Methodology

Introduction

Line Segment Detection (LSD) is the most common and essential step that has been studied thoroughly in many computer-vision research and projects. The major challenge in LSD is noise, selecting accurate segment out of several segments and variability present in an image.

In this research, Run-Length Encoded (RLE) and annotated images in JSON format were used as the initial data source for the implementation of the Line Segment Detection Algorithm. After RLE data has been decoded to an image, preprocessing and preparation steps were described in section 3.2. Then in section 3.3, Feature extraction step was described. Which is the most important step to detect all the line segments in the image. In the final step, (Section 3.4) line segments between two bounding boxes were successfully identified on the stem of each plant.

3.2 Preprocessing and Preparation

In this project, an image annotation tool called “Hasty” was used to create ground-truth dataset. During the annotation, the object classes of the image were defined as “First Section Cutting”, “Redundant Top End” and “Tip Cutting” etc. Finally, the annotated image was encoded to a run-length encoded (RLE) binary mask, which is a compressed representation of a binary image. In this encoding, the binary image is represented by a sequence of pairs (start, length), where each pair represents a consecutive run of 1's in the image.

In the 1st part of the algorithm, this mask was decoded and created a 2D numpy array. Also, image dimensions, bounding box details, selected object classes (First Section Cutting, Redundant Top End, Redundant Bottom End, Tip Cutting, Non-Viable Part, Second Section Cutting, Third Section Cutting, Fourth Section Cutting) which are useful for the line segment detection were gathered in this stage.

Usually sample images contain one or more plants, and in the second part of the algorithm, two Python lists were generated. Typically, in the Hasty Generated JSON file, plant sections are annotated with specific small bounding boxes and saved separately. The exact coordinates of these small bounding boxes were stored in the first Python list (Let’s call “small\_bbox\_list”). Also, large bounding boxes were defined for each plant (with all the section cuttings) in the JSON file and saved separately in the second Python list (Let’s call “large\_bbox\_list”). In addition, a dictionary was created to store the details of each plant section. In this case, the background of the image was saved as "0" and the foreground as "1".

Then the 3rd part of the algorithm is used to create the list of neighboring sections of the plants. As an example, 1st section, and 2nd section of the plant. This part of the algorithm is more challenging because efficiency of the algorithm mainly depends on this section. In here, OpenCV function: cv2.findContours()

was used to detect only the external contours (cv2.RETR\_EXTERNAL) of the sections of the plants and saved in a list(Let’s say “edge\_only\_list”) . Then in order to reduce the memory usage and speed up the process “cv2.CHAIN\_APPROX\_SIMPLE” method was used as the contour approximation method. Initially both internal and external contour (whole section of the plant) was used to find the neighboring sections of the plants. But during the stage of optimizing the algorithm’s efficiency, huge efficiency improvements were achieved while using this outside contour detection method. It was discussed more in the results section.

Also, another major improvement in efficiency was achieved by using parallel processing. By considering the factor of ease of use, two types of parallel processing methods based on multiple threads and based on multiple processors were tested. Using multiple processors for parallel processing means, distributing the workload across multiple physical or logical processors. Therefore, in this part of the algorithm (finding neighboring sections and finding complete sections of the plant) was written specifically for the ease of parallel processing. In here, standard Python library called “multiprocessing” was used for initial testing. But unfortunately, processing took excess time compared to normal processing. Therefore, other technique based on multiple threads was used with the help of “concurrent.futures” module in python.

A thread is a lightweight unit of execution that can run concurrently with other threads, sharing the same memory space. As shown in the **following algorithm(fig)**, multi-threading was done. Finally same combinations of the neighboring sections were eliminated. As an example (1st section cutting, 2nd section cutting) and (2nd section cutting, 1st section cutting) these two are similar combinations and these kinds of combinations were removed from the neighboring sections list.

Algorithm finding\_neighbors(small\_bbox\_list, edge\_only\_list):

Initialize an empty list bbox\_ones\_mask\_list

Initialize an empty list intersected\_comm\_ones\_list

Iterate over small\_bbox\_list with index mask\_index:

Iterate over edge\_only\_list with index mask\_index\_checked:

Iterate over edge\_only\_list[mask\_index\_checked] with index ones\_index:

If (small\_bbox\_list[mask\_index][0] <= edge\_only\_list[mask\_index\_checked][ones\_index][1] < small\_bbox\_list[mask\_index][1]) and (small\_bbox\_list[mask\_index][2] <= edge\_only\_list[mask\_index\_checked][ones\_index][0] < small\_bbox\_list[mask\_index][3]):

If mask\_index != mask\_index\_checked:

Append [mask\_index, mask\_index\_checked] to bbox\_ones\_mask\_list

Append [(edge\_only\_list[mask\_index\_checked][ones\_index][0], edge\_only\_list[mask\_index\_checked][ones\_index][1]), mask\_index, mask\_index\_checked] to intersected\_comm\_ones\_list

Return (bbox\_ones\_mask\_list, intersected\_comm\_ones\_list)

Algorithm finding\_sections(large\_bbox\_list, edge\_only\_list):

Initialize an empty list mask\_inside\_large\_bbox\_list

Iterate over large\_bbox\_list with index lg\_bb\_ind and element lg\_bb\_ele:

Iterate over edge\_only\_list with index mask\_index\_2 and element mask\_ele\_2:

Initialize ones\_count as 0

Iterate over mask\_ele\_2:

If (lg\_bb\_ele[0] <= co\_with\_ones\_ele2[1] < lg\_bb\_ele[1]) and (lg\_bb\_ele[2] <= co\_with\_ones\_ele2[0] < lg\_bb\_ele[3]):

Increment ones\_count by 1

If ones\_count is equal to the length of mask\_ele\_2:

Append [lg\_bb\_ind, mask\_index\_2] to mask\_inside\_large\_bbox\_list

Return mask\_inside\_large\_bbox\_list

Algorithm results\_function(small\_bbox\_list, large\_bbox\_list, edge\_only\_list):

Create an executor using ThreadPoolExecutor

# Submit the first for loop as a task

task1 = executor.submit(finding\_neighbors, small\_bbox\_list, edge\_only\_list)

# Submit the second for loop as a task

task2 = executor.submit(finding\_sections, large\_bbox\_list, edge\_only\_list)

# Wait for both tasks to complete

results = [task1.result(), task2.result()]

# Combine the results from both tasks and return them

return results

Then in the final part of the algorithm, based on the new neighboring sections list, every time two neighboring sections were considered for the analysis. Therefore, in this part of the algorithm, those two sections were considered for further analysis.

3.3 Feature Extraction

Feature Extraction step is the most important step in this project because the cutting line segment of two neighboring sections is situated in the edge of these sections. Therefore, several edge detection methods are tested to get optimum results.

The Canny Edge Detection method has several steps: Initially, the image is smoothed using a Gaussian filter to reduce the noise. Then, the gradients of the image intensity are calculated using the Sobel operator. Non-maximum suppression is applied to thinning the edges and keep only the maximum values. Double thresholding is used to classify pixels as strong, weak, or non-edges. Finally, weak edges that are connected to strong edges are retained as actual edges using a process called edge tracking by hysteresis. Opencv based algorithm (cv2.Canny) was used for this.

Then during the analysis of the results, some line segments were not identified. Therefore, two other methods were tested to identify those missing line segments. Sobel Operator calculates the gradient magnitude and direction of an image, which helps to identify regions with significant changes in intensity. In here Sobel operator (cv2.Sobel) was used in both x and y directions and results were combined.

Moreover, Laplacian of Gaussian (LoG) was also used to identify edges. The combination of Laplacian operator (cv2.Laplacian) with Gaussian smoothing(cv2.GaussianBlur) with the kernel size of 3x3 was applied to highlight the regions of rapid intensity changes. After that, the results of two neighboring plant sections were taken and passed through an AND operator. Then output was taken into further consideration. Finally, several types of line segments were revealed. The majority of the line segments appeared on the stem of the plants but some of the irrelevant lines were noticed on the area of the intersection of leaves and also leaves & stems etc. as shown in the Fig.

Detection of exact line Segment

This is the most challenging area of the algorithm because not all the line segments lay on the stem of the plant. Therefore, based on the results of the feature extraction stage, detection of the exact line segment was done in three steps. Initially combination of nearest sections with single line segments were considered. Then some combination of nearest sections was noticed with multiple line segments. Those were separated into two main categories. (Main reason for those line segments is due to intersection of two leaves and intersection of part of the leaves and stems). 1st separation (2nd Step) was done based on the size of the line segments. By analyzing the data, it was noticed that the majority of the samples had a noticeable difference in size. But still some samples were noticed with nearly similar line segments in size. 2nd separation (3rd step) was used for this kind of samples using the distance of the line segments to the stem.

The 1st step was easily achieved by using the clustering of the adjacent foreground pixels. Then defined a threshold value (distance in-between one adjacent pixel cluster to other) to categorize the cluster of adjacent pixels with non-adjacent pixel cluster. Then those values were stored in a list called “clustering\_list”. In the 1st step, only one cluster of adjacent pixels is detected all the time. It means clustering list length is zero. Thereafter, all the pixel coordinates are saved in another list (“cluster\_coor\_list”) that list was used as the final output (detected line segment) of the algorithm.

The second step is followed by the algorithm unless length of the clustering list is detected nonzero. In here, once again “cv2.findContours” method was used to detect all the available common line segments in two neighboring plant sections. Then the size of all the line segments were measured (using cv2.arcLength()) and discovered the range of the size of the line segment which is usually noticed on the stem as the cutting line. So based on that range, correct line segment was detected. As usual, pixel coordinates are saved in “cluster\_coor\_list”.

The third step is followed only if the size of the line segment is out of the range or if it has line segments with similar lengths. In this case, several methods were tried to identify the stem of the plant because the main concept in this part of the algorithm is to measure the distance from the point in stem to the nearest line segment.

Initially a Python library called “PlantCV” which is designed based on the OpenCV for image analysis in the field of plant was used to detect the stem. “plantcv.morphology.skeletonize() “ function is applied to obtain the skeletonized image. It was used to extract the skeleton while preserving the connectivity of the plant structure. But unfortunately, the main problem in applying skeletonization on the binary masks (plant sections) is separation of leaves and stems because of the unidentifiable boundary in between leaves and stems. Therefore, sometimes algorithms picked leaves as stems and skeletonized the leaves as well. Finally, the expected classification of stem and leaves separately was not achieved.

For the same purpose, the next approach was to segment the edge of the stem into several line segments as shown in the Fig. For this “cv2.createLineSegmentDetector” function was used and saved the result as “LSD\_img”. Based on the results, parallel or closer to parallel line segments were noticed on either side of the stem. If two or more parallel line segments are thickened, then there is a chance of creating a single line or solid object. This is the technique which was used to detect stems separately. Therefore, those line segments were dilated using “cv2.dilate” function with a 7x7 kernel (numpy.ones((7, 7), dtype=np.uint8)) and two iterations. In order to erode unwanted parts, erosion was done with OpenCV erosion function with same configurations as in dilation function. Then the difference of the result(“dilated\_eroded\_img”) and “LSD\_img” was saved as “substracted\_img” and passed through a median blur operation using a kernel size of 7, to reduce noise and outliers.

The resulted image(substracted\_img) consisted of parts of the stem not connected with each other. This issue was due to the improper boundaries of stem where the leaves joint. So once again morphological dilation operation was performed in order to connect those areas as much as possible. After that, the closed and biggest contour was assumed as the stem and other small, closed contours were neglected. Then Euclidean distance from that contour to the line segments were calculated and line segment with shortest distance was considered as the correct cutting line segment. Finally, pixel coordinates of that line segment were saved in “cluster\_coor\_list”.