Sudoku Puzzle Detection

Batuhan Erden

Abstract

In this assignment, I have implemented a **Python 3.6** script that detects rectangles in the image and if the image contains a Sudoku puzzle or not. I have used ***OpenCV 3.3*** for all image operations, ***numpy*** for matrix operations and ***glob*** for reading files from a directory.

Introduction

The script starts its operations by reading all of the images in a directory that is set at the top of the script. If no images are found, the script terminates the program. After the images are read, they are processed one by one and put in an array. All of the operations are done until now. After that, it shows the images processed in a window in which clicking any button shows the next image and clicking ESC button terminates the program.

Segmentation of the Sudoku Puzzle

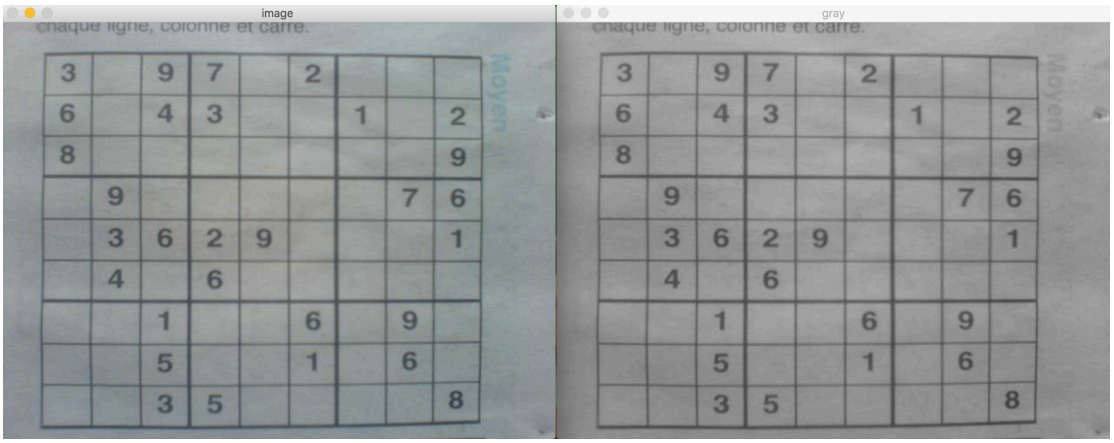
For each image read, we need to apply some filters to be able to detect the edges in images. The filters applied to images are as follows:

- ### Blur Detection (Laplacian Variance)

This operation is done using Laplacian Variance. If the variance is less than our threshold, we set its blurriness level that is *1 if it's less than 120, 2 if it's less than 100, 3 if it's less than 60, 4 if it's less than 15, 5 if it's less than 10 and 0 otherwise*. 0 blurriness level means that the image is not blurry. The Laplacian uses the 2nd derivative of the image to detect regions that contains rapid intensity changes. The idea here is that if the variance is high, we can say that there are lots of rapid intensity changes. When the variance is low, it means that there are a few edges in the image (The more blurred the image is, the less edges it contains). This blurriness level is used as an argument for Gaussian Filter that is the next step. **OpenCV's Laplacian()** is used. It takes 2 arguments. The 1st argument is the image itself when the 2nd one is the Laplacian kernel.

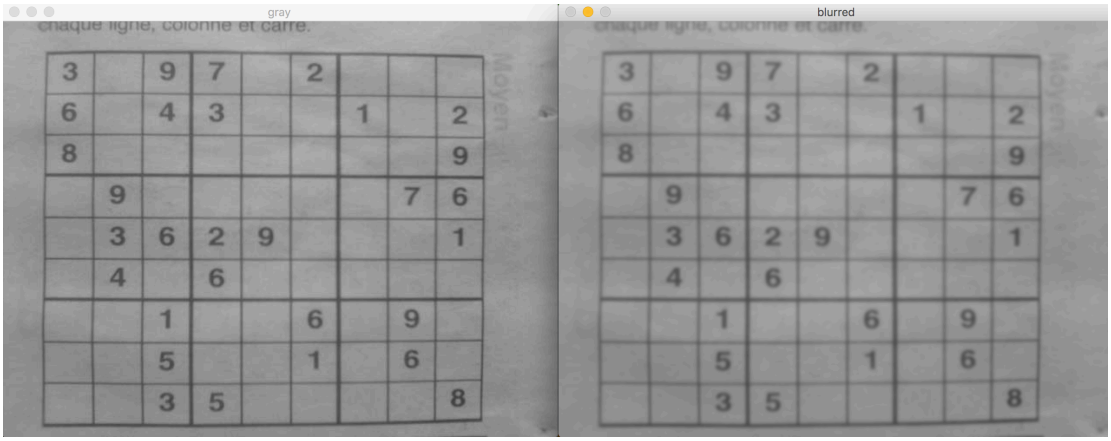
- ## Changing color space from RGB to gray scale

Original image's color space is RGB at the start and we need to change its color scape to gray scale. The reason for this conversion is because RGB color space has a lot of information that most will be useless in our case. Besides, this conversion is needed for applying the other filters that we use. **OpenCV's cvtColor()** is used. It takes 2 arguments. The 1st argument takes the original image and the 2nd is the color code.



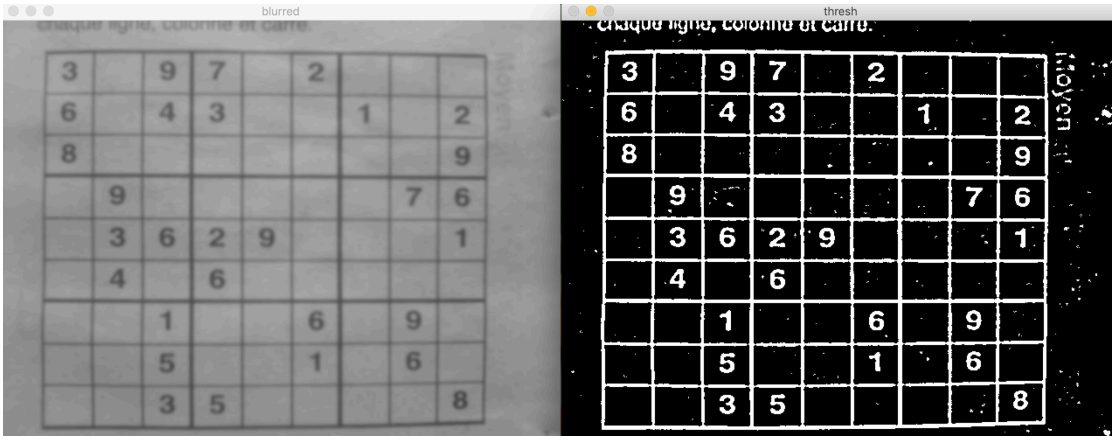
- ## Gaussian Filter

This filter is applied on the gray scale image. This filter blurs the image to reduce the noise and detail in the image. **OpenCV's GaussianBlur()** is used. It takes 3 arguments. The 1st argument is the gray scale image, the 2nd argument is the blurriness level (kernel size) and the 3rd one is SigmaX that is nothing but the standard deviation in X direction. Blurriness level is calculated using Laplacian. We simply add 3 to blurriness level and make it tuple to decide the kernel size (*3 + blurriness level, 3 + blurriness level*). And, the SigmaX is 0.



- ## Adaptive Threshold

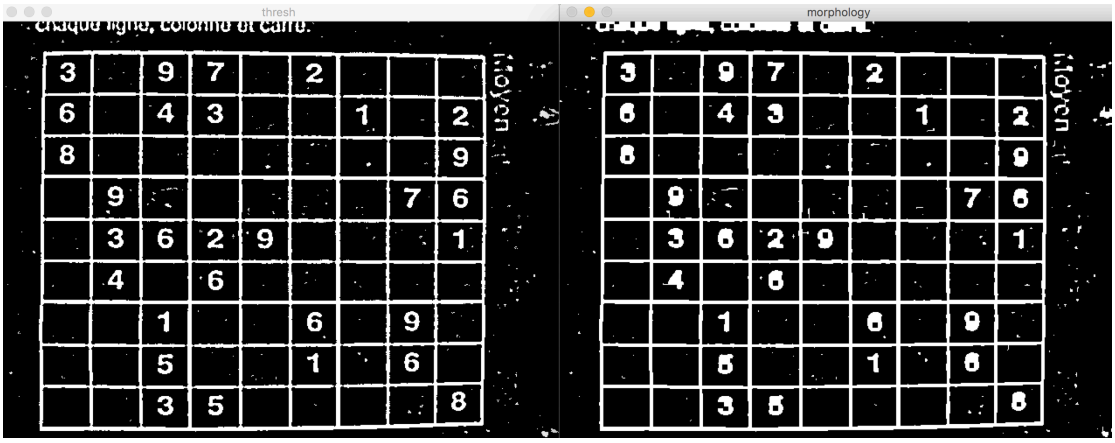
The next step is threshold the image. To do so, adaptive threshold is applied to the image. It is applied to highlight edges in the image. After it is applied, it outputs a binary image that represents the segmentation. At this point, edges are more detectable. **OpenCV's adaptiveThreshold()** function is used. It takes 6 arguments. The 1st argument is the blurred image obtained by using Gaussian Filter. The 2nd argument is the maximum value and it is set it to 255. The 3rd and 4th arguments are adaptive method and threshold type respectively. Each is set to 1 after many experiments. 5th argument is the block size which can be changed at the runtime. It's basically the size of the edges. Its default value is 11. It is incremented by 10 each time when there are no rectangles found in the image. This operation is done while detecting the rectangles. Its maximum value is 111. The last argument, the 6th argument, is a constant that is subtracted from the mean.



- ## Closing (Dilation & Erosion)

The next step is to apply some morphological transformations to the image. It is applied to the image obtained from adaptive threshold as it is normally performed on binary images. Closing algorithm first applies dilation, and then applies erosion to the image. It is used for closing small holes inside the objects. Dilation is an algorithm that increases the white region in the image or object area increases. Erosion basically erodes away the boundaries of objects. All of the pixels near boundaries are discarded because the kernel slides through the image. It is mainly used to remove small white noises in the image. **OpenCV's morphologyEx()** is used. It takes 3 arguments where the 1st argument is the thresh image, 2nd argument is the **OpenCV's Closing** algorithm and the 3rd one is the kernel. The value of the kernel is set according to the blurriness level that we calculate using Laplacian variance. The value of the kernel is this formula:

```
b ← blurriness_level
kernel ← (5 + b, 5 + b) if b is even
         (4 + b, 4 + b) if b is odd
```



Detecting Rectangles

After applying necessary filters, the image is ready to be processed. A method called contour approximation is used to detect the rectangles in an image. **OpenCV's findContours()** is used to find the contours in an image. Contours are the curves joining all the continuous points along a boundary with same color or intensity. **cv2.CHAIN_APPROX_SIMPLE** parameter is given as an argument to **findCountours()** to obtain only the endpoints of contours. After all contours are obtained, for each contour, we find its contour area and if that area is bigger than the minimum area (image.size / 500), the following operations are performed: Contour parameter is calculated by **OpenCV's arcLength()** function and the result is multiplied by 0.1 to take the 10% of arc length (epsilon). Then, contour approximation algorithm (Ramer-Douglas-Peucker algorithm) is used to decide the shape of the contour. If the length of the approximation is 4, it means that it's a rectangle and it is appended to rectangles list. After this operations are performed for each contour, out rectangles list is the rectangles detected in the image.

Detecting if image contains Sudoku Puzzle or not

After detecting the rectangles in the image, the next goal is to detect if image contains Sudoku puzzle or not. The algorithm simply iterates over all of the rectangles detected and finds if there are at least 50 rectangles in that rectangle. The rectangles found are put in a list to decide which of them is the Sudoku's bounding box. If there are no rectangles containing at least 50 other rectangles, we again apply filtering by incrementing the block size by 10 (The one which is used as an argument in adaptive threshold) and all of these steps after that are done again. This operation is repeated until block size reaches its max value (111).

To decide the Sudoku's bounding box, we sort the list by rectangle size descending and take the smallest element in the list. The reason for this is because the smallest box that contains more than 50 other rectangles is the Sudoku's bounding box. At last, Sudoku's bounding box and the rectangles detected are drawn onto the image. The color of the bounding box is red when the color of other detected rectangles are blue.

Conclusion

In conclusion, the program reads the images from the directory given regardless of their names. Then it processes the images and detects the rectangles and if image contains Sudoku puzzle or not. It uses some filters and edge detection techniques to achieve this. After the processing operation is finished, all of the images are shown one by one and pressing any button shows the next picture. Also, pressing ESC key terminates the program immediately.