Hand-drawn Flowchart Template To Digital Flowchart Template

Abdulrahman Elgendy
McMaster University
1280 Main St W, Hamilton, ON, Canada
elgendya@mcmaster.ca

Abstract

Flowcharts are a very popular tool that have a variety of applications such as providing visual clarity when communicating different processes. Many people are more comfortable creating hand-drawn flowcharts as they are quicker to create compared to creating a digital version on a software tool. However, it is always a better idea to create digital flowcharts, as they can be saved, shared, and edited on different file systems. To minimize the time spent creating a digital flowchart, this paper proposes a methodology where the user can quickly sketch an empty template of the required flowchart, then take a picture of the hand-drawn template, and finally generate a digital version of the flowchart in an editable computer file format. The transformation is done by first binarizing the flowchart using adaptive thresholding and then extracting the basic shapes and arrows of a flowchart by doing basic filtering, finding contours, and applying morphological operations such as opening. The extracted shapes are then classified by training a SVM model using a dataset of hand-drawn geometric shapes that I created specifically for this task. Moreover, the arrows in the flowchart are analyzed using connected components analysis, and the connections between all the classified shapes are established. Finally, reconstructing a digital version of the flowchart will be done using a python library called Flowgiston which is used for easy flowchart development.

1. Introduction

When working on any project, people tend to quickly draw a flowchart on a piece of paper to organize their thoughts and determine the steps and methodologies needed to achieve their ideas. However, it could be time consuming to go through the whole process of replicating this flow chart on a software tool that will allow you to store the flowchart in your file system for documentation purposes. Therefore, the aim of this project is to allow the user to quickly create a flowchart template made up of shapes and arrows in their correct order on a piece of paper and then converting that into a digital template that the user can then start working on. This application could be useful in many settings, especially in brainstorming sessions where individual don't want to spend a lot of time creating a flowchart template on a software tool.

2. Related Works

Many approaches have been attempted so far for the generation of digital flowcharts from scanned hand-drawn flowcharts. Reference [1] proposed a flowchart secretary which transforms the outline of a hand-drawn flowchart to a digital version. It does so by trying to predict the geometrical shapes of the flowchart components using area of the shape relative to the area of the bounding box. However that method may not always work especially because the shapes are hand-drawn and are drawn differently by different people. This would affect the robustness of that method. Reference [2] proposed a Methodology to Rapidly Generate Documents from Hand Drawn Flowcharts. Their pre-processing involved binarizing the image using Otsu's method to determine an optimal global threshold value from the image histogram. This might work if the image is scanned. However, if the image was captured using a regular camera, then shadows might be present on the flowchart image and varying contrast levels might exist across different parts of the image. Therefore, global thresholding may not be useful in this application. Moreover, Reference [2] tries to predict the geometrical shapes using polygon approximation and then finding the number of vertices and polygon angle. This method might work well if the shapes are synthetically generated and perfectly symmetric. However, given the variability of the way people draw shapes and the added variability caused by morphological operations and other preprocessing steps, the number of vertices and angle may not always lead to the correct prediction. This paper will work on addressing these shortcomings.

3. Assumptions

To make this project achievable as a course project, the following assumptions will be made:

- This project focuses solely on creating a digital template. Therefore, all input flowcharts should not be filled with words.
- The shapes that can be classified will be restricted to hand-drawn rectangles, diamonds, circles, and triangles.
- The arrows in the flowchart should always be straight.
- There should be no intersections among arrows.

4. Proposed Method

Figure 1 below illustrates the outline of steps taken to achieve the objectives of this project given the assumptions mentioned above.

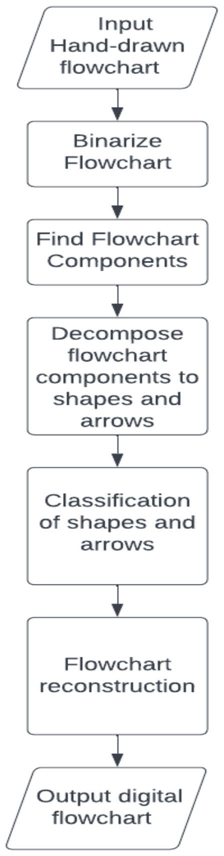

Input Hand-drawn flowchart
Binarize Flowchart
Find Flowchart Components
Decompose flowchart components to shapes and arrows
Classification of shapes and arrows
Flowchart reconstruction
Output digital flowchart

Figure 1. Methodology Overview

4.1. Pre-Processing

Pre-processing is an important step of this pipeline as it will help us setup the image for feature extraction and classification steps.

4.1.1 Input Hand-drawn Flowchart

The input to this pipeline is a camera image of a hand-drawn flowchart template. The image is read in as BGR and converted to gray-scale to prepare it for the next step.

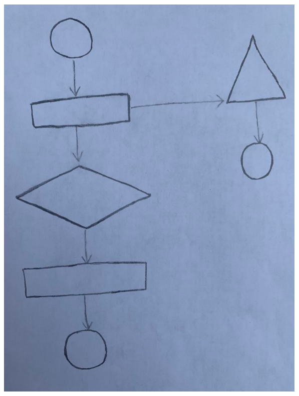


Figure 2. Input BGR Image

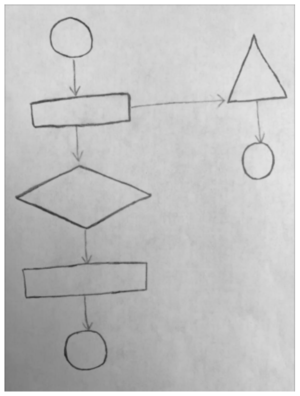


Figure 3. Gray-scale Image

4.1.2 Binarize Flowchart

This next step involves creating a binary image of the flowchart. This important as it will make the process of separating the flowchart components from the background an easier task. To do this, the input gray-scale image is first blurred using a gaussian filter, and then a thresholding technique will have to be considered to turn it into a binary image. Due to the shadows that might be present on the flowchart image and the varying contrast levels across different parts of the image, a global thresholding technique will not work satisfactorily. Therefore, adaptive thresholding was favoured over global thresholding for this application. After adaptive thresholding is employed, there might be some pepper noise left on the newly created binary image. Therefore, a median filter is used to remove the pepper noise. The final step is to swap the front-ground and background colors using a bit-wise inversion. The results of this section can be seen in Figures 4-7 below. This step would offer an improvement on what was proposed in reference [2].

4.1.3 Find Flowchart Components

Now that we have a clean binary image, we will be able to accurately separate the flowchart components from the background. To do this we start by finding the edges using

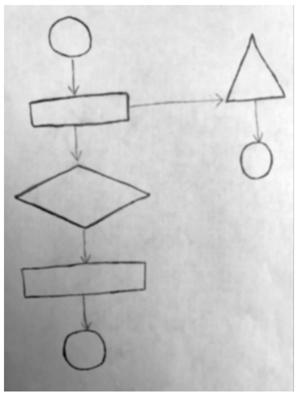


Figure 4. Gaussian Blur

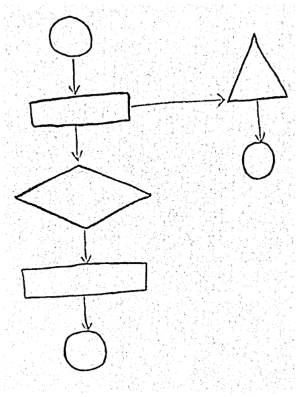


Figure 5. Adaptive Thresholding

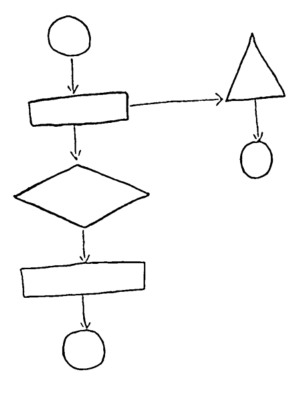


Figure 6. Median Filter

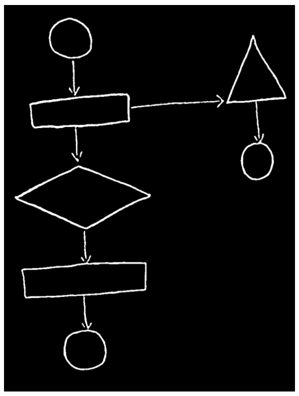


Figure 7. inverted binary image

the canny operator. Canny was selected to extract the edges as it produces smoother edges and produces less ”false” edges compared with other methods. It is important to detect the edges of the shapes as accurately as possible to produce more accurate results in the features extraction and classification steps. After extracting the edges, we find the contours of the image and fill all the contours with white. This will provide a clear separation between the flowchart and the background and will serve as an input to the morphological operations in the next step. The results of this section can be seen in Figures 8 and 9.

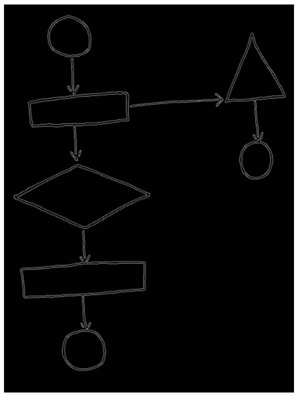


Figure 8. Canny Output

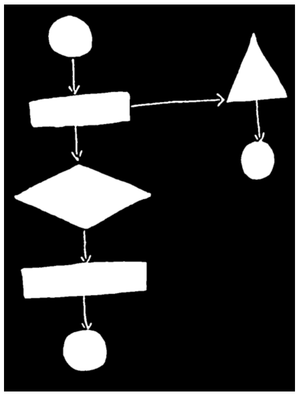


Figure 9. contours filled

4.1.4 Decomposing Flowchart Components

After completely separating the flowchart from the background, the next step is decompose the flowchart into shapes and arrows for further classifications in the future steps. The main idea here is that the arrows can be thought of as thin bridges connecting the different flowchart shapes to each other. Therefore, morphological opening will be used to remove the arrows and isolate the shapes. Now that we have the isolated shapes, we can subtract the isolated shapes from the original image to get an image containing the arrows only. Due to the arrows being very thin, we will dilate the arrows image to make sure that all the holes in the arrows are covered. The results of this section can be seen in Figures 10 and 11.

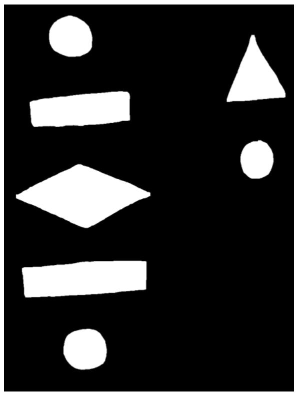


Figure 10. Morph Opening - Isolated Shapes

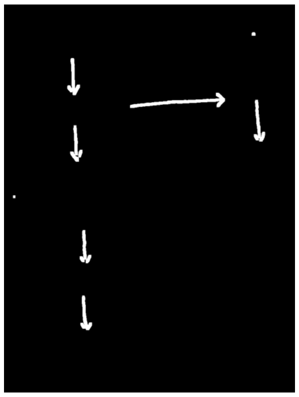


Figure 11. Dilated Arrows Only

Before moving on to feature extraction, we need to isolate and create a region of interest for every shape. To create these regions of interest, we need to find the contour for every shape in the image in figure 10 and create a bounding box for that shape. After that, the area covered by the bounding box is extracted and a padding of black pixels is added to the top, bottom, left, right of that bounding box so that the contour for the shape is clearly visible. This constructed image will constitute the region of interest for a specific shape. The process is then repeated for all shapes in the image. To maintain uniformity across the different shapes when extracting the feature vectors we need to make sure that all the constructed regions of interest are of the same size. Therefore the regions of interest are resized to 250x200 before extracting feature vectors. The results of this section can be seen in Figure 12 below.

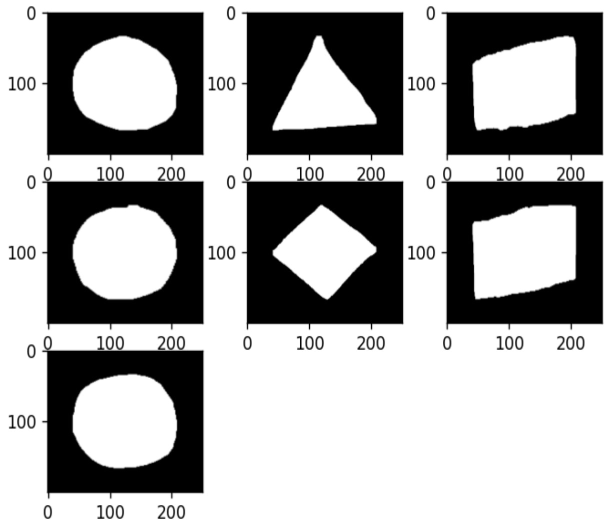


Figure 12. Extracted ROIs And Resized

4.2. Feature Extraction

Now that we have the isolated shapes and the arrows in a separate image, we need to extract a feature vector representative of the objects we are trying to classify with the aim of removing redundant and irrelevant features so that classification of new instances will be more accurate.

4.2.1 Hand-drawn Shapes Feature Extraction

The main shapes being classified in this project are circles, rectangles, diamonds, and rectangles. The outlines of these shapes are highly differentiable and HoG descriptors would be able to capture such outline information. HOG descriptor is computed by calculating image gradients that capture contour and silhouette information of gray-scale images. This HoG descriptor will make up the first part of the feature vector. More representative features could also be added to the feature vector. If we find the minimum enclosing circle for every shape, we can find the ratio of the (area of shape contour / area of minimum enclosing circle). This ratio would be different for different shapes. For example, for circular shapes, this ratio would be closer to one, whereas for a triangle shape, the ratio would be a lot less that that. Figure 13 gives a visual representation of that. This feature is also added to the feature vector, and we end up with a complete feature vector containing information that is very representative of the four shapes being classified. This feature vectors provides a more representative features about the hand-drawn shapes compared to what was offered in references [1] and [2].

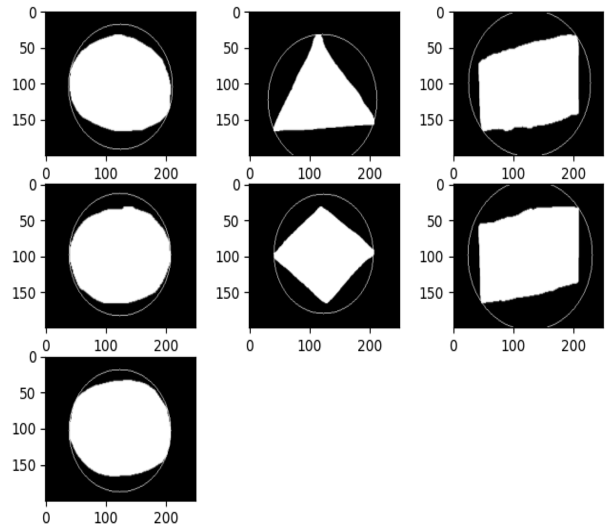


Figure 13. Minimum Enclosed Circle

4.2.2 Arrows Feature Extraction

To analyze the image containing all the arrows only, we need to iterate through every arrow individually and extract information with the aim of figuring out how the arrows are connected to the shapes in the flowchart. The main features that we are interested in for every arrow is an approximation of the coordinate of the head of the arrow and an approximation of the coordinate of the tail of the arrow. Every arrow can be looked at as a connected component since every arrow is a cluster of pixels with the same value, which are connected to each other through either 4-pixel, or 8-pixel connectivity. Therefore, we can find individual arrows by finding all the connected components in the image. After that, we need to filter out the connected components that may be noise and are too small to be arrows. For the remaining connected components, we need to create a bounding box for every arrow, and then find the centroid of every arrow. By looking at the general structure of an arrow, we can assume that the centroid is always closer to the head of the arrow compared to the tail of the arrow. Therefore, to approximate the coordinate of the head of the arrow, I found the bounding box corner point that is closest to the centroid. The corner point diagonal to that chosen corner point can be used to approximate the coordinate of the tail of the arrow. In figure 14, the red point represents the centroid, the yellow point is the center of the bounding box, and the green points are approximations of the head and tail coordinates.

4.3. Classifier

After deciding the structure of our feature vector, we need to extract feature vectors from a training set and then use them to train a hand-drawn shapes classifier. The out-

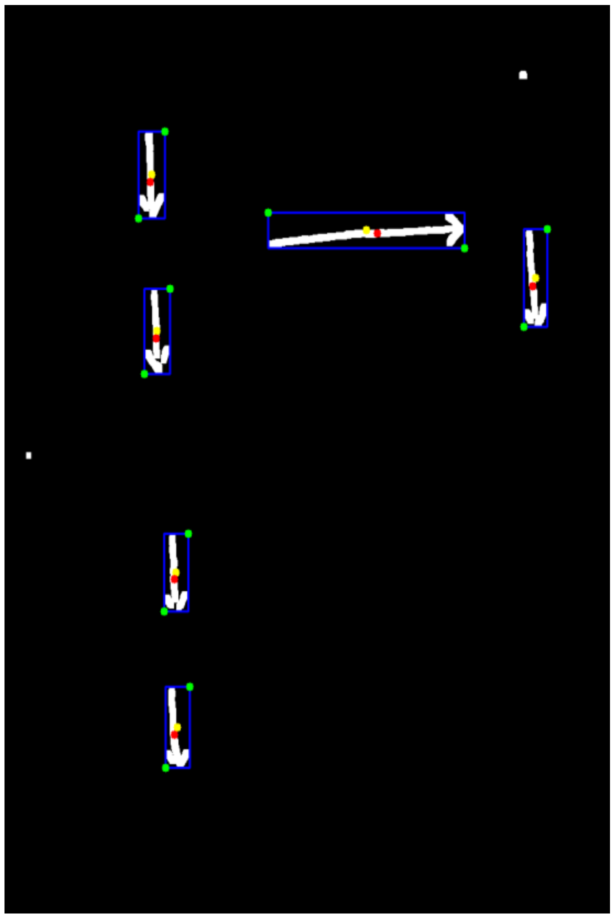


Figure 14. Arrows Features Extraction

put is its classification to one of the predefined classes (circle, diamond, rectangle, triangle). The classifier selected for this application was a SVM which creates a line or a hyperplane which separates the data into classes. Since there is a clear margin of separation between classes, SVM would work relatively well.

4.3.1 Dataset Creation

To train a classifier for this application, I tried to find a dataset of hand-drawn geometric shapes. However, I was only able to find datasets of synthetically generated shapes. The reality about hand-drawn shapes is that its very difficult to draw a perfectly symmetrical shape, and different people would draw the same shape in slightly different ways. Therefore, I decided to create my own dataset of hand-drawn shapes containing circles, diamonds, rectangles, and triangles to help with this application. The process of gathering this dataset involved asking five people to draw the four required shapes on a piece of paper. The drawn shapes were then filled with a distinctive color (pink) so that the shapes can easily be extracted when processing the data. Samples of these drawn shapes can be seen in figures 15-18. After that, a video recording is started on every drawn shape, where the camera is kept static and the paper is rotated so that the camera captures the shape at different orientations. These videos are all collected and their frames are pre-processed and added to the dataset.

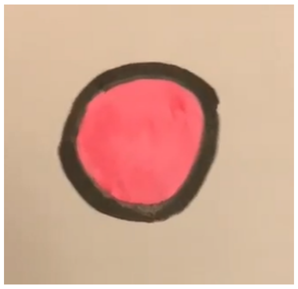


Figure 15. Sample Drawn Circle For Dataset

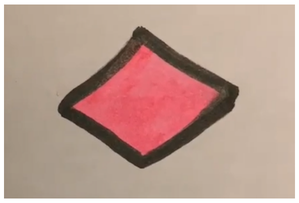


Figure 16. Sample Drawn Diamond For Dataset

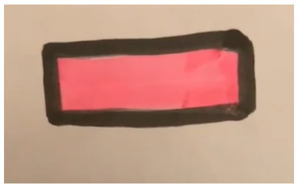


Figure 17. Sample Drawn Rectangle For Dataset

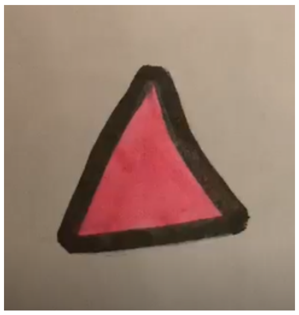


Figure 18. Sample Drawn Triangle For Dataset

The pre-processing involves converting the image to the HSV colour space and doing thresholding on the Hue channel to filter out any colours that are not pink. After that the image is binarized and contour of the image is found. This is followed by creating a bounding box around that contour. The area covered by the bounding box is extracted and a padding of black pixels is added to the top, bottom, left, right of that bounding box so that the contour for the shape is clearly visible during feature extraction and training. This constructed image will constitute an image in the dataset. The process is then repeated for all frames captured. Samples of final images in the dataset can be seen from figures 19-26 below. The created training set is made up of 2,083 circle images, 1,239 diamond shapes, 565 rectangle images, and 2,899 triangle images. This makes the training set a total of 6,786 images. To generate a test set that will be used to test the trained model, I asked five different people to draw the four required shapes on a piece of paper. The same procedure outlined above was used to process this data and create a test set. The created test set is made up of 389 circle images, 417 diamond shapes, 468 rectangle images, 327 triangle images. This makes the test set a total of 1,601 images. To label the dataset, all the data for separate classes were separated in different folders and labels are created for the data as they are being read from their respective directories during run-time.

Figure 19. Sample Rectangle In Dataset

Figure 20. Sample Rectangle In Dataset

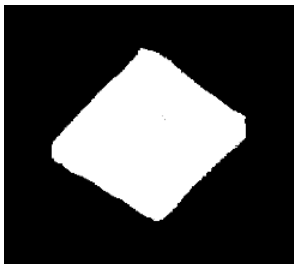


Figure 21. Sample Diamond In Dataset

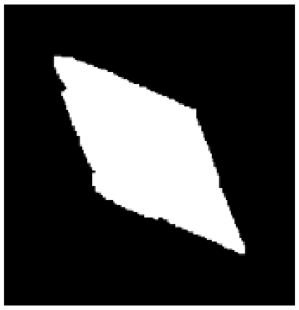


Figure 22. Sample Diamond In Dataset

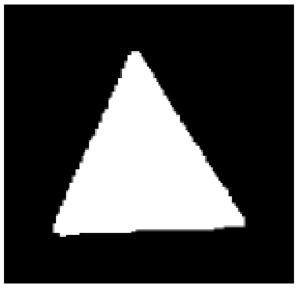


Figure 23. Sample Triangle In Dataset

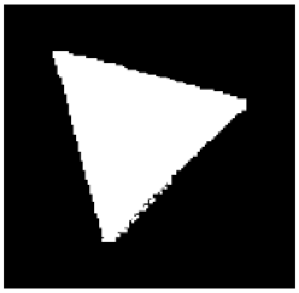


Figure 24. Sample Triangle In Dataset

4.3.2 Trained Classifier results

After using the training set to extract feature vectors and train the SVM, a model is generated which can be used to predict the shapes in a flowchart. Therefore, for the sample flowchart that we have been covering in this paper, we did pre-processing, extracted the shapes region of interest,

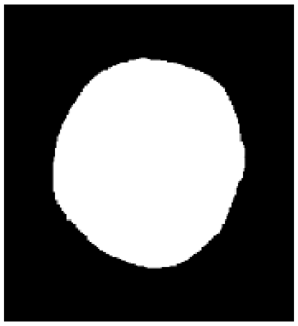


Figure 25. Sample Circle In Dataset

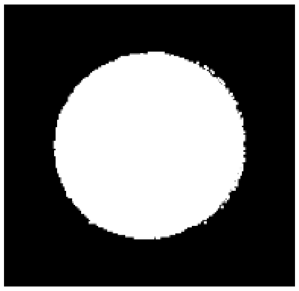


Figure 26. Sample Circle In Dataset

extracted a feature vector for every shape, and then finally passed the feature vector through the model to get the classification result. The classification output can be seen in figure 27 below. Moreover, the model was tested on the test set that was created specifically for this task, and the confusion matrix for this test set can be seen in figure 28 below. The generated metrics for the model after testing on the test set can be seen in figure 29. We can see the precision and recall for every class is really good with most of them being above 0.9. Moreover, this classifier provides a more robust solution to the one proposed in references [1] and [2] as it was trained on hand-drawn shapes by different people and shapes at different orientations as well.

4.4. Post-Processing

After classifying the shapes, and extracting the necessary information about the arrows, the next step would involve linking the arrows to their respective shapes and then digitally reconstructing the flowchart template.

4.4.1 Linking Arrows To Shapes

To link the arrows to the shapes, we iterate through every arrow individually and then we find the shape that is the source of the arrow and the shape that is the destination of the arrow. To find the source of the arrow, we calculate the distance between the approximated coordinate of the tail of the arrow and the centroid of all the shapes. The shape with the shortest distance to the tail of the arrow is considered to be the source of the arrow. The same logic is applied to finding the destination of the arrow. We calculate the distance between the approximated coordinate of the head of the arrow and the centroid of all the shapes. The shape with the shortest distance to the head of the arrow is considered to be the destination of the arrow. Figure 30 visualizes the output of this process. The red lines represent an arrow leaving the source shape, and the blue lines represent an arrow reaching the destination shape. After classifying the shapes and understanding the connections and relationships between the

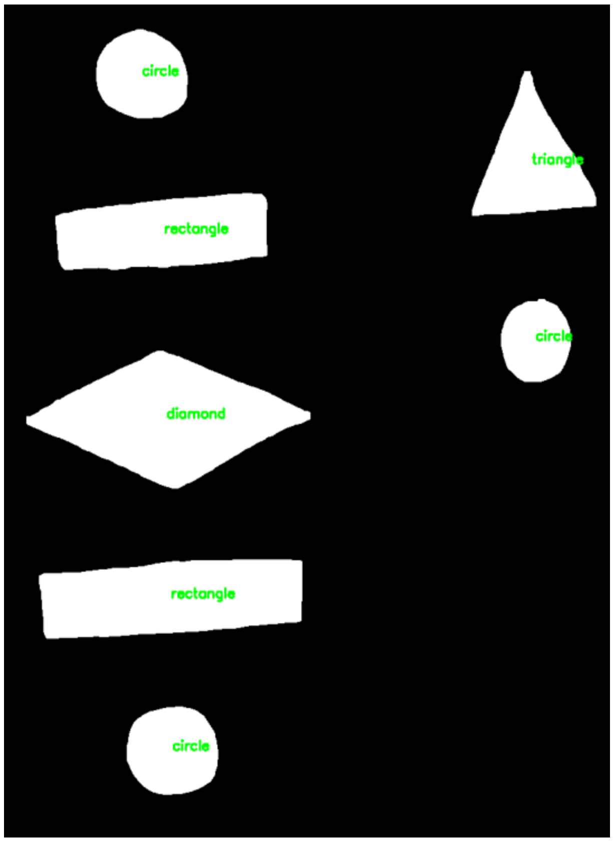


Figure 27. Shapes Classification

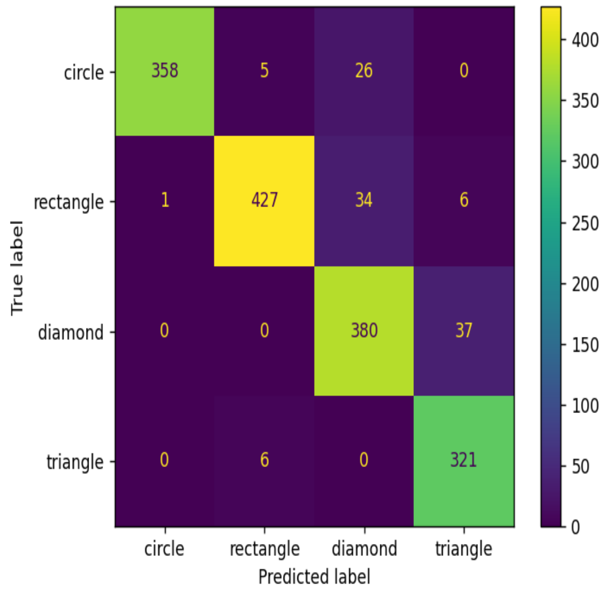
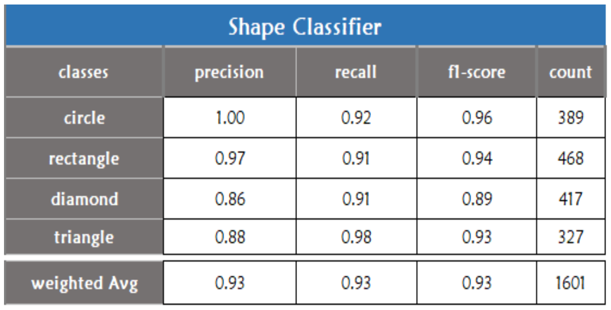


Figure 28. Confusion Matrix



Shape Classifier				
classes	precision	recall	f1-score	count
circle	1.00	0.92	0.96	389
rectangle	0.97	0.91	0.94	468
diamond	0.86	0.91	0.89	417
triangle	0.88	0.98	0.93	327
weighted Avg	0.93	0.93	0.93	1601

Figure 29. Model metrics

shapes, we can go ahead and digitally reconstruct the template.

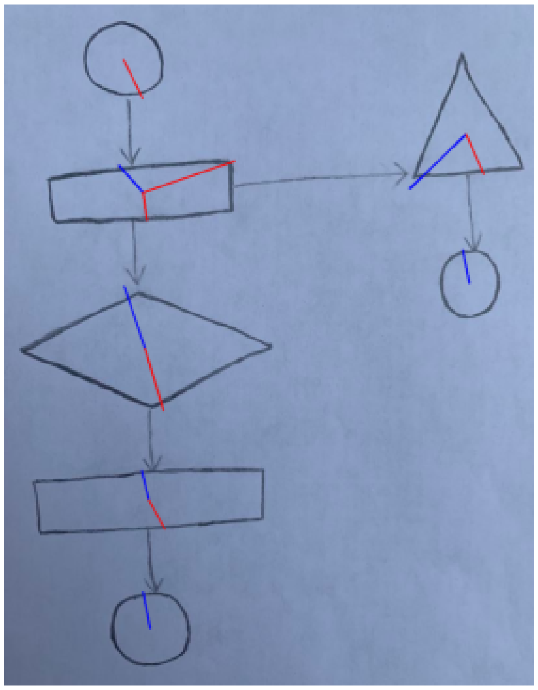


Figure 30. arrow links

4.4.2 Reconstructing Flowchart

To reconstruct the template, I will be using a python library called Flowgiston which is used for easy flowchart development in Python. I wrote a supporting class that takes in the classified shapes and their arrow relationships and does the required function calls in the Flowgiston library to construct the desired flowchart. The resulting reconstructed flowchart can be seen in figure 31. The position of the arrows and shapes may not be exactly the same as the original flowchart. However, the direction of the flow and arrows is still maintained.

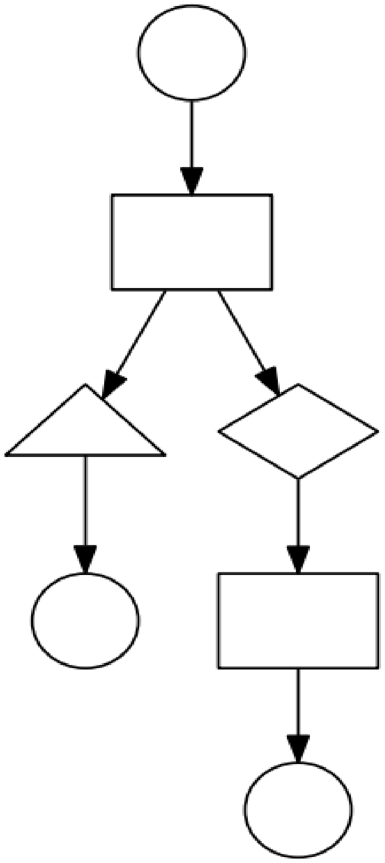


Figure 31. Reconstructed Flowchart

4.5. Conclusion

To conclude, this paper has successfully produced a robust methodology to recognize hand-drawn flowchart templates and transform them to digital versions. For future work, the proposed methodology can be improved by using a CNN and adding more complex shapes to the training set.

References

[1] Qian Yu, Rao Zhang, Tien-Ning Hsu, Zheng Lyu, ”Your Flowchart Secretary: Real-Time Hand-Written Flowchart Converter” ,2020.

[2] S. Chakraborty, S. Paul and S. M. Masudul Ahsan, ”A Novel Approach to Rapidly Generate Document from Hand Drawn Flowcharts,” 2020 IEEE Region 10 Symposium (TENSYMP), 2020, pp. 702-705, doi: 10.1109/TENSYMP50017.2020.9231033.