Your Flowchart Secretary: Real-Time Hand-Written Flowchart Converter

Qian Yu*, Rao Zhang†, Tien-Ning Hsu‡, Zheng Lyu§
Department of Electrical Engineering
{*qiany, †zhangrao, ‡tiening, §zhenglyu} @stanford.edu

I. Introduction

Often when we are prototyping new ideas, a flowchart will be an ideal way to explicit our thoughts. However, it could be super time consuming to draw up a well-organized one onto our documentation files or presentation. In that sense, it would be really convenient if we can automatically generate the flowchart template with proper order informed from hand drawing chart. Here we developed an Android implemented application that can generate electronic version when scanning the flowchart. The application can be applied in many settings from discussion and notes in daily use to formal meetings in company or education institution. In addition, this product will be implemented in Android device for users convenience.

II. Related Works

Off-line interfaces is early common path to deal with hand-written flowchart problems. A method based on recognizing hand-drawn architectural symbols was discussed in [1]. The method operated on 300 dpi bitmaps. Their method realized a 85% recognition rates. Ramel, et. al proposed another approach to recognize handwritten shapes by using OCR system [2].

CLEF-IP competition rises the attention to flowchart recognition since 2012 [4]. Rusiol, et. al present a flowchart recognition method aimed at achieving a structured representation of flowchart in CLEF-IP 2013 [3]. The principal approach (as shown in Fig 1) for these work is to retrieve the structure with the nodes of the arrows or the end of lines.

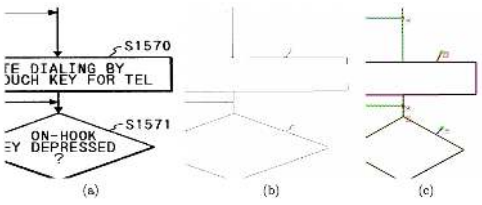


Fig. 1: Process of flowchart retrieval. (a) Origin flowchart, (b) node detection and (c) retrieved flowchart.

A recent patent reports a method for recognizing binary document such as table, pure text or flowchart [5]. The method first calculates the side of the image for top, bottom, left and right sides, then generates a boundary removal size for each side. After boundary removal for the four sides, it compares the side profiles to recognize the input document. Such method, however, cannot recognize the component within the flowchart.

III. Methodology

A. Assumptions

To make the objective achievable as course project, several assumptions are made and listed as below:

- Each components in the hand-written flowchart should be closed graph.
- The arrows in the curve must be straight.
- The shape that can be recognized is restricted to rectangle, arrow, diamond and circle.
- No intersections among arrows.

B. Pipeline

Based on the assumptions above, the algorithm and pipeline is illustrated in Fig 2. Details are explained in the following subsection.

1) Input Flowchart: The input of our pipeline is an image in .jpg format. This image can be loaded from Android device's photo library or detected from Android camera view. In this step, the image is also adjusted to a fixed size. Usually, we re-size the image for accelerating the following processing steps, since an extremely large high resolution image can take prohibitively long time.

2) Binarize: The input image usually has three channels, RGB. The first step in our pipeline is to convert this RGB image to a gray-scale image, as shown in Fig 3. After converting the image into gray-scale, we employ an Gaussian filter to remove the salt-and-pepper noise caused by camera. The improvements can be observed explicitly after an adaptive-threshold binarization operation. The filter size for this binarization is tuned carefully, because it is pretty sensitive to the change of contrast and luminance. Now, we have an image with black front-ground flow chart shapes and white background. The final operation in this step is a bit-wise inversion, to swap the front-ground and background colors. The results of binarization and bit-wise inversion are shown in Fig 4 and Fig 5, respectively.

3) De-noising: The de-noising operation is a little different from the one in the previous step. The "noise" in this step is the small areas that inappropriately generated by the binarization operation or caused by stains on the paper .Gaussian filter doesn't work again. Here, we remove all these small areas in another way. First, find all contours in the white fore-ground shapes. Second, fill black color into the contours with area smaller than the threshold we set. This procedure effectively

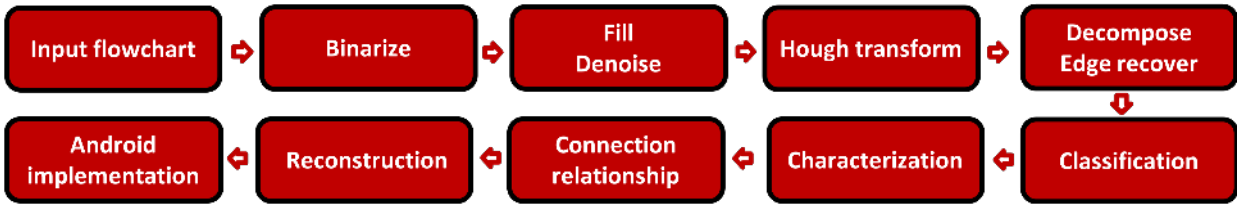
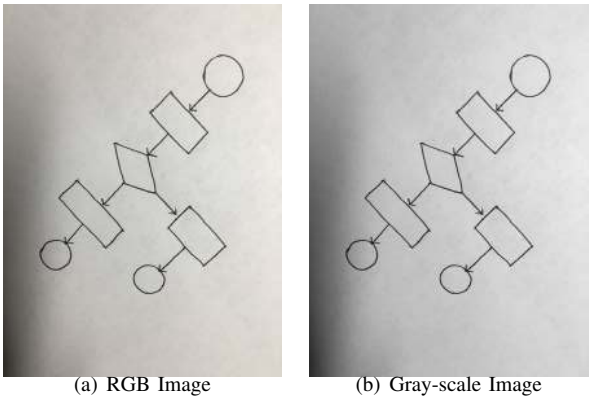
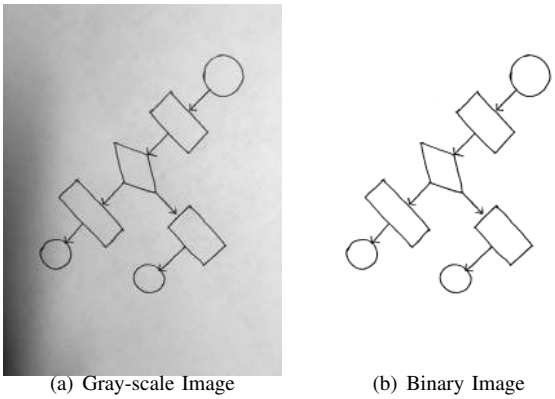


Fig. 2: Algorithm



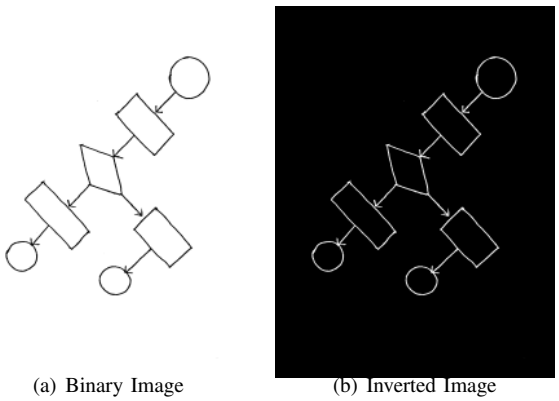
(a) RGB Image (b) Gray-scale Image

Fig. 3: Conversion from RGB to Gray-scale



(a) Gray-scale Image (b) Binary Image

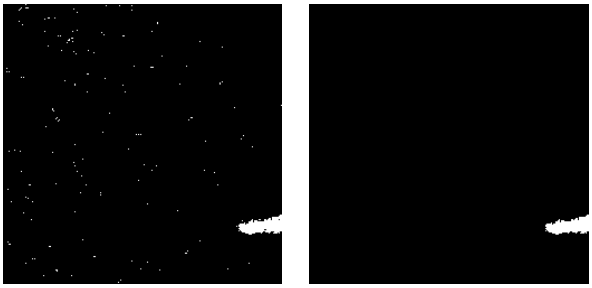
Fig. 4: Binarization



(a) Binary Image (b) Inverted Image

Fig. 5: Bit-wise Inversion

eliminates all the irrelevant small areas and leaves the flow-chart in the background. The effect of Gaussian filter is shown in Fig 6 and that of small region removal is shown in Fig 7. and small region removal are shown in



(a) Before Gaussian Filter (b) After Gaussian Filter

Fig. 6: Gaussian Filtering

(a) Before Small Region Removal (b) After Small Region Removal

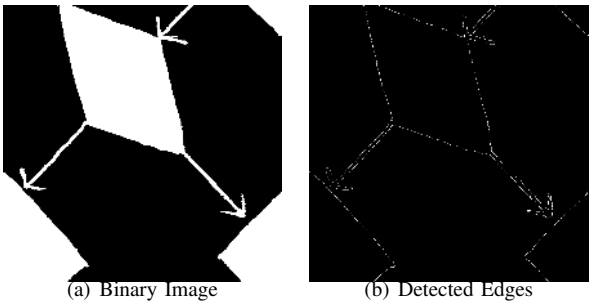
Fig. 7: Small Region Removal

4) Hough Transform and Rotation: In this step, we want to make the flow chart to stay horizontally or vertically instead of tilted. This operation will be beneficial in the shape recognition and classification. The way to implement this is first to detect all edges in the image, as shown in Fig 8. These edges will be analyzed by Hough transform to extract their angles. A histogram is generated then according to these angles. Among all the bins, the one with largest count is regarded as the primary angle. Thus, rotation is made based on this angle to reconstruct the correct direction for the flow chart. Here, we employ a two-step search to find the bin containing the primary angle. The first search is a coarse search with 180 degrees divided into 10 bins. the second search is a fine search with 18 degrees divided into 18 bins. In the example, we find the rotated angle is 45 degree clockwise, as shown in Fig 9. However, there is one more problem. The interface of

rotation provided by OpenCV doesn't keep the whole image after rotation, instead it keeps the size of the original image, therefore, the corners of the image are cropped. Hence, we need to enlarge and shift the image before rotating it. This involves some calculation of the new width, new height as well as an expected shift. With of loss of generality, let's assume the four vertices of this rectangular image are $(0,0),(x,0),(0,y)$ and (x,y). The clock-wise coordinate rotation formula is

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

After a rotation of angle θ, we can obtain the four rotated vertices. They are respectively $(0,0),(x*\cos\theta,-x*\sin\theta),(y*\sin\theta,y*\cos\theta)$ and $(x*\cos\theta+y*\sin\theta,-x*\sin\theta+y*\cos\theta)$. Now, we can clearly know that if we have a clockwise rotation (θ is positive), the new width of the rotated image is $x*\cos\theta+y*\sin\theta$, the new height is $y*\cos\theta+x*\sin\theta$ and the shift is $x*\sin\theta$ along y-axis. If the rotation is counter-clockwise (θ is negative), the new width of the rotated image is $x*\cos\theta-y*\sin\theta$, the new height is $-x*\sin\theta+y*\cos\theta$ and the shift is $-y*\sin\theta$ along x-axis. The effect of shift and rotation is shown in Fig 10.



(a) Binary Image (b) Detected Edges

Fig. 8: Edge Detection

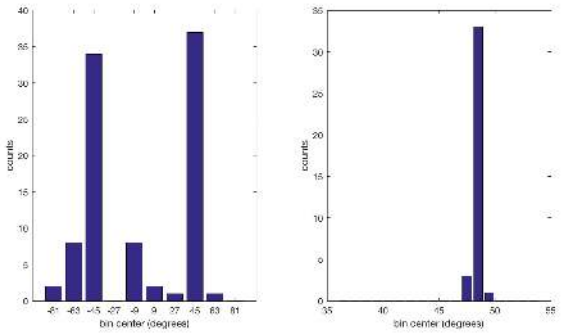
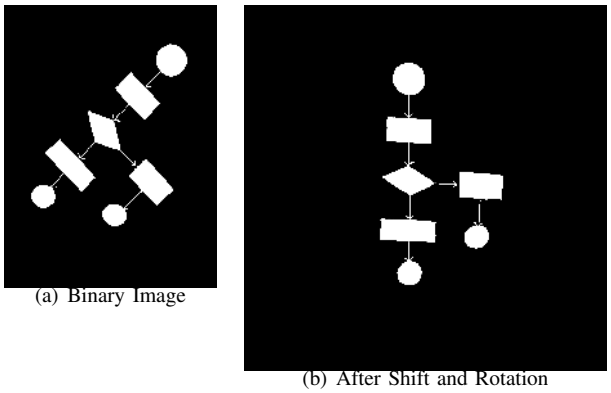


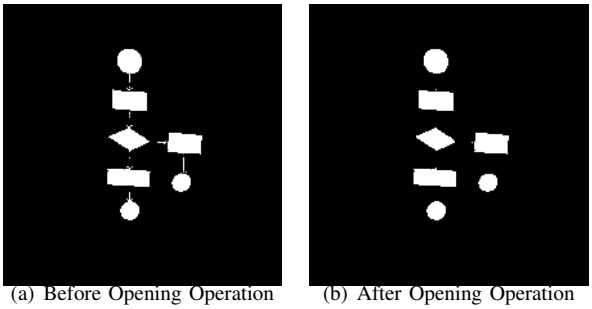
Fig. 9: Histogram of Rotated Degrees



(a) Binary Image (b) After Shift and Rotation

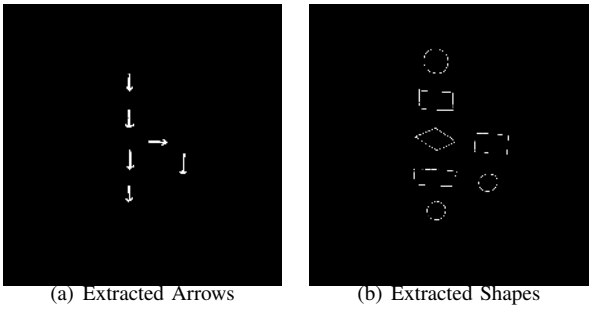
Fig. 10: Shift and Rotation

in the same way, thus we get the shapes. However, after subtractions we find that the corners of the shapes are left in the image due to the edge effect caused by open operation. Therefore, we perform small region removal algorithm one more time, to eliminate those small white areas with an appropriate threshold. The result is shown in Fig 12.



(a) Before Opening Operation (b) After Opening Operation

Fig. 11: Opening Operation



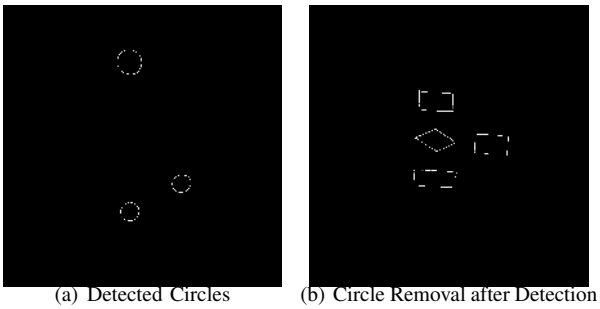
(a) Extracted Arrows (b) Extracted Shapes

Fig. 12: Extracted Arrows and Shapes

5) Decomposition: To decompose the flow chart, the key is to separate arrows from shapes in the chart. Since we have already filled the contours with white color, what is now in the image is only arrows and filled shapes. First, we apply an "open" filter to the image, which erodes the white area and remove arrows from the image, then dilates to recover the shapes to their original size, as shown in Fig 11. Second, subtract these shapes from the original image, thus we get the arrows. Third, subtract these arrows from the original image

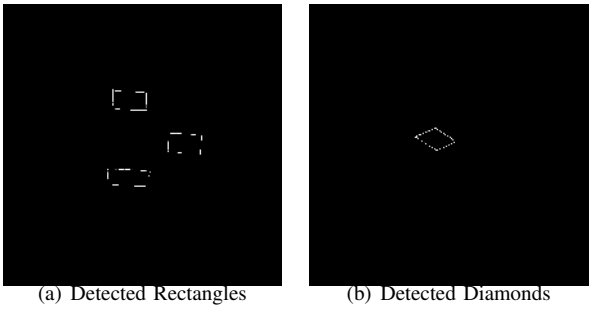
6) Classification: After all the components are disconnected after decomposition step, we found and labeled the connected ones to determine the regions of each shape. With *OpenCV.connectedComponentsWithStats*, we are able to label and get the centroids of the components, as well as the boundary box of each component. First, circles are differentiated from the other shapes by *OpenCV.HoughCircles*, as shown in Fig 13. Next, diamonds can be identified by comparing the ratio between the area of each shape and the area of the

boundary box. If the ratio is close to 1, this means the shape is a rectangle. On the other hand, if the ratio is lower than 0.75, the shape would be defined as a diamond. The distinguished rectangles and circles are shown in Fig 16.



(a) Detected Circles (b) Circle Removal after Detection

Fig. 13: Detected Circles and Circle Removal



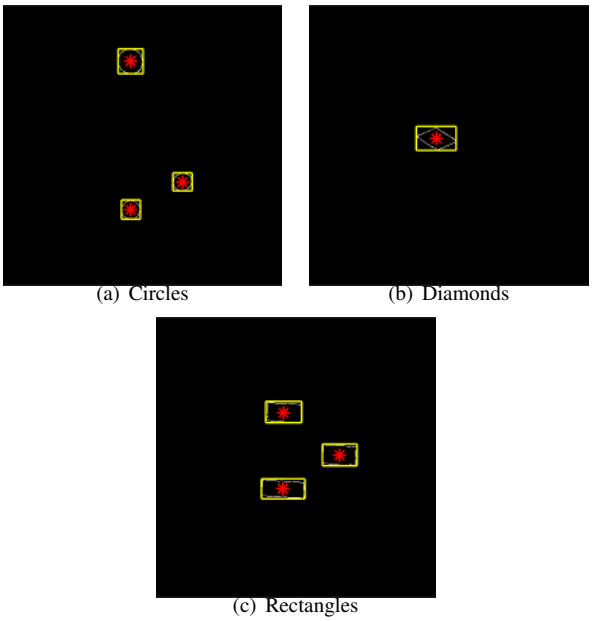
(a) Detected Rectangles (b) Detected Diamonds

Fig. 14: Detected Rectangles and Diamonds

7) Characterization: From last step, the OpenCV function *OpenCV.connectedComponentsWithStats* also returns the centroid and some statistical information as output. More specifically, the function would return the width and the height of the boundary box as well as the coordinate of the upper left corner of the boundary box. These attributes are preserved in an array for later use.

8) Connection relationship: The goal of this step is to determine the direction of arrows. Since the width, height and the upper left point of the boundary box is known, the center of the box can also be computed. By comparing the relative location of the center and the centroid of the shape, the corner that is closet to the centroid can be found. Based on the assumption that the arrows are straight, the closet corner can approximate the head of the arrow, and the diagonal corner would represent the tail of the arrow. These two end points would be passed to next step, with tails and heads labeled.

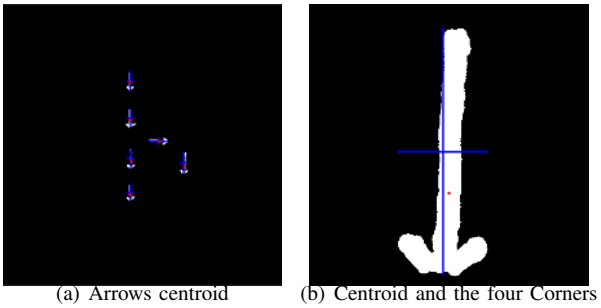
9) Reconstruction: In this step, we aim to find which shape for an arrow belongs to. For all diamonds, we set the four anchors on the corners of the shape. For all rectangles, anchors are set on the center of each side. Based on the two end points given from last step, we loop over all anchors, to find the closet anchor and attach the arrow on it so that the resulting flowchart would have a well-organized structure. This information collected so far would be utilized in the next section.



(a) Circles (b) Diamonds

(c) Rectangles

Fig. 15: Centroid and Boundary Box



(a) Arrows centroid (b) Centroid and the four Corners

Fig. 16: Arrow Direction Determination

C. Android Implementation

We encapsulate the interfaces Flowchart shape is an abstract class that all other child shapes extend with. Neighbors are the other shapes that the current shape connects to and the edges represent those connected edges. For each shape, it also has an attribute called anchors which stand for a list of points an arrow can attach to. Each shape has an abstract draw method that can be customized by each child shape. As shown in figure 17

```
class FlowchartShape {
    List<FlowchartShape> neighbors;
    List<Edge> edges;
    List<Point> anchors;
    abstract void draw(Mat mat,
    Scalar scalar, int thickness);
}
```

Rectangle, diamond and circle are three sub-classes that extend the Flowchartshape class. Each one implements helper functions and also identifies the anchor point position. When plotting the flowchart, we only need to call draw method on

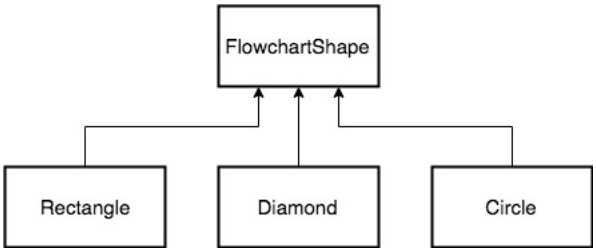


Fig. 17: Class hierarchy

each shape and this polymorphism makes the flowchart plotting very convenient.

The Graph class encapsulates the shapes and has its own draw method. In this draw method, it first draws all the shapes and then figures out how to draw arrows based on the graph data structure which is done by performing topological sort on the graph.

In order to show the real-time result while not blocking the camera frame, a separate thread is needed. When a new frame arrives, the main thread launches a background thread doing all the image processing and recognition. To avoid unnecessary computing overhead, only one background processing thread exists. Also, the result of previous processed frame is shown while the background thread is doing its job.

As mentioned above, the flowchart recognition algorithm depends on many parameters such as binarization threshold, small region removal area and threshold in Hough transform. Therefore, those parameters should be tuned in order to achieve ideal recognition result. However, this tuning becomes very hard when we try to adjust parameters in compile time because OpenCV library is very large which makes APK installation quite slow. Thus, we also introduce the idea that we can adjust parameters in run-time. In our Android implementation, we have three buttons at the bottom of screen as shown in the figure 18. The change button can switch among all those parameters and we can tune them by clicking plus and minus button. This is tuned in dynamic and can quickly show the processed result on screen.

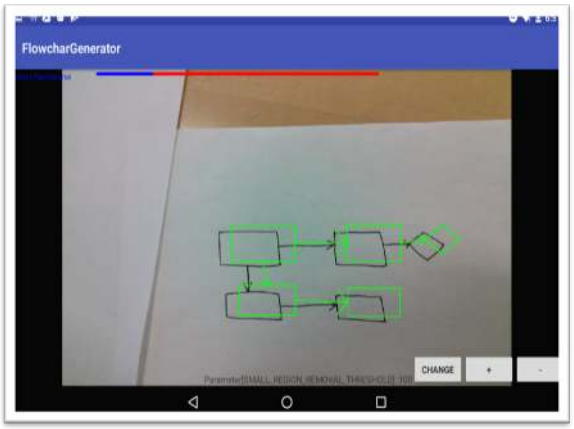


Fig. 18: Android implementation

IV. Conclusion and Future Work

A. Conclusion

Our flowchart recognition algorithm is able to detect rectangle, diamond, circle and arrow in a flowchart in real-time. It is very user friendly, interactive and easy to use.

B. Future Work

1) Orientation correction: One of very important assumption we made in the algorithm is that there should exist more rectangles in the flowchart because the correct orientation is achieved by finding the primary orientation component in the edges of rectangles. However, this is actually not generic enough in the real life conditions and would entail wrong recognition result in an extreme input. We also made an assumption that the arrow can point to any directions rather than only vertical and horizontal. This restrict us from doing correction through arrows. One solution to this problem is to make use of diamond to do orientation correction. Since the diagonals of a diamond should be perpendicular to each other, this can also indicate the flowchart orientation.

2) Denoise: In order to simplify the pre-processing step, we also made an assumption that the flowchart appears on a clean white paper without obvious noise. Although we have Gaussian filtering and small region removal in the algorithm, it is not enough to remove noise like book, pen or other real life objects. One easy solution can be implemented in the Android device side by asking the user to manually crop the ROI (region of interest). This can significantly improve the recognition accuracy while keeps the application user-friendly and interactive.

3) Testing: One big problem in our experiment is testing. Although we manually draw a lot of flowcharts and use this test set to evaluate the algorithm, it is not enough for our real-time application. This is because even if the algorithm passed all of the test cases, it is very possible that the algorithm will fail to recognize a flowchart in the real-time settings since lighting, focus and directions all vary. Thus, a more rigorous testing should be performed to make the algorithm much more robust. One very easy way is to do module or unit test. We can implement each of intermediate step in real-time, for example, only doing binarization and see if the result meets the requirement.

4) Export flowchart: Our Android implementation can recognize the flowchart and plot it on the screen. A further goal is to export the flowchart so that the user can actually use it in their writing. This can be done by using third party plot packages like python graphviz and saving as a png or jpeg file.

References

[1] Valveny, Ernest, and Enric Mart. "Deformable template matching within a bayesian framework for hand-written graphic symbol recognition." International Workshop on Graphics Recognition. Springer, Berlin, Heidelberg, 1999.

[2] Ramel, Jean-Yves, Guillaume Boissier, and Hubert Emptoz. "A structural representation adapted to handwritten symbol recognition." International Workshop on Graphics Recognition. Springer, Berlin, Heidelberg, 1999.

[3] Rusiol, Maral, Llus-Pere de las Heras, and Oriol Ramos Terrades. "Flowchart recognition for non-textual information retrieval in patent search." Information retrieval 17.5-6 (2014): 545-562.

[4] Mrzinger, Roland, et al. "Visual Structure Analysis of Flow Charts in Patent Images." CLEF (Online Working Notes/Labs/Workshop). 2012.

[5] Ming, Wei. "Method for recognizing table, flowchart and text in document images." U.S. Patent No. 9,858,476. 2 Jan. 2018.

V. Appendix

The break-down of work done by each student is listed as follows.

- Qian Yu designed data structure and realized Android device implementation
- Rao Zhang implemented the steps from image binarization (step 2) to shape classification (step 6).
- Tien-Ning Hsu collaborated work from shape classification (step 6) to component connection (step 8).
- Zheng Lyu collaborated work from shape classification (step 6) to component connection (step 8).