INTRODUCTION TO IMAGE PROCESSING AND COMPUTER VISION

LABORATORY PROJECT 1

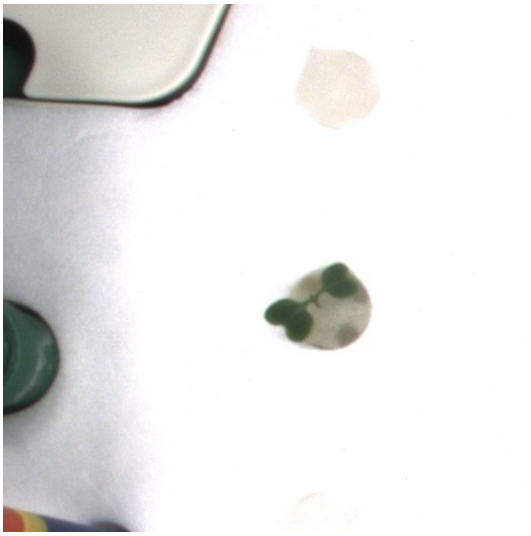
Contents

By Elif Ozdemir
ozdemire@studen.mini.pw.edu.pl

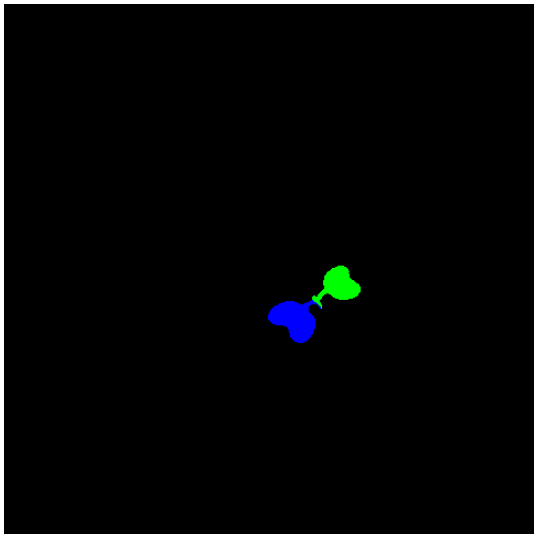
06.01.2020

Introduction

Image segmentation is a common technique in digital image processing and analysis to partition an image into multiple parts or regions, often based on specific features and criteria.
In this project we are provided with the images of plants (during growing process). There are 5 different plants and 3 cameras. Each 4 hours during 10 days picture of each plant is taken. In total we have 900 images to be segmented and labeled.
Besides, we have colored labels of each plant at each stage.

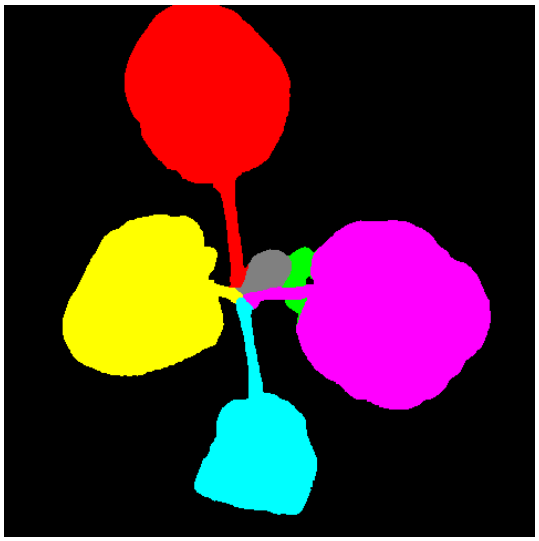


rgb_00_00_000_00.png



label_00_00_000_00.png

rgb_00_01_009_04.png



label_00_01_009_04.png

Plant Segmentation and Labeling Task 1

Task description:

• output: segmented plants (segmentation masks and bounding boxes optionally)
• binary segmentation (color multiclass masks need to be converted to binary masks)
• assessment according to Intersection over Union metric (IoU, also referred to as the Jaccard index) and Dice coefficient
• mean results for whole data set and per individual plant

Obstacles faced:

- After some time the small circle, from where the plant grows, changes its properties. The area's color becomes almost the same with plant's leaf color. This fact makes very diffiicult to differ the background and foreground.
  The following picture is a good example.

rgb_00_02_004_02.png

- Another hardship is connected to other elements in the images. In some pictures there are non-plant objects that have exactly the same tones of green. This obstacle was easier to overcome. An example of it:

rgb_02_04_009_05.png

Algorithm description:

After importing an image I start processing with CLAHE (contrast limited adaptive histogram equalization) technique since improving contrast in images in this dataset usually gives better results. My main purpose in applying this method is lightening the circle area near the root which was mentioned in obstacle number one.
I'm not using AHE (Adaptive histogram equalization) because of its tendency to overamplify the noice. CLAHE prevents such problems by limiting the amplification.
All in all, CLAHE algorithm enhances the segmentation Jaccard index by 0.0059 and Dice coefficient by 0.0033.

```
# applying CLAHE contrast limited adaptive histogram equalization
clahe = cv2.createCLAHE(clipLimit=3., tileGridSize=(8, 8))
lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB)  # convert from BGR to LAB color
space
l, a, b = cv2.split(lab)  # split on 3 different channels
l2 = clahe.apply(l)  # apply CLAHE to the L-channel
lab = cv2.merge((l2, a, b))  # merge channels
image = cv2.cvtColor(lab, cv2.COLOR_LAB2BGR)
```

How CLAHE affects the source image:

Original image
rgb_02_00_007_00.png

Processed image

In the next step I'm converting the image to HSV colorspace and perform the thresholding. HSV (Hue Saturation Value) colorspace is more suitable for thresholding as choosing specific colors is more straightforward.
And I'm segmenting using the simplest algorithm which is thresholding. I filter out the pixels that are not in the range.

```
# converting to HSV and thresholding
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
lower_green = np.array([40, 40, 40])
upper_green = np.array([70, 255, 255])
mask = cv2.inRange(hsv, lower_green, upper_green)
height, width = image.shape[:2]
```

I'm keeping on with the same '02_00_007_00' example. The result after thresholding is represented below.

02_00_007_00

After trying the following smoothing techniques I decided not to include any of them into my algorithm since they worsened the computed IoU and Dice scores.

```
# #applying smoothing techniques
# kernel = np.ones((1, 1), np.uint8)
# mask=cv2.medianBlur(mask, 5)
# mask=cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
# mask=cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
```

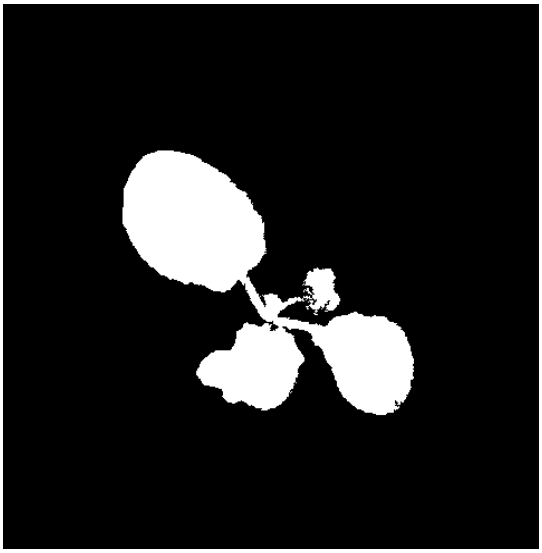
I use findContours() function to select all the contours(curves joinning the continuous points that satisfy the above mentioned thresholding conditions).
I decided to avoid the contours that are not a part of a plant by filtering areas of contours. And this is the second obstacle that needed to be solved. The following part of the code is for filtering:

```
conts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)[1]
# filtering contours according to contour area
newcnts = []
for cnt in conts:
    area = cv2.contourArea(cnt)
    if area > 300:
        newcnts.append(cnt)
```

I go through each contour and check its area. By brute force and observations the magical value is 300. For most of the cases this works well.

```
mask = cv2.drawContours(np.zeros((height, width, 3), np.uint8
                                 ), newcnts, -1, (255, 255, 255), cv2.FILLED)
mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
segmented = cv2.bitwise_and(image, image, mask=mask)
```

Then I use drawContours() method to apply my new set of desired contours. It can be seen from the picture below that I got rid of the non-plant areas. After that I get the segmentation result which is represented below.



mask_02_00_007_00.png

02_00_007_00

The next mission is drawing bounding boxes around the plant.. Since it's not obligatory I do it when I have just one countour. 681 is the number of images where I get the green rectangle. (681 out of 900) In other cases I don't alter the source image. The example plant has a bounding box below.

```
# drawing a bounding box when it's possible
if len(newcnts) == 1:
    rect = cv2.minAreaRect(newcnts[0])
    box = cv2.boxPoints(rect)
    box = np.int0(box)
    cv2.drawContours(image, [box], -1, (0, 255, 0), 3)
```

02_00_007_00

Now it's high time to check the success of this algorithm. For this purpose I compute Jaccard index and Dice coefficient. The webpage http://limu.ait.kyushu-u.ac.jp/~agri/komatsuna/ provides colored labels. I get the black and white version of it by thresholding. The formulas for the computations provided in lab task description were used here.

```
# getting black and white of label image
grayimagelabel = cv2.cvtColor(imagelabel, cv2.COLOR_BGR2GRAY)
(thresh, bwlabelimage) = cv2.threshold(grayimagelabel, 0, 255,
      cv2.THRESH_BINARY)

# black and white of the segmented image
graysegmented = cv2.cvtColor(segmented, cv2.COLOR_BGR2GRAY)
(thresh1, bwsegmented) = cv2.threshold(graysegmented, 0, 255,
      cv2.THRESH_BINARY)

# computing Jaccard index
intersection = np.logical_and(bwlabelimage, bwsegmented)
union = np.logical_or(bwlabelimage, bwsegmented)
iou_score = np.sum(intersection) / np.sum(union)

# computing Dice Coefficient
labelarray = np.asarray(bwlabelimage).astype(np.bool)
segmentedarray = np.asarray(bwsegmented).astype(np.bool)
intersection1 = np.logical_and(labelarray, segmentedarray)
dice_score = 2. * intersection1.sum() / (labelarray.sum() +
      segmentedarray.sum())
```

The mean results for whole data set and per individual plant

Plant number	IoU score	Dice score
1	0.8720010697682704	0.9306772193961936
2	0.8727257914520905	0.9310607947480266
3	0.8765090469735686	0.9323508964959659
4	0.8564248124776228	0.92220242748295
5	0.8586297143687344	0.9228195889890854
Total	0.8672580870080585	0.9278221854224439

To conclude with, the algorithm is not perfect; however, it could be used and improved. All in all, the Jaccard index is 0.87 and the Dice coefficient is 0.93. The main factor that worsened the process is the center of the plants. The algorithm had difficulties to tell the difference between the actual plant leaf and the greenish part of the ground. When it comes to algorithm complexity, it is quite fast thanks to built in optimized opencv functions.

Code for calculation of the scores:

```
def iteration_function():
    iou = 0
    dice = 0
    count = 0
    for plant in range(5):
        plant_iou = 0
        plant_dice = 0
        plant_count = 0
        for camera in range(3):
            plant_cam_iou = 0
            plant_cam_dice = 0
            plant_cam_count = 0
            for day in range(10):
                for hours in range(6):
                    image_title = str(camera) + "_0" + str(plant) + "_00" +
     str(day) + '_0' + str(hours)
                    iou_score, dice_score = process_image(image_title)

                    iou = iou + iou_score
                    plant_iou = plant_iou + iou_score
                    plant_cam_iou = plant_cam_iou + iou_score

                    dice = dice + dice_score
                    plant_dice = plant_dice + dice_score
```

```
                plant_cam_dice = plant_cam_dice + dice_score

                count = count + 1
                plant_count = plant_count + 1
                plant_cam_count = plant_cam_count + 1

        # print("iou plant " + str(plant) + " cam " + str(camera) + "  " +
    str(plant_cam_iou / plant_cam_count))
        # print("dice  plant " + str(plant) + " cam " + str(camera) + "  "
    + str(plant_cam_dice / plant_cam_count))
      print("iou plant " + str(plant) + "  " + str(plant_iou / plant_count))
      print("dice plant " + str(plant) + "  " + str(plant_dice /
    plant_count))
  print("total iou " + str(iou / count))
  print("total dice " + str(dice / count))
```

The whole source code is attached. (task1.py)

Plant Segmentation and Labeling Task 2

• output: segmented leavs (segmentation masks)
• multiclass segmentation (color multiclass masks)
• assessment according to Intersection over Union metric (IoU, also referred to as the Jaccard index) and Dice coefficient
• mean results for whole data set, per individual plant and per indicidual leaf in plant

First of all, I need to say that I tried to come up with different solutions. However, none of them were successful. I cannot describe them as average unfortunately. The main obstacle is the same as in the first task. Since the middle part of the plant cannot be segmented properly it is very difficult to separate to plant to its leaves. Nevertheless, I will be sharing my approaches.

1st Approach description

My first approach is mainly using distance transformation. A distance transform, also known as distance map or distance field, is a derived representation of a digital image. The choice of the term depends on the point of view on the object in question: whether the initial image is transformed into another representation.

As the source image I'm using the result of segmentation from the previous task. Thus, the steps till that point are the same.
Next, I get grayscale and binary images to be able to apply Distance Transform method. Normalization is for visualization purposes.

```
# Create binary image from source image
bw = cv.cvtColor(imgResult, cv.COLOR_BGR2GRAY)
_, bw = cv.threshold(bw, 40, 255, cv.THRESH_BINARY | cv.THRESH_OTSU)
dist = cv.distanceTransform(bw, cv.DIST_L2, 0)
# Normalize the distance image for range = {0.0, 1.0}
# so we can visualize and threshold it
cv.normalize(dist, dist, 0, 1.0, cv.NORM_MINMAX)
# Threshold to obtain the peaks
_, dist = cv.threshold(dist, 0.5, 1.0, cv.THRESH_BINARY)
```

In the following part of the code we dilate the image to get the peaks and create markers for watershed algorithm. At the end, I apply the algorithm.

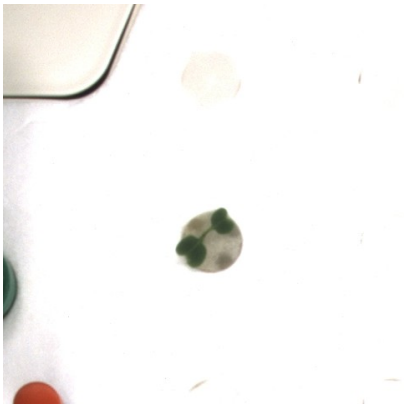
```
# Dilate a bit the dist image
kernel1 = np.ones((3, 3), dtype=np.uint8)
dist = cv.dilate(dist, kernel1)
dist_8u = dist.astype('uint8')
# Find total markers
_, contours, _ = cv.findContours(dist_8u, cv.RETR_EXTERNAL,
cv.CHAIN_APPROX_SIMPLE)
# Create the marker image for the watershed algorithm
markers = np.zeros(dist.shape, dtype=np.int32)
# Draw the foreground markers
for i in range(len(contours)):
    cv.drawContours(markers, contours, i, (i + 1), -1)
# Draw the background marker
cv.circle(markers, (5, 5), 3, (255, 255, 255), -1)
cv.watershed(imgResult, markers)
mark = markers.astype('uint8')
mark = cv.bitwise_not(mark)
```

If I get reasonable amount of contours I use the standard colors from my list. But in case algorithm results in too many contours I don’t want it to break. For this reason there are cases where I assign colors randomly. The implementation of it can be found below.

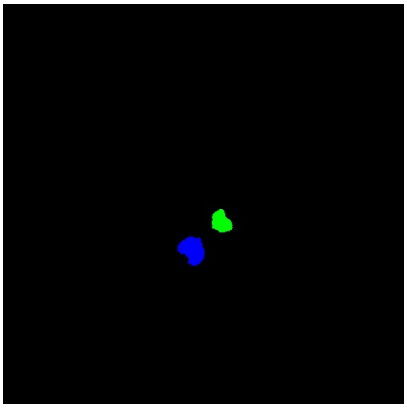
```
if len(contours) < 8:
    green = (0, 255, 0)
    blue = (255, 0, 0)
    azure = (255, 255, 0)
    red = (0, 0, 255)
    purple = (255, 0, 255)
    yellow = (0, 255, 255)
    gray = (125, 125, 125)

    colors = [blue, green, azure, red, purple, yellow, gray]
else:
    colors = []
```

```
    for contour in contours:
        colors.append((rng.randint(0, 256), rng.randint(0, 256),
      rng.randint(0, 256)))
# Create the result image
dst = np.zeros((markers.shape[0], markers.shape[1], 3), dtype=np.uint8)

# Fill labeled objects with colors
for i in range(markers.shape[0]):
    for j in range(markers.shape[1]):
        index = markers[i, j]
        if index > 0 and index <= len(contours):
            dst[i, j, :] = colors[index - 1]
```

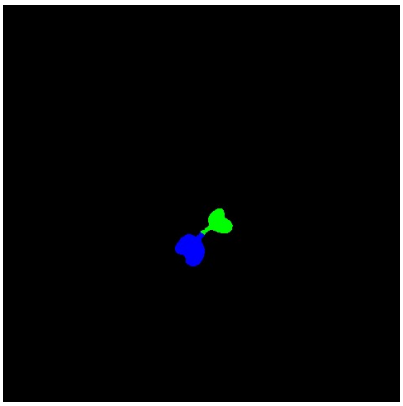
Few instance that can be considered as not failing are below.



rgb_02_00_000_00.png

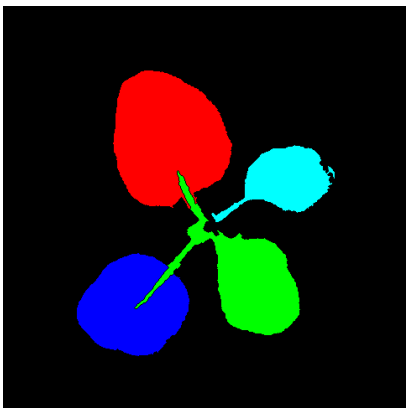


rgbm_02_00_000_00.png

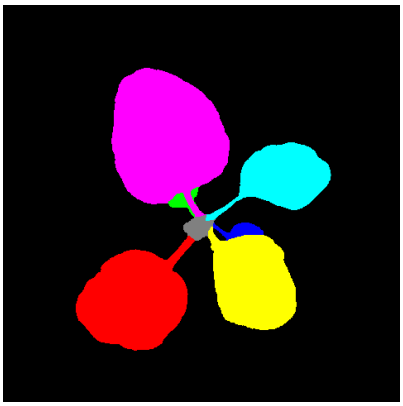


label_02_00_000_00.png
Expected

rgb_00_02_008_02.png

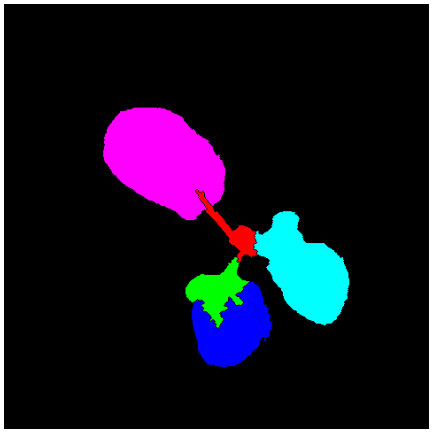


rgbm_00_02_008_02.png

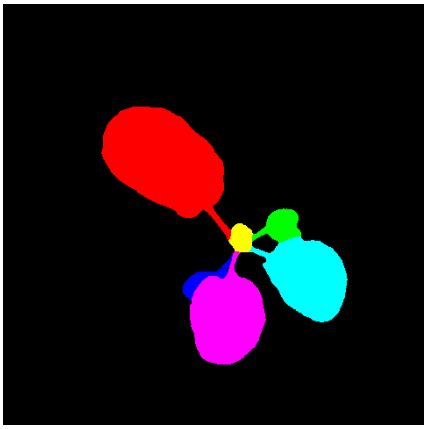


label_00_02_008_02.png

rgb_01_00_007_02.png



rgbm_01_00_007_02.png



label_01_00_007_02.png

It is quite clear that the approach mostly doesn't detect leaf borders properly. But in some cases it can detect the number of leaves correctly. The whole source code is task2apr1.py

2nd Approach Description

In this approach I'm using the fact that in some instances each leaf is a separate contour. So I get seperate mask for each leaf. I fill each contour with a different color and get labeled plant. However, as it was mentioned in the previous task in 681 cases the count of contours is 1. Which implies that this algorithm will fail at least 681 times.
The advantages of this algorithm is that it's very straightforward and fast. The whole source code is attached, here I will mention and explain few parts that I find important. (task2apr2.py)
First of all, I assign the colors that I will use.

```
green = (0, 255, 0)
blue = (255, 0, 0)
azure = (255, 255, 0)
red = (0, 0, 255)
purple = (255, 0, 255)
yellow = (0, 255, 255)
gray = (125, 125, 125)
colors = [blue, green, azure, red, purple, yellow, gray]
```

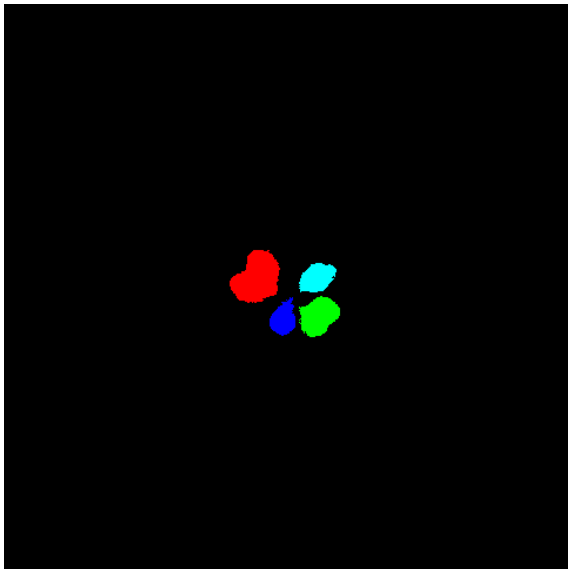
As in the first task I apply thresholding and get the contours. I filter contours based on the area each of them.

After that I create a new mask for each found contour. I apply it with the corresponding color from the previously created list of colors. And in the next step I combine all the masks in order to get the resulting mask.

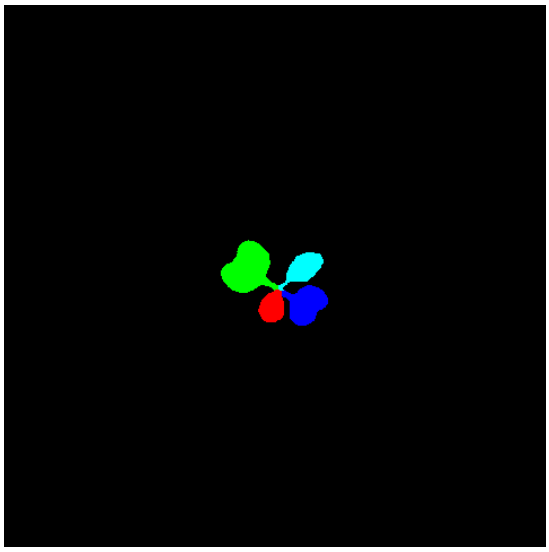
```
masks = []
foundleafcount = len(newcnts)
for x in range(foundleafcount):
    mask = cv2.drawContours(np.zeros((height, width, 3), np.uint8
                      ), [newcnts[x]], -1, colors[x], cv2.FILLED)
    masks.append(mask)

mainmask = masks[0]
if foundleafcount > 1:
    for i in range(1, foundleafcount):
        mainmask = cv2.addWeighted(mainmask, 1, masks[i], 1, 0)
```

A successful example for this algorithm can be found below.

rgb_01_02_001_04.png



rgbm2_01_02_001_04.png



label_01_02_001_04.png

For this approach to be more appropriate and appliable the segmentation technique should be improved. The maximum number of contours detected is 4. The technique will fail for sure for the plants that have more than four leaves.

Conclusion

At the end of each algorithm description some small assessment or evaluation was provided. In general, I tried other methods as well. For example, Canny Edge Detection or Laplacian Transformation. However, they reasult in another problem. When the plants grow they have strongly visible veins. Which cause the above mentioned techniques to result in more edges or conrours than leaf count in that specific plant.

As a result, I decided to stick with CLAHE, HSV thresholding and contourArea filtering. And for the second task I implemented Distance Transformation and Labeling based on contours as another approach. Resulting masks can be found in folder result and result2_1 which are for the first task and second task 1st approach correspondingly.

References and used material

1. Lecture notes
2. https://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html
3. https://www.pyimagesearch.com/2015/11/02/watershed-opencv/
4. https://en.wikipedia.org/wiki/Distance_transform
5. https://docs.opencv.org/3.4/d2/dbd/tutorial_distance_transform.html