WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

APPM4058A & COMS7238A: Digital Image Processing
Project One: Sudoku Solver

Mayur Ranchod, 1601745, Coms Hons.

10 June 2020

1 Introduction

Sudoku is an ancient Japanese number-placing puzzle that is based on logic and combinatorics which can be commonly found in newspapers, magazines, or the Internet. The objective of the game is to fill a 9x9 grid with digits such that each row, column, and each of the nine 3x3 subgrids that compose the grid contains all digits from 1 to 9[1]. Initially, the grid is partially completed and the player is required to complete the rest of the grid whilst ensuring that the aforementioned conditions are satisfied. We present an example of a typical sudoku puzzle alongside its corresponding solution in Figure 1. Generally, most sudoku puzzles are termed *well-formed* which means that a unique solution exists. The number of permutations in which the grid can be configured is innumerable and it is the initialization which determines the difficulty of the puzzle.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) Typical Sudoku Puzzle, Stellmach [2017]

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

(b) The solution to the puzzle in Figure 1a. Cburnett [2017]

Figure 1: An example of a typical sudoku puzzle alongside its solution.

[1]Definition adapted from: https://en.wikipedia.org/wiki/Sudoku

There exist numerous sudoku solvers on the Internet that are capable of solving any sudoku puzzle with ease which makes it convenient to solve sudoku puzzles found online, however, getting the sudoku solver to solve puzzles taken from newspapers and magazines requires some thought. This provides the motivation for this report - more specifically, we introduce a solution such that, given an image of a sudoku puzzle from a standardized source (i.e. from the same newspaper, with the same font, scanned in the same way each time, with even lighting), we find its digital representation which can be easily passed to a sudoku solver. We accomplish this with the additional constraint of only using image-processing techniques. This is in contrast to the typical approach taken which relies on machine-learning techniques to solve the problem. More specifically, the most common approach is to train a feature-learning model such as a convolutional neural network (CNN), support vector machine (CNN) or performing K-nearest neighbours (KNN) on the popular MNIST dataset which contains a vast collection of handwritten digits. We present the details pertaining to our simple yet effective solution in the following section which is then followed by an analysis of results.

2 Implementation

Our solution accepts an image of a sudoku puzzle (that satisfies the criteria outlined in the introduction section) as input. We provide an example of such an input image in Figure 2.

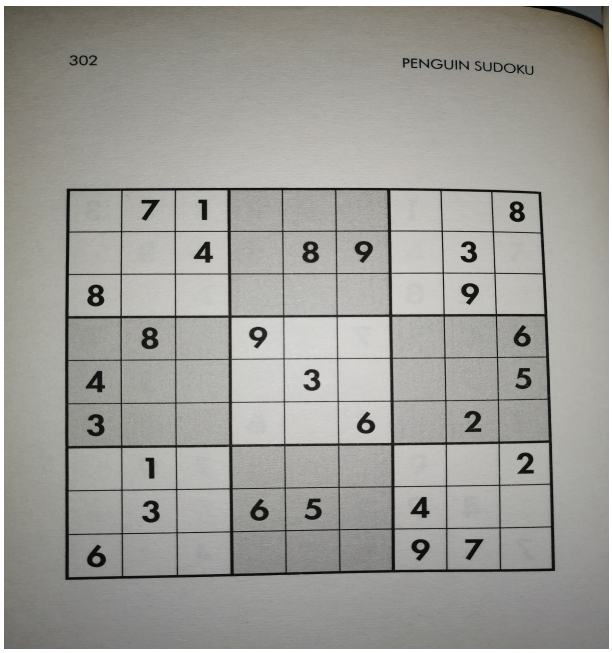


Figure 2: Example input for our solution.

Since we do not make any assumptions on the size of the input image, we resize the image to 990x990 pixels which allows us to standardize our solution to work on images with various sizes. We choose this size since it allows us to operate on an image that is smaller yet more manageable without significantly compromising the image quality. Since the entire image is not required for our task, we attempt to extract the grid from the image and thereafter perform all operations on this new image. For the time being, the decreased visibility of the digits is not a concern as we are primarily focussed on locating the grid. To locate the grid, we first attempt to filter the image using a Gaussian kernel with size 13x13 and standard deviation equal to 10 to reduce the effects of noise. Thereafter, we filter the image using the two 3x3 Sobel filters presented in Figure 3 for edge-detection purposes. These 2 kernels are used to detect edges in the x and y directions respectively.

-1	0	+1
-1	0	+1
-1	0	+1

Gx

+1	+1	+1
0	0	0
-1	-1	-1

Gy

Figure 3: Sobel Filters used for edge-detection purposes. R. Fisher and Wolfart. [2003]

We perform this edge-detection operation in an effort to make the border of the grid more pronounced. Since the colour variation in the image does not provide any useful information for our task, we perform a binary threshold operation on the result of the edge detection operation by using a threshold value of 10. Under normal circumstances, this is considered a very low threshold value, however, this is acceptable for our task since the edge-detection operation reduces the tonality of the image significantly. By setting a threshold value less than 10, the noise in the image is amplified while setting a threshold value greater than 10 introduces discontinuities in the border of the grid. It is crucial that no discontinuities present in the border of the grid for the operations that follow. The threshold operation maps all pixels with an intensity less than 10 to 0 and pixels with an intensity greater than 10 to 255. This sequence of operations produces the result in Figure 4.

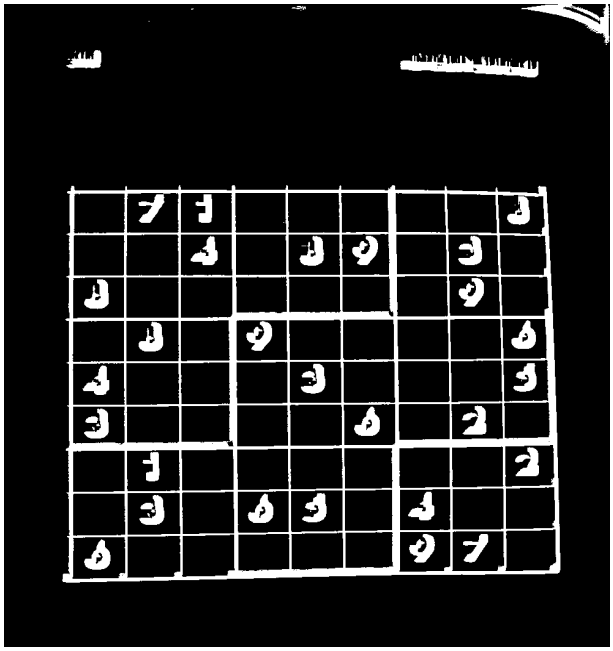


Figure 4: Result produced by performing the sequence of operations discussed above.

Since we now have a binary image, we begin locating the corners of the grid by finding the *contours* of the image. A contour is defined as a curve joining all the continuous points (along a boundary), having the same intensity. This is the reason for ensuring that no discontinuities are introduced in the border of the grid when we thresholded the image as we would not have been able to locate the border of the grid correctly. We find the contours of the image by using the `cv.findContours()` function. Since this function locates additional contours that are produced by numbers and the internal gridlines, we are required to extract the contours which define the border of the grid. We first sort the contours by size since we expect the contours of the borders to be the largest. Thereafter, we approximate the 4 largest

contours (which represents the 4 sides of the grid) using the `cv.approxPolyDP()` function which allows us to extract the endpoints of these contours. Once the endpoints are located, we find the smallest square that encompasses this collection of endpoints which we achieve by using the `cv.minAreaRect()` function. Lastly, we retrieve the coordinates of the corners of this new square which we use to extract the grid from the image, thus, producing the image in Figure 5.

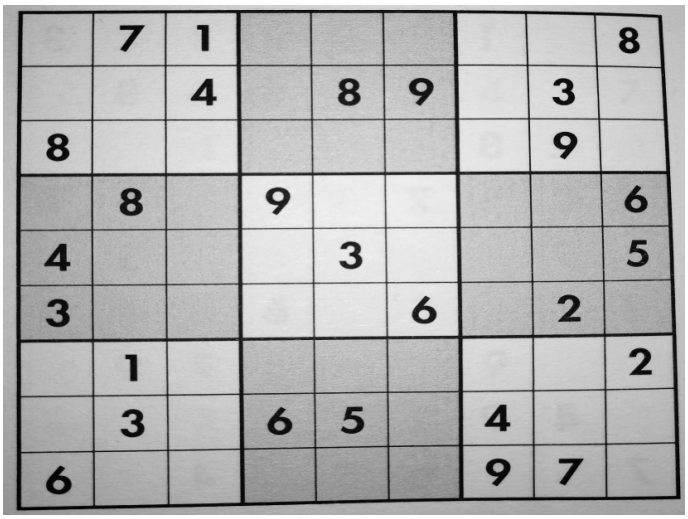


Figure 5: Result produced by extracting the grid from the input image.

Upon reviewing the result presented in Figure 5, we see that we are indeed able to narrow down the region of interest significantly by extracting the grid from the input image, however, we are convinced that this result can be vastly improved. This is motivated by the observation that the extracted grid still contains a thick border that is not part of the grid. Furthermore, upon considering the number of image-processing operations performed in relation to the quality of the result produced, we deduce that this is not a robust approach since it is unable to process images with minuscule deviations in the manner that the image is acquired. As an example, we see that the image in Figure 2 is captured at a slightly skewed angle which results in the poor extraction of the grid presented in Figure 5. As a consequence of the poor results achieved, we abandoned this approach in search of a more robust and effective solution.

Following the experimentation that we conduct as well as reviewing some of the previous approaches taken, we are convinced that we have formulated a simpler yet more effective solution than our previous approach. For our revised solution, we begin by resizing the image to 990x990 pixels as before to ensure that the process is standardized, allowing our solution to be applied to images with various sizes. We then threshold the image using a threshold value of 70 which maps pixels with an intensity less than 70 to 0 and pixels with an intensity greater than 70 to 255. By setting a too low or too high threshold value, the visibility of the digits is reduced considerably. We then take the negative of this result which sets the grid and digits to foreground pixels. Once we have a binary image, we define a cross structuring-element with size 3x3 which is used to perform morphological operations for noise removal purposes. More specifically, we perform an opening operation (which is composed of an erosion operation followed by a dilation operation) and on this result, we perform an additional dilation operation. The first erosion operation is used to eliminate single white pixels which subsequently enlarges "holes" formed by black pixels surrounded by white pixels. To remediate this, we perform a dilation operation which restores the holes to their original size which is followed by final dilation operation to fill the holes. This would typically be followed by a final erosion operation to restore the digits to their correct size; however, this is not necessary for our purpose.

Now that we have a binary image with reduced noise, we begin extracting the grid from the input image. As before, we locate the grid by finding the contours in the image using the `cv.findcontours()` function. Following from the revised image-processing operations that we perform, the border of the grid is now represented as a single contour which leaves us with the task of locating the corners of this contour. Since there is no guarantee that the grid in this extracted

region is perfectly square in shape since the image may have been captured at an oblique angle, we wish to acquire a 'birds-eye' view of the grid to achieve a better extraction of the grid with all digits clearly visible. To do this, we first construct a new (square) region that we want the grid to occupy. The length of the sides of this new region are determined by the computing the maximum Euclidean distance of the sides of the old extracted region. Given that the coordinates of the grid in the old region and the coordinates of the corners of the new region are known, we aim to project the grid from the old region onto the new region to acquire a centred view of the grid. We achieve this by performing a *perspective transformation* by using the `cv.getPerspectiveTransform()` function. The perspective transformation is a crucial operation which ensures that we do not include unnecessary regions surrounding the grid as before and that we now achieve an improved extraction of the grid. In Figure 6, we present a simple example which highlights the effect of performing a perspective transformation which is then followed by an illustration of the result achieved by performing a perspective transformation on the image presented in Figure 2. Note that the image illustrated in Figure 7 is for illustration purposes to highlight the effectiveness of the perspective transform and we continue to perform the following operations on the binary image that we have produced.

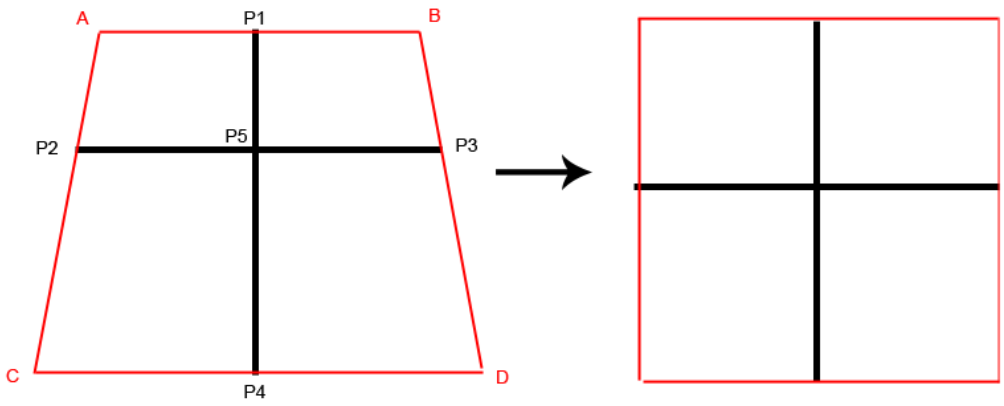


Figure 6: Illustrating the effect of performing a perspective transformation.

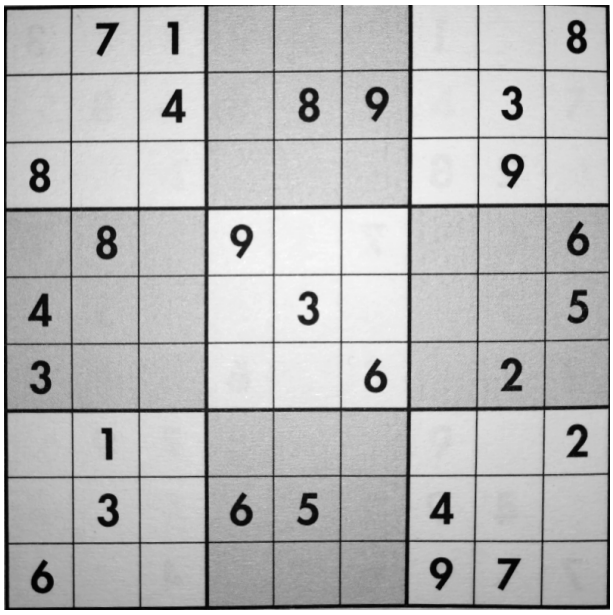


Figure 7: Extracted grid after performing a perspective transformation.

Upon analysing the result presented in Figure 7, we see that we have achieved a significantly improved extraction of the grid. Additionally, we note that the perspective transform leads to increased robustness since we are now able to process images with minor deviations in the way the image is acquired. Since we now have a clear view of the grid, we continue to use the binary image that we have produced above and begin to identify the digits in the grid. Before we proceed, we first define our a digital sudoku grid and since the sudoku solver of our choice is the *basicsudoku* library (available in Python), this library provides a method which creates an empty sudoku grid in the required format which we illustrate in Figure 8.

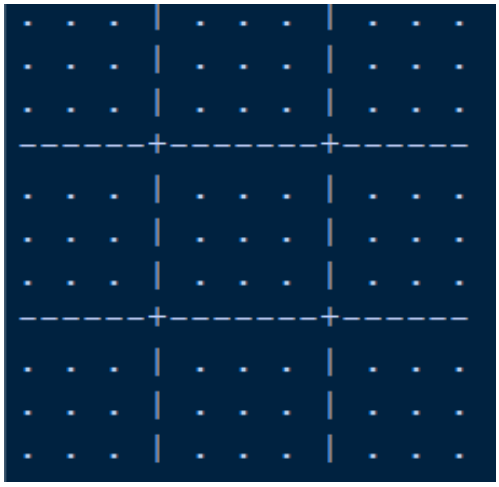


Figure 8: Empty sudoku grid provided by *basicsudoku*.

We are now left with the task of correctly identifying the digit in each cell and to populate the correct cell in the digital sudoku grid that we have defined. To the best of our knowledge, the approach that we take to accomplish this has not been embraced by any previous approaches. We first acquire 50 images of sudoku puzzles that we retrieve from the book by Bodycombe [2006] which we use to form a dataset. More specifically, we first construct a dataset containing 50 unique images (with size 64x64 pixels) of each digit. We acquire these images by individually cropping the digits from the negative of the input image. Thereafter, we capture an additional 15 images which we crop from the binary image that we have produced. We construct this large dataset containing unique images of each digit in an attempt to capture the variation that may arise during the operations that follow.

In order to identify the digits in the grid, we process a single cell at a time in a top-down manner. Given a cell, we first check if it is empty as we do not want to perform computations unnecessarily. We do this by testing whether the average intensity of the cell is greater than 20 or not. If a threshold value less than 20 is used, we would be attempting to identify a digit in an empty cell which is incorrect. In contrast, if a threshold value greater than 20 is used, cells that contain a digit will be considered as empty. Once we encounter a cell with a digit, we are required to identify the digit and since we restrict ourselves to only using image-processing techniques, we perform *template matching* [2]. The way that template matching is performed is that we require an input image and a template and we determine how well do regions in the input image correlate with the contents of the template. More specifically, we slide the template through image and at each point, and we determine how well does the region in the input image masked by the template correlate with the contents in the template itself. The result of this process is a grayscale image where each point stores a metric which reflects how well that point correlates with the template.

To use template matching for our solution, we are required to first formulate the necessary templates needed to identify each digit. Initially, we attempted to use a single image of each digit as our templates, however, this approach led to incorrect results. Thereafter, we use the 65 images of each digit to compute a mean image for that digit and we use this mean image as a template. We present the 9 formulated templates in Figure 9.

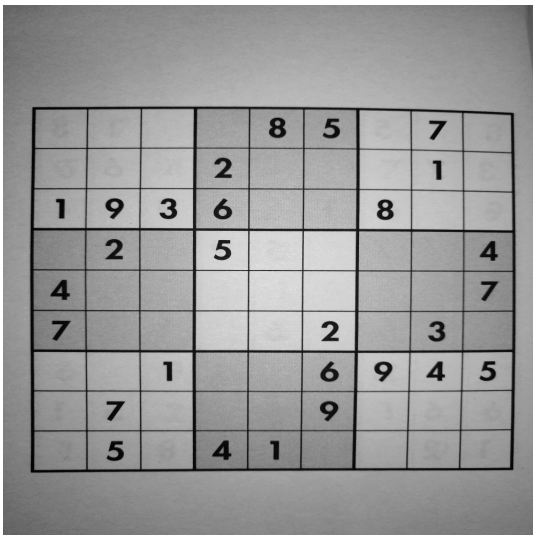
Figure 9: The templates used to perform template matching for identifying digits.

In order to use these templates, when a non-empty cell is encountered, we perform template matching with each of these templates to determine which of the templates corresponds the best with the content of the cell. Initially, we required a way to measure how well a template corresponded with the contents of the cell. The first approach that we took to achieve this was to compute the Structural Similarity Index (SSIM) metric between the cell and the template where a SSIM close to 1 indicates that the images are similar. This idea was motivated by the fact that the SSIM metric is commonly used to determine the similarity between 2 images for various image manipulations tasks, however, this method led to incorrect results. Upon conducting further experimentation, we formulated another simplistic approach which relies on the return values of the `cv.minmaxLoc()` function. The `cv.minmaxloc()` function accepts the grayscale image produced by the `cv.matchTemplate()` function as input and returns the highest and lowest values in that image as well as their locations. We use the `maxValue` parameter returned by this function to reflect the similarity between the cell and the template. For each of the templates that we have formulated, we perform template matching and we record the `maxValue` achieved by that template in an array. Once we have performed template matching with each of the templates, we use the template that produced the highest `maxValue` to indicate the digit that will be placed in the corresponding cell of the digital representation. We continue this process for each of the cells in the grid. We present the results achieved by our presented solution in the next section.

[2] We verify that template matching is an image-processing technique from the following article: https://en.wikipedia.org/wiki/Template_matching

3 Results

To determine the efficacy of our solution, we conduct experiments on numerous images of sudoku puzzles to determine the efficiency and accuracy of our solution. We accomplish this by providing our solution with images of sudoku puzzles from a standardized source with the same font and adequate lighting. We provide a handful of examples of the results produced below which is then followed by an analysis of the results.

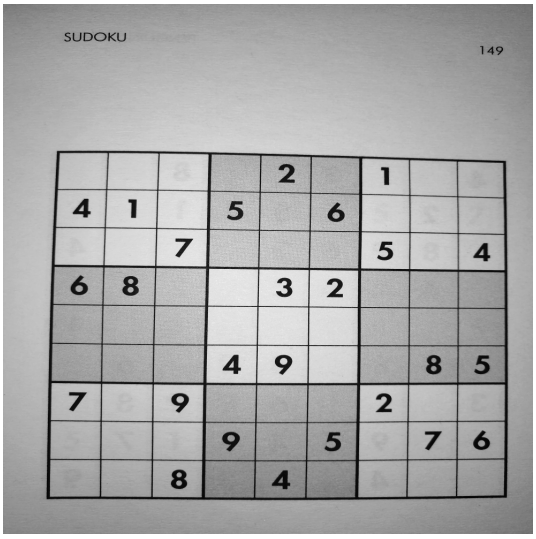


(a) Input Image.

```
. . . | . 8 5 | . 7 .
. . . | 2 . . | . 1 .
1 9 3 | 6 . . | 8 . .
------+-------+------
. 2 . | 5 . . | . . 4
4 . . | . . . | . . 7
7 . . | . . 2 | . 3 .
------+-------+------
. . 1 | . . 6 | 9 4 5
. 7 . | . . 9 | . . .
. 5 . | 4 1 . | . . .
```

(b) Digital representation of the input image in Figure 10a produced by our approach.

Figure 10: Example input and output of our solution.

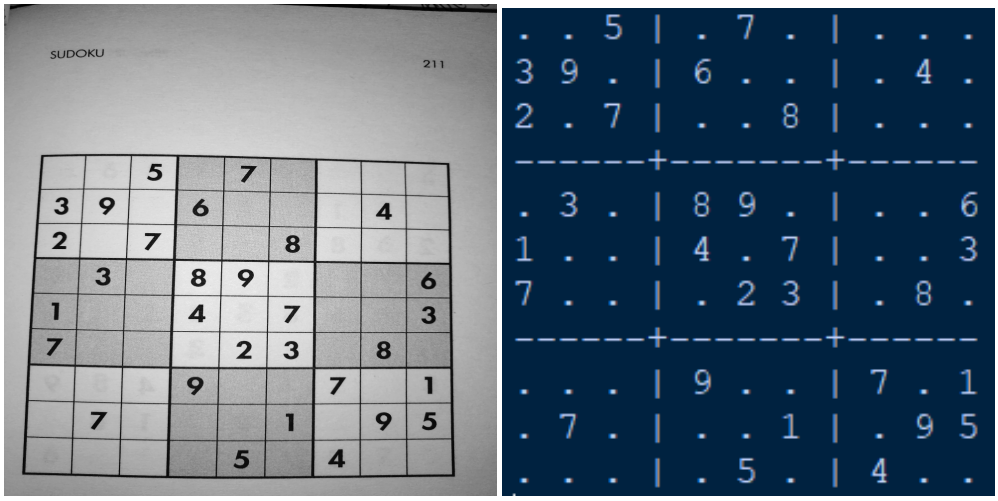


(a) Input image.

```
. . . | . 2 . | 1 . .
4 1 . | 5 . 6 | . . .
. . 7 | . . . | 5 . 4
------+-------+------
6 8 . | . 3 2 | . . .
. . . | . . . | . . .
. . . | 4 9 . | . 8 5
------+-------+------
7 . 9 | . . . | 2 . .
. . . | 9 . 5 | . 7 6
. . 8 | . 4 . | . . .
```

(b) Digital representation of the input image in Figure 11a produced by our approach.

Figure 11: Example input and output of our solution.



(a) Input image.

(b) Digital representation of the input image in Figure 12a produced by our approach.

Figure 12: Example input and output of our solution.

By taking the examples above and the results of the additional experiments that we conduct into consideration, we see that despite the simplicity of our solution, we are able to achieve a 100% accuracy rate. Upon reviewing the results produced when testing our solution, we acknowledge that we did not encounter errors where the solution incorrectly identifies a digit or places digits in empty cells. Furthermore, we see that even though alternate subgrids in the input image are darker, this does not pose an issue which highlights the robustness of our solution. Following from the simplicity of our solution, our solution is not computationally expensive as we do not experience any latency when producing a result.

4 Conclusion

Following from the analysis conducted in the results section, we are pleased with the results that our presented solution achieves despite its simplicity. This follows from the accuracy and efficiency that our solution is able to achieve. Additionally, these desirable traits emphasize the power of image-processing techniques. We therefore conclude that we have successfully addressed the problem that we have set out to tackle to find the digital representation of a sudoku puzzle taken from a standardized source that can be passed to a sudoku solver library using only image-processing techniques. Despite being able to produce impressive results, upon reviewing these results within a broader view, we are convinced that machine-learning based models would produce superior results for this task as these models would be able to process images from a much wider domain such as processing images of sudoku puzzles from varying sources with inconsistent fonts or lighting. Additionally, these models could potentially be able to process handwritten digits as well. Taking this into consideration, we can deduce that the generality of solutions based on image-processing techniques can be vastly improved.

References

Bodycombe, D. J. (2006). *Sudoku 2007*. Penguin Publishers.

Cburnett (2017). Sudoku puzzle solution. https://upload.wikimedia.org/wikipedia/commons/1/12/Sudoku_Puzzle_by_L2G-20050714_solution_standardized_layout.svg. Online; accessed 10 June 2020.

R. Fisher, S. Perkins, A. W. and Wolfart., E. (2003). Sobel edge detector. https://homepages.inf.ed.ac.uk/rbf/HIPR2/figs/sobmasks.gif. Online; accessed 10 June 2020.

Stellmach, T. (2017). A typical sudoku puzzle. https://upload.wikimedia.org/wikipedia/commons/e/e0/Sudoku_Puzzle_by_L2G-20050714_standardized_layout.svg. Online; accessed 10 June 2020.