Foundations of Computer Vision Final Project
Recognition of numbers in Sudoku puzzles using OCR

Sahil Pethe (ssp5329@rit.edu)

Imaging chain description:

The imaging chain I used was as follows:

1. **Passing the original image:**
   We take the image names from the current directory and pass those names using a for loop to the function run_analysis() which runs the techniques used to find the numbers in the Sudokus.
2. **Black Flood fill of White Space and opening:**
   We use connected regions[1] to find the first largest white space ignoring the immediate black space and flood fill it with black to improve hough line matching[3] and reducing errors. We are not performing noise reduction because this step works best without noise reduction. Opening the image (on the complement) a bit so that the smaller white spaces are filled by black which further increases the chances of correct hough line matching[3].
3. **Canny Edge detection with Hough Line matching:**
   We use Canny Edge detection to get the edges of the lines which are denoted by white lines on the boundary of the lines but on the both sides. We then pass this Canny Edge detected image to the hough() transform function to get the Hough Transform Matrix, Theta and Rho. We then use houghlines() function to get the lines along with some essential parameters. The parameters in this case are not much important since we have reduced the white islands a lot by using black flood filling. I have avoided having a huge minimum length otherwise line will not fit properly. I have used 10 as the FillGap parameter for the same reason. Having a long line gives you a better angle but chances of the line itself not fitting properly increases.
4. **Finding the correct angle and rotating original image:**
   After getting the lines from the houghlines() function, we get the endpoints of a line (any line will suffice in this case because we just want the angle of a single line and there weren't any lines with sufficiently large error to warrant using different technique. Ideally, a weighted average would suffice for lines with constantly varying slopes but this is not the case here). After getting the endpoints, we find the slope of the line and then find the corresponding angle to rotate the image by. I found out that the correct angle to rotate the image by is -angle_found so the Sudoku matrix box matches the horizontal and vertical lines perfectly.
5. **Finding individual cells:**

This is done by using bwlabel and getting the total number of islands. We then use for loop to iterate through the total number of islands. Before entering the iteration though, we check if the (max(r) - min(r) > 30 && max(c) - min(c) > 30) and only enter if this condition is true. This will inherently remove small islands which we will not require in our analysis. In each iteration, first we find the max(r), max(c), min(r) and max(r); where r and c are respectively the rows and columns of the white islands. What this gives us is the bounding box of the entire island in question. We then reduce the box size by 4 pixels both horizontally and vertically to reduce the chances of the character being detected incorrectly because of noise in the edges of the cell of the Sudoku. This means even though we are passing a smaller box than originally detected to the OCR engine[2], the amount of noise and error is reduced significantly.

6. **Passing the cropped and rotated image of the cell to the OCR engine[2]:**
   After we get the cropped image and rotated image (of the cells), we pass the image to the OCR engine[2] with parameters: 'CharacterSet', '123456789', 'TextLayout', 'word'. The parameter CharacterSet will force the OCR engine[2] to guess the letter(s) in the image from the set of the character passed to it which is '123456789' only. The parameter TextLayout will force the OCR engine[2] to recognize the letter as a word. After we do this, we find the Text inside the image by using the .Text attribute and store it in a variable. We do the same for the confidence of the word. After doing this, we rotate the image by 90 degrees and perform the same analysis each time.
7. **Finding the best probable character:**
   Each time we rotate the image, we store the text and confidence values of the recognized word. After rotating the image 3 times and analyzing it 4 times, we get the 4 degree of confidences we are going to use. We find the maximum degree of confidence and output the text corresponding to the degree of confidence we got by finding the maximum degree of confidence. At this point, the OCR has either incorrectly or correctly recognized the character. We further explore this particular characteristic in the confusion matrix that I came up after analyzing the input character in the cell and the output received after using OCR. Using the confusion matrix, we can estimate the percentage error and then analyze the patterns in the error and come up with new solutions.

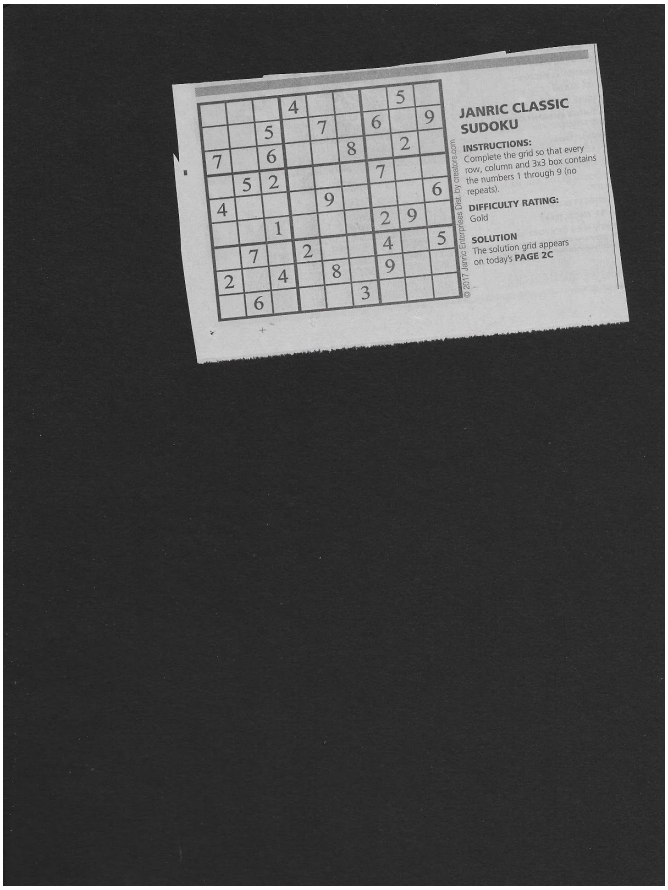

Fig 1: Original image (SCAN00281.jpg)

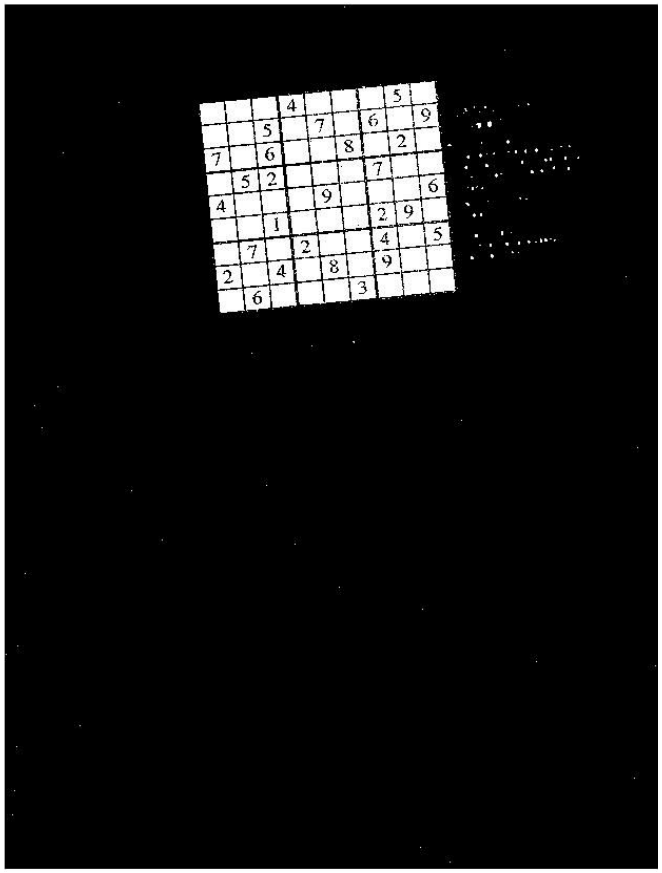

Fig 2: Black flood fill of White Spaces and opening

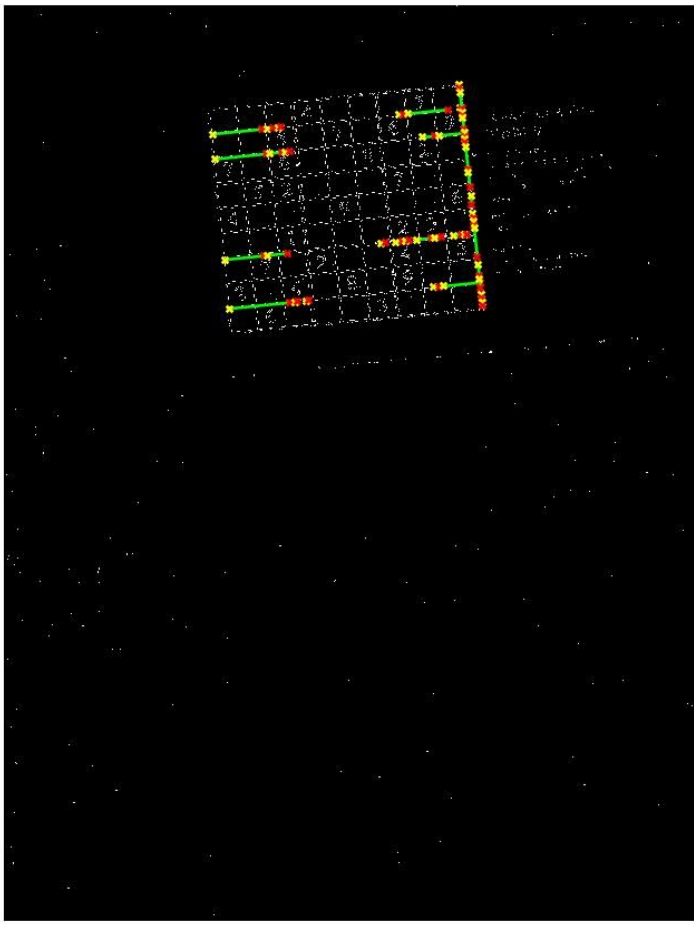

Fig 3: Canny Edge detection followed by Hough Line matching[3]

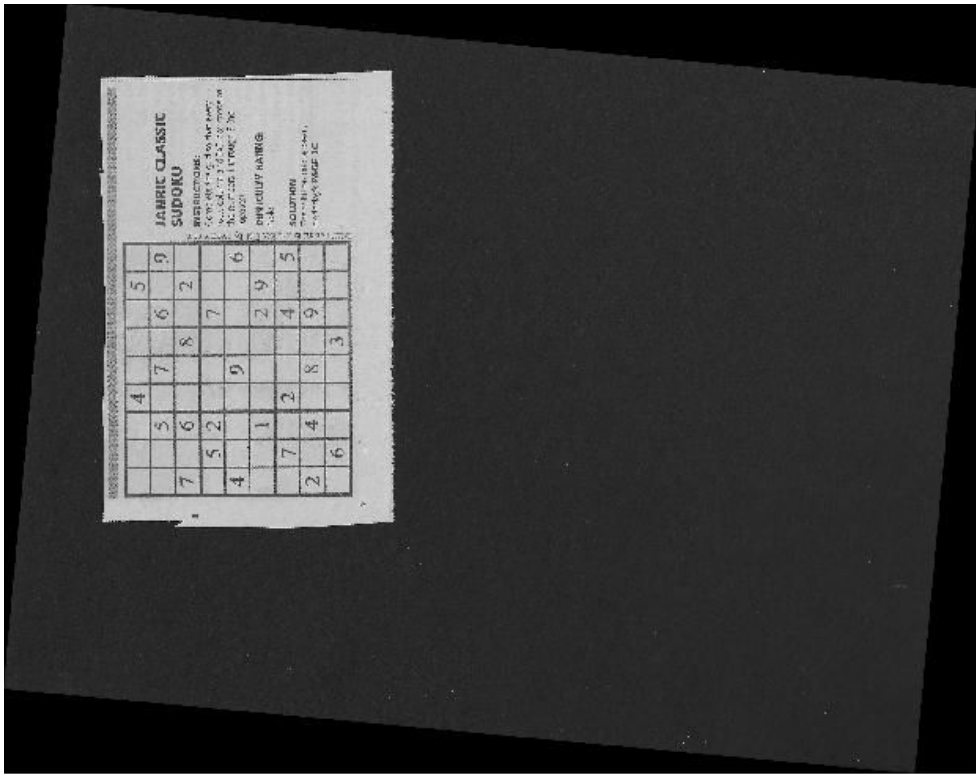

Fig 4: Finding angle and rotating the original image

Fig 6.1-6.4: Passing the cropped and rotated image of the cell to the OCR engine[2]

Techniques explored:

Many techniques I have explored did not make it into the final solution for reasons I have stated:

1. **Hough Transform and Lines (used):**
   This is being used in the solution to fit lines to the Sudoku border in order to find the angle to rotate the image by.
2. **Canny Edge Detection (used):**
   This is being used in the solution to find the edges of the lines of the Sudoku so that we can fit the Hough Lines correctly and without any error.
3. **Finding Islands using bwlabel (used):**

I am using bwlabel to find the white colored islands in the image which are nothing but the cells of the Sudoku and using the max(r), min(r) and max(c), min (c) values to find the bounding box to get the cell correctly.

4. **Harris Corners (explored but not used):**
   I explored this option to find the intersections of the lines of the Sudoku with the Hough Lines to further reduce any potential errors but firstly, it would be too compute intensive to do that, and secondly, using just the Hough Lines with the right preprocessing produced line matching with very little error. So, I scrapped this idea early on.
5. **Template Matching (failed):**
   I explored this midway in the project to only recognize the square cells and ignore other items (like in one photo there are images of some comic) but there are a lot of problems with template matching (like needing the correct template and having a lot of parameters to mess around with) which I did not want to explore midway in the project. Thus, there will be a few errors when a comic or some other image comes in the image.
6. **K-means Clustering (failed):**
   I also explored this technique midway in the project. The primary goal in my mind when using k-means clustering or any sort of clustering was to remove the small islands which we did not want to recognize as potential cells when doing further analysis like OCR on them. But I found that the clusters also included islands (cells) which were correct (and needed further analysis) and it was removing those too. So, I did not proceed further exploring different ways to remove the non-required islands.
7. **RegionProps (explored but not used):**
   I tried using RegionProps instead of simply using bwlabel but I found that using RegionProps was a bit confusing and I did not necessarily know which parameters I had to tweak in order to achieve the result I was expecting. In contrast to that, I found bwlabel as the best option as it was very basic. Sometimes having more control and fewer parameters to explore is a lot better than having more parameters and lesser control.

Results:

By running the program on all of the images which were provided in the Hough Lines Detection (for Sudokus) homework, I have gathered the input data v/s the observed data. The below confusion matrix provides how many of the observations were correct v/s how many failed and got some different values. Since the data was a lot and I had limited time to collect the results, these results, upto the best of my knowledge, are accurate, as per my observations. The garbage values which were observed contained random set of numbers for example: 00, 531, 11, 25 etc. which don't make much sense. Cells with no number in them i.e blank cells should not output anything. If they do, it's because I have not done any sort of training to the OCR for it know what is blank and what is not. The output are one cell at a time, in a single line.

Confusion Matrix

Input/ Obse-rvati-on	1	2	3	4	5	6	7	8	9	00	Garb-age values
1	54	0	0	0	0	0	0	0	0	0	0
2	0	46	0	0	0	0	0	0	0	0	1
3	0	0	44	0	0	0	0	0	0	0	3
4	0	0	0	53	0	0	0	0	0	0	1
5	0	0	1	0	54	0	0	0	0	0	3
6	0	0	0	0	0	72	0	0	0	0	0
7	0	0	0	0	0	0	53	0	0	0	0
8	0	0	0	0	0	0	0	40	0	10	0
9	0	0	0	0	0	0	0	0	27	0	0
Una-ccou-nted	0	0	0	0	0	0	0	0	0	0	16

Observations and Remarks:

I observed that when there are blank cells, the OCR sometimes tries to fit some character(s) which should not be fit at all. This only happens sometimes though. Another thing is, many times it will predict ‘8’ as ‘00’. This might be mostly because the way the OCR engine[2] is trained, it will recognize certain fonts better and some fonts worse. In this case, for this particular character, it seems to be ignoring the small line between the two circles in the ‘8’ figure. I could have solved this probably by using the CharacterSet parameter with individual numbers i.e with ‘1’, ‘2’, ‘3’, ... ‘9’ so that there is not instance of ‘00’ for it to predict. But doing this for each orientation of the cell and for each cell in the Sudoku would have increased the computation time 10 fold. After considering how less the prediction percentage for ‘00’ was, I have considered not to implement it. The accuracy observed is 92.68%.

Issues I faced:

The issues I faced during the development of this project were basically figuring out the orientation of the matrix. I considered an approach where finding certain numbers such as ‘7’ or ‘3’ or ‘2’ would give me the correct orientation and then going back to the bigger picture and finding out the characters in each cell using that particular orientation. But the way I designed this project, I could not go back after detecting the correct orientation. Instead, what I decided to do was to detect the correct character itself based on the degree of confidence of a particular cell in all orientations. One other issue I faced was that because I was independently detecting the characters without considering the entire Sudoku’s orientation, the OCR keeps jumbling up ‘6’ and ‘9’ frequently. Other than these two problems, the program works efficiently and without much errors. The hardest part was probably figuring out how to approach the orientation problem. When there were pictures of comics, those islands would also get detected and OCR would predict the output of that too. But the instances of those were far less than expected.

Conclusion:

Things I have learnt in this project include feature selection and detection along with orientation handling based on certain degree of confidence, restricting the predictions by using CharacterSet, using the OCR engine[2] with efficiency in mind. Other things I have learnt are that sometimes errors cannot be averted no matter what you try and that new techniques need to be researched in order to use those new techniques in conjunction with the proven old ones in order to solve the

problems we might face. The accuracy of my implementation is approximately 443/478 which comes to 92.68% which is not bad but there is certainly room for improvement.

References:

[1] Connected regions and Flood Filling:
https://www.mathworks.com/help/images/ref/bwconncomp.html
[2] OCR parameters and examples:
https://www.mathworks.com/help/vision/examples/recognize-text-using-optical-character-recognition-ocr.html
[3] Hough Line matching:
https://www.mathworks.com/help/images/ref/houghlines.html?s_tid=doc_ta