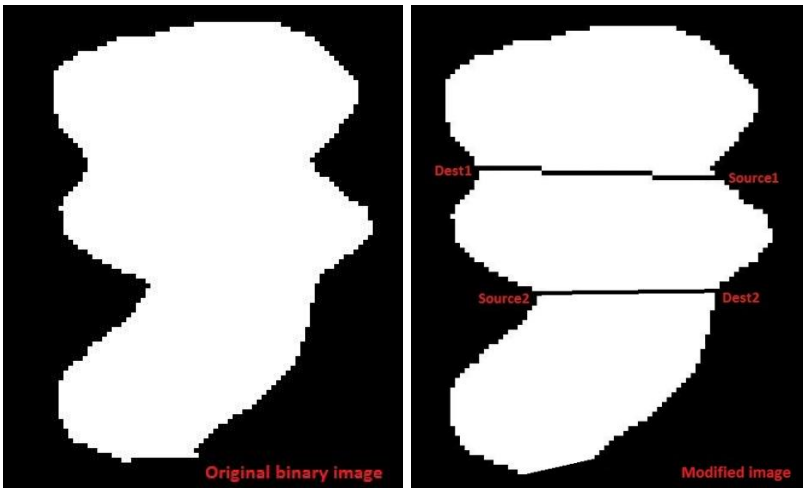

Figure 6-34 Joining the inflection points in the original binary image

Now, we again perform the Watershed segmentation on this modified binary image by determining the Euclidean Distance Map and the Ultimate Eroding Points. The left figure shows the UEPs in the contour before modification. The figure in the right shows the new UEPs in the region after the contour is divided in the binary image.

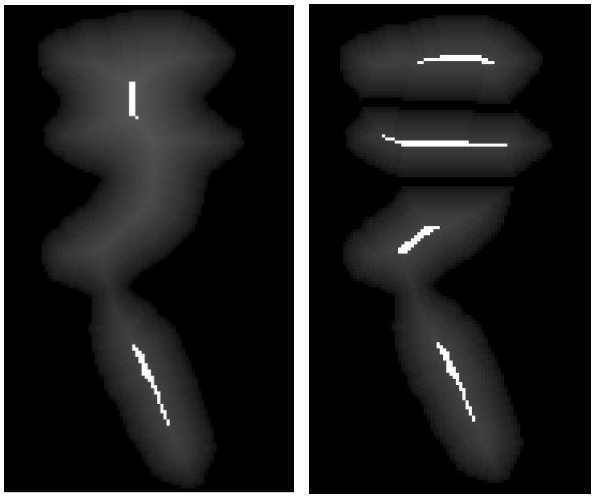

Figure 6-35 Original UEPs and new UEPs

We that we have the UEPs determined correctly. The flooding process can be visualized as follows

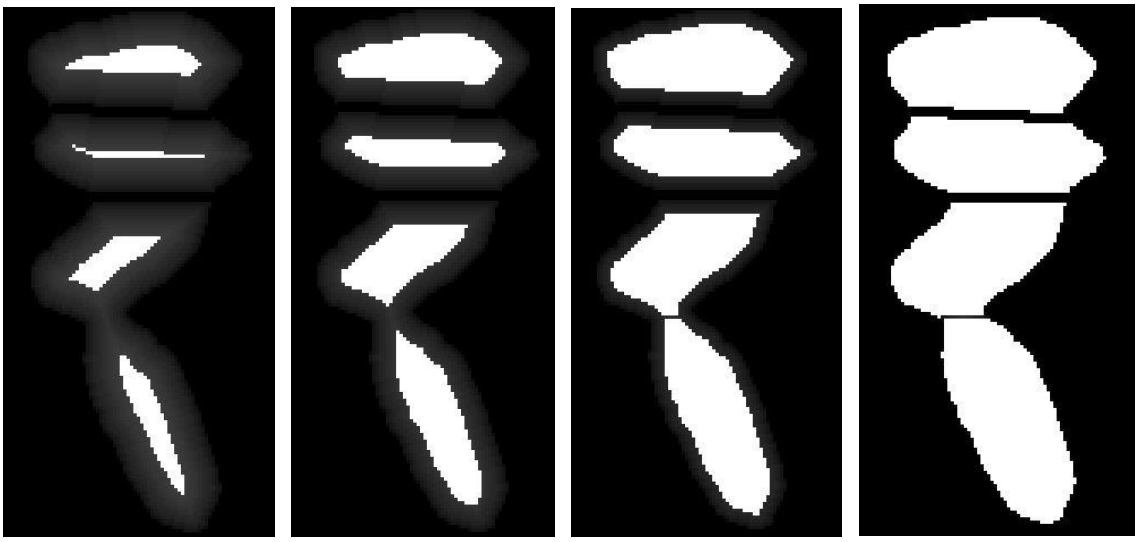

Figure 6-36 Watershed Segmentation based on new UEPs

The final representation of contours will be.

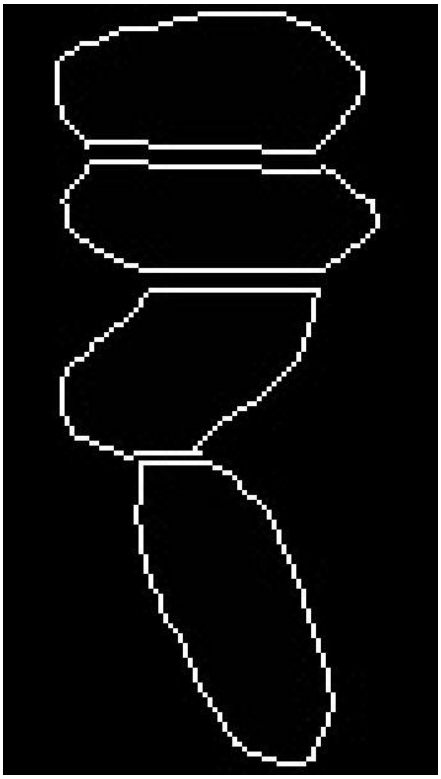

Figure 6-37 Final contours based on new UEPs

We can see that the large contour is successfully divided into 3 individual contours. Thus, we can conclude that this recursive process of counting and splitting the contours further optimizes the traditional Watershed segmentation algorithm and it will provide much better results even for wide variety of crop seeds irrespective of their sizes and shapes.

Chapter 7 - System design

7.1 State Chart diagram

Below is the State Chart diagram represtning the flow of events in the Android application. Initially, the main screen is displayed where the user can click a button to proceed to the second acreen with the navigation drawer menu. The user will then be prompted to select an image from camera or gallery. The selected image will be set to the custom image view. Then the user can click on the button to start processing.

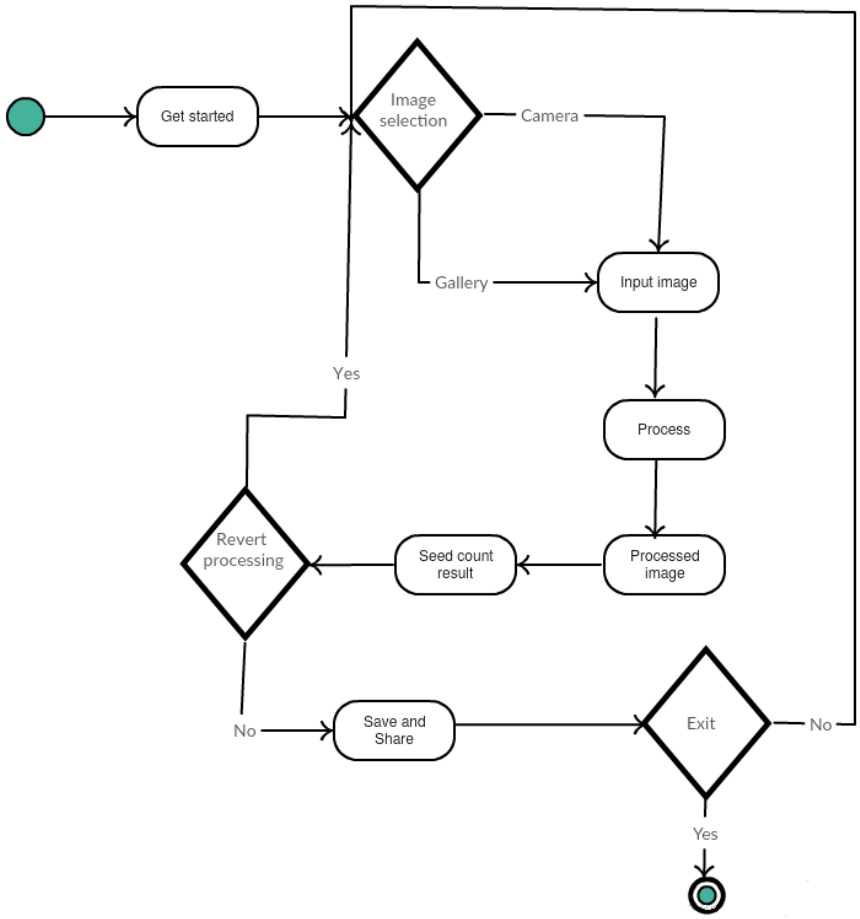

Figure 7-1 State Chart diagram

The processed image is set back to the custom image view. Then an alert dialog pops up with the seed count result in the image. The user can save and share the image. The user can further reset the image or process another image or can exit from the application.

7.2 Graphical User Interface

Before going to the design part, let us see the Graphical User Interface for the Android application. The main screen is blank with a button which directs the user to the second screen. This screen implements a navigation drawer menu. It further holds a Fragment on which a custom Image View has been built. Using this custom view, we can perform various operations such as navigate, zoom, pinch zoom on the bitmap set on the top of it. We can call this fragment whenever required. Below the fragement, on the main screen of second activity, we hold an ImageView with a Wildacat watermark and a button below it. When we call the fragment, the cutom image view wil be setup on top of the Watermark and the button can be used for processing the image. When the user initially enters the second activity, he will be prompted to select an image to be processed from Camera or Gallery. After he selects an image, the fragment will be called and the image bitmap will be held on the image view. He can then click on the button to process the image. After processing, the resultant bitmap will be set back on to the custom view along with a prompt of seed count. For significantly larger images, this may take a while. The user can revert the processing, change the threshold, clear the custom view from the options provided in the overflow menu. Further, the navigation drawer allows the user to reselect a new image, save the processed image and to share the image. Further, the user can share the saved Bitmap via E-Mail, Messaging, Slack, Facebook, WhatsApp and can even save it into his Google Drive. The screen shots of the application can be viewed below.

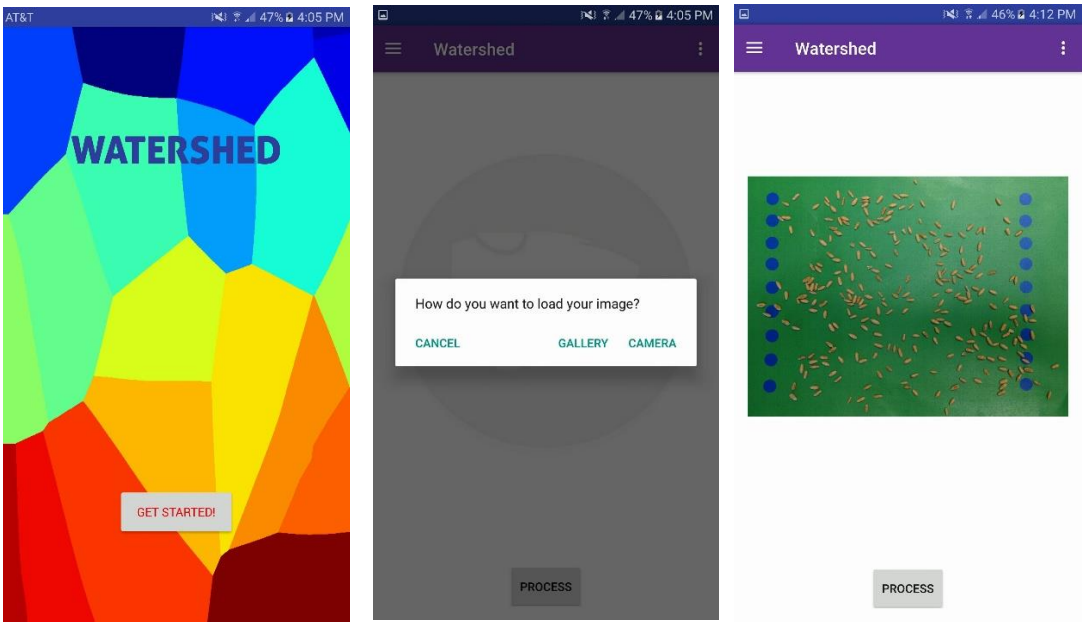

Figure 7-2 Primary screen and second screen

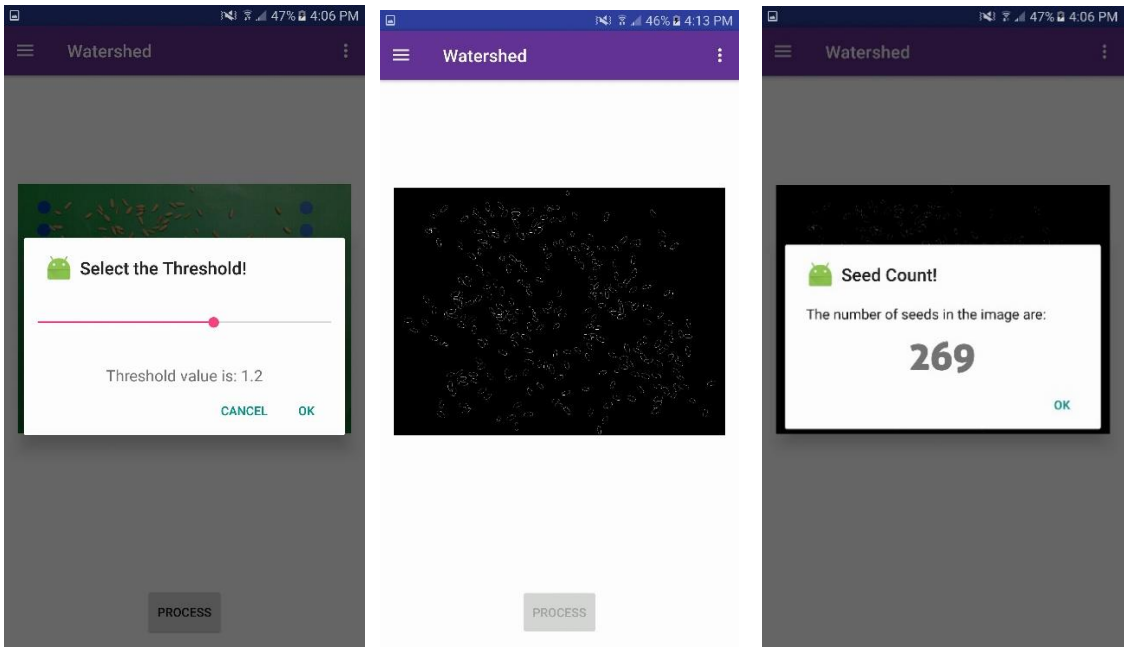

Figure 7-3 Secondary screen before processing and after processing

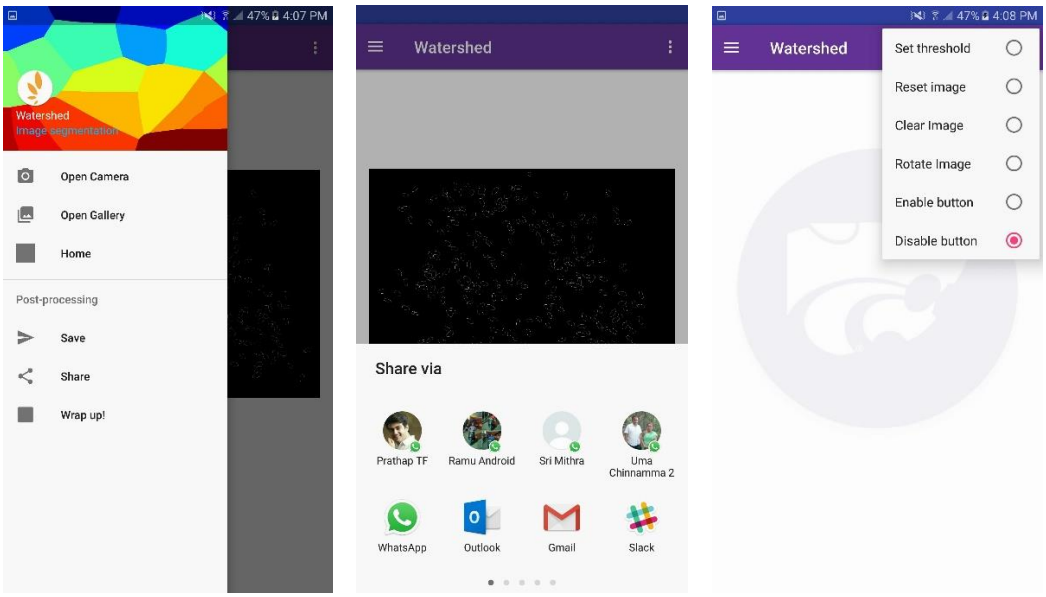

Figure 7-4 Navigation drawer and other options

Chapter 8 - Testing and comparing the results

8.1 Results

To verify the correctness of the algorithm, we have tested the algorithm extensively on a large set of images consisting of seeds from varied crops. We have also tested the speed and performance over images of varied sizes. Further, we have compared the obtained seed count from both Java and Android versions with the actual seed count. The error rate in Java is around 1% and the error rate in Android is around 1.5%. The contours generated in Android will be slightly different from that of Java due to several reasons. Primarily, we cannot use structures and methods provided by ImageJ such as ImagePlus, ImageProcessor and ByteProcessor as they depend on java AWT and Android doesn't support Java AWT packages. So, we have used Bitmaps alone as their replacement. The traversal over the image using a Bitmap is different from the ByteProcessor and ImageProcessor objects. Further, we have already seen that, indexing is completely different in Java. The ImageProcessor object as a reference to the ImagePlus object and a single dimensional byte array can be used as a reference to the double dimensional ImageProcessor object. So, any manipulation performed on the byte array in a single dimensional offset will be automatically deferenced to corresponding double dimensional index and will be updated in the corresponding index of the ByteProcessor object. As this is not possible in Android, we have replicated the single dimensional access into a double dimensional access and manipulated the original double dimensional Bitmap directly. However, the results are very close and the differences are negligent. Let us compare the actual seed count with the count obtained from the extended algorithm and that of the seed count obtained from the traditional algorithm. Consider few examples below.

Test case 1 (Number of seeds: Moderate)

Figure 8-1 Input image 1

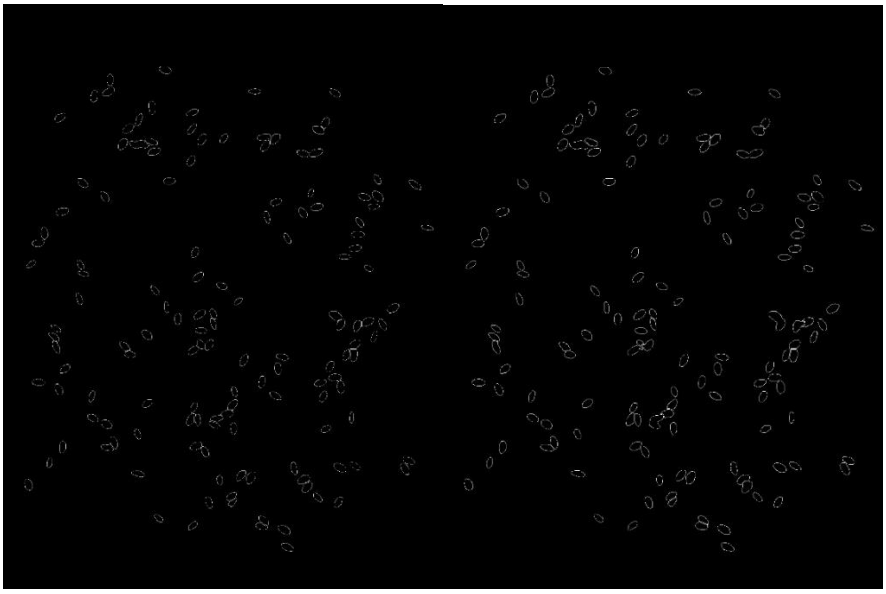

Figure 8-2 Contours in Java and Android for Image 1

Actual seed count	Java		Android	
141	Traditional algorithm	Extended algorithm	Traditional algorithm	Extended algorithm
	133	140	131	141

Table 8-1 Results for Image 1

Test case 2 (Number of seeds: High)

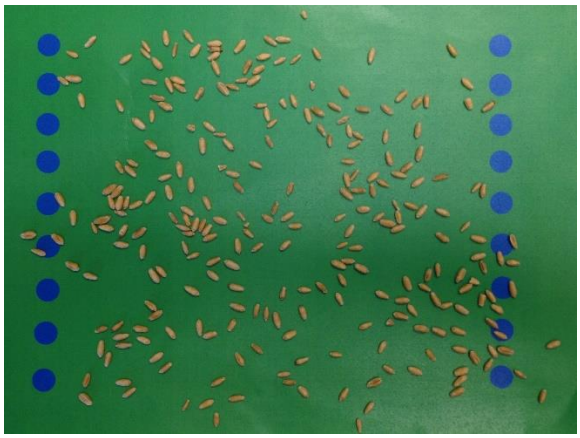

Figure 8-3 Input Image 2

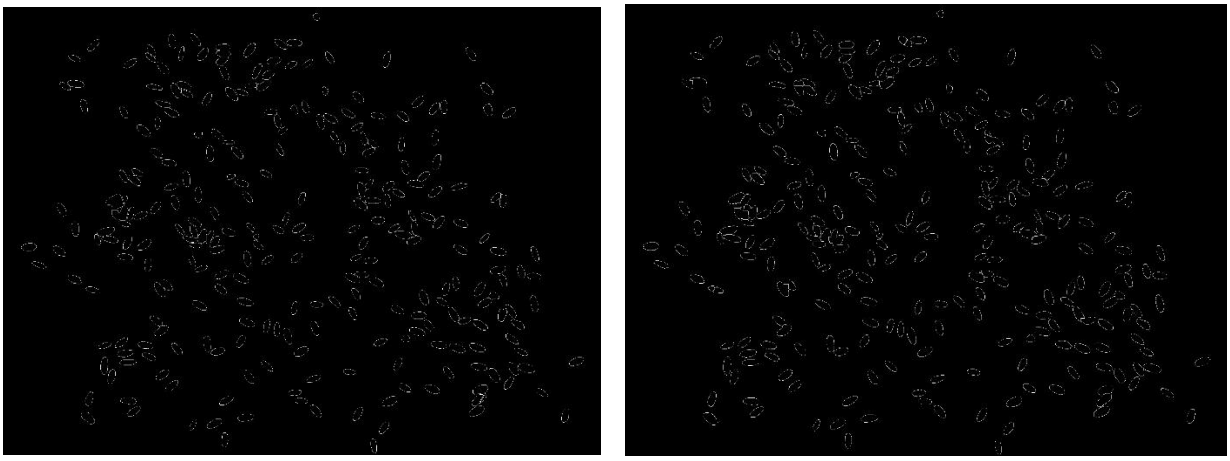

Figure 8-4 Contours in Java and Android for Image 2

Actual seed count	Java		Android	
267	Traditional algorithm	Extended algorithm	Traditional algorithm	Extended algorithm
	256	266	258	269

Table 8-2 Results for Image 2

Test case 3 (Number of seeds: Very high)

Figure 8-5 Input Image 3

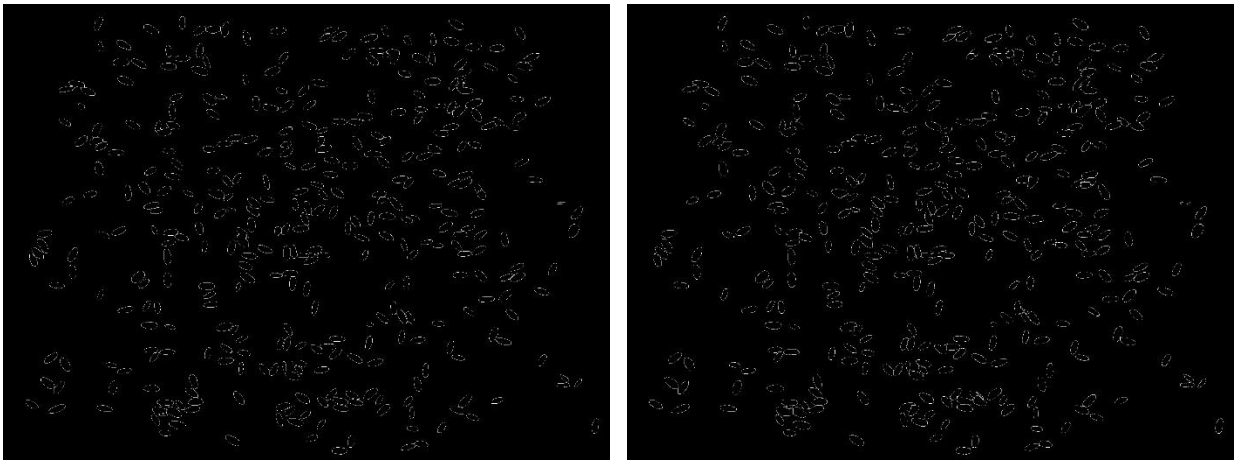

Figure 8-6 Contours in Java and Android for Image 3

Actual seed count	Java		Android	
363	Traditional algorithm	Extended algorithm	Traditional algorithm	Extended algorithm
	345	363	348	366

Table 8-3 Results for Image 3

8.2 Processor and Memory utilization in Android

8.2.1 Memory

This application requires a device with a RAM of 1 Gigabytes or more. The memory utilization by the application for various cases can be seen in the below graphs. The free and allocated RAM in megabytes are represented along the y-axis. Time elapsed is represented along the x-axis. Light blue color represents the free memory whereas the allocated memory is represented by dark blue color.

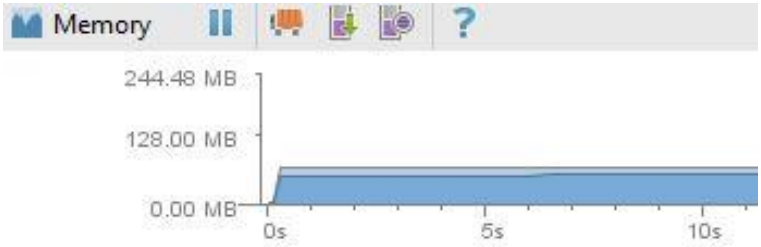

Figure 8-7 Memory utilization when the application is opened initially

The application is allocated with 57.48 MB of RAM when it is run initially.

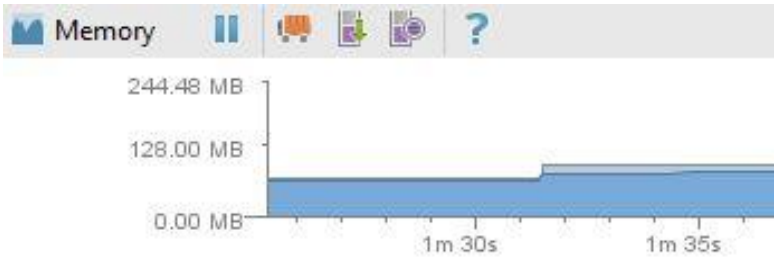

Figure 8-8 Memory utilization when the fragment is loaded

When the fragment is loaded, it consumes additional 20 MB and the consumption will be at 77.62 MB. This can be seen by the sudden hike in the graph after opening the second screen after staying idle in the primary screen for 1 minute 30 seconds.

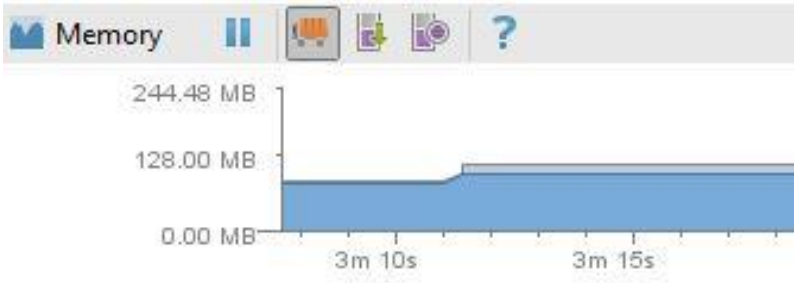

Figure 8-9 Memory utilization when the fragment is loaded with the selected image

When the image is captured or selected, the custom image view will be loaded with the bitmap of the selected image. The image selected has the size of 2520 * 1886. Additional 20 MB is allocated to hold the bitmap at this point. Total allocation is 99.27 MB.

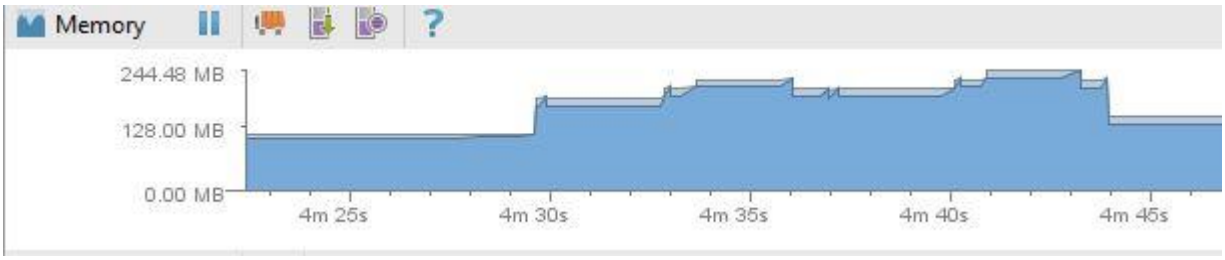

Figure 8-10 Memory utilization when the picture is being processed

When the button is clicked, processing starts. During processing the algorithm is run twice over the bitmap object. During each run, a double dimensional array is maintained as a reference for the bitmap and is manipulated all the way and is copied back into the bitmap in the end. The initial bump in the graph represents the additional memory allocated to the OpenCV Mat objects. The next bump represents the first call to the algorithm. The sudden fall represents the invocation of garbage collector which freed all the unused memory after the first run. The middle part represents the middle operations and the last bump represents the second call to the algorithm. Here it reaches to the peak value at 228.21 MB. When the bitmap is returned to the fragment we have a sharp fall to 99.27 MB.

8.2.2 CPU utilization

Here, y-axis represents the percentage of the processor being utilized while the x-axis represents the time elapsed. When the application is run initially, 0% of CPU is used as the main screen is blank with a single button.

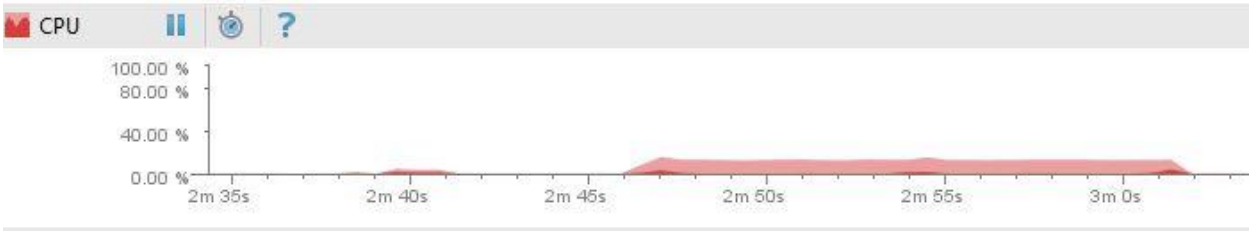

Figure 8-11 CPU utilization

When the button is clicked 5.58% of the CPU is used and the went back to 0% as the application remains idle. When the bitmap is set to the image view, 0.45% of total is used and when the image is being processed, an average of 9.76% with a peak value of 13.15% processor is utilized. When the image is zoomed and navigated, 3.38% of the processor is utilized.

8.2.3 Graphics Processing Unit Monitoring

As our application deals with significant amount of graphics. The GPU monitor gives a representation of the time taken to render the frames of the current window in the User Interface. The horizontal green line represents 60 Frames per second whereas the horizontal red line represents 30 Frames per second. The y-axis represents the amount in milliseconds taken by the GPU to execute different stages of graphics rendering. The x-axis represents the time elapsed. We can visualize the GPU monitor in the device by enabling it in the developer actions. In the below graphs, Red color represents the time taken for rendering the commands, Blue represents the time taken to render draw operations, light green color represents the time taken to render the current

window layout, Orange represents the internal buffers and dark green line color represents the miscellaneous time taken by background processes.

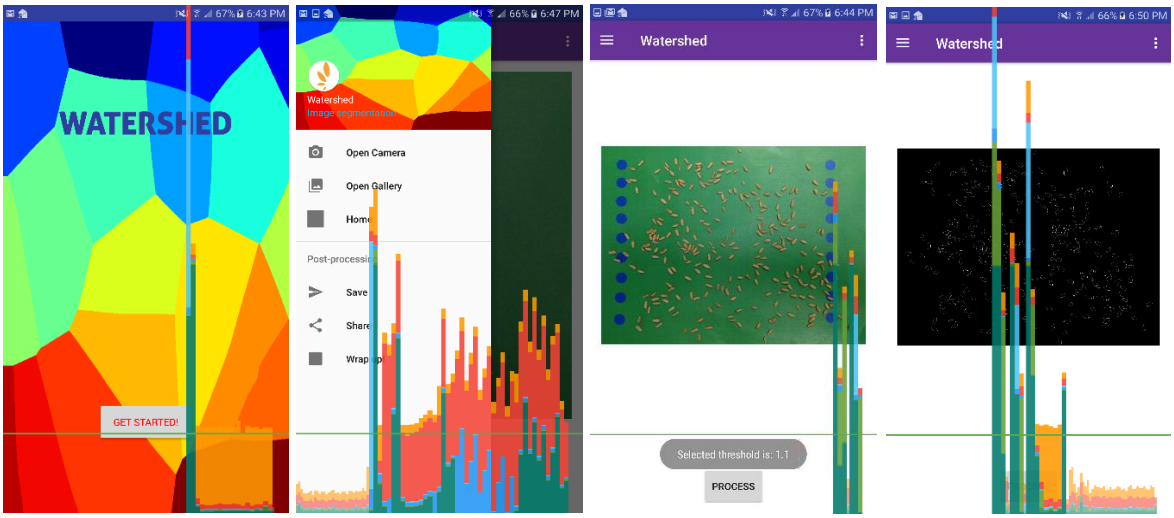

Figure 8-12 GPU monitoring when the graphics and processing is high

As we can see, the main screen has various colors with high end graphics. We have used custom fonts for the text on the button which is the reason for the blue line. Further, there are three different views appended over the navigation drawer which are the reason for all the red lines. Further, custom font are used for the title of the drawer which is the reason for blue and green lines. When the image is set onto the custom view, background processes include conversion of bitmap and calling the fragment, placing ImageView over the fragment which increase background rendering. The blue lines in the right most image represent time taken to draw lines are drawn over the binary image during processing. Green lines are for setting up the fragment over the layout. All other operations such as zooming, navigating, rotating, clearing, saving are under the green line as they do not require any particular graphical rendering.

Figure 8-13 GPU monitoring during other operations

8.2.4 Speed

The algorithm is relatively faster in Java for larger images when comparted to Android. We have used the ImageProcessor object in Java as a reference to ImagePlus object. The copy operation is very faster which is performed by the inbuilt function getProcessor(). In addition, to process the ByteProcessor object, we used byte array as a reference. All the pixels from the ByteProcessor object will be copied to the byte array by just using the getPixels() function provided by the ByteProcessor class. Further, the algorithm manipulates the binary image several times in various steps. In java, manipulations take minimal time as the manipulations made in reference objects will directly be reflected into the referenced object. This avoids all the copy operations after pixel manipulations. However, in Android, there are no such structures which can minimize the overhead of copy operations and further we can only use a Bitmap object to store and manipulate image data. A Bitmap is nothing but a two-dimensional array. All the manipulations and copy operations which are performed recursively back and forth in the

algorithm will affect the speed to a much greater extent which might even crash the application. But, we have taken a single double dimensional array to which all the pixel intensities are copied initially and we then manipulated this double dimensional array itself in all the operations and set this array to the returned Bitmap. We reduced the number of copy operations to just two in the whole algorithm in Android. So, both the versions work with equal speed. However, for larger images these copy operations might cost some time in Android. Further, the speeds of the algorithm in both the versions increase with the increase in the number of seeds. As our processing depends on the foreground objects which are the white pixels after the input image is converted to binary. If there are very less seeds, it results in lesser white pixels and reduces the delay. However, for images with large number of seeds, the processing speed is affected. For larger images with lesser number of seeds, Android version is much faster than Java as the overhead for maintain references for image objects affects the speed in java. Further, Android version is much faster than Java for both the cases where smaller images with many seeds and smaller images with many seeds. These specific test cases are for a system with a 2.70 GHz Intel® Core ™ i7-2620M processor with 16 Gigabytes of RAM and an Android device with a 4 x 2.1 GHz Cortex-A57 processor with a 4 Gigabytes of RAM. We have tested the algorithm on several systems and Android devices with varying processors but the results turned out to be the same.

Case 1 Large image with extreme number of seeds

Image size	Seed population	Java	Android
2520 x 1886	350 to 500 seeds	9.87 seconds	16.22 seconds

Table 8-4 Large image with extreme number of seeds

Case 2 Large image with moderate number of seeds

Image size	Seed population	Java	Android
2520 x 1886	200 to 350 seeds	6.21 seconds	14.68 seconds

Table 8-5 Large image with moderate number of seeds

Case 3 Large image with less number of seeds

Image size	Seed population	Java	Android
2520 x 1886	50 to 150 seeds	4.68 seconds	5.65 seconds

Table 8-6 Large image with less number of seeds

Case 4 Large image with very less number of seeds

Image size	Seed population	Java	Android
2520 x 1886	1 to 50 seeds	3.63 seconds	2.48 seconds

Table 8-7 Large image with very less number of seeds

Case 5 Medium image with large number of seeds

Image size	Seed population	Java	Android
1508 x 1508	200 to 250 seeds	8.57 seconds	8.21 seconds

Table 8-8 Medium image with large number of seeds

Case 6 Medium image with moderate number of seeds

Image size	Seed population	Java	Android
1508 x 1508	100 to 200 seeds	7.65 seconds	7.49 seconds

Table 8-9 Medium image with moderate number of seeds

Case 7 Medium image with less number of seeds

Image size	Seed population	Java	Android
1508 x 1508	50 to 100 seeds	4.63 seconds	3.16 seconds

Table 8-10 Medium image with less number of seeds

Case 8 Small image with less number of seeds

Image size	Seed population	Java	Android
462 x 462	30 to 50 seeds	1.85 seconds	0.76 seconds

Table 8-11 Small image with less number of seeds

Case 9 Small image with very less number of seeds

Image size	Seed population	Java	Android
462 x 462	1 to 30 seeds	0.98 seconds	0.33 seconds

Table 8-12 Small image with very less number of seeds

As we can see here, the performance of the algorithm in Android is very faster the Java for small and medium sized images regardless of the number of seeds present in the image. However, Java is faster than Android for images with large sizes. Based on the varied number of tests performed on many Android devices, this implementation of the extend algorithm can efficiently perform Mobile High Throughput Phenotyping.

Chapter 9 - Ongoing research

Currently, we have implemented the auto crop functionality which allows the user to capture the image from any distance and the application will automatically crop the region of interest. That is, the application will crop the image along the green background edges and process the image accordingly. In order to support the algorithm for seeds from additional crops such as Soybeans, Canola, Cassava, Potato and Silphium. Each seed has a Hue frequency depending on the color shade. We plan to extend the algorithm further to allow the user to input multiple hue ranges dynamically to allow the corresponding color ranges. Also, we are planning to optimize the pre-processing in the application. The application will be optimized in such a way that the user can click the picture from any direction and in any angle. He application automatically adjusts the image based on the camera angle and project the image onto the plane. Further, the processing also depends on the lighting conditions. The application will further be modified to handle images with various lighting conditions. To support plan breeding for rapidly extract plant phenotypes, several user-friendly applications are being developed and deployed which are currently in use by hundreds of breeding programs around the world. Currently, we are integrating this application with one such application named ‘1KK’ [33], an image based HTP application which gives accurate size and shape parameters for the seeds and tubers/roots. 1KK which stands for one thousand kernel weight, a primary selection criterion in plant breeding programs. This application uses a non-parametric algorithm along with an Elane USB scale to measure and analyze the length, width and area of each individual seed in the input image. Our goal is to integrate the extended Watershed algorithm with the 1KK app to accurately count seeds in the input image to determine the average weight of the seeds. This count will further be used to determine the yield of a crop.

Chapter 10 - Conclusion

Today, Phenotyping is considered as one of the major bottlenecks in plant breeding programs. With the increase in the availability of genomic data and lack of availability of methods to rapidly collect precision phenotypic data, there is a need for advancement in this area. Building on the success of Field Book in BreadPheno, a collection of user friendly mobile applications for high-throughput phenotyping (HTP) have been developed. This application is the first ever Android application which implements Watershed segmentation for rapid field-based phenotyping. In addition, this application optimizes the existing traditional Watershed segmentation algorithm by merging and splitting to rapidly characterize and count the number of seeds in the image. With an error rate of less than 2%, this application increases the ability of plant breeders to analyze and generate data accurately, rapidly and cheaply. As a part of our projects, several other applications to study determine different phenotypes have been deployed. All these applications have been adopted by hundreds of breeding programs around the world. Thus, this research is successfully and continuously building an active platform to decipher plant genomes and accelerate plant breeding.

Chapter 11 - References

[1] Mitchell L. Neilsen, Shravan D. Gangadhara and Trevor Rife. *Extending Watershed Segmentation Algorithms for High-Throughput Phenotyping*. 29th International Conference on Computer Applications in Industry and Engineering (CAINE-2016). September 26–28, 2016.

[2] Wikipedia (n.d.). *Phenotype*. Retrieved on March 16, 2017 from: https://en.wikipedia.org/wiki/Phenotype

[3] Rensselaer Polytechnic Institute (n.d.). *What is computer vision?*. Retrieved on March 16, 2017 from: https://www.ecse.rpi.edu/Homepages/qji/CV/3dvision_intro.pdf

[4] Serge Beucher. May 18, 2010. *The Watershed Transformation.* Retrieved on March 18, 2017 from: http://cmm.ensmp.fr/~beucher/wtshed.html

[5] Itseez. 2016. *Open Source Computer Vision Library*. Retrieved on March 19, 2017 from https://github.com/itseez/opencv.

[6] Gary Bradski and Adrian Kaehler. September 2008. *Learning OpenCV*. O'Reilly Media. pp. 1-5.

[7] Marvin Smith. (n.d.). *Introduction to OpenCV*. Retrieved on March 16, 2017 from: https://www.cse.unr.edu/~bebis/CS48the 5/Lectures/Intro_OpenCV.pdf

[8] OpenCV dev team. April 12, 2017. *Mat – The Basic Image Container*. OpenCV 2.4.13.2 documentation. Retrieved on March 16, 2017 from: http://docs.opencv.org/2.4/doc/tutorials/core/mat_the_basic_image_container/mat_the_basic_image_container.html

[9] OpenCV dev team. April 12, 2017. *Basic Drawing- Scalar*. OpenCV 2.4.13.2 documentation. Retrieved on March 18, 2017 from: http://docs.opencv.org/2.4/doc/tutorials/core/mat_the_basic_image_container/mat_the_basic_image_container.html

[10] OpenCV dev team. April 12, 2017. *The OpenCV Tutorials.* Release 2.4.13.2. Retrieved on March 18, 2017 from: http://docs.opencv.org/2.4/opencv_tutorials.pdf. pp. 150-151.

[11] Students Gymkhana – IIT Kanpur. August 29, 2016. *OpenCV tutorial: Image Processing.* Retrieved on March 19, 2017 from: http://students.iitk.ac.in/eclub/assets/tutorials/OPENCV%20TUTORIAL.pdf. pp. 8-21.

[12] University of Washington. May 05, 2000. *Image Segmentation.* Retrieved on March 19, 2017 from: https://courses.cs.washington.edu/courses/cse576/book/ch10.pdf

[13] Steve Eddins. 2002. *The Watershed Transform: Strategies for Image Segmentation.* Retrieved on March 19, 2017 from: https://www.mathworks.com/company/newsletters/articles/the-watershed-transform-strategies-for-image-segmentation.html

[14] Gary Bradski and Adrian Kaehler. September 2008. *Learning OpenCV*. O'Reilly Media. pp. 295-297.

[15] Doxygen. December 23, 2016. *Image Segmentation with Watershed Algorithm.* OpenCV 3.2.0 documentation. Retrieved on March 19, 2017 from: http://docs.opencv.org/3.2.0/d3/db4/tutorial_py_watershed.html

[16] Patrick Pirrotte. (n.d.). *The ImageJ Eclipse Howto.* ImageJ Documentation Wiki. Retrieved on March 21, 2017 from: http://imagejdocu.tudor.lu/doku.php?id=howto:plugins:the_imagej_eclipse_howto

[17] Wikipedia. December 03, 2016. *Comparison of Java and Android API.* Retrieved on March 21, 2017 from: https://en.wikipedia.org/wiki/Comparison_of_Java_and_Android_API#Look_and_feel

[18] Chris Thompson. June 12, 2016. *Android Studio – android.jar and rt.jar conflicts.* Retrieved on March 21, 2017 from: http://stackoverflow.com/questions/37778738/android-studio-android-jar-and-rt-jar-conflicts

[19] Google Code Archive. (n.d.). *awt-android-compat.* An attempt to facilitate AWT rendering on the Android platform. Retrieved on March 26, 2017 from: https://code.google.com/archive/p/awt-android-compat/

[20] Curtis Rueden and Tobias Pietzsch. March 22, 2017. *ImgLib2 (Generic Image Processing for Java).* Retrieved on March 26, 2017 from: http://imagej.net/ImgLib2

[21] Curtis Reuden. August, 2016. *Using ImageJ with Android.* ImageJ forum- shravandg. Retrieved on March 26, 2017 from: http://forum.imagej.net/t/using-imagej-with-android/2517

[22] Mike James. July 27, 2015. *Android Adventures – Beginning Bitmap Graphics.* Retrieved on March 26, 2017 from: http://www.i-programmer.info/programming/android/8797-android-adventures-beginning-bitmap-graphics.html

[23] HigherPass. (n.d.). *Working With Images In Android.* Retrieved on March 26, 2017 from: http://www.higherpass.com/android/tutorials/working-with-images-in-android/2/

[24] Dima. June 22, 2012. *Why do we use the HSV colour space so often in vision and image processing?*. Stack Overflow. Retrieved on March 26, 2017 from: https://dsp.stackexchange.com/questions/2687/why-do-we-use-the-hsv-colour-space-so-often-in-vision-and-image-processing

[25] John W. Shipman. October 16, 2012. *The hue-saturation-value (HSV) color model.* Introduction to color theory. Retrieved on March 26, 2017 from: http://infohost.nmt.edu/tcc/help/pubs/colortheory/web/hsv.html

[26] Wikipedia. March 22, 20110. *HSV cylinder*. HSL and HSV. Retrieved on March 26, 2017 from: https://en.wikipedia.org/wiki/HSL_and_HSV

[27] OpenCV dev team. November 10, 2014. *Changing Colorspaces.* OpenCV 3.0.0-dev documentation. Retrieved on March 26, 2017 from: http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html

[28] Android Developers. (n.d.). *Color.* Retrieved on March 26, 2017 from: https://developer.android.com/reference/android/graphics/Color.html

[29] Outfitters Supply, Inc. (n.d.). *Tucker Equitation Endurance saddle.* [Photograph]. Retrieved on April 1, 2017 from: https://www.outfitterssupply.com/tucker-endurance-saddle-comparison.asp

[30] Wikipedia. (n.d.). *A saddle point on the graph of $z=x^2-y^2$ (in red).* Saddle point. Retrieved on April 1, 2017 from: https://en.wikipedia.org/wiki/Saddle_point

[31] Rong Ge. March 22, 2016. *Escaping from Saddle Points.* Retrieved on April 1, 2017 from: http://www.offconvex.org/2016/03/22/saddlepoints/

[32] Laerd statistics. 2013. *What is Histogram?* Retrieved on April 1, 2017 from: https://statistics.laerd.com/statistical-guides/understanding-histograms.php

[33] Trevor Rife and Jesse Poland. 2015. *1KK*. Wheat Genetics and Germplasm Improvement. Retrieved on April 1, 2017 from: http://www.wheatgenetics.org/1kk. Github source: https://github.com/trife/1KK