Report of Computer Vision with OpenCV

(Project 2, part 2)[1]

Author: ZHUO Xu (卓旭); Email: zhuoxu@seu.edu.cn

All the previous procedures in Part 1 are integrated into a function called *sudoku_part1*, which takes a Sudoku Puzzle Photo and gives the calibrated Sudoku Puzzle Image. We'll start from this image in Part 2.

4. Localize the Cells

We simply ignore the different line thicknesses of a 3×3 cell and a 1×1 cell. Crop the image into 81 1×1 cells with (height, width) = (h//9, w//9) uniformly, where h and w are the height and width of the image respectively, and // indicates divide and floor. In our case, each cell has shape 80×80.

By doing this, some black borders still remain in the cells. To remove them, I first tried to center crop each cell into 64×64. However, the results are not satisfying enough as shown below. Some cells still have black borders.

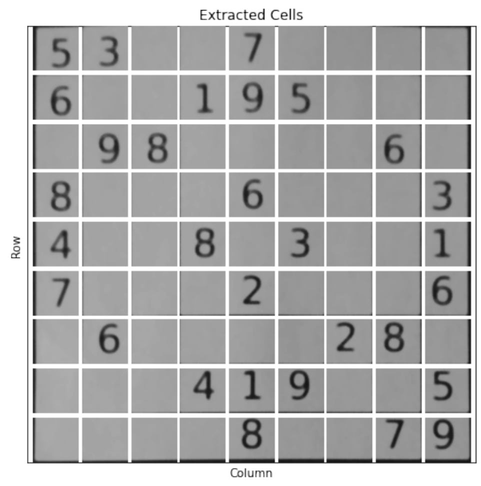


Figure 1 – Extracted Cells via Center Crop.

I turned to another approach: Remove all the grid lines. Since all the grid lines are connected to each other, if we can determine any seed point on them and perform region grow from it, we can extract the whole grid. This can be done by *cv2.floodFill*, and the seed point is selected as any black point on a line with more than 90% black points scanned bottom up on the Adaptive Threshed image. Intermediate results are shown below. Note that to ensure the grid removal is complete, the extracted grid mask is dilated using a 3×3 all-one kernel. After that, corresponding pixels in the desk wed image where the mask is 1 is set to pure white to remove the grid.

[1] Corresponding code: *sudoku_part2.ipynb*; Opensource at github.com/z0gSh1u/rennes1-seu-cv .

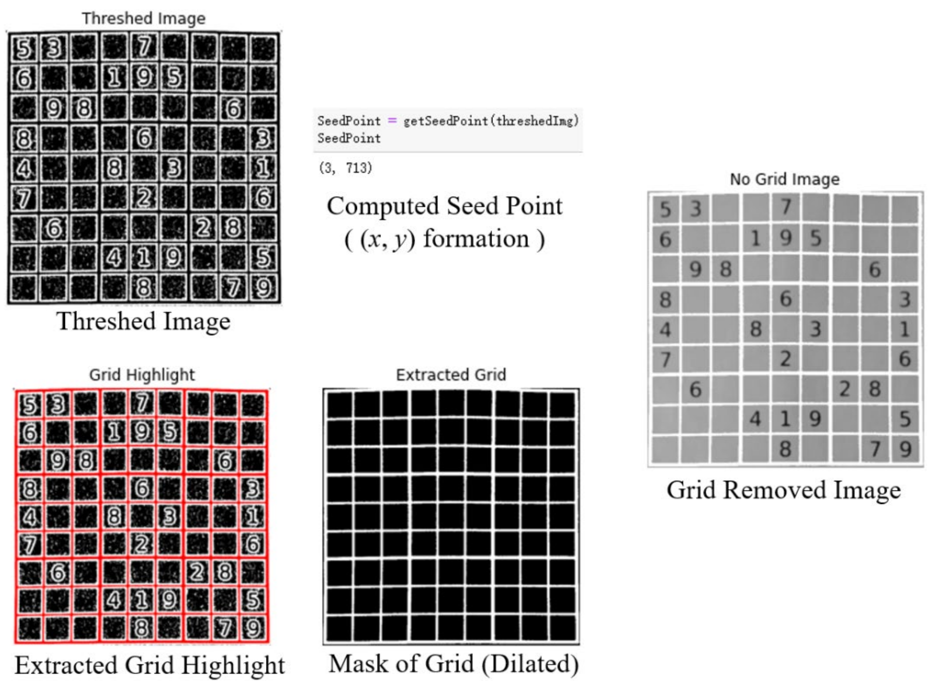

```
SeedPoint = getSeedPoint(threshedImg)
SeedPoint
(3, 713)
```

Figure 2 – Intermediate Results of Grid Removal.

Now, we can perform the previous 9×9 partition and 64×64 center crop operations. Extracted cells are shown below. The outcome is much better than Figure 1.

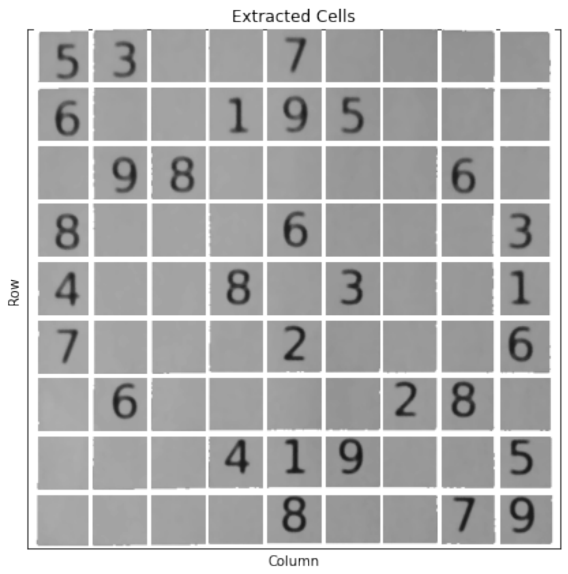
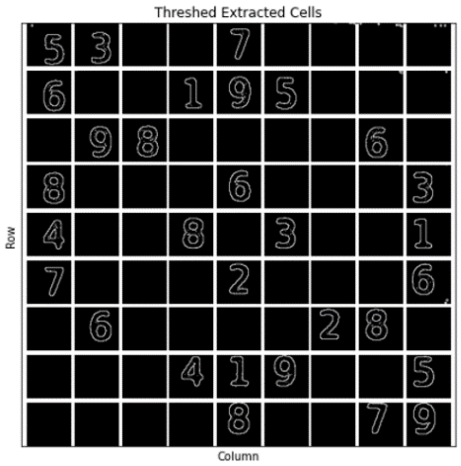


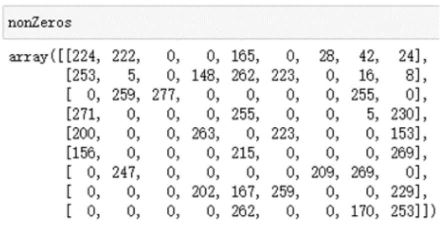
Figure 3 – Extracted Cells via Grid Removal and Center Crop.

5. Identify Empty/Filled Cells

Some cells are blank in the Sudoku Puzzle, while others are not. To differentiate them, we first use an empirical threshold 255//2 = 127, to set those pixels greater than 127 to 255, i.e., gray pixels to pure white, so that the following thresholding will be more robust. Then we use *cv2.adaptiveThreshold* to threshold each cell. After that, we use *cv2.countNonZero* to count non-zero (not black) pixels in each threshed cell. Intermediate results are shown below.



Threshed Cells


```
nonZeros
```

```
array([[224, 222,   0,   0, 165,   0,  28,  42,  24],
       [253,   5,   0, 148, 262, 223,   0,  16,   8],
       [  0, 259, 277,   0,   0,   0,   0, 255,   0],
       [271,   0,   0,   0, 255,   0,   0,   5, 230],
       [200,   0,   0, 263,   0, 223,   0,   0, 153],
       [156,   0,   0,   0, 215,   0,   0,   0, 269],
       [  0, 247,   0,   0,   0,   0, 209, 269,   0],
       [  0,   0,   0, 202, 167, 259,   0,   0, 229],
       [  0,   0,   0,   0, 262,   0,   0, 170, 253]])
```

Non-zero Counts

Figure 4 – Intermediate Results of Non-zero Counting.

The number of existing digits of the Sudoku's initialization is uncertain. So, we use an empirical threshold 125 non-zero pixels to identify empty/filled cells (this might not be very robust). Finally, the identification result is shown below. A white cell means there is a digit inside it, while black cell isn't. The result is correct after checking.

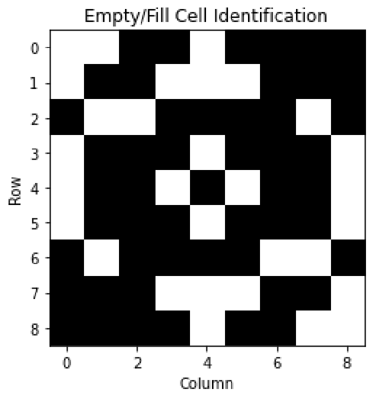


Figure 5 – Identification of Empty/Fill Cell.