Transformation of a hand drawn flow diagram into a digital image

Tanya Bhadouria.
Computer Science and Engineering.
Sharda University
Greater Noida, India

Vidushi Parashar.
Computer Science and Engineering.
Sharda University
Greater Noida, India

Vikalp Arora
Computer Science and Engineering
Sharda University
Greater Noida, India

Abstract—**The goal of this work is to apply the concepts of perceptual grouping to digitize hand-drawn flow diagrams and render an aestheticized image made of polygons and connectors. We address typical imperfections in human drawing such as missed intersections of lines, overshooting of lines beyond the intersection point and curved line segments. Our method works on camera acquired images of flow diagrams and is able to handle illumination variations, shadows and speckle noise in the input. In this paper, we describe an algorithm to recognize all the types of polygons connected through lines. We have tried to redress many human errors and background noises. Moreover, we have digitalized hand written text and placed them into their respective positions. The scope of this algorithm is not only hand drawn flow diagrams, but it can also be modified to extend its application to architectural drawings, circuit diagrams, or may be a toddler friendly application to help them learn about polygonal shapes.**

I. INTRODUCTION

We often face many situations where we need to present our ideas in the form of flow diagrams, for they are the best representations of dataflow concepts. The easiest way to make a flow diagram is hand drawing on paper using a pen. Therefore, our aim is to process an image of a hand-drawn flow diagram of polygons and will convert it into an aestheticized digital flow diagram. We make use of the Gestalt principles of proximity, continuity, connectedness and closure to recognize polygons connected through lines. Our aim is to convert a hand-drawn image consisting of polygons connected through lines into a digitalised image, while removing the human errors and other kinds of noise. The main feature point of our algorithm is that it makes use of heuristics which reduces time complexity. Human errors such as broken edges, incomplete polygons (missed intersections of line segments), crossing edges (line segments overshooting past the intersection points) and somewhat curved lines which are intended to be drawn as straight lines. Discrete steps of the algorithm is shown in Figure 9.It also completes an incomplete polygon. Many times, while drawing a polygon in a hurry, we do not complete the figure, i.e. edges are open. Since our algorithm first breaks a polygon into set of edges, therefore this human error is automatically rectified. We achieved a high accuracy of 93correct rendering of the intended convex polygonal shapes on a dataset of more than 500 hand-drawn polygons. The main feature point of our algorithm is that it handles the problem heuristically, which reduces time complexity in comparison to the existing works based upon machine learning. Moreover, it handles many forms of errors and noise. Cluttered and nested polygons have also been worked upon successfully.

II. RELATED WORK

Previous research in hand-drawn shapes recognition is done by either online or offline approach. Very few research has been done in the field of offline approach. In this paper, we discuss about the offline approach

For off-line sketch interpretation Notowidigdo and Miller [1]developed UDSI (User Directed Sketch Information) which used heuristic approach to recognize 3 shapes (Circle, Rectangle and Diamond) and arrows. Shapes were recognized using corner detection, then a heuristic filter is applied to recognize unrecognized shapes (i.e. broken edges etc) and finally a greedy elimination algorithm is run which provides an effective filter among the false and true positives. But this work was limited to recognition of only 3 shapes.

For recognition of hand-drawn flowcharts Wioleta Szwoch and Micha Mucha [2], [3] created a Flow Chart Analyzer (FCA) system for recognizing, understanding and aestheti- cization of freehand drawing flow charts. The recognition algorithm counts similarity measure of a recognized figure and ideal patterns. If a recognized figure does not match with any ideal pattern, then it is considered as line. They used flowgram programming approach to create the flowchart. One of the limitation of this research is that the recognition is limited only to the given ideal polygons. Unlike [1] and [2] our algorithm can recognise every polygon.

Research on offline hand-drawn sketch using images of pen-and-paper diagrams is less common. Valveny and Marti [4] discussed a method for recognizing handdrawn architectural symbols using deformable template matching. They achieve recognition rates around 85%. For digitization of hand-drawn diagrams Regina Altmann [5]implemented generalised Hough Transform to detect the shapes in the flowchart. However the approach worked only for diagrams created with a digital image editor and did not give correct results for camera captured images of hand-drawn diagrams. Processing

time of this method was very large and memory requirement was high.

Perceptual organization has been widely used in computer vision to extract 2D and 3D structures and generate descriptions [6] , [7] . Cohen and Deschamps [8] used perceptual grouping to find a set of contour curves in 2D and 3D images. Their method could find new complete curves from a set of edge points. We adopt the same philosophy of perceptual organization and develop a computationally frugal method to identify hand-drawn polygons and connectors. The input diagram is broken down into line segments which are then grouped to form polygons. The detailed steps are described in Section III. Experimental results are given in Section IV and the paper is concluded in Section V.

Generation of Slides from Hand-Drawn Sketches (Muneeb Ahmed and Jeff Wheeler, 2014) used the same approach for shape recognition as used [9] by us. They separated every shape into line segments and later joined those line segments using heuristic to generate a digitalised image. But their algorithm rejected the open shapes, crossing egdes, broken edges and therefore is less robust. Whereas our research has covered all possible human error.

III. ALGORITHM

The algorithm is developed in Python using OpenCV libraries and various other image processing tools of Python. All the discrete steps of the algorithm have been explained in detail below.

A. Image Input

A web application implemented using Python flask takes an image as an input which has a flowchart drawn over it using hands. The image is captured using a smart-phone camera whose camera specification ranges from 8 mega pixels to 21 mega pixels. The software is designed to handle only completely focused images. This input image is rescaled down to width of 850 to improve the time of algorithm. Sample images can be seen in Figure 1 and Figure 2.

B. Image Binarization and Attenuation of noise

Beginning with the algorithm, we first binarize the rescaled image using Adaptive threshold with varying parameters suiting our image type. Adaptive thresholding takes into account spatial variations in illumination and thus eliminate the background noises, shadows and other factors that could degrade the algorithmic performance. Adaptive Binarization is a very critical step because it separates out the actual figure from the background noise and shadows, therefore allowing our algorithm to work on any kind of paper, whether it is of good quality or bad, ruled or blank.

Adaptive Thresholding leads to breaking of long edges and some irregularity in the figure, therefore, to make the curves smooth, morphological closing is done. To apply morphological closing, first the image is inverted so that the curves appear in white colour with black background. The morphological close operation is a dilation followed by an erosion, using the same structuring element for both operations. In our algorithm, we used 9x9 box structuring element. Output of this step can be clearly seen from Figure 2 to Figure 3.

Figure 2: Input image with noise

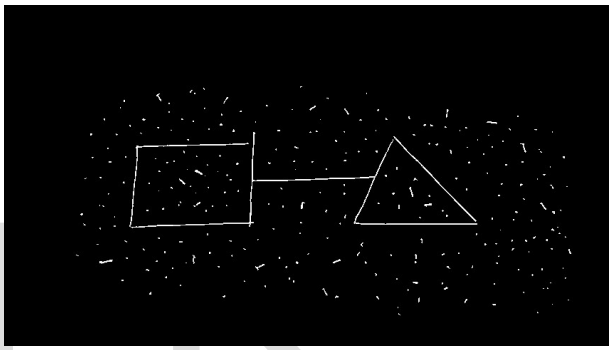


Figure 3: Inverted image with adaptive thresholding and morphological closing

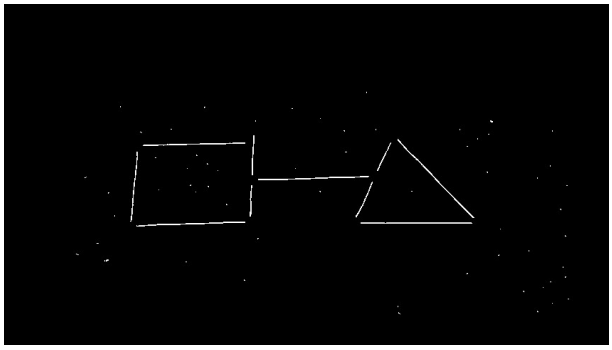


Figure 4: Final image with minimum noise and corner removed

c. Text region extraction

After Image Binarization and adaptive thresholding, text regions need to be extracted and put into hand written text recognition function. The process starts with finding the connected components of the image. Then each connected component is iterated once and bounding box of the connected component is calculated. If the bounding box is too small or large than a particular value, then that could be considered as shapes of flowchart, else it is the text region. Now that bounding box is cropped and feeded into hand written text recognition function.

D. Hand written text recognition

Hand written text recognition is done using SVM. This process consists of many parts. Firstly, we have to train the neural network on a very large data set, and secondly, we have to test it. We used NIST dataset for handwritten text to train the network. We used support vector machines (SVMs, also support vector networks) which is a supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. SVM classifier was used to train the network. After training, the network was tested on 10% of the dataset, which resulted in the accuracy of 97%.

Now after training of the network is done, we created a function that takes out input, pre-process it properly and finally feed it to the character recognizer function. We have used Scikit-learn tool for using SVM classifier. After the preprocessing, an image is given to the SVM classifier to extract features and using the training matrices, predict the character written there. Then finally the predicted character is given as an output.

E. Placement of text in the image

After the character recognition algorithm returns the correct character written, we place that character in a new blank white image exactly at that position. After that, we blackens the bounding box of that character from the main image, so that the image can be given later to the flow diagram aestheticization function. As you can clearly see in the Figure 9 (b), firstly characters are removed from the image and the left out image is processed later.

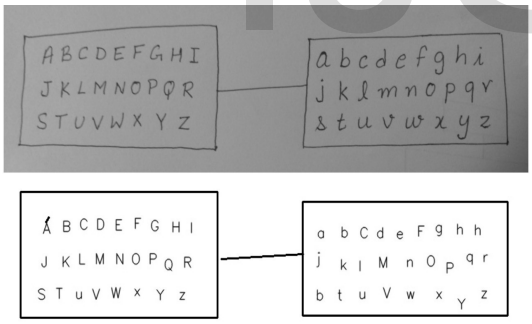


Figure 5: Text Recognition from the image

F. Corner detection in the polygons

This algorithm separates all the figures in a set of line segments and then using heuristics, joins them together for shape recognition. To detect the corners in the polygons and to separate out the connecting lines from the figure, we use Harris Corner detector. Corner is the intersection of two edges, i.e., it represents a point at which the directions of these two edges change. Therefore, the gradient of the image develops a high variation, which can be used to detect corners. OpenCV function for corner detection has many parameters. We have adjusted them according to our requirements, wherein we don't want any corners to be detected, if the angle between the edges is more than 150 degrees. After corner detection, that point is marked with black. Now, this point is enlarged, so as to disconnect the edges of the polygon and leave the whole image with line segments only as shown in Figure 4.

G. Line Detection

There are many ways to detect lines in an image. We started from the Hough line transform, but the results were not fruitful because Hough Line transform is able to detect only straight lines whereas our image has hand drawn lines which could range from being straight to trigonometric curves of sine or cosine. Therefore, we came up with an approximation algorithm which will approximate a line segment with a line equation.

For line detection, we have used contour analysis. Python OpenCV has a function cv2.findContours() which has many arguments like the image and the contour approximation method. For our purpose, we used cv2.CHAIN APPROX SIMPLE. It outputs all the contours in the image.

After contour analysis, all the contours are sent to a function which approximates these contours with a straight line segment. The function iterates through each contour and applies line approximation algorithm over it. Line approximation algorithm works as follows:

1) Use cv2.minEnclosingCircle() function over the contour to find the smallest circle that could enclose this contour completely. The function gives radius and centre of the circle as output. Logically considering the common geometrical deductions, the diameter of this circle will be the length of approximated line segment. Therefore, any circle with diameter less than a certain threshold value is considered as noise and cleared out.
2) Second step of algorithm is approximating the contour with a line equation. Python OpenCV function cv2.fitLine() takes contour value as input and outputs four points for the line equation.
3) Now, we have the equation of the minimum enclosing circle and the equation of the line. Using these 2 equations, we find the points of intersection, which are supposedly the end points of the required line segment, and save them in a list which stores all the line segments.

Please refer to Figure 5 for pictorial explanation. Output of this section is a list which has the equations of all the line segments. By equation, it is meant that it contains the end coordinates of the line segments.

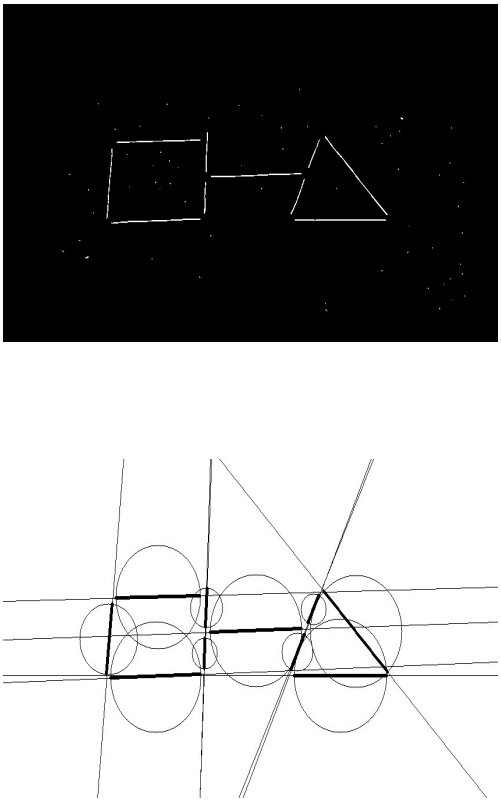


Figure 6: Algorithm for line detection

- The first image consists of 2 polygons connected through a line. After corner detection and removal, the image is left with only line segments.
- The second image shows how our algorithm works, by drawing a circle and an approximate contour with lines and getting it's intersection to draw the straight lines.

H. Merge 2 lines into a single line disconnected due to corner detection and Adaptive thresholding

The list of line segments is iterated and analysis is done over every pair of two different lines. Given 2 line segments, there could be total 4 combinations of distance between their end points. If the distance between any one of the four combinations is less than a particular threshold value (decided heuristically), then those 2 end points are taken into analysis. Angle between those 2 lines is calculated and if the value of acute angle is less than a heuristically decided threshold value, then those 2 line segments are merged together into a single line segment and the list of line segments is updated accordingly. The 2 line segments are marked using a hash array and finally removed from the list of line segments. Demonstration of this step is shown in Figure 6.

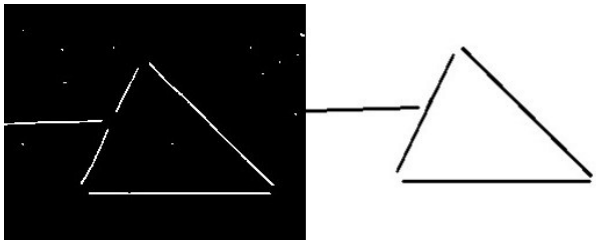


Figure 7: Broken/Discontinuous lines merged together

I. Join 3 concurrent line segments

The list of line segments is iterated and this time every triplet of 3 different line segments is considered. A total of 3 line segments means there would be 8 combinations of distances between their end points. Similar to the previous sub section, if the distance between any one of the eight combinations is less than a decided threshold value, then those 3 end points are merged together and this point of intersection of the three respective lines is shifted to the centroid of those 3 points.

J. Join the line segments to form a complete polygon

Point of intersection is calculated for every pair of line segments. Then the distance from point of intersection to the 4 possible pairs of points is calculated. If any of the 4 combinations gives fruitful value, then it can be inferred that these two line segments were actually intersecting in the original image. After that, as shown in Figure 7, the two end points near the point of intersection are changed into the point of intersection.

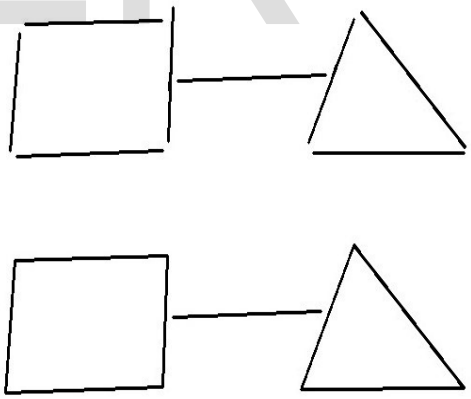


Figure 8: The algorithm iterates through every line segment and tries to make a closed polygon out of it.

K. Shape Detection

By the end of the previous subsection, we have a list of all the line segments without any noise modified such that all the discontinuous and intersecting lines are joined together. Now, for shape detection, we call a function that iterates through every line segment and for that particular line segment, find the next line segment joining it and continue to do the same. If for a given line segment, no such next line segment is found, it would mean that the shape is a straight line. We

implemented the shape recognition algorithm using cycle detection in a forest of graphs. Considering each line segment as a node in a graph where each node consists of 2 end points, we created a graph and used DFS (Depth First Search) using every node and if a node contributes to a cycle, then those nodes are removed. Basically after each iteration, if a cycle is found, then those set of lines are removed and saved into a list, and if no cycle is found, then that line is pushed at the back of the list, and again cycle detection algorithm is run. After few iterations, if no cycle is found, then the algorithm stops and return a list consisting of closed shapes, and another list consisting of remaining lines. Algorithm for shape detection is shown in Algorithm 1 and 2.

Algorithm 1 Shape Detection algorithm

procedure GETCLOSEDPOLYGON(*line list*) . line list is an arroy of line segments
polygon _list ← []. List of polygon identified
count ← 0 . Count the number of iteration of loops
while line list is not empty and *count* 6= *len*(*line list*) do . This loop choses first line of the line list and find if it is a part of a cycle or not
count ← *count* + 1
visited ← [] . visited bool array initialized to false
first ← *line list*[0]. first line in line list
vertices *first.x ,first.y* *first.x ,first.y* . Initialize a list of vertices for tracing a polygon
visited[0] ← *False* . Marking first line visited
RECURPOLYGON$(line_list, vertices, visited)$
check,vertices =
if *check* == *True* then
polygon _list.append(*vertices*) . Append newly found polygon
line list.remove(*vertices*) . Delete detected lines from line list
$temp = line\_list.pop(0)$
$line\_list.append(temp)$
$polygon\_list, line\_$
else. Remove the first line and push it in back
return *list* . line list contains remaining lines

L. Recognition of arrow heads

After the execution of shape recognition algorithm, we have a list that contains all the shapes and another list, that contains all remaining lines. Those remaining lines are considered as connecting lines and arrow heads. Now all these lines are again analysed and if any three lines intersect again at a single point, then that means 2 shorter line among those 3 lines are arrows. Therefore, that corner of line is considered as arrow head. As you can see in Fig 8, there is a line with a arrow head which is finally retained and get separated from other shapes.

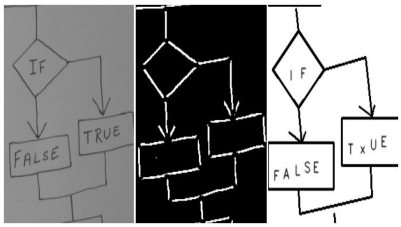


Figure 9: The image clearly shows how arrows are separated from the shapes and line incident on the rhombus is adjusted accordingly.

M. Shape Recognition

After the previous step, we have a list whose each element in turn has a list of points in anti-clockwise or clockwise manner depicting the vertices of a polygon. A list has 'n' pointstands for a polygon with 'n-1' sides. This step involves recognizing different types of polygon and coming up with a generalised algorithm to draw it. There are 4 functions made for drawing the polygon i.e. parallelogram, rectangle, 'n' sided polygon and a line, and 2 other functions to check if the quadrilateral is a rhombus or a parallelogram. The description of these functions is as follows:

1) Check rhombus function basically calculates the length of the diagonals and their slopes. If the length is nearly same, and angle between the diagonals is approximately 90 degrees, then the quadrilateral is a rhombus.
2) Check parallelogram function is called after the check rhombus function. Thus reducing the possibility of the quadrilateral being a rhombus. This function calculates the length of the diagonals, and if the length of one of the diagonal is greater than the other, then it is concluded to be a parallelogram.
3) Draw parallelogram function is called if check parallelogram approves that the quadrilateral is a parallelogram. This function first calculates the length and the values of slopes of both the diagonals. Then, taking the shorter diagonal as the primary diagonal to draw the figure, the function shortens the larger diagonal so that the parallelogram is parallel to the x-axis. Then we use Python OpenCV function cv2.line() function to draw line on the image.
4) Draw rectangle function is called if the the quadrilateral is neither rhombus nor parallelogram. Draw rectangle function is capable of drawing both, rectangles and squares. This function uses the same technique as the draw parallelogram function, i.e., it first calculates the length of both the diagonals and then takes shorter one to draw the rectangle. We know that if the rectangle is parallel to x-axis, given a diagonal, we can easily find the other two coordinates of the second diagonal. Following this, cv2.line() function is used to draw all the lines.

5) Draw regular polygon is a generalised function that can draw a regular convex polygon of any number of sides. If values of vertices are given in continuous way, it first calculates the centroid of the polygon using an algorithm. To draw a regular polygon, we need a centre as well as a radial distance at which all the vertices will exist. Therefore, average distance of all the current vertices from the centroid is taken as radius.
Now taking the first point as the point just above the centroid at a distance equal to the radius, we find out the coordinates of all the other points in clockwise manner to make a regular polygon. Using these continuous points, a regular polygon is drawn on the image.

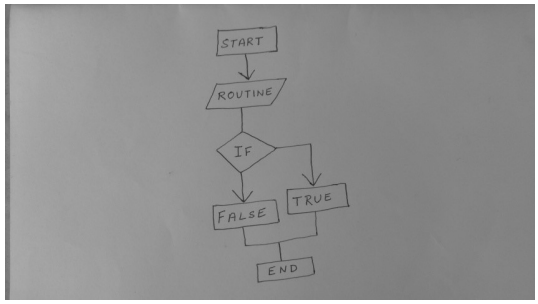
6) For drawing a line segment, we use an inbuilt function of Python OpenCV, cv2.line() which takes 2 arguments as starting and ending points of the line segment and an image on which the line has to be drawn.

Algorithm 2 Cycle Detection Recursion Function (DFS)

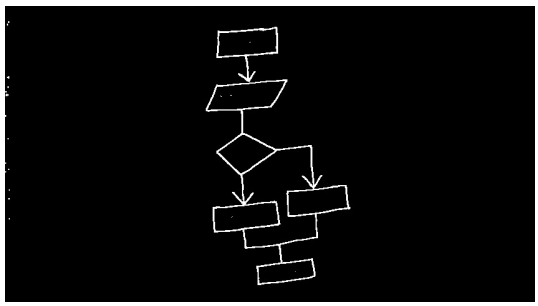
```
procedure RECURPOLY-GON(line_list, vertices, visited )
    next_vertex ← vertices[−1]   Last point of vertices for matching
    second check ← vertices[−2]   . Second last point for same line check
    for each line in line list do
    v1 ← [line.x1,line.y1] . Possbility of any two end points to match
        v2 ← [line.x2,line.y2]
    if next _vertex == v1 and second check 6= v2 then
    if visited[line] == False then   . If match found and it's not visited
    visited[line] ← True   . Mark the line visited
    vertices.append(v2)
    if vertices[−1] == vertices[0] then . If first and last points are same, then cycle is found
                return True,vertices
            check,list till = RECURPOLYGON(line list,vertices,visited)
                if check == True then   . If recursive function gives True, then that means the path made a complete cycle
        return True,list till
        else.   Backtracking
                vertices.pop()
            else
                return False,empty
        if next _vertex == v2 and second check 6= v1 then
    if visited[line] == False then.   . Similar steps as previous one
            visited[line] ← True
            vertices.append(v1)
            if vertices[−1] == vertices[0] then
                return True,vertices
            check,list till = RECURPOLYGON(line list,vertices,visited)
            if check == True then
                return True,list till
            else
                vertices.pop()
        else
            return False,empty
return False,empty   . If nothing match is found, then it will not make a cycle
```

N. Adjustment of edges according to the rendered shapes

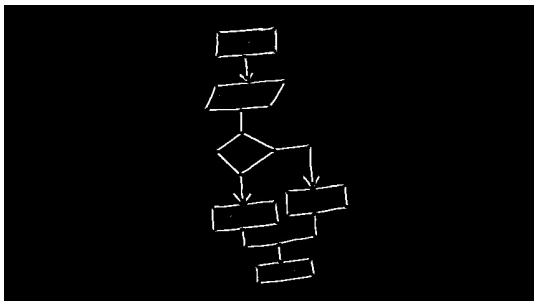
After recognition of different shapes and rendering it on a blank image, the edges incident on those shapes must be adjusted accordingly. As you can see in Figure 8, after rendering a perfect shape, the new shape is compared with the original one, and if any edge was incident on the original one, then the end points of that edge is changed according to the new shape.



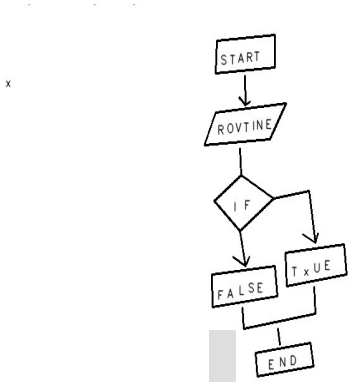
(a) Input image



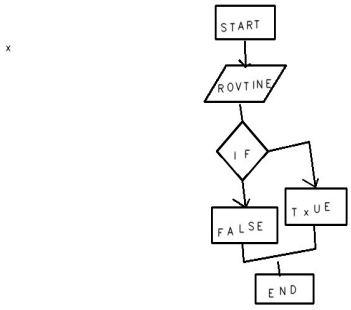
(b) Noise removal using Adaptive thresholding and Morphologicalclosing, Character detection and removal from the main image



(c) Corner detection and removal from the image



(d) Joining the line segments to complete a polygon and placement ofrecognized text back to the image



(e) Shape detection, recognition and drawing of perfect shape with adjustment of the connecting lines

Figure 10: Steps of the algorithm

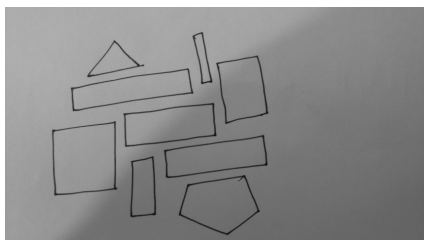
V. EXPERIMENTAL SETUP

In order to evaluate the recognition algorithm, we asked more than 100 subjects to draw 5 polygons of each type. The polygons were namely triangle, square, rhombus, rectangle, parallelogram, pentagon, hexagon and few shapes with a line connecting them. Each subject drew the figures on a blank white paper with blue dot pen. Nothing prior information was given to the subjects about the size or shape of the polygon for which our algorithm can work best. Subjects were told that the input is for shape recognition, therefore they did not draw any other non polygonal or open shape.

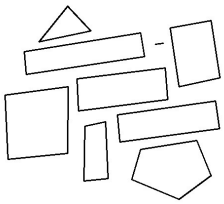
VI. RESULTS

Our algorithm is able to detect the closed polygonal shapes connected using a line, thus making an aestheticized figure of the hand drawn flowchart. It is also able to neglect some human errors, rectifying them and extraction of hand written text and recognition. Some of the salient features of the algorithm are as follows:

1) Using adaptive thresholding, morphological closing and ignoring small line segments, our algorithm is able to remove maximum noise and human error from the image.
2) It is able to extract and recognize hand written character upto an accuracy of 97%.
3) Our algorithm is able to separate the polygon from the connecting lines and detect them.
4) It also takes human error into consideration. If an edge is not continuous, it joins them together
5) It also completes an incomplete polygon. Many times, while drawing a polygon in a hurry, we do not complete the figure, i.e. edges are open. Since our algorithm first breaks a polygon into set of edges, therefore this human error is automatically rectified.
6) Sometimes, we mistakenly draw crossing edges in the figure. During edge detection, our algorithm divides a crossing edge into 2 parts, i.e. an actual edge and a small line segment that crossed the other edge. The small line segment is treated as noise and removed from the analysis.
7) Our algorithm is able to recognize polygons from clutter of shapes. Shapes close to another shapes are easily separated.
8) As shown in Figure 11 , our algorithm is also able to distinguish shapes nested into other shapes. For example, if a square lies inside a triangle, it is detected separately.
9) Our generalised algorithm can draw a polygon with any number of sides. It uses a formula to find centroid and takes mean distance of all the vertices from the centroid as radius to draw a regular convex polygon.

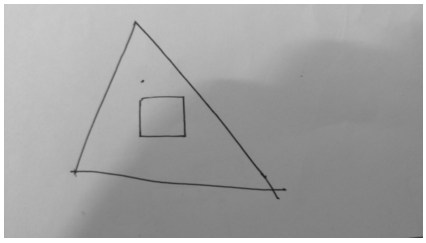


(a)Input test image with clutter

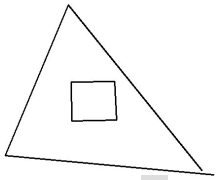


(b) Recognition of shapes in clutter

Figure 11: Clutter of shapes



(a) Input test image with square in side triangle



(b) Recognition of both shapes

Figure 12: Nested Shapes

10)As shown in Figure 12, it is able to distinguish between a rectangle, square, parallelogram and a rhombus from a given 4 sided polygon.

Our algorithm can easily detect an isolated ellipsoid figure, but an ellipsoid figure connected using a line with some other polygon is broken due to corner detection algorithm. Therefore, that figure is approximated as a line segment.

Of all the test cases given as inputs to our application, we could generate desired outputs for 93 percent of them. When considered specifically for each polygon, we could detect a triangle correctly 97% of the times, and differentiate between different quadrilaterals 93% of the times. Averaging out, it would not be wrong to stipulate an efficiency of 93% for this algorithm.

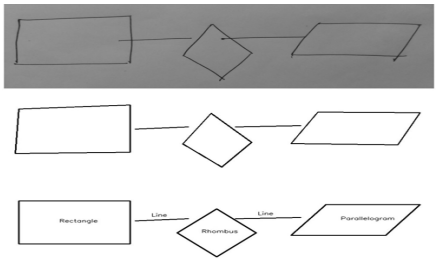


Figure 13: Different quadrilaterals

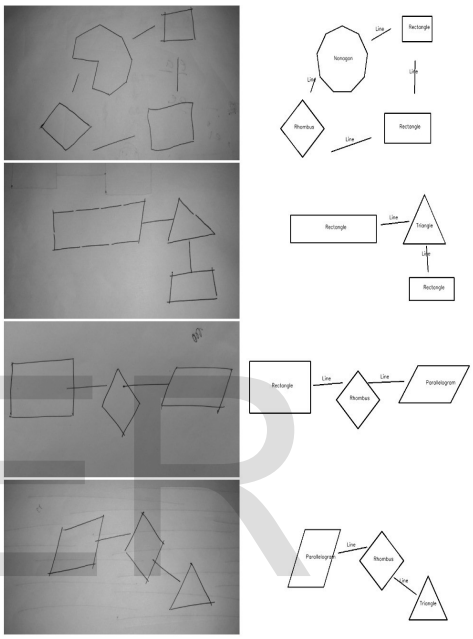


Figure 14: Different test cases worked upon

VII. Conclusion and Further Scope

We were able to implement the algorithm to detect polygons. However, our algorithm only detects a flowchart without ellipsoid shapes. Further works can be done to develop a new approach for detecting and recognizing ellipsoid shapes. Features like detection of cluttered and nested images can be used for further work in the digitalization of hand drawn architectural drawings.

Reference

[1] M. N. a. R. C. Miller, "“Offline-sketch interpretation,” in Making Pen-Based Interation Intelligent and Natural," *AAAI Fall Sym- posium, Menlo Park, California,* 2004.

[2] W. Szwoch and M. Mucha, " Recognition of Hand Drawn Flowcharts.," *Berlin, Heidelberg: Springer Berlin Heidelberg,* 2013.

[3] W. Szwoch, "Aestheticization of Flowcharts.," *Berlin, Heidelberg: Springer Berlin Heidelberg,,* 2008.

[4] E. Valveny and E. Marti, "“Application of deformable template matching to symbol recognition in hand-drawn architectural drawings,”," *Inter- national Conference on Document Analysis and Recognition,,* 1999.

[5] R. Altmann, "“Digitization of hand drawn diagrams,”," *Thesis, Aalto University School of Sicence and Technology, Helsinki, Helsinki,,* 2015.

[6] G. N. Khan and D. F. Gillies, " “Extracting Contours by Perceptual Grouping,”," 1992.

[7] R. Mohan and R. Nevatia, "“Using perceptual organization to extract 3-D structures,”," 1989.

[8] L. D. Cohen and T. Deschamps, " “Grouping connected components using minimal path techniques. Applications to reconstruction of vessels in 2D and 3D images,”," *Proc. Computer Vision and Pattern Recognition,,* 2001.

[9] Muneeb Ahmed and Jeff Wheeler, "Generation of Slides from HandDrawn Sketches," *Stanford University,* 2014.