Institute of Automation

Projektarbeit

Line Segment Detection

Dasanayake Mudiyanselage, Hasith Thilanka Dasanayake
6003143

May 23, 2023

Betreuer:
Philip Tietjen

Gutachter:
Prof. Dr.-Ing. Kai Michels

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich meine Abschlussarbeit ohne fremde Hilfe angefertigt habe und dass ich keine Anderen als die von mir angegebenen Quellen und Hilfsmittel benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entommen sind, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Die Abschlussarbeit darf nach der Abgabe nicht mehr verändert werden.

Datum:_____ Unterschrift:_____

Erklärung zur Veröffentlichung von Abschlussarbeiten

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs.2 BremArchivG) im Universitätsarchiv f‘ür wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin *nicht* damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum:_____ Unterschrift:_____

Contents

1 Introduction

This section is under construction.

2 Methodology

Line Segment Detection (LSD) is the most common and essential step in computer vision. The major challenges in LSD such as accurate segment selection out of several segments, and the efficient processing of the algorithm, have been extensively studied.

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 subtopic 2.1. Then in subtopic 2.2, the feature extraction step was described. Which is the most important step to detect all the line segments in the image. In the final step, (subtopic 2.3) correct line segments between two neighboring plant sections were successfully identified on the stem of each plant. Figure 2.1 shows the methodology in a flow chart.

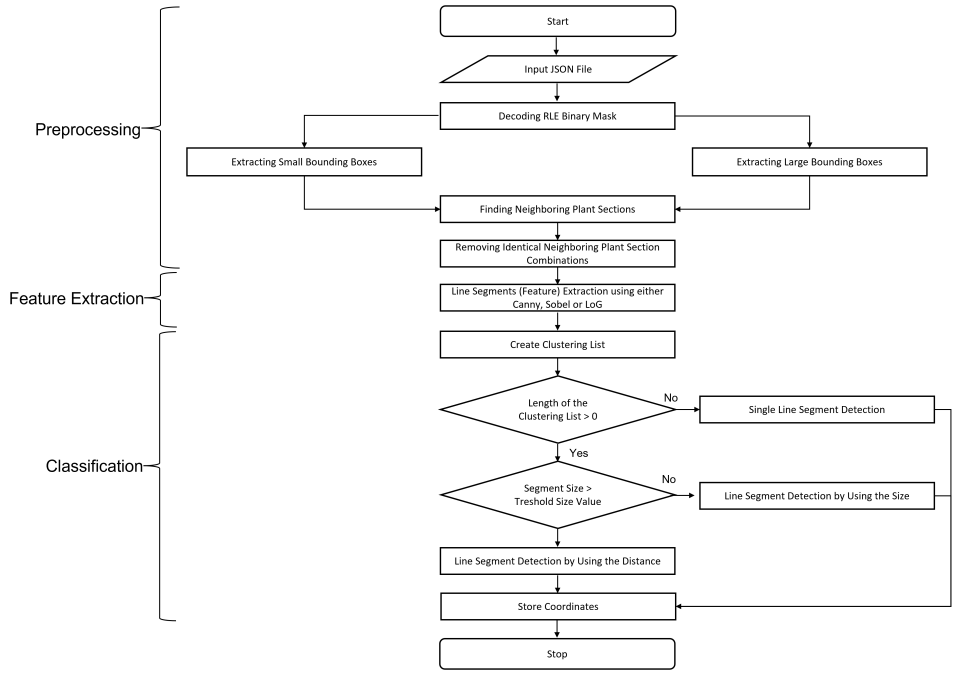

Figure 2.1: Flow Chart of the Overall Algorithm

2.1 Preprocessing and Preparation

In this project, an image annotation tool called “Hasty” was used to create a 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, and 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". As shown in the Figure 2.2 one large bounding box consisted of multiple small bounding boxes.

Then the 3rd part of the algorithm was used to create the list of neighboring sections of the plants. As an example, 1st Section and Redundant Bottom End section of the plant. This part of the algorithm is more challenging because the efficiency of the algorithm mainly depends on this section. Here, OpenCV function: cv2.findContours() was used to detect only the external contours (cv2.RETR_EXTERNAL) of the sections of the plants and saved them in a list(Let’s call “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 (whole solid section of the plant) and external contours were used to find the neighboring sections of the plants. But during the stage of optimizing the algorithm’s

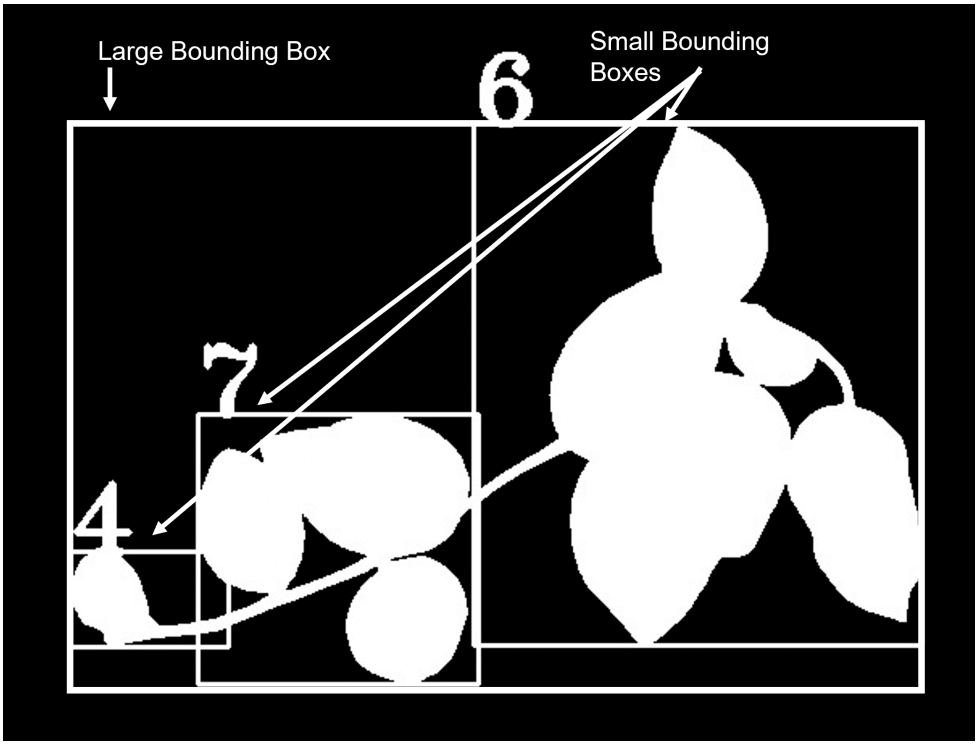

Figure 2.2: Small and Large Bounding Boxes

efficiency, huge efficiency improvements were achieved while using this outside contour detection method. It was discussed more in the results section.

Also, an additional major improvement in efficiency was achieved through the implementation of parallel processing. The most popular two types of parallel processing methods were tested, which were based on multiple threads and multiple processors. The use of multiple processors for parallel processing involves the distribution of the workload across numerous physical or logical processors. Two separate functions were used in the algorithm (function for the identification of neighboring sections and function for the identification of complete sections of the plant) to facilitate easy parallel processing. For initial testing, a standard Python library called "multiprocessing" was employed. Regrettably, the processing time exceeded the duration of normal processing time. Consequently, an alternative technique based on multiple threads was adopted, utilizing the "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 1, algorithm 2, algorithm 3 , multi-threading was used. After that, identical combinations

of neighboring sections were eliminated. For instance, combinations such as (1st section cutting, redundant bottom end cutting) and (redundant bottom end cutting, 1st section cutting) were considered similar combinations and were removed from the list of neighboring sections.

Input : small_bbox_list, edge_only_list
Output : neighboring_sections_list, intersected_area_list

Initialize an empty list neighboring_sections_list; Initialize an empty list intersected_area_list;

foreach *mask_index in small_bbox_list* **do**
  foreach *mask_index_checked in edge_only_list* **do**
    foreach *ones_index in edge_only_list[mask_index_checked]* **do**
      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]*)* **then**
        if *mask_index* ≠ *mask_index_checked* **then**
          Append [mask_index, mask_index_checked] to neighboring_sections_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_area_list;
        end
      end
    end
  end
end

return (neighboring_sections_list, intersected_area_list);

Algorithm 1: Finding Neighboring Sections

Input : large_bbox_list, edge_only_list
Output : mask_inside_large_bbox_list

Initialize an empty list mask_inside_large_bbox_list;

foreach *lg_bb_ind and lg_bb_ele in large_bbox_list* **do**
  foreach *mask_index_2 and mask_ele_2 in edge_only_list* **do**
    Initialize ones_count as 0;
    foreach *co_with_ones_ele2 in mask_ele_2* **do**
      if $(lg\_bb\_ele[0] \leq co\_with\_ones\_ele2[1] < lg\_bb\_ele[1])$ ***and*** $(lg\_bb\_ele[2] \leq co\_with\_ones\_ele2[0] < lg\_bb\_ele[3])$ **then**
        Increment ones_count by 1;
      end
    end
    if *ones_count is equal to the length of mask_ele_2* **then**
      Append [lg_bb_ind, mask_index_2] to mask_inside_large_bbox_list;
    end
  end
end

return (mask_inside_large_bbox_list);

Algorithm 2: Finding Small BBoxes Inside Large BBox

Input : small_bbox_list, large_bbox_list, edge_only_list
Output : results

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);

Algorithm 3: Parallel Processing

2.2 Feature Extraction

The feature extraction step is the most important step in this project because the cutting line segment of two neighboring sections is situated on the edge. Therefore, several edge detection methods were tested to get optimum results.

In this project, the most important step is the feature extraction step due to the

placement of the cutting line segment of two neighboring sections on the edge. Consequently, several edge detection methods were employed to achieve optimum results.

The initial detection method employed in this study was the Canny Edge Detection method (by using Opencv based algorithm (cv2.Canny)), which theoretically encompasses several steps: Firstly, a Gaussian filter was utilized to smooth the image and reduce noise. Subsequently, the gradients of the image intensity were calculated using the Sobel operator. To thin the edges and keep only the maximum values, non-maximum suppression was applied. Pixels were classified as strong, weak, or non-edges through the utilization of double thresholding. Finally, weak edges that are connected to strong edges are retained as actual edges using a process called edge tracking by hysteresis.

During the analysis of the results, certain line segments were not identified. Therefore, an alternative method was deployed to identify these missing line segments. In this study, the Sobel operator (cv2.Sobel) was applied in both the x and y directions, and the results were merged. Usually, the Sobel Operator is used to compute the gradient magnitude and direction of an image, facilitating the detection of regions exhibiting substantial changes in intensity.

Moreover, the identification of edges was performed using the Laplacian of Gaussian (LoG) method. The regions of rapid intensity changes were highlighted by applying a combination of the Laplacian operator (cv2.Laplacian) and Gaussian smoothing (cv2.GaussianBlur) with a kernel size of 3x3. After that, the results obtained from two neighboring plant sections were subjected to an AND operator.

Finally, various types of line segments were revealed. The majority of the line segments were on the stem, although some irrelevant lines were observed in the area where two leaves intersected with each other or with the stems. (see Figure 2.3).

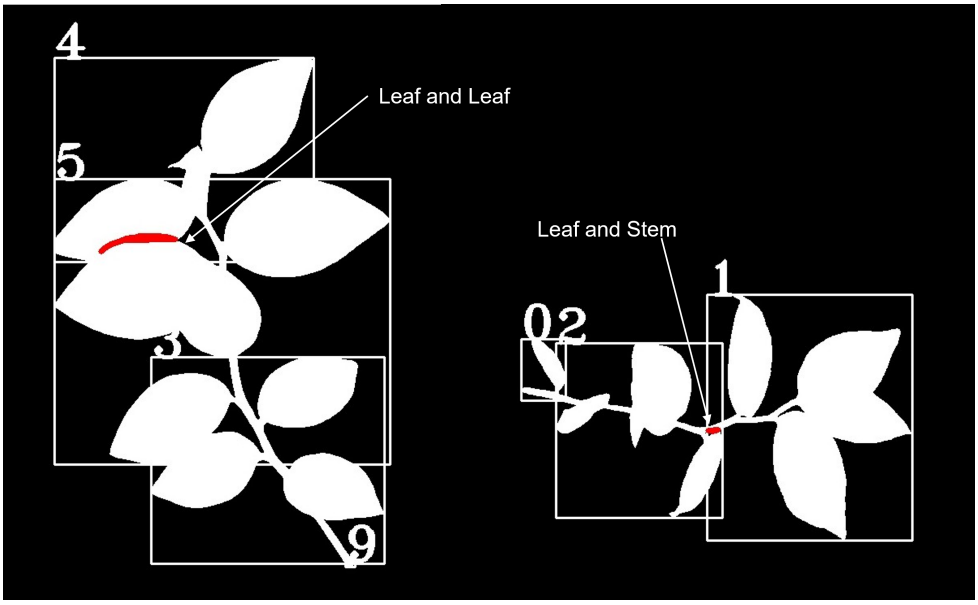

Figure 2.3: Intersection of Two Leaves and Leaf and Stem in Two Neighboring Sections

2.3 Detection of Exact Cutting Line Segment (Classification)

When not all line segments are situated on the stem of the plant, then it's a bit challenging to identify the correct segment. Therefore, the detection of the exact line segment was performed in three steps based on the results of the feature extraction stage. Initially, a combination of nearest sections with single line segments was considered(first step). Subsequently, combinations of nearest sections were identified with multiple line segments, which were further classified into two main categories. (These line segments arise primarily from the intersection of two leaves or the intersection of a portion of the leaves and stems). The first classification (second step) was conducted based on the size of the line segments. Data analysis revealed that the majority of the samples depicted a noticeable difference in size, although some samples showed line segments that were nearly similar in size. For this type of sample, the second classification (third step) was employed using the distance of the line segments to the stem.

The first step was easily achieved by clustering of the adjacent foreground pixels because only a single line segment needed to be detected. A threshold value was defined to categorize the cluster of adjacent pixels with non-adjacent pixel clusters.

If there is another cluster that is above the threshold value, the index of that cluster's closest pixel were stored in a list called "clustering_list". (Here, nothing was stored in the "clustering_list" because all the pixels were adjacent and there was no more than one cluster in this step). Throughout the first step, only one cluster of adjacent pixels (one line segment) was consistently detected. Finally, all the pixel coordinates were saved in another list called "cluster_coor_list", which served as the final output (detected line segment) of the algorithm.

The second step is executed if the length of the "clustering_list" is found to be non-zero which means multiple line segments are detected. Here, the "cv2.findContours" method was once again used to detect all available common line segments in two neighboring plant sections. The size of each line segment was measured using "cv2.arcLength()" to identify the range of sizes typically observed on the stem, as the cutting line. Based on this range, the correct line segment was detected. As before, the pixel coordinates were saved in the "cluster_coor_list".

The third step is pursued only if the size of the line segment falls outside the size range or if there are line segments with similar lengths. In this scenario, several methods were attempted to identify the stem of the plant, with the primary concept being to measure the distance from a point in the stem to the nearest line segment.

Initially, a Python library called "PlantCV," designed for image analysis in the field of plant research and based on OpenCV, was utilized to detect the stem. The "plantcv.morphology.skeletonize()" function was applied to obtain the skeletonized image, extracting the skeleton while preserving the plant structure's connectivity. However, a major challenge arose when applying skeletonization to the binary masks representing the plant sections due to the indistinguishable boundary between leaves and stems. Then, the algorithm occasionally misidentified leaves as stems and skeletonized them as well. Ultimately, the desired separate classification of stems and leaves was not achieved.

For the same purpose, the next approach involved segmenting the edge of the stem into multiple line segments. The "cv2.createLineSegmentDetector" function was employed, and the result was saved as "LSD_img". Based on the outcomes, parallel or nearly parallel line segments were observed on either side of the stem. If two or more parallel line segments were sufficiently thickened, the possibility of creating a single line or solid object arose. This technique was utilized to detect stems separately. Consequently, these line segments were dilated using the "cv2.dilate" function with a 7x7 kernel (numpy.ones((7, 7), dtype=np.uint8)) and two iterations. To remove unwanted parts, an erosion operation was applied using the

OpenCV erosion function with the same configurations as the dilation function. The difference between the result ("dilated_eroded_img") and "LSD_img" was saved as "subtracted_img" and subjected to a median blur operation with a kernel size of 7 to reduce noise and outliers.

The resulting image ("subtracted_img") consisted of disconnected parts of the stem due to improper boundaries where the leaves join. Therefore, another morphological dilation operation was performed to connect those areas as much as possible. Subsequently, the closed and largest contour was assumed to represent the stem, while other small, closed contours were disregarded. The Euclidean distance from that contour to the line segments was calculated, and the line segment with the shortest distance was considered the correct cutting line segment. Finally, the pixel coordinates of that line segment were saved in the "cluster_coor_list".

3 Results and Analysis

In this section, the results of the algorithm for detecting accurate cutting line segments are presented under several subtopics. A JSON file was used, containing a total of 316 images along with varying numbers of plants captured in each image. As an example, in some images, 21 section cuttings were available, while in others, only one or two were available.

In subtopic 3.1, a selected sample image was considered, and it served as an illustrative example, displaying all the algorithm steps employed. In subtopic 3.2, the accuracy and precision of the algorithm had been evaluated. Furthermore, the efficiency of the algorithm had been considered a critical factor to measure the quality of algorithm. This aspect has been thoroughly described in subtopic 3.3

3.1 Visualization of the Algorithm Steps

As shown in the Figure 3.1 , all the steps of the algorithm were visualized with the help of an illustrative example(stn2_pkg004_0_1077_rep). In the feature extraction stage, of the Figure 3.1 detected line segments were circled in yellow. The details of the Classification stage were summarized in the Table 3.1. For the index [0, 1], coordinates were received as (887, 1076), (888, 1076), (889, 1077), (890, 1077), (892, 1079), (893, 1079), (892, 1078), (891, 1078), (890, 1077), (889, 1077), (888, 1076). Then for the index [1,2], coordinates were received as (935, 787), (936, 787), (937, 788), (936, 787).

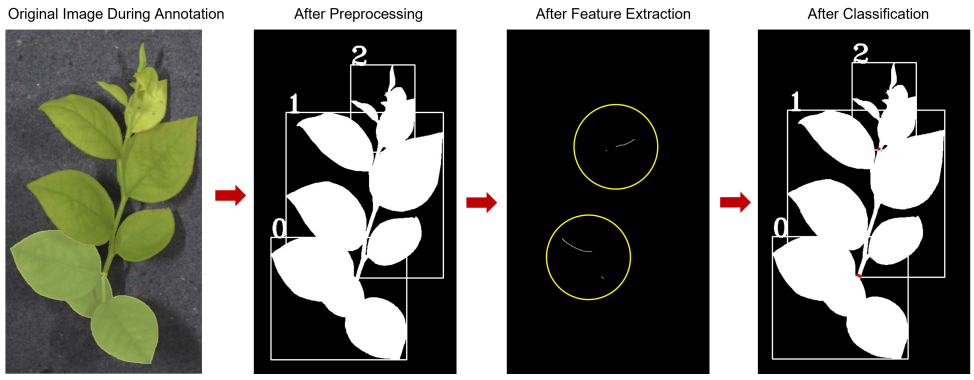

Figure 3.1: Visualization of Algorithm Steps

Neighboring Sections	Index	Classification
1st Section and Redundent Bottom End	[0, 1]	Using Size
1st Section and Raw Cutting	[1, 2]	Using Size

Table 3.1: Line Segment Coordinates

3.2 Evaluation of Algorithm's Performance

As mentioned in the introduction, evaluation of this algorithm's performance was done by using 316 samples. Through the analysis of various evaluation metrics and the visualization of results, insights can be gained into how well positive and negative instances are accurately identified by the algorithm. In this project, a fundamental tool for summarizing classification results called a confusion matrix was used. Moreover, the calculation and interpretation of key evaluation metrics such as accuracy, precision, and the F1 score were examined. Furthermore, the Receiver Operating Characteristic (ROC) curve and its significance in assessing the algorithm's performance in terms of Recall and false positive rates were investigated.

The confusion matrix is a tabular representation of the performance of the algorithm by the counts of true positive (TP), true negative (TN), false positive (FP), and false negative (FN) instances. Then it was used as the basis or the foundation of finding other evaluation metrics.

As shown in the, the actual classes of "correct line segment available and correct line segment not available" are represented by the columns, while it's detected(predicted) or not detected are represented by the rows. The number of instances falling into each category is correspondingly indicated by the values in the cells.

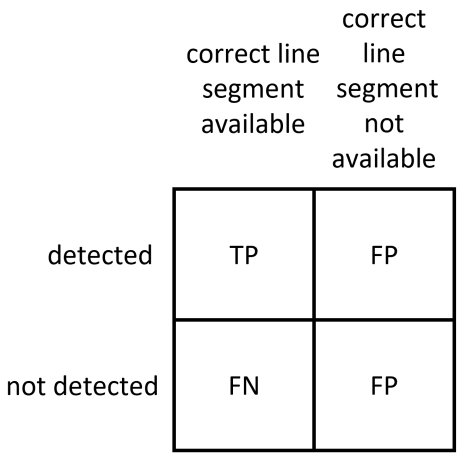

Figure 3.2: Confusion Matrix

The evaluation metrics were computed using the values derived from the confusion matrix. These metrics provide an overall assessment of the algorithm's performance.

The following formula was used to calculate the accuracy, which measures the overall correctness of the algorithm's predictions.

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{3.1}$$

Initially, the accuracy of the algorithm was calculated with only two steps. Only the 1st step and 3rd detection steps of the algorithm (see subtopic 2.4). Then the accuracy was detected as 0.8912(89.12%). As shown in the Figure 3.3 after implementing all three detection steps (see subtopic 2.4) to the algorithm, the accuracy of all the samples was plotted, and the average accuracy was taken as 0.9616 (96.16%). Therefore, an improvement of 0.0704(7.04%) was achieved by expanding the detection step.

The classification algorithm's overall predictions were highly accurate, according to this accuracy score. Also, the algorithm demonstrates its effectiveness in distinguishing between positive and negative instances (detection and non-detection).

However, a comprehensive analysis of other evaluation metrics is necessary to gain a proper understanding of the algorithm's performance, as well as its strengths and limitations. Therefore, precision and the F1 score were further calculated.

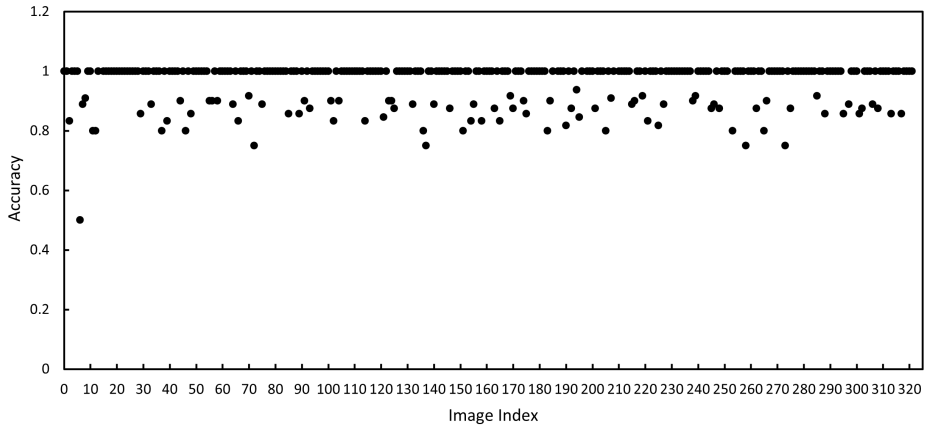

Figure 3.3: Accuracy vs Image Index Graph

The precision metric evaluates the algorithm's ability to avoid false positive predictions by measuring the proportion of correct positive predictions out of all instances predicted as positive. The following formula represented the precision in terms of the confusion matrix.

$$Precision = \frac{TP}{(TP + FP)} \tag{3.2}$$

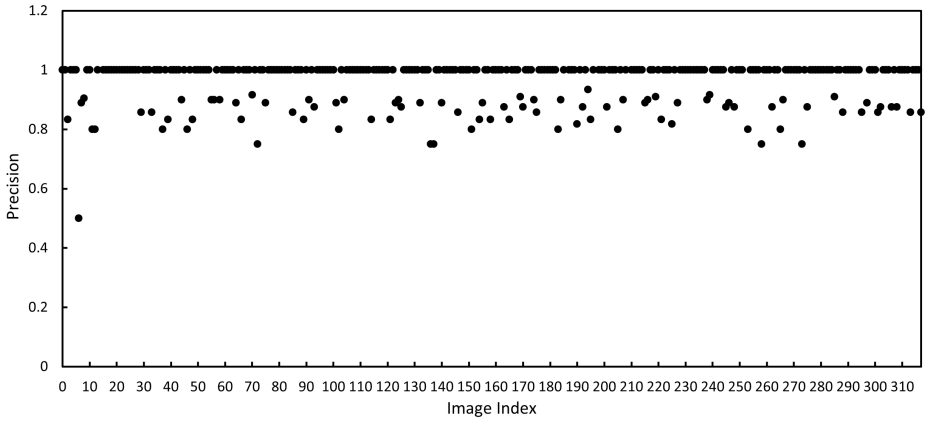

Figure 3.4: Precision vs Image Index Graph

As shown in the Figure 3.4. after implementing all three detection steps, the precision of all the samples was plotted and the average precision was taken as 0.9608 (96.08%). As a result, the occurrence of false positives was minimized and making it reliable in distinguishing positive instances from negative ones.

The F1 score, which is determined using the equation provided, represents a balanced measure of the algorithm's performance as it combines both precision and recall in a harmonic mean. These two metrics offer an overall assessment of the algorithm's ability to correctly identify positive instances while minimizing false positives and false negatives.

$$F1\_score = \frac{2 * (Precision * Recall)}{(Precision + Recall)} \tag{3.3}$$

The average F1 score was achieved as (0.9785) 97.85%, which indicates that the algorithm achieves a high balance between precision and recall. Here, recall for all the samples was calculated as one because there were zero instances in which the correct line segment was not detected. (Unless it was not passed from the feature extraction section to the classification section of the algorithm.)

As depicted in the Figure 3.5 higher F1 score was shown. It shows that the algorithm is performing well in terms of correctly identifying positive instances (exhibiting high precision) and minimizing false negatives (demonstrating high recall).

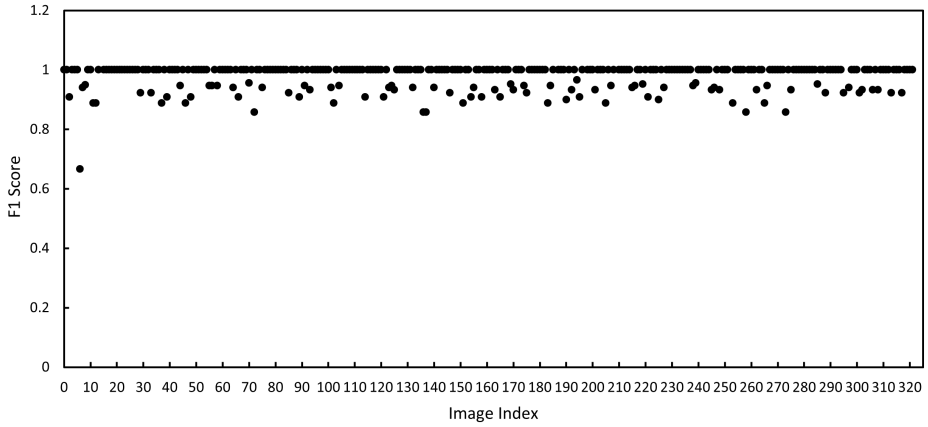

Figure 3.5: F1 Score vs Image Index Graph

The Receiver Operating Characteristic (ROC) curve was constructed by plotting the true positive rate (Recall) against the false positive rate (FPR) at various classification thresholds. For best-performing algorithms, the top-left corner of the graph would be touched by a ROC curve. Here it was achieved as shown in Figure 3.6

In summary, the algorithm's reliability and effectiveness in classification tasks are signified by the high F1 score, accuracy, and precision values. Furthermore, the

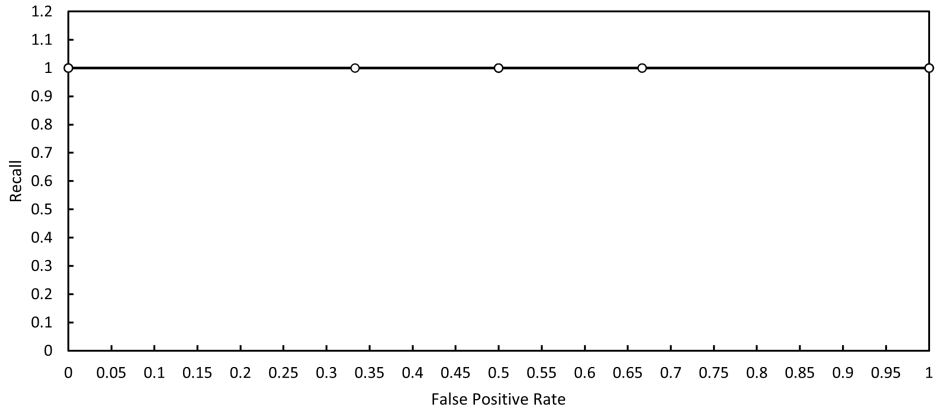

Figure 3.6: ROC Curve

perfect sensitivity suggested by the straight line in the ROC curve and the absence of false negatives further affirms the strong performance of the algorithm.

3.3 Efficiency Analysis of the Algorithm

The efficiency analysis of the algorithm is based on the processing time of the algorithm. As mentioned in the methodology, there are main three sections in this algorithm. But for the efficiency analysis, time spent on preprocessing and the feature extraction were combined into one single section and called “initial preparation time”. The time spent for the classification step was divided into three sections “single line detection”, “detection based on the size” and “detection based on the distance”. To measure the processing time, the Python library “timeit” was used.

Based on the results, the most time-consuming area in the algorithm was identified as the preprocessing and the feature extraction area. Initially, processing time was compared by changing the hardware configurations. For this, two completely different computers were used to run the program and capture the processing time as shown in the Table 3.2. 1st configuration was Intel Core i5 - 9300H CPU @ 2.40GHz Processor (9th Gen), 16.0 GB RAM, 512 GB SSD Harddisk, and 2nd configuration was Intel Core i5 - 4210U CPU @ 1.70GHz Processor (4th Gen), 8 GB RAM, 750 GB SATA Harddisk.

According to the Table 3.2, a huge contrast in the processing time was noticed. Usually, the CPU is responsible for executing instructions and performing calcula-

Configuration	Min Proc. Time (ms)	Max Proc. Time (ms)
1st configuration	66.42	716.73
2nd configuration	103.24	2078.21

Table 3.2: Performance of the Algorithm

tions. The 1st computer has a higher clock speed (2.40GHz) compared to the 2nd computer. A higher clock speed generally means faster processing, Therefore, one reason for the less processing time in the 1st computer is the performance of the CPU. In the computer architecture, RAM is used to store data that the CPU needs to access quickly. Having more RAM allows this kind of complex algorithm to be processed efficiently. The first computer has 16GB of RAM, which is twice the capacity of the second computer's 8GB. This means that the first computer can handle larger amounts of data in memory, potentially leading to faster algorithm execution. The storage drive, whether it's an SSD or an HDD, affects the algorithm's speed primarily during data read/write operations like loading JSON files, etc. SSDs are generally faster than traditional HDDs, offering quicker data access times. The first computer's 512GB SSD is likely to provide faster read/write speeds compared to the second computer's 750GB SATA drive. Therefore, the 1st computer was taken into further processing.

When further analyzing the preprocessing and feature extraction stages, a hypothesis was formulated to examine the impact of the number of neighboring plant section combinations (in a sample image) on the efficiency of the algorithm. The relationship between these two parameters was investigated using Big O notation, a mathematical notation commonly used in computer science. Big O notation, denoted as O(f(n)), where "O" represents the order of growth and "f(n)" denotes the algorithm's growth rate as a function of the input size "n," is used to analyze and compare algorithm performance as the input size increased. Typically, the function "f(n)" represents the worst-case time complexity of the algorithm, providing a clear indication of its efficiency with increasing input size.

Based on the algorithm's performance behavior in relation to the input size, various types of Big O notations have been defined. For instance, O(1) represents constant performance, O(log n) denotes logarithmic performance, and O(n) signifies linear performance. Considering the processing structure of this algorithm, it was theoretically assumed that the algorithm's Big O notation complexity should be linear (O(n)). To validate this assumption, a graph was plotted, as depicted in Figure 3.7, representing the algorithm's processing time against the number of neighboring plant section combinations. Then the graph was examined to identify patterns

in the growth of the processing time. Additionally, to compare the algorithm's performance, a processing time reduction technique (refer to subtopic 2.2) utilizing the external contour detection method(this is called "Optimized" graph in the Figure 3.7)instead of the internal contour detection method (utilizing the whole solid section of the plant) was used as shown in the Figure 3.7.

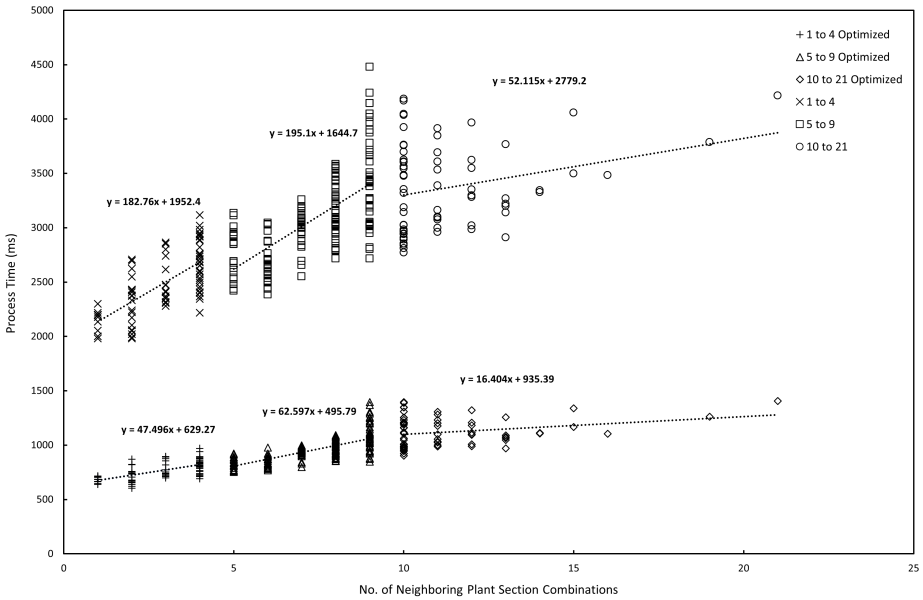

Figure 3.7: Process Time vs No. of Neighboring Plant Section Combinations

Observing Figure 3.7, significant differences in the growth rate of the processing time (gradient) were noticed as the number of combinations increased. The relevant Big O notation or the relationship between the number of combinations and the processing time was summarized in the Table 3.3. This finding is explained by the fact that the number of combinations increases in an image, the size of the plant, the plant's section size, and the size of the outside contour decreases. Therefore, less time is required for processing the contours. This is further explained in the following graph. (See Figure 3.8).

Range of Combinations	Relationship
1 to 4 Not Optimized	O(n) = 182.76n + 1952.4
5 to 9 Not Optimized	O(n) = 195.1n + 1644.7
10 to 21 Not Optimized	O(n) = 52.115n + 2779.2
1 to 4 Optimized	O(n) = 47.496n + 629.27
5 to 9 Optimized	O(n) = 62.597n + 495.79
10 to 21 Optimized	O(n) = 16.404n + 935.39

Table 3.3: Big O Notation ("O(n)" represents processing time and "n" represents No. of combinations)

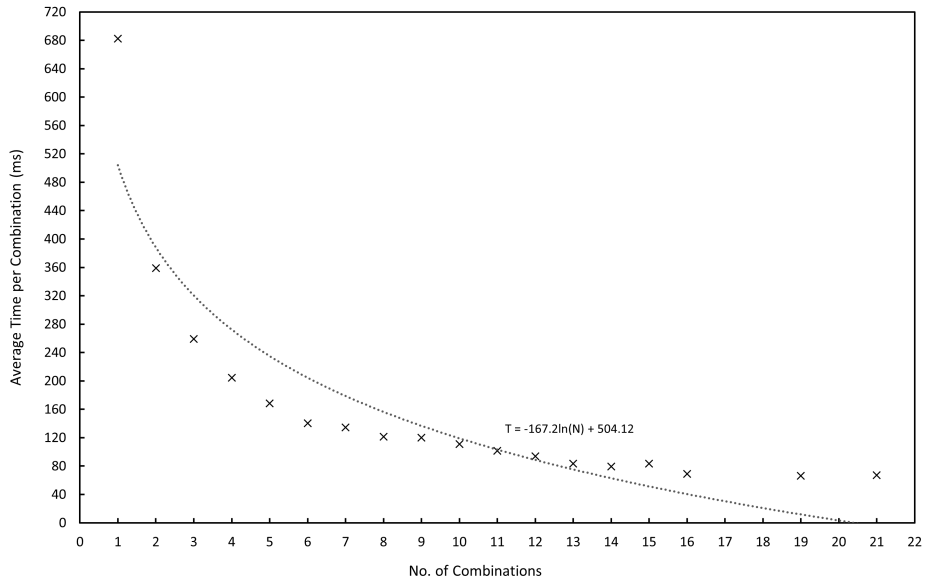

Figure 3.8: Average Time per Combination vs No. of Neighboring Plant Section Combinations

Also, a logarithmic relationship was found in the average time per combination vs No. of combinations.

"T" represents the average time per combination and "N" represents the number of combinations(N>0)

$$T = -167.2ln(N) + 504.12 \tag{3.4}$$

Also based on the optimized algorithm utilizing the external contour detection method, roughly 3 times the processing time reduction was achieved due to less

amount of pixels available on the external contour compared to the whole solid section of the plant.

A visual representation of single-line detection is provided in Figure 3.9, where the histogram illustrates a left-skewed distribution. This signifies that the range of time dedicated to detecting a single line is from 39ms to 239ms.

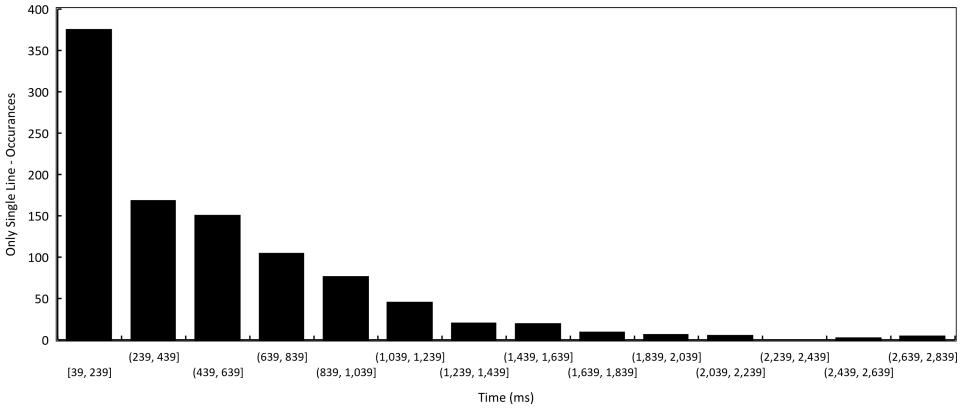

Figure 3.9: Histogram - Detection using Only Single Line

With respect to the detection based on size, as depicted inFigure 3.10, the time distribution follows a similar left-skewed pattern. Comparable to the previous instance, the duration for this type of detection also lies in a similar range, that is, between 41ms to 241ms.

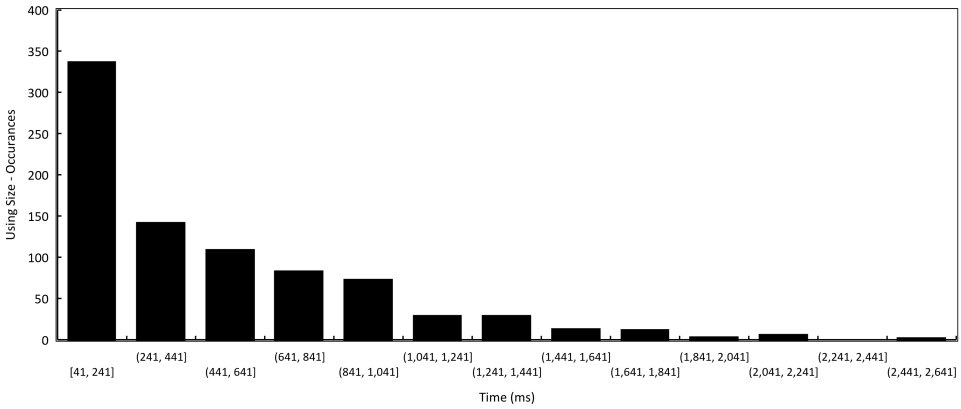

Figure 3.10: Histogram - Detection using Size

However, a deviation can be observed when we examine the time spent for detection based on distance, as represented in Figure 3.11. In this particular scenario, the duration is slightly increased, ranging from 286ms to 486ms, which is higher than the time required for the other two forms of detection.

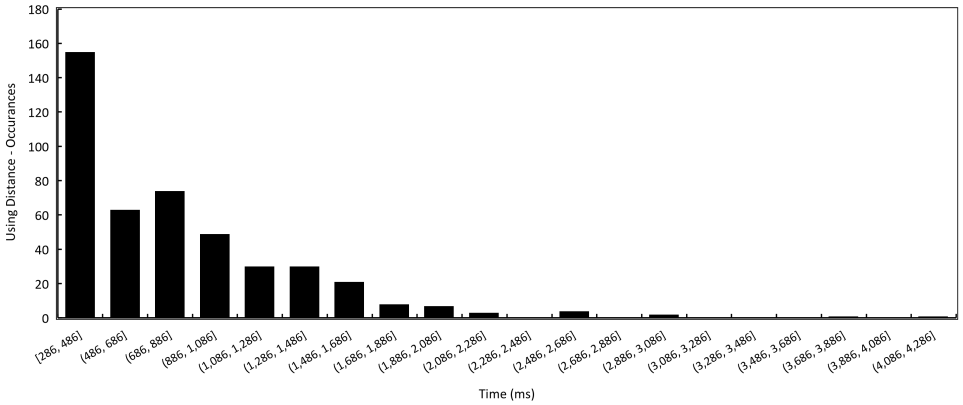

Figure 3.11: Histogram - Detection using Distance

Therefore, it can be interpreted from the histograms that while the time distributions for single-line detection and size-based detection are quite similar and relatively lower, the time required for distance-based detection is marginally greater.

It may be worth considering whether the increased time for distance-based detection is due to the complexity of the algorithm, or whether it could be optimized further. Understanding all of these factors could help to determine which detection method has to be improved in the future.

4 Conclusion and Discussion

This is under construction