Solving Word Jumbles

Debabrata Sengupta, Abhishek Sharma
Department of Electrical Engineering, Stanford University
{ dsgupta, abhisheksharma }@stanford.edu

***Abstract—* In this report we propose an algorithm to automatically solve word jumbles commonly found in newspapers and magazines. After pre-processing, we segment out the regions of interest containing the scrambled words and detect the circles indicating the positions of the 'important' letters. Then we extract the text patches and use an OCR engine to identify each letter and unscramble them to create meaningful words. The entire system has been implemented as a mobile application running on an Android platform.**

Keywords- circle detection; image binarization; OCR

I. Introduction and Problem Formulation

In our daily lives we often encounter puzzles in the form of word jumbles in newspapers and magazines. In the absence of solutions, being unable to unscramble one or two words can be very nagging for many of us. The motivation for our project stems from the idea that if we could build a mobile app which could assist the user in solving these word jumbles from an image of the puzzle, it would make life so much more awesome!

Figure 1 shows a sample word jumble which might commonly be found. We envisage that our app would be able to do the following : (i) automatically identify the patches of scrambled letters, (ii) detect each individual letter and then unscramble them to form meaningful words and (iii) identify the circled letters which are needed to form the 'mystery answer'. We do not however intend to provide the 'mystery answer' since this would spoil all the fun and make the entire puzzle meaningless.

After exploring a lot of word jumbles commonly found, we concluded that most of them have four scrambled words and a picture clue to obtain the mystery answer. The positions of important letters are indicated by circles. We have designed our algorithm assuming that the letters are placed in boxes (as shown) and are all in uppercase. We do not require each individual letter to be within a separate box. As is commonly the case with these jumbles, we also insist that each group of letters form a unique word when unscrambled.

The rest of this report is organized as follows. In Part II we provide the block diagram and explain the basic building blocks of our algorithm. Part III provides some insight into the Android implementation of our algorithm. Some observations about our algorithm are mentioned in Part IV. Finally, Part V summarizes our results and lists possible improvements to our application.

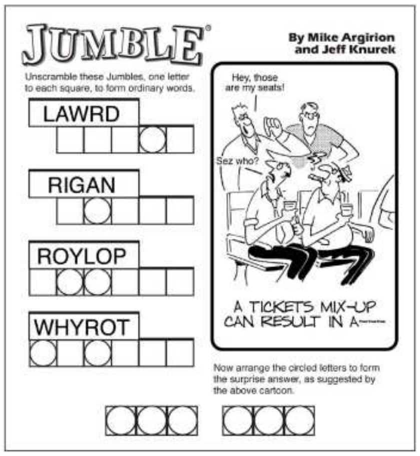

JUMBLE®
By Mike Argirion and Jeff Knurek
Unscramble these Jumbles, one letter to each square, to form ordinary words.
LAWRD
RIGAN
ROYLOP
WHYROT
Hey, those are my seats!
Sez who?
A TICKETS MIX-UP CAN RESULT IN A—
Now arrange the circled letters to form the surprise answer, as suggested by the above cartoon.

Figure 1 : A sample word jumble

II. Our Algorithm

The block diagram for our algorithm is shown in Figure 2 and details of individual phases are discussed below.

A. *Pre-processing*

Once we have obtained the image, we convert it to gray-scale and use this image in subsequent steps. As the first step in pre-processing, we convolve the image with a gaussian low pass filter of size 5x5. This smoothes out the image and helps in removing a lot of noise which subsequently helps in obtaining cleaner edges. This step is crucial to our algorithm since we do not want any false positives when we do circle detection later on to detect the 'important' letters.

B. *Segmentation of text patches and removal of clutter*

Our next goal is to automatically segment out every jumbled word and its associated set of rectangles and circles. In order to do this, we use the following steps :

- Perform an edge-detection on the image using a Canny edge detector. We choose a canny edge detector over other detectors since it gives a better edge localization and also gives us two thresholds to play around with, tuning them according to our present requirements. We chose the thresholds such that all rectangles in the image are completely detected while reducing clutter at the same time. In some cases, the contour of the rectangle may be broken. Since we need the complete contour to be detected, we perform a closing operation with a square structural element to merge all discontinuities before moving on. Results are shown in Figure 3(b).

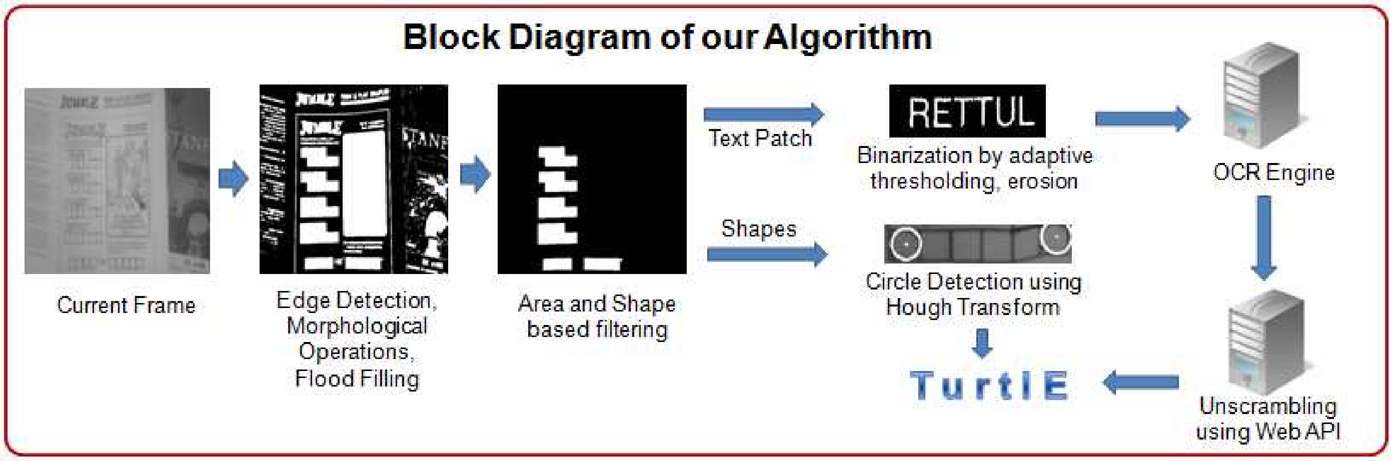

Block Diagram of our Algorithm
Current Frame
Edge Detection, Morphological Operations, Flood Filling
Area and Shape based filtering
Text Patch
RETTUL
Binarization by adaptive thresholding, erosion
OCR Engine
Shapes
Circle Detection using Hough Transform
TurtlE
Unscrambling using Web API

Figure 2 : Block Diagram

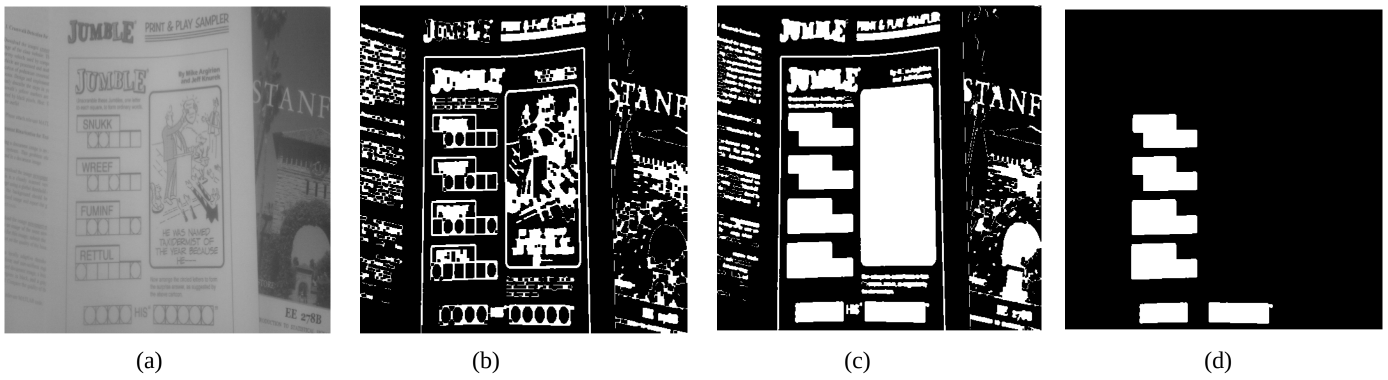


Figure 3 : (a) Image obtained using a mobile phone's camera. The illumination and quality of image is quite poor and there is additional background clutter. (b) After edge detection and morphological closing using a square structuring element (c) After flood filling to fill up holes and create 'blobs' (d) Relevant blobs detected after filtering out the rest.

- The next step was a flood filling operation performed on the edge map obtained above to completely fill all holes in the image. If we can ensure that the contours of all rectangles are completely detected in the previous step, then flood filling produces blobs at all the text patches along with some additional filled areas corresponding to the big rectangle containing the picture clue and the word 'Jumbles' at the top. Results are shown in Figure 3(c).
- Having produced blobs in the previous step, we need to identify only those blobs which correspond to the scrambled letters and remove all remaining clutter. In order to do this, we use the following heuristics which we developed based on observing the general structure of many such puzzles : (i) The box containing the picture clue is always the largest blob detected in the entire image (ii) The top-left corner of all blobs corresponding to the letters are below the top-left corner of the largest blob (corresponding to the picture clue) (iii) All blobs corresponding to the letters are rectangles that are horizontally aligned. Using these heuristics and additionally rejecting all blobs with an area less than some fraction of the largest blob, we are able to isolate only those blobs which are of interest to us. Results are shown in Figure 3(d).

C. *Shape Detection*

Having detected all the relevant blobs, our next task is to detect the positions of circles in order to identify the letters which will be required to obtain the mystery answer. We crop out only the lower half of the bounding box of each blob and use it to detect the relative position of circles. We produce an edge map for this patch using a canny edge detector and use this binary image for shape detection. The circles are detected using a Hough Transform. Exploiting the symmetry in the image, we consider only those circles whose diameter is approximately 1/6th the width of the patch. In order to further restrict false positives, we also require that the centers of adjacent circles be separated by a distance approximately equal to 1/6th the width of the patch. Figure 4 shows the circles detected in one such patch, overlaid on the original image.

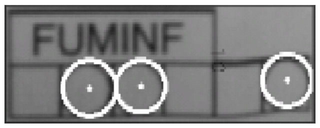


Figure 4 : An example of circle detection

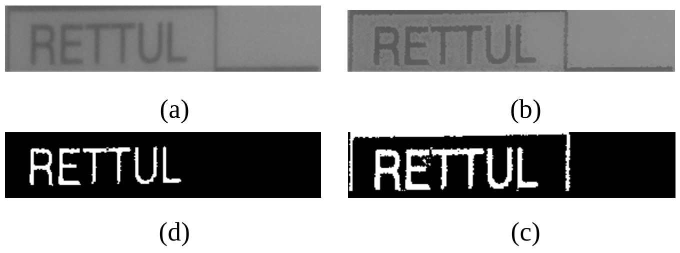


Figure 5: (a) Original patch of text (b) Contrast enhanced text (c) Result of locally adaptive thresholding (d) Final binarized text after filtering out blobs which are not a part of the text.

D. Binarization of text patches

In order to detect the letters of the anagrams, we need to feed them into an OCR engine. The OCR performs significantly better if we binarize the text patches before feeding it into the OCR engine. With this in mind, we processed the upper half of each blob in the following manner:

- We perform a contrast enhancement step first. This helps in getting a cleaner binarization and reduces the post-processing we need to do on the binary image. We used an iterative grayscale morphological image processing algorithm to enhance the sharpness, as listed below.

```
Im := Input Image
For Iteration = 1:NumIterations
     Im_d = dilate(Im, W)
     Im_e = erode(Im, W)
     Im_h = 0.5(Im_d + Im_e)
     For each pixel
     If Im > Im_h
     Im := Im_d
     Else
     Im := Im_e
End
```

- We then perform locally adaptive thresholding on this enhanced image using blocks of size 8x8. Using a global Otsu threshold does not work as well since there is usually a change in illumination over the patch.
- Following adaptive thresholding, we need to eliminate any random noisy patch which may have been misclassified as foreground, along with the rectangular boundary of the patch which is also detected. We do this by region labeling, followed by filtering based on area and location of centroid.
- Finally, we perform erosion using a small square structural element so that adjacent letters which are merged together (eg. TT in RETTUL) get separated and are recognized correctly by the OCR engine.

Figure 5 illustrates the intermediate results of every step of the binarization process.

E. OCR engine and unscrambling of letters

The final stage in our pipeline was feeding the binarized text patches into an Optical Character Recognition (OCR) engine to identify each individual letter and then unscrambling them to produce meaningful words. We used the open source Tesseract OCR engine since it is one of the most accurate ones available. One major challenge in character recognition specific to our project is the fact that the letters are jumbled and hence do not form meaningful words. Therefore, the OCR engine cannot use any lexicographic information to automatically correct any wrong predictions it might have made.

In order to reduce misclassifications, we fed the binarized and eroded letters into the OCR, along with the additional parameter that all the characters are uppercase letters of the english alphabet. The performance of the OCR engine depends critically on how well the letters have been binarized and also on whether they are clearly separated from each other. In our experiments, we found that binarization followed by erosion to reduce merging of letters was extremely crucial and a patch like what is shown in Figure 5 works really well and gives a 100% accuracy in character recognition.

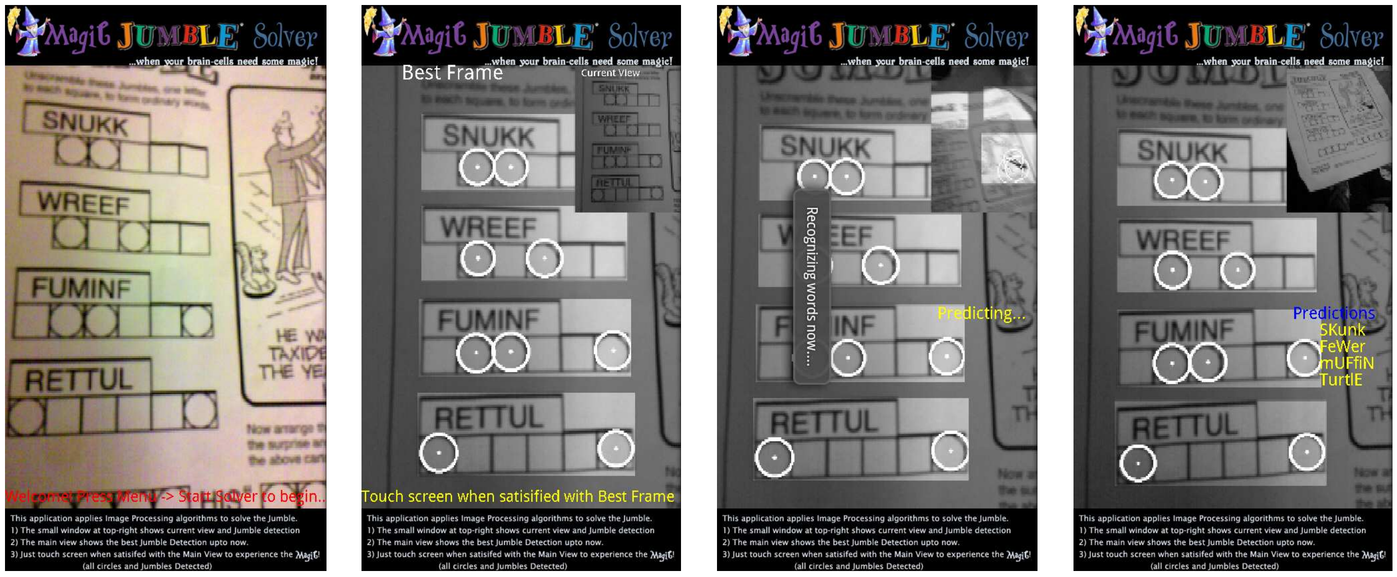
The OCR engine writes the scrambled letters into a file. Since our entire system was implemented on an Android mobile phone, to avoid unnecessary computation and to improve speed, we made an API call to an online server to get the desired unscrambled word instead of writing our own code which runs on the phone itself. After receiving the unscrambled words, we highlight the 'important' letters corresponding to the circles we had detected and display these on the screen.

III. ANRDROID IMPLEMENTATION

Our entire algorithm was first implemented in MATLAB and then on an Android platform using OpenCV. While the basic algorithm remains the same, we incorporated a few additional features to make it more robust as a mobile application.

The first difference comes from the fact that while in our MATLAB implementation we were working with an image of the word jumble taken using a mobile phone's camera, we decided to process frames in real-time while transplanting our algorithm to the Android phone. This has a few obvious advantages – while a single image of the puzzle may be blurry due to sudden movements of the hand and might not produce correct results, we have better chances of detecting all the letters correctly if we track a bunch of frames instead. With this idea in mind, we processed frames in real time and kept a track of the 'best frame' in a 2 second window. We defined the 'best frame' as that frame which has exactly four patches of text detected and in which the number of circles detected were maximum. If the user does not keep his hand very steady, the current frame along with all the detections has a very jerky appearance. In order to counter that we decided to display the 'best frame' in the main window instead and the current frame was displayed in a smaller window at one corner of the screen.

Secondly, implementation on a mobile platform brings with it issues concerning speed. In order to improve upon the speed of our algorithm, we shifted a few of our operations to the server and made API calls to the server with appropriate inputs


Magic JUMBLE Solver
...when your brain-cells need some magic!
Welcome! Press Menu -> Start Solver to begin..
Best Frame
Current View
Touch screen when satisifed with Best Frame
Recognizing words now....
Predicting...
Predictions
SKunk
FeWer
mUFfiN
TurtlE

This application applies Image Processing algorithms to solve the Jumble.
1) The small window at top-right shows current view and Jumble detection
2) The main view shows the best Jumble Detection upto now.
3) Just touch screen when satisifed with the Main View to experience the Magic!
(all circles and Jumbles Detected)

(a) (b) (c) (d)

Figure 6 : (a) Opening Screen of our app (b) Circles and letter patches detected (c) The user then touches the screen and the app connects to the OCR engine running on the server to obtain predictions (d) Predictions are displayed with the letters at circled positions capitalized.

to obtain the desired results. For instance, the entire Tesseract OCR engine was set up on a server and we provided only the binarized text patches of the 'best frame' to the server and got back the characters recognized in the form of a text file. We also shifted the entire process of unscrambling the letters to the server since this would otherwise require writing recursive code and having the entire english dictionary on the mobile phone. Presently, our app takes around 3-4 seconds to process the image and make predictions.

Figure 6 shows a snap-shot of the user interface for our android application. The 'best frame' is shown on the left and the 'current frame' is shown on the top right corner. The predictions with the important letters highlighted are shown in red on the bottom right. It must be noted that once the best frame is obtained by the application, it is no longer necessary to keep the camera focused on the words for the predictions to occur.

IV. Observations

We conducted several experiments on word jumbles downloaded from e-newspapers and magazines. From our experiments, we observed the following :

- Our algorithm is fairly robust to scaling. Bounding boxes of the letter patches and the circles are successfully detected for puzzles at different scales as long as it follows the same structure as the puzzle shown in Figure 1. However, for very small scales, the letters become quite small and tend to get merged together during the binarization process, leading to few errors in character recognition.
- Our algorithm is not robust to rotation. We can tolerate minor rotations, but our app fails if the puzzle is rotated significantly. We expect the user to keep the puzzle approximately horizontal while using the application.
- Binarization is perhaps the most crucial step in our entire pipeline. The accuracy of predictions of the OCR engine heavily depends on whether the letters are binarized properly and whether they are separated from each other. In order to ensure this, we performed sharpening before binarization and further post-processed on the binarized letter patches as detailed in section II.
- In some cases, even if binarization is done properly, the OCR engine makes errors if the letters are too close to each other. We illustrate one such example in Figure 7 in which the space between 'L' and 'I' in the word 'NELIR' is not significant and the OCR combines these two letters and classifies it as a 'U' instead.

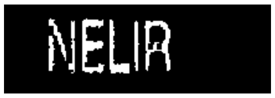


Figure 7 : An example where the OCR engine makes an error. This word is detected as 'NEUR' instead of NELIR.

V. Conclusion and Possible Improvements

In this project, we have successfully built our own Android application to solve word jumbles that are commonly found in daily life. While the app works quite well for most cases, there's still a lot of scope for improvement. Some possible extensions might be making our application robust to rotation and also exploring better algorithms for binarization of text so that the OCR engine makes fewer errors. Incorporating some error correction algorithm on the OCR predictions, based on confusion matrix might lead to a better performance.

Acknowledgment

We would like to thank David Chen for his invaluable help and advice regarding the Android implementation of our algorithm. We would also like to extend special thanks to Sam Tsai for his help with the OCR part of our project, specially for setting up the Tesseract OCR server for us. We would also like to acknowledge that some of the algorithms used in our project were based on homework problems we had for this course.

References

[1] "An Overview of the Tesseract OCR Engine" , R. Smith ,9th International Conference on Document Analysis and Recognition.

[2] "Use of the Hough Transformation to Detect Lines and Curves in Pictures," R. O. Duda, and P. E. Hart, Comm. ACM, Vol. 15 , pp. 11 – 15 (January, 1972)

[3] OpenCV resources available at www.stackoverflow.com and www.opencv.willowgarage.com

[4] R. C. Gonzalez and R.E. Woods, Digital Image Processing 2nd Edition, Prentice Hall, New Jersey, 2002.

Appendix

Debabrata was mostly involved with designing the algorithmic pipeline and implementing it in MATLAB. Abhishek was mostly involved in implementing the algorithm on the Android platform. The poster and report had equal contribution from both members.