Visual Sudoku Solver

Nikolas Pilavakis

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2020

Abstract

In this report, the design, implementation and testing of a visual Sudoku solver app for android written in Kotlin is discussed. The produced app is capable of recognising a Sudoku puzzle using the phone's camera and finding its solution(s) using a backtracking algorithm. To recognise the puzzle, multiple vision and machine learning techniques were employed, using the OpenCV library. Techniques used include grayscaling, adaptive thresholding, Gaussian blur, contour edge detection and template matching. Digits are recognised using AutoML, giving promising results. The chosen methods are explained and compared to possible alternatives. Each component of the app is then evaluated separately, with a variety of methods. A very brief user evaluation was also conducted. Finally, the limitations of the implemented app are discussed and future improvements are proposed.

Acknowledgements

I would like to express my sincere gratitude towards my supervisor, professor Nigel Topham for his continuous guidance and support throughout the academic year.

I would also like to thank my family and friends for the constant motivation.

Table of Contents

1 Introduction **7**

2 Background **9**

2.1 Sudoku Background 9
2.2 Solving the Sudoku 10
2.3 Sudoku recognition 12
2.4 Image processing 13
2.4.1 Grayscale 13
2.4.2 Thresholding 13
2.4.3 Gaussian blur 14

3 Design **17**

3.1 Design decisions 17
3.1.1 Development environment 17
3.1.2 Programming language 17
3.1.3 Solving algorithm 17
3.1.4 Vision Library 20
3.1.5 Real-time vs static recognition 21
3.1.6 Grid Detection 21
3.1.7 Image processing 21
3.1.8 Clue extraction 22
3.2 Requirements 22
3.2.1 Solver 22
3.2.2 User Interface 22
3.2.3 Grid detector 23
3.2.4 Clue extractor 23
3.3 Split between two years 23

4 Implementation **25**

4.1 App Overview 25
4.2 User Interface 26
4.2.1 Board 26
4.2.2 Puzzle Representation 28
4.2.3 Puzzle Editing 28
4.2.4 Displaying multiple solutions 28
4.3 Solving Algorithm 32

4.3.1 Kotlin Implementation 32
4.3.2 Multithreading 33
4.4 Recognition 33
4.4.1 Library Setup 33
4.4.2 Camera Use 33
4.4.3 Puzzle Detection 34
4.4.4 Orientation Correction 35
4.4.5 Clue detection 35

5 Evaluation 41
5.1 User interface 41
5.1.1 Response time 41
5.1.2 Error handling 41
5.2 Solving 42
5.2.1 Time taken 42
5.3 Recognition 43
5.3.1 Grid detection 43
5.3.2 Extracting clues 44
5.4 Requirements evaluation 47
5.4.1 Solver 47
5.4.2 User Interface 48
5.4.3 Grid detector 48
5.4.4 Clue extractor 48
5.5 User Evaluation 49

6 Conclusions 51
6.1 Project Achievements 51
6.2 Project limitations 51
6.3 Future Work 52
6.4 General remarks 53

7 Appendix 55

Bibliography 57

Chapter 1

Introduction

Peter Gordon and Frank Longo [26] refer to Sudoku as $n^2 \times n^2$ grid that is initially filled in with a certain number of cells (referred to as clues hereafter). The objective is to fill every cell with numbers 1 to n^2, without using any number more than once in the same row, column, or $n \times n$ block. In this report, only the most popular type of Sudoku is considered, for which $n = 3$. A famous Sudoku example is shown in figure 1.1. Nicknamed Al Escargot, it is considered the hardest Sudoku for human solvers. Various advanced solving techniques are necessary to solve this puzzle.

The initial goal of this project was ”to recognize a Sudoku puzzle in an image and then solve it”. More specifically, the project required that ”the phone user should be able to point the phone camera at a printed puzzle and see the image overlaid with the solution”. The description of the project suggested that the app was developed for IOS devices. After careful consideration, the goal was changed to the implementation of an android app. Instead of the puzzle being overlaid with the solution, a picture of the puzzle must be taken and the solution is presented separately. The rationale behind these decisions and the resulting advantages are discussed in Chapter 3.

The capabilities of the developed app include detecting and isolating a Sudoku puzzle using the phone’s rear camera. If necessary, the orientation of the detected puzzle can then be corrected. Afterwards, the clues can be extracted from the puzzle. Incorrect clues can also be corrected. The Sudoku with the corresponding clues can then be solved instantaneously. If the Sudoku has multiple solutions, navigation between solutions is also available. The app can be used as a companion when solving printed Sudoku, or to attempt to solve custom puzzles entered by the user. The app can also be used to verify whether the solution found is the only solution to the puzzle. Some printed puzzles might have more than one solution even though the puzzle was thought to have a single solution As this is a two-year project, the accuracy of detection is expected to improve next year. Capability to manually solve puzzles in the app will also be added.

In chapter 2, a review of existing literature is presented. The chapter begins with some general information about Sudoku puzzles. A review of existing solving algorithms is followed by a summary of literature involving Sudoku recognition. The chapter con-

cludes with a description of some common image processing techniques. In the next chapter, various decisions that were made before the beginning of the implementation of the app are discussed, including possible alternatives and the requirements that the app must fulfil. The design of the solving algorithm used by the app is also presented. In chapter 4, the implementation of the app is explained, including various obstacles that were encountered. The process that was followed in order to create the app and connect the various capabilities is presented. In chapter 5, methods used to evaluate the various aspects of the app are discussed, followed by some user evaluation. In the final chapter, the goals achieved during the project are summarised and then followed by some possible improvements.

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

Figure 1.1: AI Escargot.

Chapter 2

Background

2.1 Sudoku Background

There are 6,670,903,752,021,072,936,960 valid Sudoku grids. The number was first calculated by user QSCGZ on a google groups thread [3]. In 2006, Felgenhauer and Javris [22] confirmed this number and calculated the number of essentially different grids to be 5472730538 [23]. Most printed Sudoku have a single valid solution, however there exist examples of puzzles with more than one solution as show by figure 2.1

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

9	4
4	9

4	9
9	4

Figure 2.1: A Sudoku puzzle with two solutions

In 2010, Lin and Wu [40] used a checker created by McGuire and proposed an algorithm called Disjoint Minimal Unavoidable Set (DMUS) to conclude that the minimum number of clues required for a Sudoku to have a unique solution is 17. McGuire [48] confirmed this number in 2012.

According to some publications, the difficulty of the Sudoku is determined by the

number of cells that are initially filled, the fewer the cells, the harder the puzzle. Lee [38] separates Sudoku on 5 difficulty levels ranging extremely easy to evil, depending on the total number of clues, as shown by table 2.1 or the lower bound of number of clues in each row or column, as show by table 2.2.

Difficulty level	Number of clues
1 (Extremely easy)	> 46
2 (Easy)	36-46
3 (Medium)	32-35
4 (Difficult)	28-31
5 (Evil)	17-27

Table 2.1: Difficulty of puzzles based on number of given clues.

Difficulty level	Lower bound on number of clues in row or column.
1 (Extremely easy)	5
2 (Easy)	4
3 (Medium)	3
4 (Difficult)	2
5 (Evil)	0

Table 2.2: Difficulty of puzzles based on lower bound of clues in any row or column

However, this approach ignores the position of the clues in the puzzle. A more accurate method of determining the difficulty of a puzzle, is based on the complexity of the techniques that must be used for the puzzle to be solved by a human. Maji and Pal [31] made use of this system. This method has a significant drawback. The puzzle in question must be solved by a human for its difficulty to be determined. Ravasz and Toloczkai [21] proposed a system to quantify the difficulty of a puzzle. The proposed scale ranges from 1 to 4. Under this scale, the hardest puzzle, nicknamed ”platinum blond” was found to have a difficulty of 3.5789.

2.2 Solving the Sudoku

A plethora of different approaches were developed to solve the puzzle. In 2003, Yato and Seta [73] proved Sudoku to be ASP-complete and hence NP complete. Because of this, any know approach has a complexity which increases exponentially with the size of the grid. The most popular algorithm for solving Sudoku is backtracking, which is widely used in computer science [29]. Backtracking is a brute force algorithm that uses depth first search paradigm, completely exploring one branch before expanding another branch. Due to the finite number of valid grids, backtracking can be practical and is guaranteed to find all solutions to the puzzle, if they exist, given enough time. In its simplest form, the algorithm visits empty cells in an orderly fashion, filling in one number at a time and checking if all constraints are satisfied, in which case it moves to the next empty cell. If any of the row, column or box constraints is violated, then the value is incremented. If none of the 9 numbers (1-9) are allowed in the specific

cell, the algorithm backtracks and changes the value of the previously set cell. If one solution is sufficient, the algorithm terminates when the first solution is found, otherwise continues to search the entire search space. There are many ways to improve the algorithm. The simplest example would be only assigning numbers that will not violate any rule, instead of ranging from 1-9. A variety of different techniques have been proposed [30] [31] . Lee [38] proposed an approached based on elimination, that incorporates forward checking while picking the most constrained value first.In 2000, Knuth [37] implemented Dancing links algorithm which solves the puzzle in very short time. The methods mentioned above include frequent guessing which leads to redundant computation. Schottlender [59] explained how guesses can influence the efficiency of a backtracking algorithm. In 2014, Maji and Pal [42] proposed a guess-free solver that considers individual 3x3 grids.According to a survey hosted on a 2011 blog post by user attractivechaos [9], which was later referenced by Iyer et al. [30], The fastest known Sudoku solvers use a version of backtracking or dancing links.

A variety of soft-computing algorithms were also proposed, all of which use either bee colony or genetic algorithm. The Artificial Bee Colony (ABC) algorithm was first introduced by Karaboga [35] in 2005. An improved version [53] was presented in 2008 and was used by Pacurib et al. [52] in 2009. Yusiong and Pacurib [74] in 2010 and Mantere [44] in 2013. Genetic Annealing (GA) was first used to solve Sudoku by Mantere and Koljonen [45] [46] in 2006-7. Genetic algorithms are inspired by natural evolution to generate solutions, such as crossover, selection, mutation and inheritance. Those algorithms are exhaustive and hence tend to be extremely time consuming. In 2010, two algorithms were proposed to solve Sudoku using genetic annealing [56] [58]. In 2015, Wang et al. [70] proposed using GA with filtered mutations, yielding higher success rates as the algorithm was able to solve all test instances. In 2016, Chel et al. [15] proposed a multistage GA algorithm that is solving Sudoku puzzles with more than one solution.

Malakonakis et al. [43] made use of simulated annealing to solve a large Sudoku of size 15x15x15x15 on FPGA.

Kamal et al. [33] compared the results obtained when using backtracking, simulated annealing and genetic algorithms to solve the puzzle. Results indicated backtracking to be the best algorithm to solve Sudoku puzzles in a short time frame. Other solutions on FPGAs include Bok et al. [68], Gonzalez et al. [25], Dittrich et al. [19] and [34] that used brute-force. Brute-force was also used by Wicht and Hennebert [71] as well as Job and Paul [32].

Chowdhury and Akhter [16] proposed a solver using Boolean algebra.

Boolean satisfiability and Constrained Programming (CP) [11] were first used by Simonis [62] in 2005 and then by Rossi et al. [54] in 2006 and Crawford et al. [17] in 2008 but the efficiency of this approach was heavily influenced by the complexity of the problem. An advantage of CP, similar to backtracking, is that it will always find a solution if there is one.

In 2006, Moraglio et al. [51] suggested a method using metaheuristics and came to some encouraging results, followed by simillar succes of Lewis [39] a year later. Dur-

ing 2013-2015, Soto et al. [63] [64] [65] suggested Hybrids combining metaheuristics and CP filtering.

Herzberg [28] described Sudoku puzzles as a graph colouring problem. This is possible because each cell interacts with other cells as described by Bartlett [12].

Other techniques used include rewriting rules [57], Sinkhorn balancing [50], entropy minimization [27], Hall's marriage theorem [66], integer programming [12], harmony search [24], membrane computing [18] and Hybrid Ant Colony Optimization (ACO) [55].

2.3 Sudoku recognition

Literature for recognising a Sudoku puzzle from a picture, appropriately processing the picture and extracting the puzzle in a solvable format is very limited and much more recent compared to that of solving it.

In 2012, Simha et al. [61] proposed a model based on MATLAB that uses template matching [49] [14] to recognize the puzzle and the clues enclosed in it. While this method is very straightforward it tends to be laborious and struggles to adapt to new problems. The supplied image is cropped for efficiency and then converted into a binary image using block processing [36]. By applying a median filter to the produced blocks, most of the noise including grid lines is removed. The filter is used with an adaptive threshold [69] to tackle uneven illumination of the image. Any components that are connected to the border are then removed then removed for additional noise removal. The puzzle is then recognized to be the largest box in the processed image. The advantage of this approach for detecting the puzzle instead of finding extreme distance points from the origin of the image is that no extra steps are required for different angles of the image. A visual grid is then created. Anything on the grid is then removed using blob analysis [67]. Most of the errors occur in this stage, so it must be handled with extreme caution. The width/height ratio of each digit is then matched to that of the template and then compare to identify the digit. Overall, the model was relatively successful with a success rate of 90%.

In 2014-15 Wicht et al. [71] [72] used a model based on convolutional deep belief networks. Image processing includes Hough transformations and contour detection. The extracted features are then extracted using a support vector machine. reported results show an accuracy of 92%

In 2015, Ly and Vo [41] created a model based on Neural networks. While this requires massive quantities of training data and training can be computationally expensive, it can be very fast and reliable once it's trained. After the image is binarized, the angle is detected and the image is rotated accordingly. The study reports an accuracy of 95% , which can be improved even further with a larger training set.

In the same year,Kamal et al.[34] [33] used adapting thresholding, Hough and geometric transformations to process the image. The digits were then extracted using Optical character recognition (OCR). This approach led to excellent results, close to 100% accuracy.

In 2016, Dutta and Ghosh [20] suggested a character recognition in MATLAB using k-Nearest Neighbor algorithm. This algorithm does not require training, however it can be computationally expensive for large sets of data. This method also uses appropriate pre-processing techniques and template matching for extracting the digits.

2.4 Image processing

In this section the most popular techniques used to process an image of the puzzle in order to extract the clues are discussed.

2.4.1 Grayscale

Any colour present in the image to be processed does not contribute any useful information, it can be discarded by converting the image to grayscale. To store a grayscale image, only one channel of 8 bits is used for each pixel, compared to three channels(Red, Green, Blue) used in RGB, cutting the file size to a third of the original. As a result, processing of the image is simpler and faster. There are multiple algorithms available for converting an image to grayscale. The simplest algorithm takes the average of the three values for each pixel:

$$Gray = \frac{R+G+B}{3} \tag{2.1}$$

As show by figure 2.3 the resulting image looks unnatural. This happens because the human eye is much more sensitive to green compared to the other two colours. An alternative algorithm for, taking this into account is:

$$Gray = 0.299 * R + 0.587 * G + 0.114 * B \tag{2.2}$$

This algorithm produces images that reflect luminosity as perceived by the human eye, as seen in figure 2.4 and is used by major image processing libraries such as openCV [6] and MATLAB. Ahmed at al. [10] compared different algorithms for converting images to grayscale and found that the algorithm chosen can have significant effect on the efficiency of edge detection, which is used on recognising Sudoku.

d

2.4.2 Thresholding

Thresholding is a process used to transform the grayscale image, so that only two colours exist in the image, black and white. The process is commonly used when recognising Sudoku as it is an effective tool to distinguish an object from the background. A variety of methods exist [60]. The most commonly used ones are discussed in this section.

2.4.2.1 Global Thresholding

Global thresholding is the simplest form of thresholding. A threshold value T is chosen, ranging from 0 to 255. Every pixel value of the image is compared with the threshold. If it smaller, it is set to 0 (black), otherwise it is set to 1(white).

$$dst(x) = \begin{cases} 0 & src(x) \leq T \\ 1 & otherwise \end{cases}$$

2.4.2.2 Adaptive Thresholding

Global thresholding is not very effective in cases where different lighting conditions are present in the image, such as glare. Adaptive thresholding is a similar process to global thresholding that tackles differences in spatial illumination across the image [13]. This is achieved by partitioning the image to various ”neighbourhoods” and calculating a different threshold value for each. This method is the most effective when recognising Sudoku and is hence used widely in existing literature. The two most common ways of calculating the threshold for each region is either by taking the mean of all the pixels in the region or by using a weighted sum of all the pixels. Usually, the latter method works better for Sudoku as show by figure 2.2

2.4.3 Gaussian blur

Gaussian blur is a technique used to smooth out any noise that could be introduced during the camera's capturing process. It is based on the Gaussian function:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2 / 2\sigma^2} \tag{2.3}$$

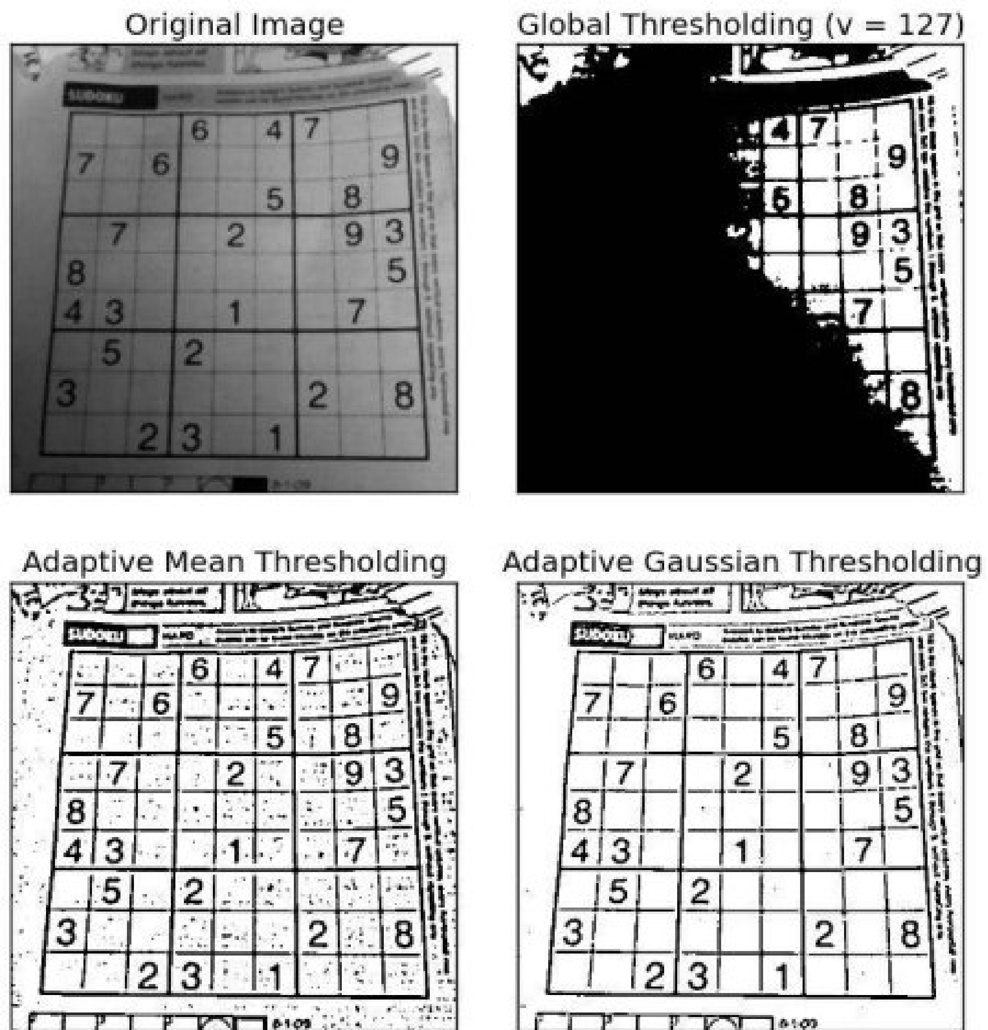

Figure 2.2: thresholding. Credit [5]

Chapter 3

Design

3.1 Design decisions

In this section, decisions made before the start of implementation are explained.

3.1.1 Development environment

The project proposal required an app to be running on IOS however, I decided to develop an android app to take advantage of prior experience I had in the domain, which allowed faster development. Furthermore, android is the most popular OS for smartphones, meaning that the resulting app would be suitable for more people, making finding test subjects easier.

3.1.2 Programming language

The programming languages available for android development are Java, Kotlin and C++. Since I only had experience with the first two, C++ was ruled out. Between Java and Kotlin, I chose to use the latter as it is the official programming language for android as of 2019 [2]. Furthermore, Kotlin and Java compile to very similar bytecode, resulting to similar performance. Translation from Java to Kotlin code can happen very easily as there are tools available. Files for both languages can co-exist within the same project without any extra cost, as a result, this decision is not binding and can be revised at any point. Last but not least, due to the fact that Kotlin is relatively new, no major similar projects exist on Kotlin. The comparison of the performance of the two is of personal interest and could happen during the project.

3.1.3 Solving algorithm

It was decided to solve the puzzles using a backtracking algorithm due to the significant performance advantage discussed in section 2.2. To reiterate, the fastest available algorithms all use some version of backtracking, as shown by the review conducted by blog post user attractiveChaos [9]. Speed is one of the most important factors for

commercial apps. Furthermore, some of the algorithms found in literature are not guaranteed to find a solution, while most of these algorithms assume that a Sudoku only has one valid solution. A well designed backtracking algorithm is guaranteed to find all solutions to any given puzzle, given enough time.

3.1.3.1 Algorithm design

Even though countless solvers are available online, implementing an algorithm from scratch was of personal interest. An outline of the simplest possible backtracking algorithm is shown below:

Algorithm 1 SolveSudoku(grid)

```
 1: if FindEmptyCell(grid) == NULL then
 2:     return true {Solving successful}
 3: else
 4:     for digit 1 to 9 do
 5:         if digit is legal for cell then
 6:             Assign digit to cell
 7:             if SolveSudoku(grid) then
 8:                 return true
 9:             else
10:                 Undo Assignment
11:                 return false {backtrack}
12:             end if
13:         end if
14:     end for
15: end if
```

Algorithm 2 FindEmptyCell(Grid)

```
1: for Cell in grid do
2:     if cell is empty then
3:         return cell
4:     end if
5: end for
6: return NULL
```

This algorithm blindly assigns all possible numbers to a random cell, until a solution is found. It can be further improved by using various heuristics when choosing a cell. The most widely used heuristics are:

1. Minimum remaining values: Choose the cell with the least legal values. In the case that there is a cell with only one remaining legal value, the heuristic implements the hidden singles technique.

2. Most constraining: Choose the cell that maximises constraints on other cells. Using this heuristic minimises branching.

When selecting a value for a cell, selecting the least constraining value maximises the probability that the branch to be explored will lead to a solution.

The algorithm was improved by keeping track of the illegal values for each empty cell using a set. The sets are initialised before the execution of the algorithm. Every time the sets are updated, they are ordered depending on their size. The cell with the most illegal values is selected and assigned a number. The sets corresponding to cells of the same row, column or grid are updated and the process is repeated. When a cell is filled, the corresponding set is removed. The revised algorithm is shown below:

Algorithm 3 SolveSudokuRevised(grid,setsArray)

Require: a set of illegal values for each empty cell. Sets are ordered by descending size and stored in an array.

```
1: largestSet ∈ setsArray s.t.size(largestSet) == max{size(A)|A ∈ setsArray}
2: if largestSet == NULL then
3:     return true {Solving successful}
4: else
5:     for digit ∈ {1..9} − largestSet do
6:         Assign digit to cell corresponding to largestSet.
7:         UpdatedSets= UpdateSets(grid,sets,cell)
8:         if UpdatedSets == NULL then
9:             return false {Branch cannot lead to solution, backtrack}
10:        end if
11:        if SolveSudokuRevised(grid,UpdatedSets) then
12:            return true {further explore branch}
13:        else
14:            Undo Assignment
15:            return false {backtrack}
16:        end if
17:    end for
18: end if
```

Algorithm 4 UpdateSets(grid,sets,cell)

```
1: for each set in row,column, grid do
2:     add content of updated cell to set
3:     if set.size == 9 then
4:         return NULL {Empty cell has no more legal values}
5:     end if
6: end for
7: sort sets
8: return sets
```

It is worth noting that the revised algorithm uses much more space compared to the previous one. This is a small price to pay for modern smartphones and should not be noticeable at run-time due to the small depth of each explored branch.

The algorithm stops when the first solution is found. To allow for multiple solutions, the algorithm was tweaked to store a solution when found and continue the search. The final version of the algorithm is shown below:

Algorithm 5 SolveSudokuMultiple(grid,setsArray)

```
Require: a set of illegal values for each empty cell. Sets are ordered by descending
    size and stored in an array.
 1: largestSet ∈ setsArray s.t.size(largestSet) == max{size(A)|A ∈ setsArray}
 2: if largestSet == NULL then
 3:    return true {Solving successful}
 4: else
 5:    for digit ∈ {1..9} − largestSet do
 6:       Assign digit to cell corresponding to largestSet.
 7:       UpdatedSets= UpdateSets(grid,sets,cell)
 8:       if UpdatedSets == NULL then
 9:          return false {Branch cannot lead to solution, backtrack}
10:       end if
11:       if SolveSudokuMultiple(grid,UpdatedSets) then
12:          if grid full then
13:             print solution
14:             Undo assignment
15:             return false {backtrack to try and find more solutions}
16:          end if
17:          return true {further explore branch}
18:       else
19:          Undo Assignment
20:          return false {backtrack}
21:       end if
22:    end for
23: end if
```

Naturally, this version of the algorithm takes more time to terminate, as every branch of the search tree is explored. This was taken into account during the evaluation of the algorithm, discussed in section 5.2.

3.1.4 Vision Library

The most common vision library for android is OpenCV [1]. OpenCV is a cross platform open source library with a large user community, exceeding 18 million downloads at the time of writing. Multiple projects can be found online that make use of OpenCV. One popular alternative is Google vision API. This option is mostly suitable for standardised tasks such as face recognition or scanning barcodes and is not so flexible. Also, the community is much smaller compared to OpenCV. Another alternative is OpenIMAJ. This option was rejected because of the very small community. Furthermore, no specific advantages were found over OpenCV. As far as speed is concerned, no benchmarks specific to android were found. As a result, the deciding factors were

community size and existing projects. As there were no apparent disadvantages of using OpenCV, it was believed to be the most suitable for this project.

3.1.5 Real-time vs static recognition

Two different approaches were considered regarding the way that the app completes its goal of recognising a puzzle and displaying the solution(s) to the user. The first approach was to solve the puzzle in real time using the phone's camera and overlay the camera feed with a solution. This approach requires efficient use of vision techniques as any additional delay would be easily noticed by the user. This task can be difficult to implement without prior experience in vision.

The second approach considered was to prompt the user to take a picture of a Sudoku and then use the static picture to display the solution(s) to the puzzle. Even though the results of this approach would not be as visually attractive, there are numerous advantages. Firstly, this approach allows the user to ensure that the puzzle detected is correct, perhaps allowing editing of clues before solving. Secondly, the first approach can only display one solution for any puzzle, where multiple solutions can be displayed with a static approach. Thirdly, better performance is likely when the camera is positioned at an angle towards the puzzle. Fourthly, static image processing is much easier to test, as the same image can be reused without having to worry about difference in lighting. Last but not least, this approach would allow the user to try and solve the puzzle on the phone. The only throwback of this solution is the need to design a responsive user interface to cover the extra functionality.

Due to the above factors, the use of a static picture was preferred instead of real-time solving.

3.1.6 Grid Detection

For the detection of the grid, it was decided to use a closed feedback loop, highlighting the grid before the picture is taken. This approach allows the user to adjust the position of the camera to increase the efficiency of the recognition of the puzzle. The puzzle can be highlighted and extracted using contour edge detection. The biggest rectangle in the camera feed can be assumed to be the Sudoku.

3.1.7 Image processing

Any information except the highlighted puzzle is unnecessary, and is therefore cropped and discarded. Color is another factor that provides no useful information, and can be removed by converting the image to grayscale, using equation 2.2 for reasons outlined in section 2.4.1. The grayscale image can then be converted into a black and white image, using adaptive thresholding, using a weighted sum of neighbouring pixels, as described in section 2.4.2.2. Gaussian blur can also be added to the image to smooth out any noise.

3.1.8 Clue extraction

For the detection of the clues from an image, two possible approaches were considered. One option was to train a custom machine learning model, such a neural network. This approach required training data. The most commonly used database for such tasks is the MNIST database. This database consists of handwritten digits and is not ideal in this use case, as the digits in question will always be printed. Another option considered was to use a preexisting library. An example of such library is Firebase AutoML, acquired by google in 2014. This library was preferred over possible alternatives due to prior experience with Firebase for possible alternatives. AutoML includes a database of mixed printed and handwritten digits. This database is more suitable for this application, compared to MNIST. A template matching feature was also offered, that could simplify the clue extraction process. As the latter approach was not as widely used in existing literature and had some advantages over the former, it was prioritised.

Cropping images to only contain the puzzle allows for division of the puzzle to 81 smaller images, in case template matching does not work as expected.

3.2 Requirements

In this section, the requirements that each individual component of the app must fulfill to be considered successful are specified. When applicable, an additional set of requirements is also specified, which are not essential but are nice to have.

3.2.1 Solver

- **Capability:** Given any Sudoku, the solver must be able to find all the solutions of the puzzle, subject to time and space constraints for edge cases, such as an empty grid.
- **Determinism:** The solver must always provide the same results for the same puzzle. If there is more than one solution, the order of the solutions returned must be fixed for the same puzzle.
- **Correctness:** The solver must only return a solution if the puzzle provided is solvable, and return an error otherwise.
- **Speed:** The solver must take no more than 1 second for each solution found to ensure that the app remains responsive and no downtime is noticed by the user. small loading time is tolerable for nearly empty grids that may have many solutions.
- **Flexibility:** The maximum number of solutions returned by the solver must be flexible.

3.2.2 User Interface

- **Intuitiveness:** The implemented interface must be easy to understand and must be usable without any prior training.

- **Responsiveness:** The interface must be responsive to the user's actions without any noticeable delay. Loading icons must be displayed when a delay is anticipated.
- **Robustness** The app must not crash under normal circumstances.
- **Functionality** Given an internal representation of a puzzle and its solutions, the interface must be capable of displaying the unsolved puzzle and its solution(s).
- **Speed:** The interface must take no longer than 0.5 seconds to update after a state change, such as displaying the solution of a puzzle.

3.2.3 Grid detector

- **Responsiveness:** The detector must respond to the camera moving and adjust without the user noticing any delay and provide feedback to the user.
- **Capability** Given a clear view of a Sudoku puzzle, the detector must detect and highlight the puzzle. The detector must extract the highlighted puzzle when the user confirms that the required puzzle is indeed highlighted.
- **Stability:** The detector must be stable to ensure that the highlighted portion does not change while the user attempts to capture it
- **Time required:** During evaluation, up to 8 seconds were required to detect a puzzle, with an average time of 5 seconds.

3.2.4 Clue extractor

- **Responsiveness:** The extractor must show some indication that the extraction is happening, perhaps with a loading icon.
- **Capability:** For pictures that are well lit and of high resolution, the extractor must recognise at least 90% of the clues correctly.
- **Additional Capability** For pictures that are of variable lighting and of medium resolution, the extractor should recognise some of the clues correctly.
- **Correctness:** The extractor must only place a clue in cells that contain one.
- **Speed:** The detector must take no more than 10 seconds to extract the clues from any given puzzle.

3.3 Split between two years

Because this is a two year project, different goals were set for each year. For the first year, the main objectives were the implementation of an app that can recognise and solve Sudoku, along with a suitable test suite to evaluate the performance of the app. The main objectives for next year are user evaluation as well as improvement of the performance of the app. A more detailed outline of directions the project could take next year can be found in section 6.3

Chapter 4

Implementation

In this chapter, the implementation of the different components of the app is discussed, in implementation order.

4.1 App Overview

The sequence of steps that are followed in order to scan a puzzle and find its solution(s) using the app are listed below.

1. The first time the app is launched, camera permission is requested
2. The camera feed is displayed on screen
3. When the phone is pointed at a Sudoku puzzle, it is detected using contour edge detection. For feedback purposes, the detected Sudoku is highlighted.
4. Once the user is confident that the Sudoku is highlighted correctly, it is captured with the click of a button.
5. The Sudoku is cropped off the camera feed and is then displayed.
6. If necessary, the orientation of the cropped puzzle can be corrected.
7. Clues can be extracted with the click of a button. For the extraction of clues, the image is divided to 81 equally sized images, one for each cell. The content of the divided images is then recognized using AutoML.
8. Once clue extraction is completed, the detected puzzle is presented.
9. If necessary, errors in the puzzle can be corrected.
10. The puzzle can then be solved with the click of a button.
11. If multiple solutions exist for the puzzle, navigation between solutions is possible.

4.2 User Interface

Implementing the User Interface(UI) was the first step to ensuring the results produced by the app are useful for the user. the UI was developed first to allow for easier integration of the other parts.

4.2.1 Board

Visualising the board was a much more complicated task than one would expect. To the best of my knowledge, there was no recommended way to create a board in android. The most common way is using a GridView [4], which "shows items in two-dimensional scrolling grid". Because of the way GridView is updated to accommodate changes in the sizes of the grid and enable scrolling when necessary, there is an additional performance cost. Since the puzzle to be represented is of constant size, alternatives were considered. A grid with constant size could be implemented as a Table [7], however, the documentation of the table component suggested " For better performance and tooling support, you should instead build your layout with ConstraintLayout". After carefully examining the documentation of each of the three options, the use of a ConstraintLayout was preferred, as the board is quite simple. By using this component, the layout was kept lightweight, improving performance and making changes easier. Instead of a hard-coded XML layout, it was decided to create the grid programmatically. This decision allowed for better encapsulation of the board, as only one file would need to be changed when changes to the layout were necessary. Furthermore, producing a layout through algorithms instead of using XML, removes the temptation of hard-coding values, improving compatibility across devices. From my experience, the only disadvantage of this approach is a delay during the creation of the layout. This delay was negligible compared to the time taken to recognise or solve a puzzle and was ignored.

For the creation of the board, 81 TextView objects are created using a loop, which are positioned in a constraint layout according to their position. The id of each TextView is stored in an array to allow changes to the object, such as contents and colours, at run-time.

In order to visually differentiate the different blocks, the use of alternating colours between cells of neighbouring blocks was decided. Finding the indexes of cells that must have the same color was not trivial. A Karnaugh map was used and a formula was derived. Taking the row number of the cell as x and the column number as y (starting for 1) the color of the cell is decided according to the following Boolean expression:

$$Colour(x,y) = ((x\,mod\,7) <= 3) \oplus ((y\,mod\,7) <= 3) \tag{4.1}$$

The resulting board is shown in figure 4.1

The cell borders were not implemented as adding partial borders to a TextView object was not trivial. For this to happen, implementation of a custom layout for each variation of borders would be required. Such implementation would be time consuming without any significant benefit and was postponed for next year.

Visual Sudoku Solver EDIT

9	0	6	0	7	0	4	0	3
0	0	0	4	0	0	2	0	0
0	7	0	0	2	3	0	1	0
5	0	0	0	0	0	1	0	0
0	4	0	2	0	8	0	6	0
0	0	3	0	0	0	0	0	5
0	3	0	7	0	0	0	5	0
0	0	7	0	0	5	0	0	0
4	0	5	0	1	0	7	0	8

SOLVE

Figure 4.1: Board implementation

4.2.2 Puzzle Representation

For the internal representation and storage of the puzzle, a 9X9 array was preferred to allow for one-to-one correspondence with the UI. char would seem like the obvious choice of data type, however int was preferred as it is often used by vision libraries as a hint when scanning characters. The only drawback of this approach is that int used 4 bytes of memory, compared to 1 Byte used by char. The difference is insignificant, as the size of the array is constant. The array was encapsulated inside a class called ”Board”. The class also contains methods to validate the board against the constraints of the puzzle. To check whether repeated clues appear in any row, the rows are traversed one by one, adding numbers already seen to a set. If an element already exists in the set prior to addition, the row violates the constraints of the puzzle. Without using a Set, all elements would need to be compared pairwise, leading to a complexity of O(n^2). Checking whether an element is contained and adding elements takes constant time, hence the cost of the operations is O(1). Hence, by introducing a set structure, the complexity of checking for duplicates was reduces from O(n^2) to O(n).

4.2.3 Puzzle Editing

Giving the user the option to edit the recognised clues before solving it is a feature that was not initially planned. The use of a ConstraintLayout as described above simplified the process, as any additional components required could be added to the layout without altering the board. One way to get user input would be using the phone’s keyboard. This approach requires thorough validation of the input as well as scaling of the rest of the screen when the keyboard appears or disappears. The fact that the number of possible inputs is limited to 10 meant that each input could be mapped to a button on screen, eliminating all the disadvantages of the aforementioned approach. The edited puzzle is saved in a separate puzzle to allow discarding of any unwanted changes.

The resulting implementation is shown in figure 4.2. Edit mode can be entered by clicking ”edit” shown in figure 4.1. A cell is selected when clicked. The selected cell is highlighted. The content of the cell can then be changed by clicking the button with the desired input. Changes can then be saved by clicking ”DONE” or discarded by clicking ”X”

4.2.4 Displaying multiple solutions

Navigating between solutions for a solved puzzle is a key functionality of the UI. Buttons were added below the board to display the next or previous solution. When the first solution is displayed, the previous button is grayed out and becomes unclickable. Similar care was taken for displaying the last solution A TextView displaying the index of the solution displayed was also added.However, this functionality is only necessary when multiple solutions are found. If there is only one solution, the objects described above are hidden. This functionality is shown in figures 4.3 and 4.4. The puzzle was then edited so that it only has one solution. The result is shown in figure 4.5

Visual Sudoku Solver DONE X

9	0	6	0	7	0	4	0	3
0	0	0	4	0	0	2	0	0
0	7	0	0	2	3	0	1	0
5	0	0	0	0	0	1	0	0
0	4	0	2	0	8	0	6	0
0	0	3	0	0	0	0	0	5
0	3	0	7	0	0	0	5	0
0	0	7	0	0	5	0	0	0
4	0	5	0	1	0	7	0	8

1	2	3	4	5
6	7	8	9	0

Figure 4.2: Editing a puzzle

Visual Sudoku Solver

Solution 1/2

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

PREV NEXT

Figure 4.3: First solution

Visual Sudoku Solver

Solution 2/2

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

PREV NEXT

Figure 4.4: Second solution

Visual Sudoku Solver

Solution 1/1

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

Figure 4.5: Puzzle with one solution

4.3 Solving Algorithm

An outline of the developed algorithm was described in section 3.1.3.1. Out of the requirements set in section 3.2.1, the speed requirement is the most challenging. As a result, care was taken where possible to make the algorithm faster.

4.3.1 Kotlin Implementation

Many implementation details were not clear from the design of the algorithm. One such example is the data type that was used for the internal representation of the contents of the puzzle. In order to be consistent with the UI, as outlined in section 4.2.2, a 9x9 array of integers was used. This decision also allowed for easier testing, as the benchmark that the algorithm would be compared to, also used integers. This is discussed in section. 5.2.

The set of illegal values for each empty cell was implemented using a HashSet. As the insertion order is not important, HashSet will give better performance compared to LinkedHashSet. TreeSet has the worst performance of the three options as the order of the elements has to be maintained after insertion or deletion of an element, which is not necessary.

To order the sets based on their size, a custom class named "CellHashSet" was created to couple each HashSet with the id of its corresponding cell. The id of the cell is calculated as:

$$id(row, col) = col + row * 9 \tag{4.2}$$

and ranges from 0 to 80. Hashsets are also stored in an array based on the id of their corresponding cell to allow faster retrieval. As objects in Kotlin are stored by reference, no duplicate storage is produced. A method that returns the size of the HashSet and a comparator method was also implemented in the class to allow direct comparison between instances. An instance of CellHashSet is initialised for each cell, and then stored in an array. Once initialisation is completed, the array is sorted in descending order, resulting in the instance corresponding to the most constrained cell being placed first. Kotlin sort method uses a version of MergeSort, that has a time complexity of O(nlogn) When a cell is filled, the HashSets of cells that are in the same row, column or block are updated. When a HashSet is updated, order is maintained by swapping the updated HashSet with the one that is ordered before it, until the updated HashSet is ordered correctly. Maintaining order has a time complexity of O(n).

When a solution is found, a Board object is initialised and added to an arrayList. The solution must be cloned before being added to the board. Cloning takes time and will impact the performance of the algorithm for puzzles with a large number of solutions. Unfortunately, there is no way around the issue, as Kotlin does not allow passing arguments by value, something that is common in languages such C or C++.

To handle extreme cases such as the empty grid, the algorithm was programmed to halt the search once a predefined number of solutions was found. This value was fixed to 10.

4.3.2 Multithreading

Utilising multiple threads for the execution of tasks that require a lot of processing power is common practice. In android, updating the UI is often handled by the main thread, to ensure that the UI is prioritised and remains responsive. Tasks that can be run independently with the UI are run on separate threads. However, it was decided that no multithreading would be used. The reason for this decision was to ensure that the solving time was consistent. Prior experience showed that secondary threads can be slightly delayed. Only using a single thread ensured that the results reported during evaluation were as accurate as possible.

4.4 Recognition

For reasons discussed in section 3.1.4, OpenCV library was used to implement all vision related tasks. All implementation tasks related to the detection of a puzzle and extraction of clues, including processing of the puzzle are discussed in this section. When applicable, alternatives considered are also discussed.

4.4.1 Library Setup

Setting up OpenCV for my android project was not as trivial as expected. A significant amount of troubleshooting was necessary to ensure that OpenCV was compatible with all the android dependencies used by the UI. Surprisingly, all libraries of OpenCV are installed together and a separate library manager is required to isolate libraries that end up being used by the project. As the required libraries could change in the future, the library manager was ignored. Consequently, the size of the app and the build time increased. This could be easily fixed once the set of libraries used by the project is finalised.

4.4.2 Camera Use

The first step towards detecting a puzzle was enabling the phone's camera. For this to happen, the app must be granted camera permission. Recent versions of android require that permissions are granted at run-time. A popup dialogue was implemented, prompting the user to give permission when the app is loaded, as shown in figure 4.6. If no permission is given, the app is terminated.

The displayed camera feed was not oriented correctly, but rotated 90 degrees anti-clockwise. Investigation showed that this is a common issue for OpenCV projects on android. The reason this problem occurred was because OpenCV ignores the configuration of the phone's camera. The app was tested on mobile phones of 4 major phone companies (Samsung, Google, Oneplus and Xiaomi) and the same behavior was observed. The most common solution found was rotating the camera feed by 90 degrees clockwise. However, there was no confirmation that this solution would work on every android phone. To improve this solution, the configuration of the phone's camera is checked at run-time to ensure that a rotation is indeed necessary. One example where the feed displayed was correct without any additional rotation is my laptop's webcam.

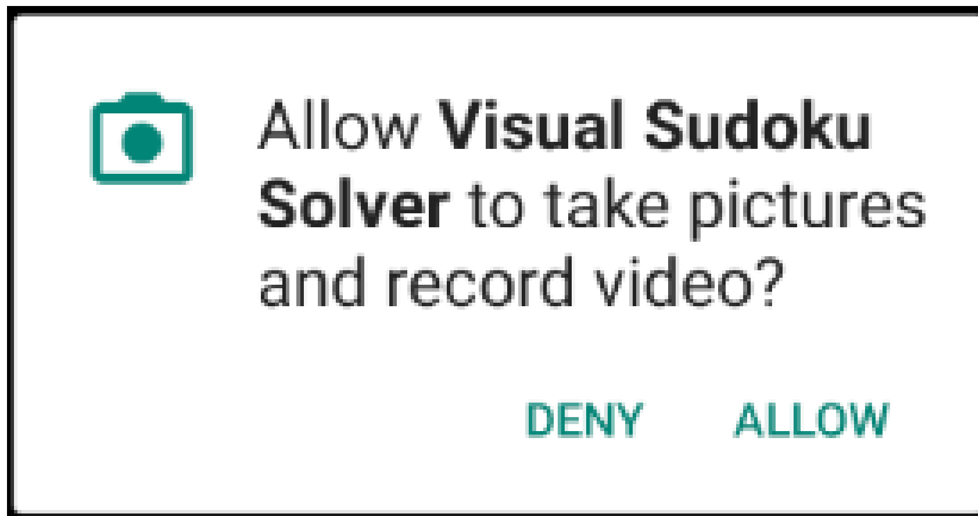

Figure 4.6: Camera permission request

Using the implemented solution, it is deduced that no rotation rotation is required and the step is skipped.

Rotating the camera feed 90 degrees clockwise when necessary was also complicated. Most solutions found online resulted in rotation of the feed displayed without rotating the feed itself. To ensure that the feed is indeed corrected, functionality allowing picture capturing was added. Different solutions were tested until the captured picture was oriented correctly.

4.4.3 Puzzle Detection

Before detecting the puzzle, copies of frames from the camera feed are converted to grayscale, using the OpenCV method ”RGB2GRAY”. This method uses equation 2.2 so no additional implementation was required for this step. It is worth noting that color in OpenCV is stored as BGR instead of traditional RGB format. Implementation was not influenced by this detail, as all color manipulation was done by preexisting method of OpenCV.

Gaussian adaptive thresholding was then applied to the image. Even though global thresholding is often used in literature for the detection of the puzzle, its use would heavily degrade the performance of the detection algorithm in cases with insufficient lighting.

Canny edge detection is applied to the image to detect the edges. After experimentation with pictures of Sudoku taken at different lighting, a minimum threshold value of 155 was determined to be the most efficient for canny edge detection. Use of lower values resulted in unnecessary detection of irrelevant edges. On the contrary, higher values resulted in failure to detect the edges of some puzzles. Similar apps use a lower threshold between 145 and 180.

Gaussian blur is added to the resulting image to smooth out noise.

Contour edge detection is applied in order to find the rectangles in the image. Contours are then sorted based on area. The largest contour is assumed to be the Sudoku. The above steps are for internal purposes only and do not affect the image displayed by the app, the original feed from the camera is displayed instead.

The largest contour must be displayed to create a closed feedback loop. For this to happen, the camera feed is cloned and the largest contour is drawn on top of the cloned image. To draw the contour, lines connecting edges of the that share the same x or y coordinate are drawn. The result is shown in figure 4.7

Once the puzzle is correctly highlighted, the user can click the scan button to isolate the puzzle. To crop the puzzle, the vertices of the rectangle are used. The result is show in figure 4.8.

4.4.4 Orientation Correction

For unknown reasons, the orientation of the cropped puzzle is not always correct. Sometimes the cropped puzzle is displayed rotated by 90 degrees clockwise, or anti-clockwise, even when the exact same process is followed for the capture of the image. The orientation problem explained in section 4.4.2 was ruled out as a suspect, as the orientation error is no longer consistent. Examining the vertices of the cropped rectangle did not lead to a solution. One way to tackle the issue was to add functionality that allows manual correction of orientation. Addition of this functionality would enable approaching puzzles from various angles. The image of the puzzle can be rotated clockwise or anticlockwise, using buttons 1 and 2 in figure 4.8. Once the orientation is correct, the clues can be extracted using button 4. If the picture taken is not satisfactory, a new picture can be captured by clicking button 3. The rotated image is shown in figure 4.9

4.4.5 Clue detection

To detect the clues, a firebase AutoML model was used for reasons outlined in section 3.1.8.

Initially, detection of clues using preexisting methods of AutoML was attempted. Put simply, the character recognition engine started from the top left of an image, moved to the desired cell and scans the cell's content. A hint was given to the engine regarding the type of the scanned values (integer). Experimentation showed that an integer hint did not eliminate other characters. The only benefit of using such hints was that sometimes, if the engine could not make a decision regarding the scanned content, a question mark was returned. This functionality could be beneficial, as cells marked with a question mark could easily be corrected by the user. If the content of a cell was matched to a character between '1' and '9'. the clue for the corresponding cell was set to that character, otherwise the cell was assumed to be empty. To scan the contents of all the cells, a nested for loop was used, fully scanning one row before moving on to the next. To communicate with the UI, the scanned values were stored in an array. A value of 1-9 represented a valid clue, a value of 0 represented an empty cell and a value of -1 represented the question mark.

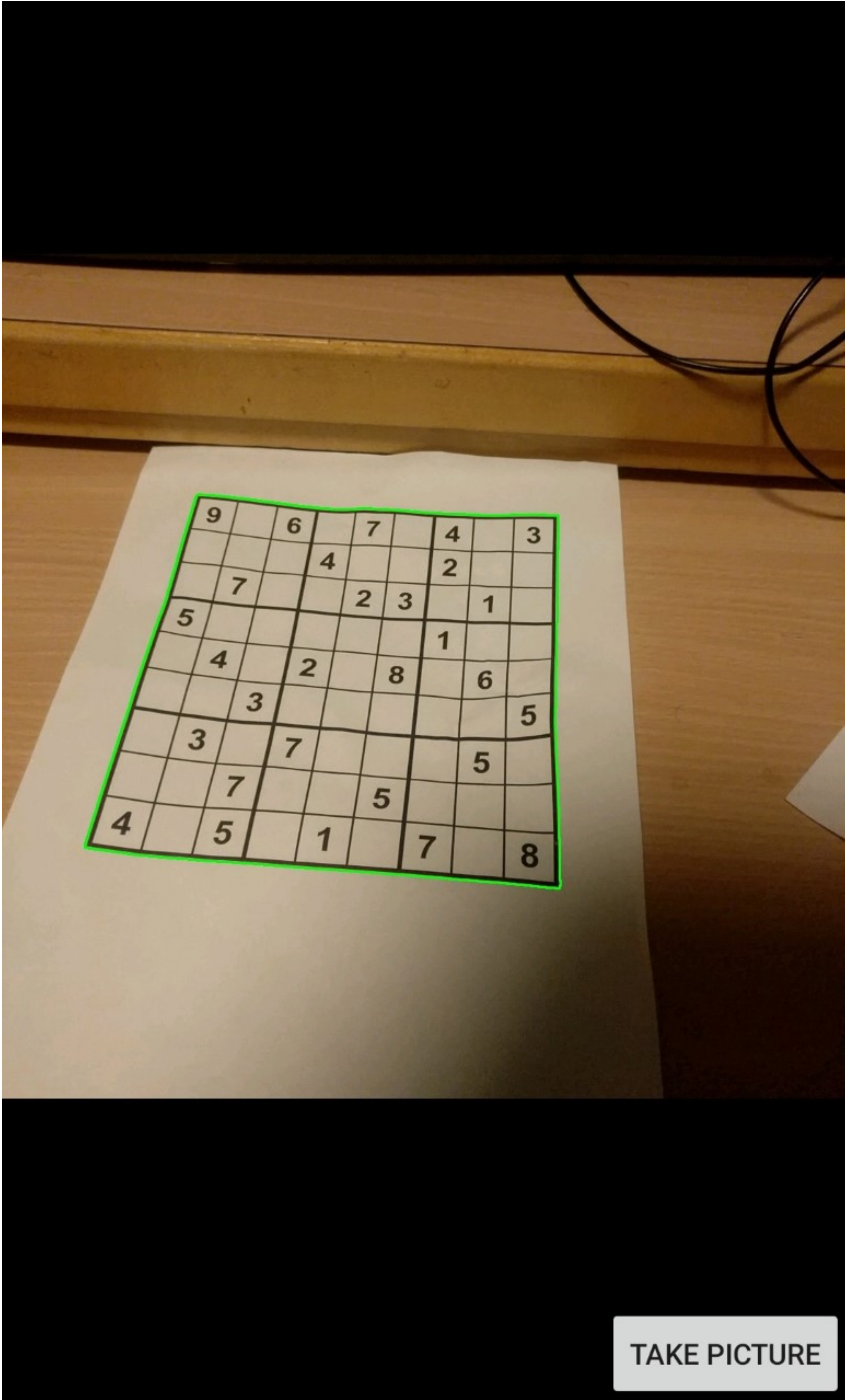

Figure 4.7: Detection and highlighting of Sudoku

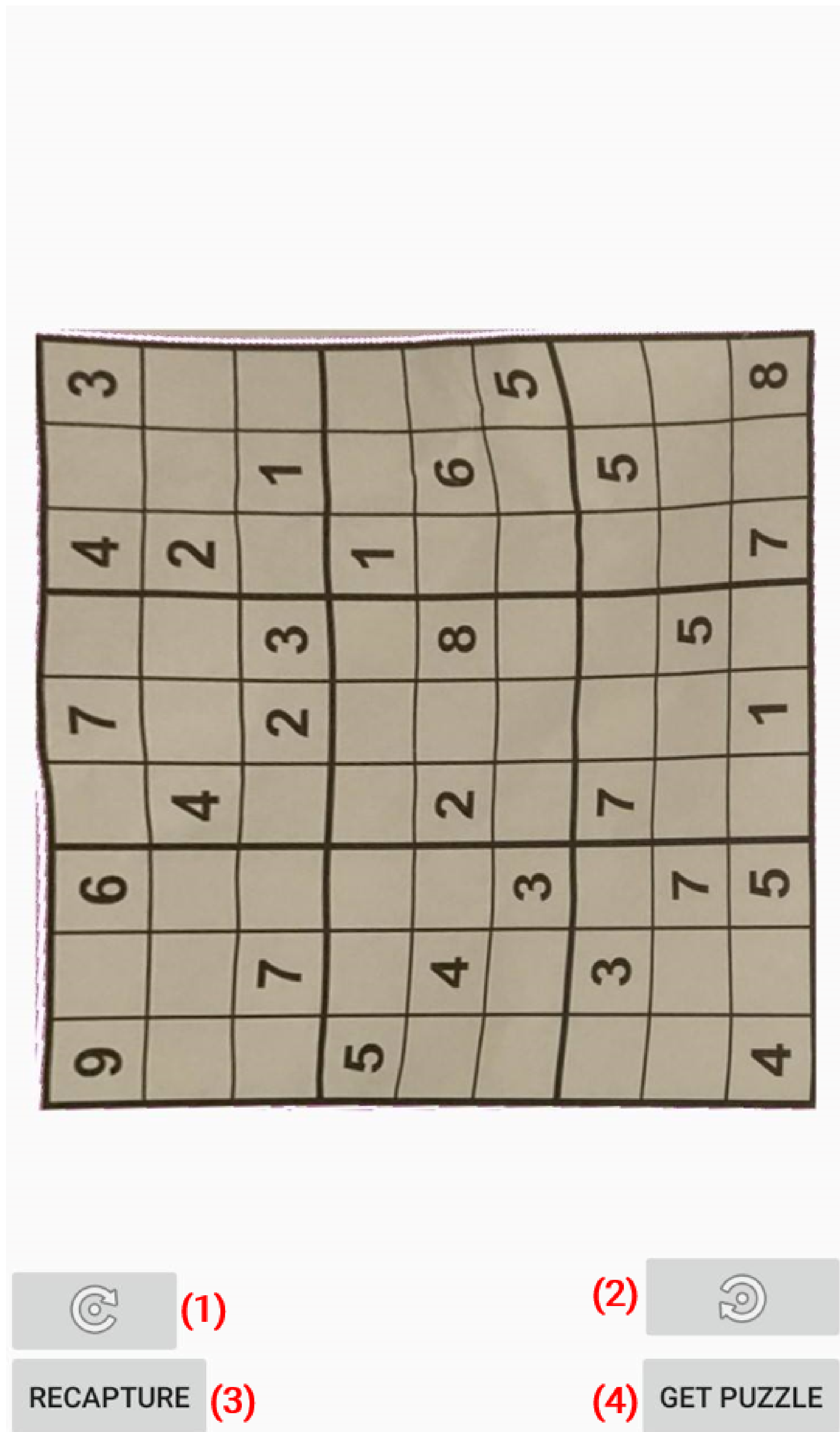

Figure 4.8: Incorrectly oriented grid

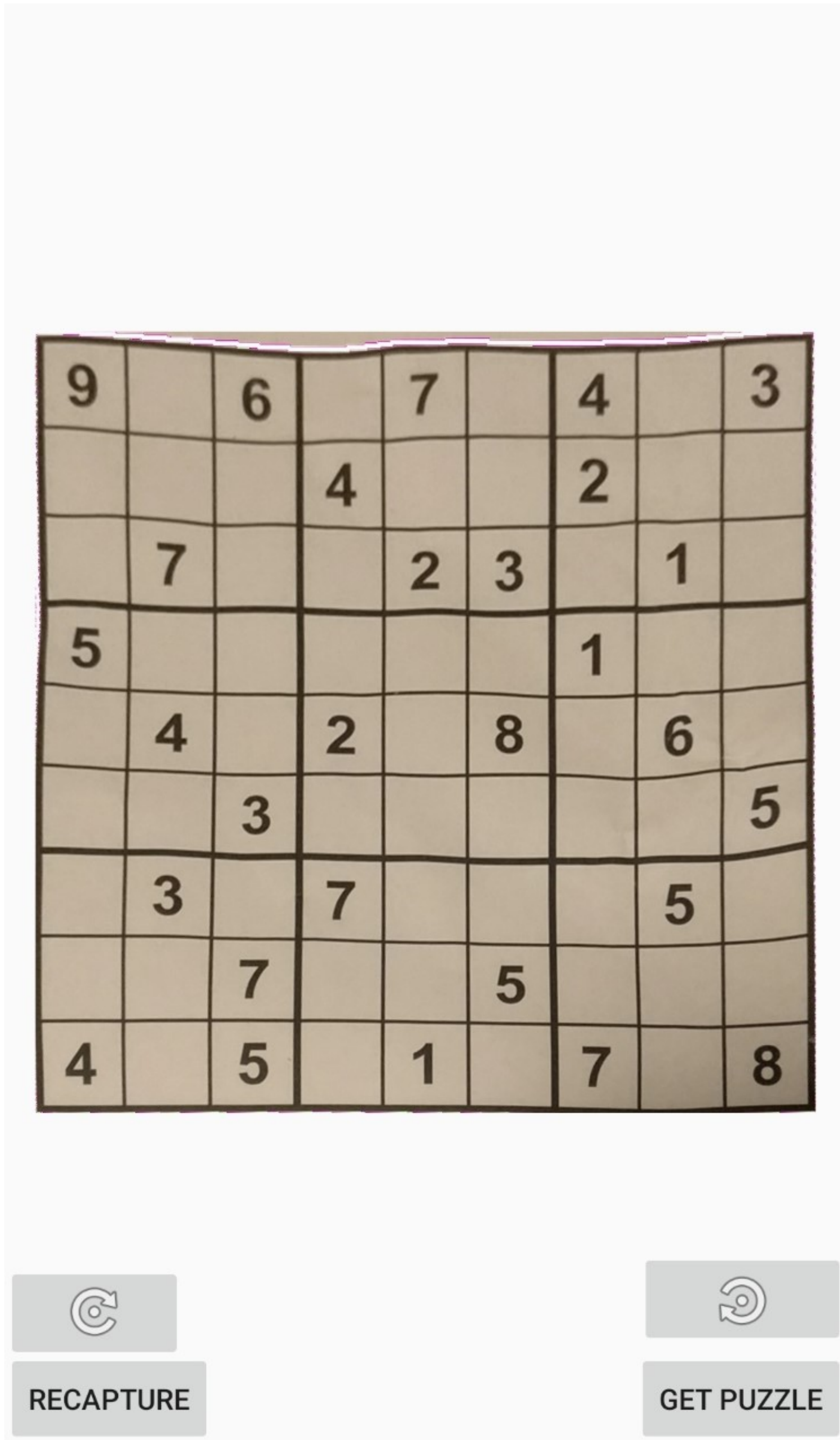

Figure 4.9: Correctly oriented puzzle.

Even though this method showed some very positive results and clues were correctly identified, they were not always placed in the correct cell. Experimentation showed that the error could occur in any cell adjacent to the target cell. This error could be attributed to the way that the puzzle was passed to the recognition engine. Further experimentation did not resolve the issue, and no patterns were observed during testing. After some time, this attempt was deemed a failure and was abandoned.

As the above approach gave encouraging results regarding the accuracy of digit recognition, an alternative use of AutoML was attempted next. The fact that the puzzles are isolated allowed segmentation of the puzzle to 81 smaller images, one for each cell.

For an image of height h and width w the x and y coordinates of a cell are calculated by:

$$(w/9) * (column - 1) <= x <= (w/9) * column \tag{4.3}$$

$$(h/9) * (row - 1) <= y <= (h/9) * row \tag{4.4}$$

AutoML was used in the same way described for the previous approach, giving positive results

Chapter 5

Evaluation

The various components of the app were tested independently and the results are presented in this chapter. When applicable, the limitations of the test are also discussed and possible alternatives are suggested. The chapter concludes with some user evaluation.

Due to the COVID-19 pandemic, evaluation did not go ahead as planned. Reasons include lack of test subjects and limited printed puzzles due to social distancing measures.

The app was tested using a OnePlus 3t smartphone, running android 9. The phone uses a snapdragon 821 processor, 6GB of RAM and a rear camera of 16MP. Use of emulators was not preferred as prior experience showed that they can be significantly slower than physical phones. Furthermore, existing literature does not provide quantitative results for tests ran on emulated devices. As a result, benchmarking would not benefit from the use of emulators.

5.1 User interface

5.1.1 Response time

The time taken for the board to be updated when a puzzle is solved or a different solution is displayed was measured. As all the cells are updated sequentially, the process takes constant time for every solution. Measurements showed an update time between 14 and 32 milliseconds. This delay is unnoticeable and no further improvement is required.

5.1.2 Error handling

Even though the app does not crash under normal circumstances, extra care was taken to ensure that detailed error messages are displayed in cases that no clues were found or the puzzle detected. Suggestions regarding steps to resolve the issue were also included in the messages. Further improvement is possible with the addition of popup dialogue boxes, explaining the possible options.

5.2 Solving

As expected from a backtracking algorithm, the solver gave correct results for all the puzzles that were tested. Up to 10 solutions were found for puzzles that have as many, while no solutions were found for unsolvable puzzles.

5.2.1 Time taken

To evaluate the speed of the algorithm, multiple puzzles of different difficulty were tested. Mantere, Timo and Janne Koljonen [47] used various puzzles of different difficulties to test three different algorithms. The puzzles and the results were later published on the web [8].

The test instances appearing in the second column (a) were used to evaluate the algorithm. The solutions found were compared to the solutions provided and were correct. As the algorithmic is deterministic, the same solutions were provided every time the algorithm was used.

The average solving time for each puzzle is shown in table 5.1. To make a fair comparison, the algorithm was tweaked to terminate when the first solution is found. The experiment was repeated 5 times for each puzzle.

Difficulty level	Average time taken (milliseconds)
1	62
2	84
3	147
4	259
5	518
Easy	55
Medium	587
Hard	1074
AI Escargot	958

Table 5.1: Average solving time for puzzles of varying difficulty

The above results can be summarised to the following key points:

- Naturally, solving time is proportional to the difficulty of the puzzle. Puzzles with less empty cells are solved faster.
- A puzzle that is harder to solve for humans is not necessarily harder to solve for a backtracking algorithm. This is evident by the last two rows of the above table. A possible explanation would be because more branches need to be explored before a solution is reached.
- Initialisation overhead makes up for a big portion of the solving process. This is evident by the time taken to solve easy puzzles. The results reported by Mantere, Timo and Janne Koljonen for the easy puzzles are better for the easy puzzles.

5.3 Recognition

5.3.1 Grid detection

For the detection of the grid, multiple aspects were evaluated. The time taken before a user can detect and isolate a grid is of prime importance. The probability with which the puzzle is oriented correctly is also important, but can be corrected easily using the rotation functionality. Finally, The performance of the detector was evaluated under extreme lighting conditions

To test the orientation of the puzzle, the puzzle shown in figure 7.1 was scanned 10 times, in constant lighting. The results are tabulated below:

Test Instance	Time taken (nearest second)	Orientation offset (degrees)
1	4	90
2	6	0
3	3	0
4	7	0
5	2	270
6	8	270
7	3	90
8	5	0
9	6	0
10	6	0

Table 5.2: results from 10 detection of puzzle 7.1

An average detection time of 5.1 seconds was required for this puzzle. Detection time varied between 2 and 8 seconds. Even though the time required could be improved, the results are encouraging.

It is worth noting that holding the phone at approximately 45 degrees angle towards the page greatly increased the detection success rate. This can be attributed to the camera configuration of the phone used, or to the fact that lighting was not blocked by the phone's shadow. Once the puzzle was correctly highlighted, it remained highlighted until its capture, despite minor camera movements.

Regarding the orientation, the puzzle was oriented correctly 60% of the time, while the remaining 4 times could be corrected with the click of a button.

The detector was also tested against the image shown in figure 7.2 to evaluate its performance when multiple puzzles appear on the same page, something common in newspapers. The results were also positive. If all four puzzles are visible in the camera frame, one of them is consistently highlighted (usually the top left). Moving the camera towards the desired puzzle, making it the only puzzle visible in the frame, resulted in the detection of the puzzle. This behavior is intended and is considered a big success.

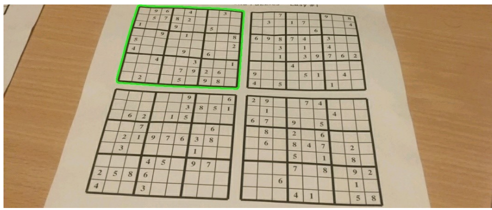

Figure 5.1: Left puzzle detected

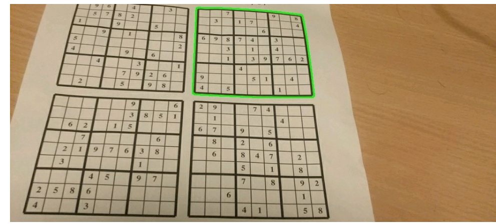

Figure 5.2: Right puzzle detected

5.3.1.1 Detection issues

Even though the implementation of the detector is considered a success. Multiple scenarios where the observed behavior is not ideal exist. For example, if the camera feed is centered between two puzzles, the detected puzzle will alternate, even with slight camera movements as shown roughly by figures 5.1 and 5.2. This behavior makes capturing of either puzzles almost impossible without moving the camera, as the same puzzle does not remain highlighted for sufficient time to allow capture.

Another issue has to do with the behavior of the detector when the whole page containing the Sudoku is part of the frame. In this case, the whole, or sometimes part of the page is detected as the contour, as shown in figure 5.3. This behavior is natural as the whole page is indeed a rectangle and is interpreted as a contour due to the contrast with the background. This problem can be easily solved by moving the camera closed to the puzzle.

Finally, the detector does not work well in dark settings. This can be attributed to the use of thresholding before detecting the contours. In low lighting, the grayscale value of the whole image is below the minimum threshold set. As a result, the whole image is considered black and no contours can be detected. An example is shown in figure 5.4

5.3.2 Extracting clues

In this section, the position of the clues detected is evaluated. This evaluation is crucial to determine whether the partition of the image to 81 smaller grids was a reasonable decision. Once again, image shown in figure 7.1 was used. This puzzle contains 27 clues, of which 9 are positioned at the edge of the grid (first and last row or column). These clues are particularly important to evaluate the efficiency of the Sudoku cropping performed after detection.

Out of 10 trials, all recognised clues were placed correctly. All errors occurred were due to incorrect recognition of a number, including the case where no clue was returned at all. This result confirmed that cropping the puzzle and dividing the resulting picture to 81 partitions was implemented correctly.

Out of 270 total clues, 21 were recognised as empty and only 4 were recognised as a different number. The reason the majority of errors are reported as empty cell could be attributed to the fact that the extraction process is too strict. Inspecting the project logs showed that sometimes numeral 2 is recognised as an upper or lower case 'z'.

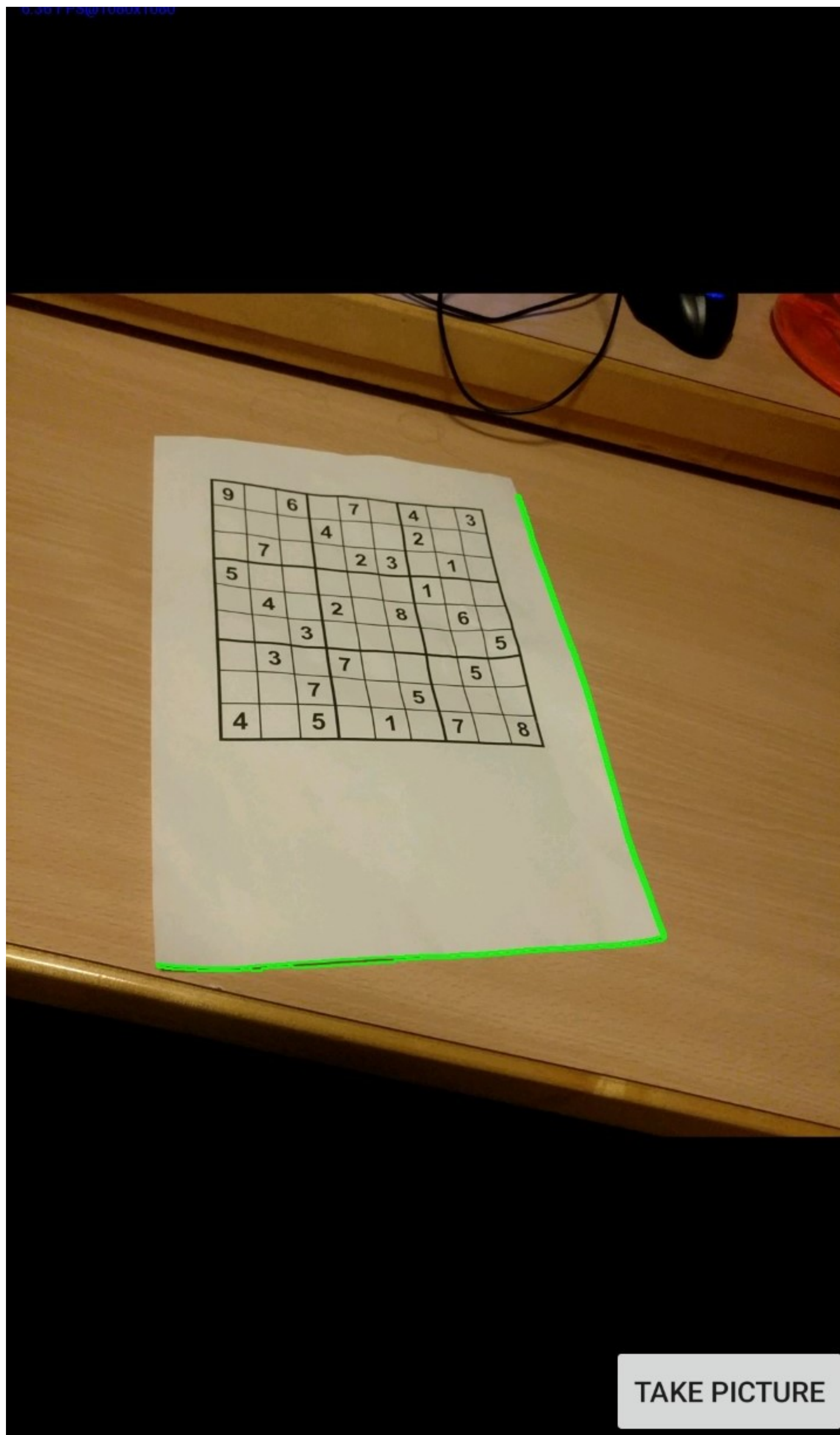

Figure 5.3: Page detected instead of Sudoku

Figure 5.4: Undetectable puzzle due to low lighting

Similar behavior was observed with '5' and 'S' and 'T' and '7'. These cases were then discarded by the extraction process, and the content of the cell was assumed to be empty. Out of the 21 cells that were incorrectly reported as empty, 9 of these were on clues based on the edge of the grid. This can be attributed to the fact that the page was slightly creased, resulting in the specific cell being imperfectly cropped. The 4 cells incorrectly recognised were all due to a 1 being recognised as a 7 (3 times) and vice-versa (1 time). It is worth noting that, for the grid selected, 7 appears as a clue 5 times, while 1 only appears three times.

The extraction process was also timed. Reported times varied between 4 and 12 seconds. 9 out of 10 trials required less than 10 seconds. Since the same puzzle was used, the large variance in extraction time is most likely observed because of the way AutoML works internally.

Even though there is plenty of room for improvement, the extractor is considered a success and could be further improvement by making the extraction criteria less strict.

5.4 Requirements evaluation

In this section, the performance of the app is evaluated against the requirements set in section 3.2.

5.4.1 Solver

The solver fulfils all the requirements set during the design stage. A correct solution will be found to any puzzle if it exists. If the puzzle has more than one solution, a total of 10 solutions will be found before the solver terminates. The order of solutions found remains constant if the same puzzle is solved multiple times.

The only requirement that was only partially fulfilled was the one regarding solving speed, which was set to 1 second per solution. The harder puzzles require slightly more time to be solved. In hindsight, the requirement set was too optimistic. However, the algorithm could be further improved for the second part of the project to meet the requirement.

- **Capability:** Given any Sudoku, the solver is able to find up to 10 solutions to any puzzle.
- **Determinism:** The solver always provides the same results for the same puzzle. If there is more than one solution, the order of the solutions returned is the same for the same puzzle. This happens because the order branches are explored is the same.
- **Correctness:** The solver provides solution(s) to any puzzle, given their existence. No solutions are provided for puzzles that are unsolvable.
- **Speed:** The speed requirement of up to one second per solution is fulfilled for most puzzles. Nearly empty grids take slightly longer than one second for the first solution.

- **Flexibility:** The maximum number of solutions returned by the solver is flexible, as it is dictated by a constant.

5.4.2 User Interface

- **Intuitiveness:** The implemented interface is easy to understand and is usable without any prior training. This could be further improved with a tutorial feature and use of better icons.
- **Responsiveness:** The interface is responsive to the user's actions without any noticeable delay. Feedback is generated during the puzzle recognition phase and a loading circle is displayed when the clue extraction process is initiated. The only scenario where the app does not respond for a few seconds is during the solving process of puzzles with many solutions.
- **Robustness** The app was tested thoroughly and did not crash.
- **Functionality** Given an internal representation of a puzzle and its solutions, the interface is capable of displaying the unsolved puzzle and its solution(s). Navigation between solutions is also possible.
- **Speed:** The interface only takes a few milliseconds to update the grid.

5.4.3 Grid detector

- **Responsiveness:** The detector responds to the camera moving and adjusts without any noticeable delay, while providing feedback.
- **Capability** Given a clear view of a Sudoku puzzle, the detector detects and highlights the puzzle. The detector extracts the highlighted puzzle when the user confirms that the required puzzle is correctly highlighted.
- **Stability:** The detector tolerates small movements of the camera, making capturing of detected puzzles easy. This excludes the edge case where the frame is centered between two puzzles.
- **Time required:** During evaluation, less than 10 seconds were required to detect a puzzle, with an average time of 5 seconds.

5.4.4 Clue extractor

- **Responsiveness:** The extractor displays a loading circle while the extraction takes place.
- **Capability:** For pictures that are well lit and of high resolution, the extractor recognises 91% of the clues correctly.
- **Additional Capability** For reasons mentioned at the start of the chapter, this requirement cannot be evaluated.

- **Correctness:** For the puzzles used, clues are only found for cells that contain a clue.
- **Speed:** The speed requirement of 10 seconds is met the majority of the time. Only one of the 10 extraction trials that were timed took more than 10 seconds,

5.5 User Evaluation

Even though user evaluation was originally planned for the second part of the project, a very basic attempt was made during the year in order to detect any fundamental flaws and ensure that app would appeal to potential users. Due to the COVID-19 pandemic, finding test subjects became impossible towards the end of the project, in which period the app would have the greater benefit. A total of 7 students were asked to use to the app to recognise and solve a puzzle. To ensure that the survey complied with GDPR regulations, no data that could be used to identify the participants was stored. The app received positive feedback from all the participants.

A summary of the problems encountered by the users and suggestions for improvements is outlined below. Unless explicitly stated, the implantation of the suggested improvement was postponed for next year.

- Addition of a tutorial feature that explains the features available to the app.
- Addition of title screen instead of the camera feed being displayed when the app is loaded.
- Use of more intuitive icons for actions such as confirm or discard editing.
- Use of more instructive error messages. Most common example raised was the occasion that no solution were found to the given puzzle, or if the puzzle violated the necessary constraints. The error messages produced were updated to prompt the user to make sure that the scanned puzzle is correct and edit it in case it was not.
- Editing feature currently gives the impression of a necessary step while it is optional. This is deliberately the case, to ensure that the functionality is visible. Once a tutorial is added, editing would not need as much advertising.
- When editing, legal values for a cell could be highlighted to improve user experience. It is worth noting that allowed values should not be constrained during editing. Editing should not require a particular order of changes to achieve the desired result.
- Addition of a ”cancel” button, to halt the solver in case solving takes too long, or the puzzle entered is incorrect. A maximum running time allowed for the solver was also suggested.

Chapter 6

Conclusions

6.1 Project Achievements

The achievements of the project can be summarised as follows:

- Extensive literature review covering solving and recognition techniques.
- Creation of a responsive and intuitive user interface with functionality to edit puzzles and display multiple solutions.
- Implementation of a fast Sudoku solving algorithm from scratch that can find all solutions to a puzzle given enough time. This includes various decisions regarding internal representation of puzzles and optimisation steps.
- Setup of and familiarisation with OpenCV library and usage of phone cameras.
- Implementation of a simple yet efficient algorithm for detection of puzzles using contour edge detection, providing feedback to the user.
- Implementation of functionality for cropping and displaying a puzzle, allowing correction of orientation.
- Use of AutoML vision library for the detection and recognition of clues from a cropped image.
- Design of various methods to test each component of the app.

All of the goals set at the start of the project were reached, As the developed app can indeed detect a puzzle using the phone's camera, extract its clues with relatively high accuracy and find its solution(s).

6.2 Project limitations

Despite the overall success of the project, various limitations exist. Some of these limitations are summarised below:

- As the detected puzzle is divided to 81 smaller images for the recognition of the clues, the success rate is highly dependent on the quality of the picture taken. Blurry or out of focus images will result in poor results. This is offset by the recognition mechanic used, as holding the phone stable is encouraged for the detection of a puzzle. The quality of the camera used also influences the efficiency of recognition. Fortunately, cameras used on smartphones have improved drastically over the past decade. Use of the app on any modern android phone should produce excellent results.

- Due to the relatively small size of the training data, the recognition performance is likely to be impacted if the app is used on a puzzle that uses an uncommon font. However, most published Sudoku use very similar fonts.

- Naturally, the app will not perform well if used in extreme lighting conditions or if the puzzle is not clearly separated from other objects. The app is still useful in these circumstances as the clues can be entered manually to produce a solution.

6.3 Future Work

Some possible extensions to the project are discussed in this section. Most of the proposed ideas are expected to be realised next year, for the second part of the project.

- **User testing/evaluation:** User evaluation is a core part of any software engineering project. Some user evaluation took place during this year, as discussed in chapter 5.5. A more thorough evaluation will be possible with a working app and looser time constraints.

- **Accuracy improvement:** Investigating ways to improve the accuracy of detection is the goal of the project. Many alternatives exist for each recognition component. The most promising alternatives can be implemented and compared with this year's implementation. One such example would be the recognition of the grid lines using Hough transformations and training a machine learning model for the recognition of the digits.

- **Detection and correction of errors:** Some of the puzzles recognised will violate the constraints of a Sudoku. Developing an algorithm that corrects some of these puzzles will further increase the success rate of recognition. No such algorithms were found during the literature review. Only a small number of recognised instances will benefit from such an algorithm. Another approach with greater benefit would be to highlight parts of the puzzle that violate the constraints to make user correction faster.

- **Speed improvement:** Changing the implemented solving algorithm could improve the solving speed. Possible changes include use of different data structures or converting the algorithm to Java.

- **Comparison of solving algorithms:** Implementing different solving algorithms would be a good way to benchmark the efficiency of the implemented algorithm.

An example of an algorithm that could prove faster than the one implemented is the dancing links algorithm.

- **Manual solving:** Functionality that allows the user to solve the puzzle manually and confirm steps by comparing with the solution found by the solver could be added.
- **Hint functionality:** Functionality to display hints when the user solves the puzzle manually could be added to improve the solving experience.
- **Tutorial:** As discussed in section 5.5, it was suggested that the app would benefit greatly from the inclusion of a tutorial explaining its capabilities.
- **Grid visualisation improvement:** The appearance of the grid could be improved. Grid lines to separate lines, columns and blocks are present in all printed Sudoku puzzles and would improve user experience.

6.4 General remarks

In this report, the implementation of an android app that can recognise and solve Sudoku was discussed. All the goals of the project were achieved to a satisfactory degree. The main areas that could be improved are the speed of the solving algorithm and the extraction of clues from a puzzle. Improvement in both areas is possible with further experimentation on solving algorithms and vision techniques used. Some parts of the project led to very encouraging results and very few changes will be necessary next year. Overall, the first part of the project is considered successful, considering the lack of prior experience with vision libraries and OpenCV in particular.

Chapter 7

Appendix

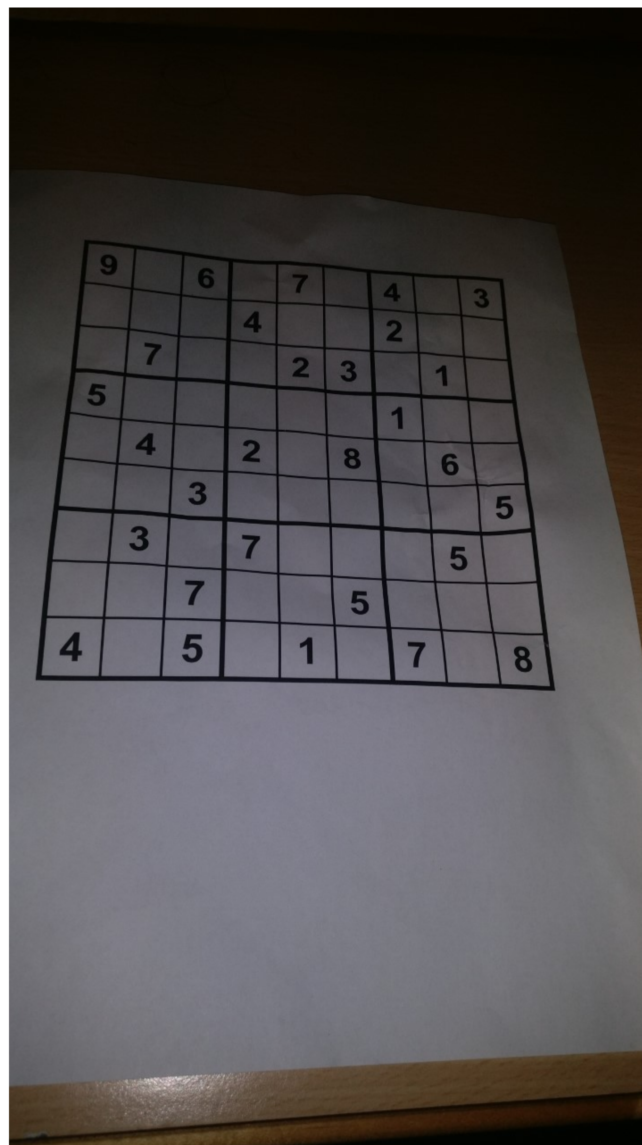

Figure 7.1: A single printed Sudoku

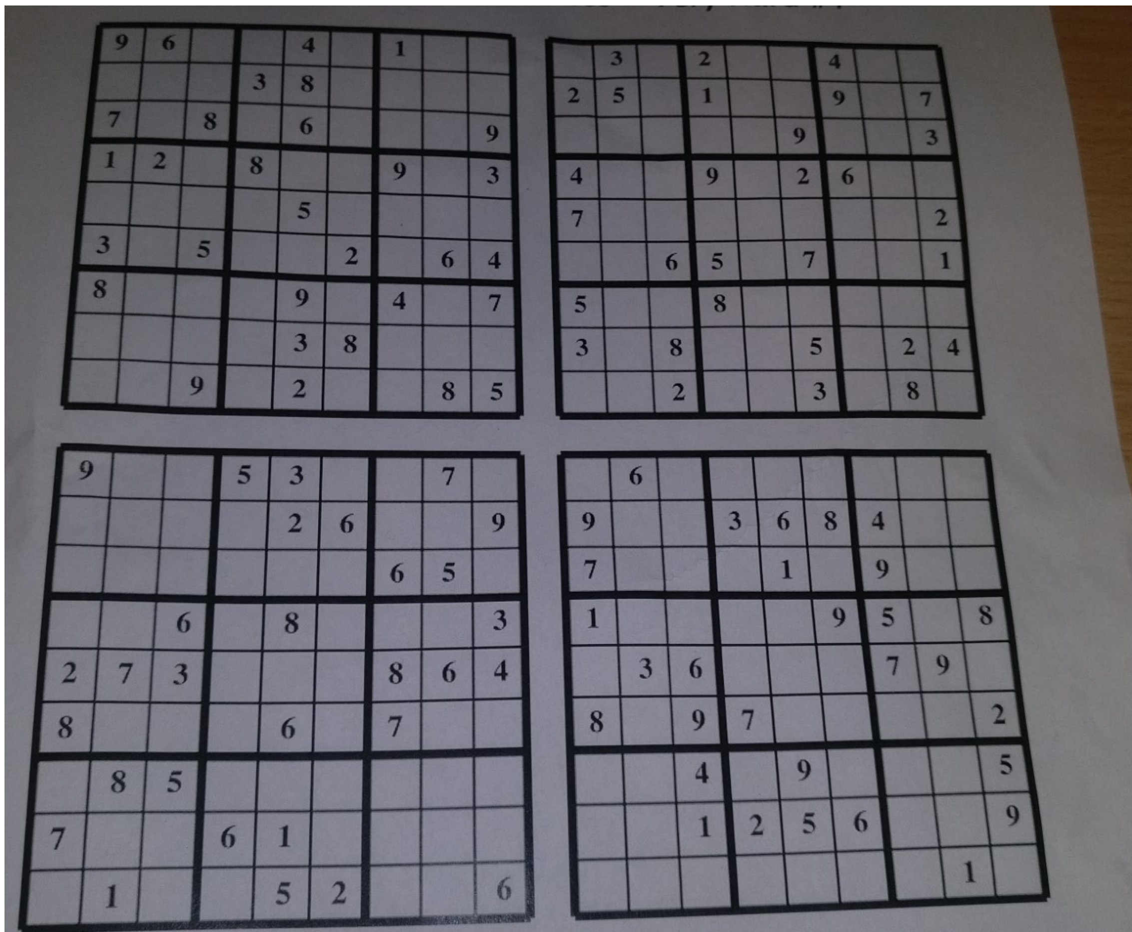

Figure 7.2: Multiple Sudoku on same page.

Bibliography

[1] About opencv. https://opencv.org/about/.

[2] Google android developers blog. http://back.ly/uLU0Z#https://android-developers.googleblog.com/2019/05/kotlin-is-everywhere-join-global-event.html.

[3] Google groups. https://groups.google.com/forum/#!topic/rec.puzzles/A7pi7S12oFI.

[4] Gridview documentation. https://developer.android.com/reference/kotlin/android/widget/GridView.

[5] Open cv image thresholding documentation. https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html.

[6] Opencv color conversions documentation. https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html.

[7] Table documentation. https://developer.android.com/guide/topics/ui/layout/grid.

[8] Test instances by mantere, timo and janne koljonen. http://lipas.uwasa.fi/~timan/sudoku/.

[9] An incomplete review of sudoku solver implementations, Jul 2011. https://attractivechaos.wordpress.com/2011/06/19/an-incomplete-review-of-Sudoku-solver-implementations/.

[10] I. Ahmad, I. Moon, and S. J. Shin. Color-to-grayscale algorithms effect on edge detection — a comparative study. In *2018 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4, Jan 2018.

[11] Krzysztof R. Apt. Principles of constraint programming. 2003.

[12] A. C. Bartlett, Timothy P. Chartier, Amy Nicole Langville, and Timothy D. Rankin. An integer programming model for the sudoku problem. 2006.

[13] Derek Bradley and Gerhard Roth. Adaptive thresholding using the integral image. *J. Graphics Tools*, 12:13–21, 2007.

[14] Roberto Brunelli. Template matching techniques in computer vision: Theory and practice. 2009.

[15] Haradhan Chel, Deepak Mylavarapu, and Deepak Sharma. A novel multi-stage genetic algorithm approach for solving sudoku puzzle. *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pages 808–813, 2016.

[16] Abu Sayed Chowdhury and Suraiya Akhter. Solving sudoku with boolean algebra. 2012.

[17] Broderick Crawford, Margaret Aranda, Carlos Castro, and Eric Monfroy. Using constraint programming to solve sudoku puzzles. *2008 Third International Conference on Convergence and Hybrid Information Technology*, 2:926–931, 2008.

[18] Dipti Deodhare, Shailesh Sonone, and Anubha Gupta. A generic membrane computing-based sudoku solver. *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, pages 89–99, 2014.

[19] Michael Dittrich, Thomas B. Preußer, and Rainer G. Spallek. Solving sudokus through an incidence matrix on an fpga. *2010 International Conference on Field-Programmable Technology*, pages 465–469, 2010.

[20] Akash Dutta and Arunabha Ghosh. Development of a character recognition software to solve a sudoku puzzle. *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 1–5, 2016.

[21] Mária Ercsey-Ravasz and Zoltán Toroczkai. The chaos within sudoku. In *Scientific reports*, 2012.

[22] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible sudoku grids. 2005.

[23] Bertram Felgenhauer and Frazer Jarvis. Mathematics of sudoku ii. 2006.

[24] Zong Woo Geem. Harmony search algorithm for solving sudoku. In *KES*, 2007.

[25] Cristina Gonzalez, Javier Olivito, and Javier Resano. An initial specific processor for sudoku solving. *2009 International Conference on Field-Programmable Technology*, pages 530–533, 2009.

[26] Peter Gordon and Frank Longo. *Mensa guide to solving sudoku: hundreds of puzzles plus techniques to help you crack them all*. Sterling Pub. Co., 2006.

[27] Jacob H. Gunther and Todd K. Moon. Entropy minimization for solving sudoku. *IEEE Transactions on Signal Processing*, 60:508–513, 2012.

[28] Agnes M. Herzberg and M. Ram Murty. Sudoku squares and chromatic polynomials. 2007.

[29] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. Computer algorithms. 1996.

[30] Rohit Iyer, Amrish Jhaveri, and Krutika Parab. A review of sudoku solving using patterns. 2013.

[31] Sunanda Jana, Arnab Kumar Maji, and Rajat Kumar Pal. A novel sudoku solving technique using column based permutation. *2015 International Symposium on Advanced Computing and Communication (ISACC)*, pages 71–77, 2015.

[32] Dhanya Job and Varghese Paul. Recursive backtracking for solving 9*9 sudoku puzzle. 2016.

[33] Snigdha Kamal, Simarpreet Singh Chawla, and Nidhi Goel. Detection of sudoku puzzle using image processing and solving by backtracking, simulated annealing and genetic algorithms: A comparative analysis. *2015 Third International Conference on Image Information Processing (ICIIP)*, pages 179–184, 2015.

[34] Snigdha Kamal, Simarpreet Singh Chawla, and Nidhi Goel. Identification of numbers and positions using matlab to solve sudoku on fpga. *2015 Annual IEEE India Conference (INDICON)*, pages 1–6, 2015.

[35] Dervis Karaboga. An idea based on honey bee swarm for numerical optimization. 2005.

[36] Byung-Gyu Kim and Dong-Jo Park. Adaptive image normalisation based on block processing for enhancement of fingerprint image. 2002.

[37] Donald E. Knuth. Dancing links. 2000.

[38] Wei-Meng Lee. *Programming Sudoku*. Apress, 2006.

[39] Rhyd Lewis. Metaheuristics can solve sudoku puzzles. *Journal of Heuristics*, 13:387–401, 2007.

[40] Hung-Hsuan Lin and I-Chen Wu. Solving the minimum sudoku poblem. *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pages 456–461, 2010.

[41] Tanh Minh Ly and Dung Trung Vo. A novel and automatic character extraction and recognition for sudoku puzzle solving. *2015 International Conference on Advanced Technologies for Communications (ATC)*, pages 546–550, 2015.

[42] Arnab Kumar Maji and Rajat Kumar Pal. Sudoku solver using minigrid based backtracking. *2014 IEEE International Advance Computing Conference (IACC)*, pages 36–44, 2014.

[43] Pavlos Malakonakis, Miltiadis Smerdis, Euripides Sotiriades, and Apostolos Dollas. An fpga-based sudoku solver based on simulated annealing methods. *2009 International Conference on Field-Programmable Technology*, pages 522–525, 2009.

[44] Timo Mantere. Improved ant colony genetic algorithm hybrid for sudoku solving. *2013 Third World Congress on Information and Communication Technologies (WICT 2013)*, pages 274–279, 2013.

[45] Timo Mantere and Janne Koljonen. Solving and rating sudoku puzzles with genetic algorithms. 2006.

[46] Timo Mantere and Janne Koljonen. Solving, rating and generating sudoku puzzles with ga. *2007 IEEE Congress on Evolutionary Computation*, pages 1382–1389, 2007.

[47] Timo Mantere and Janne Koljonen. Solving and analyzing sudokus with cultural algorithms. *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 4053–4060, 2008.

[48] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem. *Experimental Mathematics*, 23:190–217, 2012.

[49] Krystian Mikolajczyk and Cordelia Schmid. Scale and affine invariant interest point detectors international journal of computer vision. 2004.

[50] Todd K. Moon, Jacob H. Gunther, and J. J. Kupin. Sinkhorn solves sudoku. *IEEE Transactions on Information Theory*, 55:1741–1746, 2009.

[51] Alberto Moraglio, Julian Togelius, and Simon M. Lucas. Product geometric crossover for the sudoku puzzle. *2006 IEEE International Conference on Evolutionary Computation*, pages 470–476, 2006.

[52] Jaysonne A. Pacurib, Glaiza Mae M. Seno, and John Paul T. Yusiong. Solving sudoku puzzles using improved artificial bee colony algorithm. *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*, pages 885–888, 2009.

[53] Haiyan Quan and Xinling Shi. On the analysis of performance of the improved artificial-bee-colony algorithm. pages 654 – 658, 11 2008.

[54] Francesca Rossi, Peter van Beek, and Toby Walsh. Handbook of constraint programming (foundations of artificial intelligence). 2006.

[55] Ibrahim Sabuncu. Work-in-progress: Solving sudoku puzzles using hybrid ant colony optimization algorithm. *2015 1st International Conference on Industrial Networks and Intelligent Systems (INISCom)*, pages 181–184, 2015.

[56] Liu San-yang. Algorithm based on genetic algorithm for sudoku puzzles. 2010.

[57] Gustavo Santos-García and Miguel Palomino. Solving sudoku puzzles with rewriting rules. *Electr. Notes Theor. Comput. Sci.*, 176:79–93, 2007.

[58] Yuji Sato and Hazuki Inoue. Genetic operations to solve sudoku puzzles. In *GECCO*, 2010.

[59] Moriel Schottlender. The effect of guess choices on the efficiency of a backtracking algorithm in a sudoku solver. *IEEE Long Island Systems, Applications and Technology (LISAT) Conference 2014*, pages 1–6, 2014.

[60] Mehmet Sezgin and Bülent Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *J. Electronic Imaging*, 13:146–168, 2004.

[61] Prithvi Simha, K V.R.K. Suraj, and T. Ahobala. Recognition of numbers and position using image processing techniques for solving sudoku puzzles. *IEEE-International Conference On Advances In Engineering, Science And Management (ICAESM -2012)*, pages 1–5, 2012.

[62] Helmut Simonis. Sudoku as a constraint problem. 2005.

[63] Ricardo Soto, Broderick Crawford, Cristian Galleguillos, Eric Monfroy, and Fernando Paredes. A hybrid ac3-tabu search algorithm for solving sudoku puzzles. *Expert Syst. Appl.*, 40:5817–5821, 2013.

[64] Ricardo Soto, Broderick Crawford, Cristian Galleguillos, Eric Monfroy, and Fernando Paredes. A prefiltered cuckoo search algorithm with geometric operators for solving sudoku problems. In *TheScientificWorldJournal*, 2014.

[65] Ricardo Soto, Broderick Crawford, Cristian Galleguillos, Fernando Paredes, and Enrique Norero. A hybrid alldifferent-tabu search algorithm for solving sudoku puzzles. In *Comp. Int. and Neurosc.*, 2015.

[66] Ricardo Soto, Broderick Crawford, Cristian Galleguillos, Francisca C. Venegasy, and Fernando Paredes. A marriage theorem based-algorithm for solving sudoku. *2015 Fourteenth Mexican International Conference on Artificial Intelligence (MICAI)*, pages 117–121, 2015.

[67] George C. Stockman and Linda G. Shapiro. Computer vision. 2001.

[68] Kees van der Bok, Mottaqiallah Taouil, Panagiotis Afratis, and Ioannis Sourdis. The tu delft sudoku solver on fpga. *2009 International Conference on Field-Programmable Technology*, pages 526–529, 2009.

[69] Luc Vincent. Morphological grayscale reconstruction in image analysis: applications and efficient algorithms. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 2 2:176–201, 1993.

[70] Zhiwen Wang, Toshiyuki Yasuda, and Kazuhiro Ohkura. An evolutionary approach to sudoku puzzles with filtered mutations. *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 1732–1737, 2015.

[71] Baptiste Wicht and Jean Hennebert. Camera-based sudoku recognition with deep belief network. *2014 6th International Conference of Soft Computing and Pattern Recognition (SoCPaR)*, pages 83–88, 2014.

[72] Baptiste Wicht and Jean Hennebert. Mixed handwritten and printed digit recognition in sudoku with convolutional deep belief network. *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 861–865, 2015.

[73] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. 2003.

[74] John Paul T. Yusiong and Jaysonne A. Pacurib. Sudokubee : An artificial bee colony-based approach in solving sudoku puzzles. 2010.