Sudoku Solver using Dancing links in the form of Algorithm X and Augmented Reality

Timo Andrei-Claudiu

19000915

UXCFXK-30-3

Digital Systems Project

UWE Bristol | University of the West of England

1. Abstract

Sudoku is a well-known logic-based number placement problem that has gained popularity in recent years. The puzzle's purpose is to fill a 9×9 grid with numbers so each column, row, and each 3×3 sub-grids have all digits from 1 to 9.

A Sudoku Solver in form of Algorithm X employing Dancing links is a well-known approach for solving Sudoku puzzles. Algorithm X is a generalization of Knuth's Algorithm X that is used to solve exact cover issues. Dancing links is a strategy for effectively searching for solutions to exact cover problems using algorithm X.

Augmented Reality (AR) is a technology that overlays digital content in the real world, providing the user with a more immersive experience. A Sudoku Solver can be created by combining the strength of Algorithm X and Dancing links with the immersion of AR, allowing users to engage with the puzzle in a more natural and intuitive manner.

The Sudoku Solver would allow users to scan a physical Sudoku puzzle and then interact with it in real-time, delivering instant feedback and direction as they strive to solve the challenge using Dancing links in the form of Algorithm X and Augmented Reality. This technology is suitable for both novices and experts, making it an excellent tool for anybody wishing to enhance their Sudoku abilities.

2. Acknowledgements

I would like to extend my sincere thanks to my programme leader, Dr. Elias Pimenidis, for his invaluable patience and support to make this thesis possible. His guidance and advice carried me through all stages of writing my project.

I would also like to give my special thanks to my deceased uncle who managed to plant a seed of knowledge of the secrets of computers since I was a small child, and my family as a whole for their continuous support and understanding when undertaking my research and writing my project. Your prayer for me was what sustained me this far.

Last but not least, I would like to thank God, for letting me through all difficulties. I have experienced your guidance day by day. You are the one who gives me strength to move forward and let me finish my degree in Computer Science. I will keep trusting you for my future.

3. Table of Contents

4. Table of Figures

5. Introduction

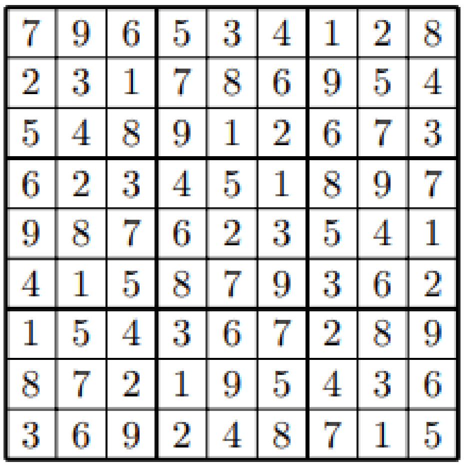
Sudoku is a logic puzzle with a 9×9 grid that is further subdivided into 3×3 “mini-grids”. Each row, column, and mini-grid contains the numbers from 1 to 9, having a unique solution. *Sudoku*, a puzzle that appears to be based primarily on Latin Squares and Magic numbers, is a relatively new phenomenon. This puzzle was originally known as *Number Place*, but after gaining popularity in Japan, the Nikoli puzzle group trademarked it as “*Sudoku*” (“Su” meaning number, and “Doku” meaning single). Since around 2005, the puzzle had become internationally popular, sparking a trend of similar puzzles of varying sizes and border constraints – being linked to real-world applications due to its strong relationships with Latin Squares and other combinatorial structures including AI (solving algorithms), Mathematics (colouring problems), Steganography (data hiding techniques and they are used like a key to hide data behind images), conflict-free wavelength routing in wide band optical networks, statistical design and error correcting codes, timetabling, and experimental design.

A grid of 81 cells divided into 9 rows and 9 columns is the most common Sudoku puzzle. A single integer between 1 and 9 can be stored in a single cell. The grid is also divided into 9 boxes, each of which has three rows and three columns. Each new puzzle begins with an arbitrary number of given integers being placed in the grid. Every such integer is known as a clue. The goal of the puzzle is to fill in the remaining grid cells with integers so that each integer appears once in each row, column, and box.

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

Figure 1: *Sudoku Puzzle*

For a grid to be considered valid, there must be only one solution for a given grid with clues. Given the clues, this means that there can only be one way for the integers in the grid to be placed to fill it. The smallest known number of clues needed for a unique solution is 17. **Figure 2** (R. Dechter, 2013) shows an example of a one-of-a-kind solution to the challenge in **Figure 1**, which has 22 clues (T. Bäck, D. B. Fogel, 2000).



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

Figure 2: *Solution for previous Sudoku*

The simplest way for a computer to solve a Sudoku puzzle is to test all possible values for each remaining cell until a solution is found, or to end when all values have been tested. A *brute force algorithm* or *search* is used to accomplish this (T. Bäck, D. B. Fogel, 2000). The term *brute force* is commonly used in many computer-related subjects, but it all boils down to the same basic idea: test all possible values.

The performance represents a problem for a brute force algorithm. The time complexity of brute force is exponential, which is only feasible for small problem instances.

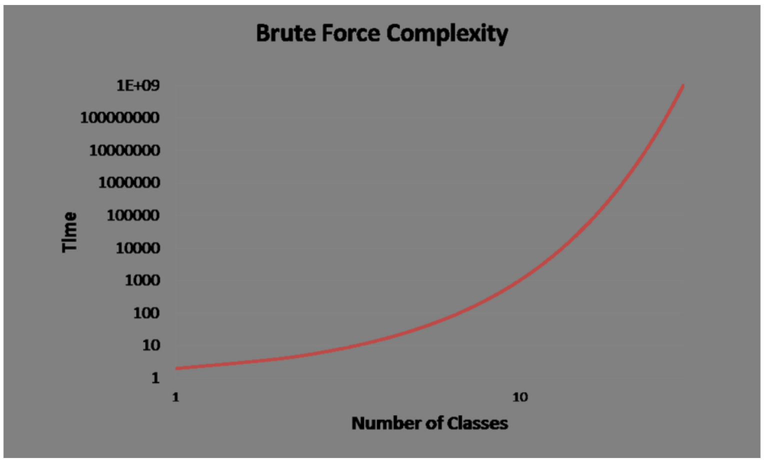


Figure 3: *Time complexity of brute force technique*

6. Literature Review

6.1. Overview

The application's goal is to build a software that scans random Sudoku problems of varied complexity, allowing us to attempt to finish puzzles while simultaneously providing assistance through optional help features. Players may also scan their own Sudoku puzzles using Augmented Reality, which the program will validate before providing the user with aforementioned support options to help them solve the puzzle. Overall, this product will have a simple Sudoku scanner, a Sudoku solver using Augmented Reality and a good user interface that will provide tools to help the player solve the puzzle and comprehend established techniques.

6.2. Research Techniques Used to Solve Sudoku Puzzles

To be able to solve a Sudoku puzzles and assist users of Sudoku Generator and Helper application, established Sudoku puzzle-solving techniques must be explored. The research into proven techniques used to solve Sudoku puzzles will be divided into three parts: focus on understanding a Sudoku puzzle, techniques used by human users to solve the puzzles, and the third on existing and proposed algorithms that computers can use to solve Sudoku puzzles.

6.2.1 Understanding Sudoku

The puzzle from **Figure 1** shows how a grid may be divided into three sections: columns, rows, and sub-grids. When dividing a grid into columns, each column is numbered from one to nine, starting from left. When dividing a grid into rows, the rows are numbered from one to nine, beginning at the top. When a grid is split into sub-grids, each sub-grid is numbered from one to nine, beginning at the top left and working clockwise. It's worth noting that each set (particular row, column, sub-grid) will have nine squares.

As previously stated, a Sudoku grid is made up of 81 squares, and each square will be referred to as a cell throughout this project. When an individual cell must be identified, it is designated as C_{xy}, where x is the row number and y is the column number. In **Figure 1**, for example, cell C_{34} has number 9.

Sudoku is essentially a basic game with only one rule that must be followed in order to successfully solve the puzzle. This rule states that the numbers one through nine must be arranged within the cells of the Sudoku grid so that each row, column, and sub-grid only has one of these numbers. This rule implies that if a number is missing from a set, it must be found in one of the vacant cells inside the set. When a number appears in a set, none of its empty cells can contain that number.

The cells of a Sudoku board, such as the one shown in **Figure 1**, either have a number or they are unfilled. For the purposes of this project, provided numbers are any pre-assigned numbers on a Sudoku grid. Following the Sudoku rule, the vacant cells inside the grid might theoretically contain between one and nine numbers with these supplied numbers in place. These prospective numbers will be referred to as candidates during the course of this project.

For example, if a cell contains the number ***5***, the following candidates will be found in the other cells in the set: {1, 2, 3, 4, 6, 7, 8, 9}.

The single rule in Sudoku helps in the demonstration of a link between cells on the Sudoku grid. Each cell belongs to one of three possible sets: one column, one row, and one sub-grid. This indicates that each cell in the Sudoku grid is related to at least twenty other cells.

6.2.2 Human Solving Techniques

Every Sudoku puzzle is solved by eliminating possibilities. There are several established strategies for completing a Sudoku problem, but according to Thomas (2006, p.12), each strategy is based on one of following two approaches:

1. Taking a number and selecting which cells it should be placed in.
2. Choosing a cell and selecting what numbers should be placed within.

Understanding these ideas facilitates comprehension of the following problem-solving approaches. Werf (2007) and Johnson (2007) demonstrate the application of these strategies to solve all levels of Sudoku problems (2005).

6.2.3 Maintaining Candidates

Johnson (2005) suggests that Sudoku players keep a list of options for each blank cell, as shown in **Figure 4**. The goal is to assist players in seeing patterns inside a problem, which is also assisted by Werf (2007).

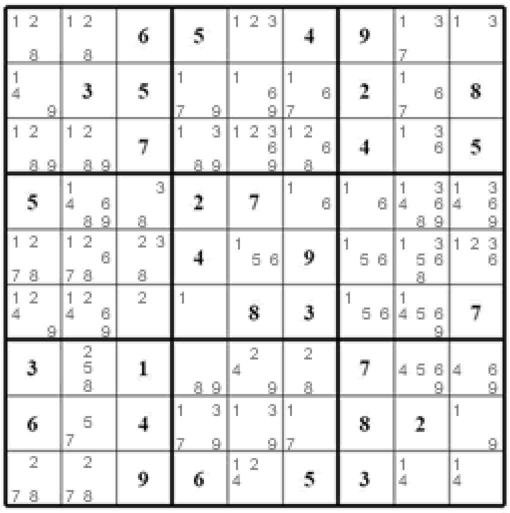


Figure 4: *Cells and their candidates*

When a player is employing simpler solution approaches, Thomas (2006) disagrees with this idea since he feels the additional information might generate confusion and inaccuracy. However, Johnson (2005) argues that the margin for error is gone and the player can use reasoning while completing the problem if this is done by a computer program rather than by hand.

6.2.3.1 Naked Single

Any cell with a single candidate, such as cell C_{64} (highlighted in **Figure 5)** from **Figure 4**, can be assigned that value because the cell cannot carry any more integers.

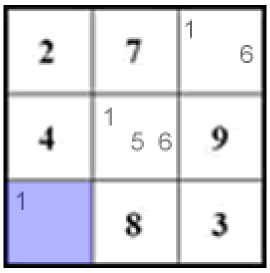


Figure 5: *A single candidate cell*

The allocated number must be deleted from the cell's buddy cells once the value has been assigned to it. The number 1 must be eliminated from any cell that is a buddy of C_{64} (depicted in **Figure 6)** in this circumstance.

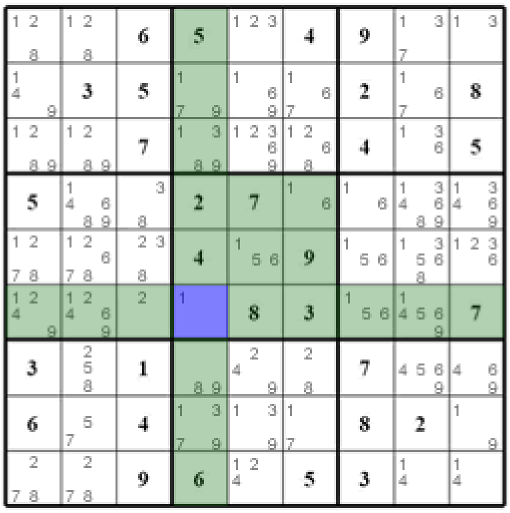


Figure 6: *A single candidate cell and its buddies*

6.2.3.2 Hidden Singles

A hidden single, like a single, happens when there is only one potential number in a cell, but it is hidden amongst other options. It is possible to notice in the example Sudoku puzzle that cell C_{21} has a hidden single.

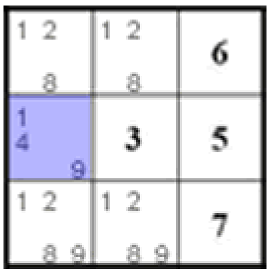


Figure 7: *A hidden single for Cell C_{21}*

Figure 7 demonstrates that cell *C_{21}* has three candidate numbers: 1, 4, and 9. However, by inspecting the cells in the sub-grid, you can observe that no other cells contain the candidate number 4. This indicates that cell *C_{21}* must be allocated the value 4 according to Sudoku rules.

6.2.3.3 Locked Candidates (type 1)

Locked candidates (type 1) restrict a candidate number to a sub-grid by locking it to a single row or column. In column eight of the sample Sudoku problem, **Figure 8** shows an example of type 1 locked candidates.

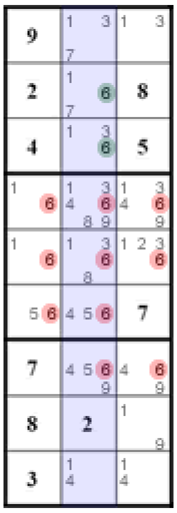


Figure 8: *Locked Candidate (type 1)*

Figure 8 shows that there are only two cells in the top sub-grid that may have the number 6 as their final value: cells *C_{28}* and *C_{38}*. When looking at the center sub-grid, there are eight cells that might potentially store the number 6 as their value, and there are two cells in the bottom sub-grid. According to Sudoku rules, the number 6 must be held in either cell *C_{28}* or *C_{38}*, else the top sub-grid will be lacking that value. As a result, candidate 6 may be removed from cells *C_{48}*, *C_{58}*, *C_{68}*, and *C_{78}*.

6.2.3.3 Locked Candidates (type 2)

Locked Candidates (type 2) is an alternative to Locked Candidates (type 1), which concentrates on locking candidates in a row or column within a sub-grid as shown in **Figure 9**.

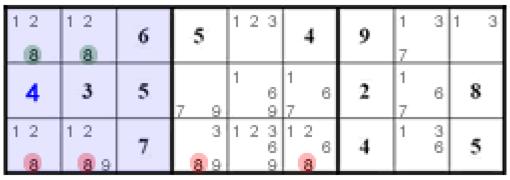


Figure 9: *Locked Candidate (type 2)*

Figure 9 shows that there are only two cells in row 1 that might conceivably have the number 8 as their final value; cells *C_{11}* and *C_{12}*. Looking at row 3, we can see that there are four cells that might potentially have the value 8 as their value. According to Sudoku rules, number 8 must be held in either cell C_{11} or C_{12}, else row 1 will be missing that value. As a result, candidate 8 may be removed from cells C_{31} and C_{32}.

6.2.3.4 Naked Pairs

When two cells in a set have the same pair of numbers, no additional cells in that set can have those numbers.

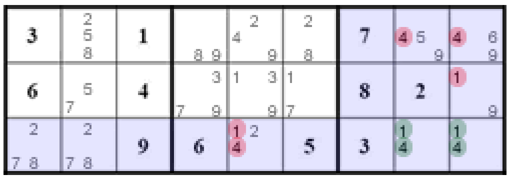


Figure 10: *Naked Pairs*

Figure 10 shows that cells C_{98} and C_{99} share the pair of candidates 1 and 4. According to Sudoku rules, any additional cells within the sets row nine or sub-grid nine cannot have the value 1 or 4. As a result, one or both of the candidate numbers 1 and 4 can be removed from cells C_{78}, C_{79}, C_{89}, and C_{95}. Cells C_{89} and C_{95}, as a result, reveal naked singles and may be filled with the digits 9 and 2, respectively.

6.2.3.5 Hidden Pairs

The notion of hidden pairs is similar to that of naked pairs, except that the emphasis is on the cells that store the pairs rather than on the other cells. When two cells in a set share a pair of numbers concealed among their candidates that are not found in any other cell in the set, the other candidates belonging to the two cells can be deleted.

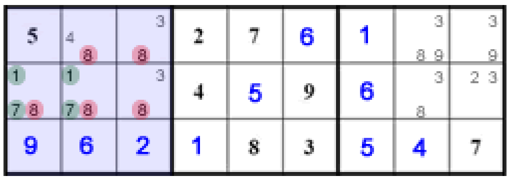


Figure 11: *Hidden Pairs*

Figure 11 shows that candidates 1 and 7 only occur in cells C_{51} and C_{52}. These pairings are concealed by the candidate 8, but because 8 appears in other cells inside sub-grid 4, we may safely delete the candidate 8 from cells C_{51} and C_{52} using the Sudoku rule.

6.3. Sudoku Solving Algorithms

Sudoku solving strategies for computers do not have to be as varied and sophisticated as those used by humans. A computer is capable of performing thousands of tasks in a short amount of time, which means that a brute force assault might answer a simple Sudoku problem. However, as problems get more difficult, optimizing algorithms to apply intelligence to answer the most difficult Sudoku puzzles in polynomial time becomes critical.

6.3.1 Proposed Algorithm – Algorithm X

Donald Knuth developed the algorithm in 2000, and it can be used to find all solutions to a specific cover problem (D. Knuth, 2000). The algorithm is based on a very simple technique that Knuth believed should be more widely known. Given a node *x* pointing to two elements in a doubly linked list, *L[x]* points to *x*'s left element and *R[x]* points to *x*'s right element. For example, *L[R[x]]* is pointing to *x* if the right element of *x* is pointing back to *x*. This could be much more clearer if we let *y = R[x]* and *L[R[x]] = L[y]*, which describes the left element of *y*:

$$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x]$$

Figure 12: *Removing x from the list*

$$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x$$

Figure 13: *Inserting x back into the list*

The first operation should be identified very easily, but Knuth believes the second one is less well known. Because *x* still has the references it had before, being removed from the doubly linked list, the insertion works. The first operation should be identified very easily, but Knuth believes the second one is less well known. Because *x* still has the references it had before being removed from the doubly linked list, the insertion works. It may appear pointless to insert an element that has just been removed, but Knuth claims that there are several applications where this is useful, the most notable being *backtracking technique*.

Algorithm X is a good approach to solve sudoku because it needs to run in two ways:

- ***To generate a solved grid at random***. In this mode, the solver is stopped as soon as it finds a solution (D. Knuth, 2000). At step 2 of the algorithm, the solver chooses rows in a random order, so that a good subset of the grid space is reachable. Before running the solver, the top row of the grid is filled with a random permutation of the nine symbols. This results in a 5% speed-up over starting on an empty grid.
- ***To check if a candidate puzzle has multiple solutions***. In this mode, the solver is stopped when it finds a second solution (if there is one) or when it has examined the entire search tree and determined that there is only one solution (M. Ercsey-Ravasz and Z. Toroczkai, 2012). It is not important in what order it chooses rows in at step 2, so skip the effort of generating random numbers in this mode can be skipped.

In a study of 24 "easy" and "intermediate" puzzles and 10.000 hard puzzles, Knuth (2000, p.4–5) found that the algorithm solves these problems in polynomial time.

6.3.2 Analysis and Interpretation of Proposed Algorithm

Donald Knuth proposed the Dancing Links in 2000, or DLX, technique for efficiently implementing Algorithm X. DLX will represent the 1s in a binary matrix as data objects. The fields in each data object will link up to any other cell with an occupied 1 to the left, right, up, and down. Any link that does not have a corresponding *1* in a suitable cell will instead link to itself.

The last field represents a link to the column object y, which is a special data object with two additional fields, *S[y]* and *N[y]*, which represent the column size and an arbitrary symbolic name. The number of data objects currently linked together from the column object is the column size. Each row and column is a circular doubly linked list, and removing a data object from a column decrements the size. Remember that the removed data object's links still point to the same places they did before the data object was removed.

6.3.3 Dancing Links

For Sudoku, The "*S heuristic*" field (selecting the column with the fewest rows) is extremely efficient. In a test of 100 randomly generated puzzles, the *S* heuristic algorithm could use 345.810 search nodes to demonstrate that each puzzle had a single solution (D. Knuth, 2000). Without the heuristic (always looking at the leftmost remaining column), the same solver could require 49.879.323 nodes to complete the same task.

$$\begin{array}{ccccccc}
A & B & C & D & E & F & G \\
\end{array}$$
$$\begin{pmatrix}
0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1
\end{pmatrix}$$

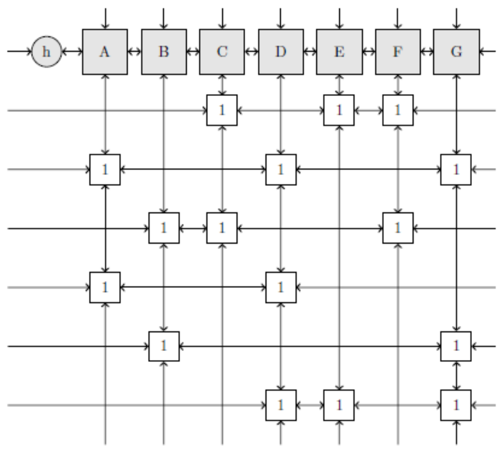


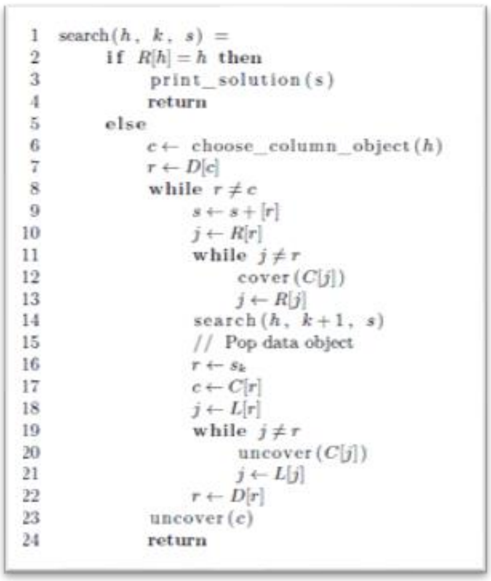
Figure 14: *A matrix representation in DLX*

This test indicates that the *S heuristic* provides a speedup of about 144 times. However, the distribution is quite broad. For simple puzzles, the speedup is usually no more than ten times. When tackling the sudoku puzzle in **Figure 1**, the solver using the *S heuristic* needs only 64 nodes to prove that there is a single solution, while the solver without the heuristic needs 21.677.544 nodes, for a speedup of 338.711 times.

Figure 14 depicts how the DLX data structure represents a given binary matrix. It demonstrates how each 1 in the matrix is represented by a data object that is linked to any nearby data object. The arrow head at the data or column objects denotes a link (D. Knuth, 2000); also note how the links are circular for both the row and columns.

Each column has a symbolic name ranging from A to G, with *h* representing the root column object. The root column object is a special case column object that uses the same data structure but cannot directly link to any data objects; DLX only uses it as an entry point to the data structure.

Below is defined a function *search*(h, k, s), where *h* represents the root column object, *k* represents the current depth, and *s* represents the solution with a list of data objects. The method should be invoked using *k* = 0 and s = []. If *s* is a linked list, *k* can be left out.



```
1   search($h$, $k$, $s$) =
2         if $R[h] = h$ then
3               print_solution(s)
4               return
5         else
6               $c \leftarrow$ choose_column_object($h$)
7               $r \leftarrow D[c]$
8               while $r \neq c$
9                     $s \leftarrow s + [r]$
10                    $j \leftarrow R[r]$
11                    while $j \neq r$
12                          cover($C[j]$)
13                          $j \leftarrow R[j]$
14                    search($h$, $k+1$, $s$)
15                    // Pop data object
16                    $r \leftarrow s_k$
17                    $c \leftarrow C[r]$
18                    $j \leftarrow L[r]$
19                    while $j \neq r$
20                          uncover($C[j]$)
21                          $j \leftarrow L[j]$
22                    $r \leftarrow D[r]$
23              uncover($c$)
24              return
```

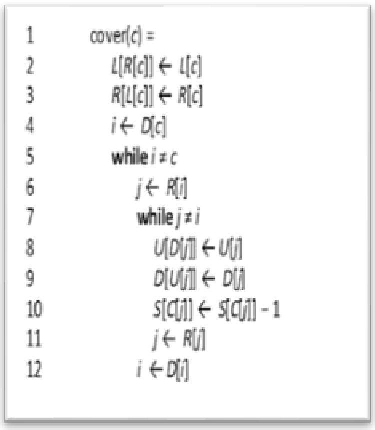
Figure 15: *DLX Algorithm*

The printing of the solution will be handled in the upcoming section for reducing Sudoku, which is specific to a solution. Add a function pointer argument to search to have more options on how the solution is printed. On row 6 of **Figure 15**, the column object is chosen, which can be done in two ways: by selecting the first column object after the root column object; by selecting the column object with the fewest number of 1s in a column – by choosing the latter, the solver minimises the branching factor(D. Knuth, 2000).

The other two functions, *cover* and *uncover*, will either cover or uncover a specified column object *c*.

6.3.4 Cover function

Figure 16 depicts the cover operation, which removes *c* from the doubly linked list of column objects and all data objects under *c*. The operation depicted in **Figure 12** will be used within a function along with Algorithm X.

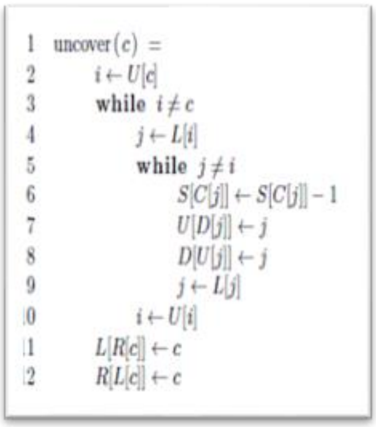


```
cover(c) =
    $L[R[c]] \leftarrow L[c]$
    $R[L[c]] \leftarrow R[c]$
    $i \leftarrow D[c]$
    while $i \neq c$
        $j \leftarrow R[i]$
        while $j \neq i$
            $U[D[j]] \leftarrow U[j]$
            $D[U[j]] \leftarrow D[j]$
            $S[C[j]] \leftarrow S[C[j]] - 1$
            $j \leftarrow R[j]$
        $i \leftarrow D[i]$
```

Figure 16: *Cover Function*

6.3.5 Uncover function

The other function revealed in **Figure 17** is more intriguing because it makes use of the list operation depicted in **Figure 13**, which Knuth wanted to be more widely known.



```
uncover(c) =
    $i \leftarrow U[c]$
    while $i \neq c$
        $j \leftarrow L[i]$
        while $j \neq i$
            $S[C[j]] \leftarrow S[C[j]] - 1$
            $U[D[j]] \leftarrow j$
            $D[U[j]] \leftarrow j$
            $j \leftarrow L[j]$
        $i \leftarrow U[i]$
    $L[R[c]] \leftarrow c$
    $R[L[c]] \leftarrow c$
```

Figure 17: *Uncover Function*

The insertion operation in **Figure 16** by performing the reverse order of the covering operation in cover because the operation of inserting *x* in the list “undo” the operation of deletion. The rows were removed from top to bottom and must now be added from bottom to top. The same logic applies to columns that were removed from left to right and must now be added from right to left to reverse the operation.

6.4. Augmented Reality

One of the key aims of augmented reality, in the midst of the development of data collecting and analysis, is to emphasize certain elements of the physical environment, raise knowledge of those qualities, and extract clever and accessible information that can be used to real-world applications. Big data may help organizations make better decisions and acquire insight into customer purchasing habits, among other things. The second part of the application implies that the user has an option to use Augmented Reality and solve the sudoku by showing the puzzle to the webcam.

6.4.1 Processing the image from the camera

For processing the image, we use the python ***OpenCV*** library. It is a simple step by step approach in the following order:

- Detect the sudoku’s outer grid and return the area included within it.
- Apply warp perspective transformation by locating the contour’s corners to change the sudoku from a part of the picture to the entire image itself.
- Get the sudoku’s dimensions, or the number of rows and columns in a sub-box of the problem.
- Applying a convolution neural network model to each cell to extract the digit value yields the final matrix.

6.4.1.1 Detect the outer sudoku grid

To locate the image’s outer edges, we must first locate the image’s contours. The outline or silhouette of an object, which in our instance is the outline of the Sudoku Puzzle, is referred to as a contour. After locating the greatest contour, we do a sanity check to guarantee that its area is bigger than 250x250 pixels. Following that, we create two masks, one black with white as the inner region of the selected contour and one white with white as the outside region of the selected contour. Then we replicate the original grey image’s white value places onto the white mask with the outside white region to produce the pictures below. This is our ROI, or Region of Interest.

6.4.1.2 Apply warp perspective transformation

We perform a sanity check to ensure that the previous step was successful, and then find the perimeter of the puzzle using cv2.arcLength. Since we know the shape of the box, we can approximate the perimeter of the contour using 1.5% precision.

We approximate the positions of the corner coordinates of the output warped image by finding the coordinates of the corner coordinates of the original image, followed by the four points we specified for our output image. We then use cv2.warpPerspective to perform the perspective transformation.

6.4.1.3 Get dimensions of the puzzle

The dimensions of an image after binarization and post-binarization are estimated by summing the image’s matrix across the vertical and horizontal axes as depicted in **Figure 18**.

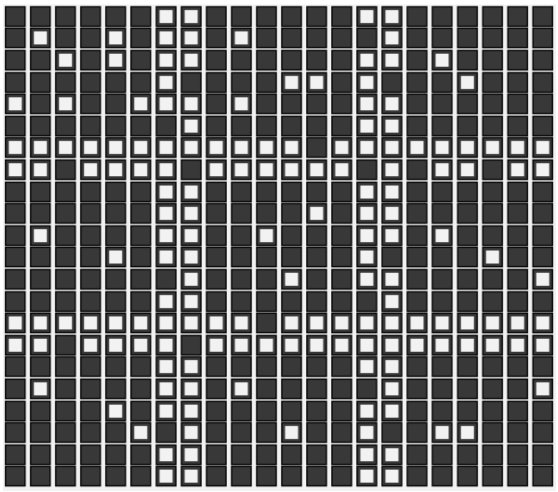


Figure 18: *A scanned sudoku grid*

6.4.1.4 Pre-process digit cells and get the final matrix

First, the image is resized to 112×112 pixels. Then, it is blurred and thresholded to convert the values to either 0 or 1. Finally, the border is cleared to remove noise from the edges. The image is then resized back to 28×28 pixels to make it suitable for the digit recognition model.

If the number of white pixels in the cell image is less than 10, we assume the cell is empty and return None. Then, we scale the pixels with values less than 150 to 75% of their value and the rest to double their value.

To get the final matrix, first we get the warped image and the dimensions of the sudoku puzzle. Next, we initialise an empty matrix and load our trained model file. We then determine the width and height of each cell and iterate over each cell. Once we have done the pre-processing of the cell, we input it into our model to find the digit value of the cell.

We check to see if any other cells in the same row, column, or box have the same value. If they do, we get the prediction score for the cell with the higher score. The cell with the higher score gets the value, and the other gets its second-highest score.

7. Requirements

7.1. System specifications

The system requirements have been separated into seven areas in order to make the review process easier and to ensure that the product offers all desired characteristics.

7.1.1 Solving Algorithm

The solving algorithm must be capable of solving any Sudoku puzzles in polynomial time, regardless of complexity.

7.1.2 Solution Generation Algorithm

The Sudoku problem generator must be able to accept a valid Sudoku solution generated by the solution generation method.

7.1.3 Puzzle Generation Algorithm

The problem generation method must be capable of producing a Sudoku puzzle with a unique solution and of the desired complexity.

7.1.4 Unique Solution Algorithm

The unique solution algorithm checks for multiple solutions to a particular Sudoku puzzle and checks to see if any of them is the correct one. When the procedure is finished, it provides a Boolean value indicating if the problem has a unique solution.

7.1.5 Difficulty Calculation Algorithm

The difficulty calculation algorithm determines the complexity of a Sudoku problem.

7.1.6 Help Tools

The helper tools will provide a selection of help features available to Sudoku Generator and Augmented Reality options via the GUI. These features will mirror the problem-solving strategies described in sections **section 6.3** and **6.4**, respectively.

7.1.7 User Interface

The user interface must be a well-designed human-computer interface with user-friendly and simple-to-use features. The user interface will be used to apply the help tools available to the user – AR option and solver when a puzzle is uploaded from a file.

7.2. Non-functional Requirements

The following non-functional requirements have been created based on system specifications, use case definition, and expert system analysis:

- The system should provide a simple method for designing a Sudoku solver.
- A Graphical User Interface to integrate the prototype.
- Acceptable performance speed for loading, solving and validating puzzles.
- An option for solving and displaying the grid-solution within the GUI.
- An option for loading a grid using Augmented Reality.
- An option to generate a new Sudoku grid.
- Solve and validate Sudoku puzzles that a player has entered.
- Allow the player to input numbers into a Sudoku puzzle.
- An option to automatically resolve the current grid.
- Save and display the latest used grid.
- An option to enter the numbers on grid ranging from 1 to 9.
- Allow the player to use the help tools:
    - The Augmented Reality option to display the solution on the webcam.
    - To automatically solve the grid by pressing "Space".
- Allow the player to exit from the Augmented Reality by pressing "Q".
- The system should display an error message when the grid does not respects the constraints.
- The system should display a message when, in Augmented Reality, the grid is not a clear image/too far from the camera.
- The ability for the user to graphically see the solution when uploading a Sudoku puzzle image.

These non-functional requirements will draw out more information about the system's functionality. The main components for building the system will be outlined in the functional requirements section – the next step to design and implementation.

7.3. Use Cases Diagrams

The requirements for the Sudoku Generator and Helper system from the perspective of a user were analysed using the following use case diagram and use case description.

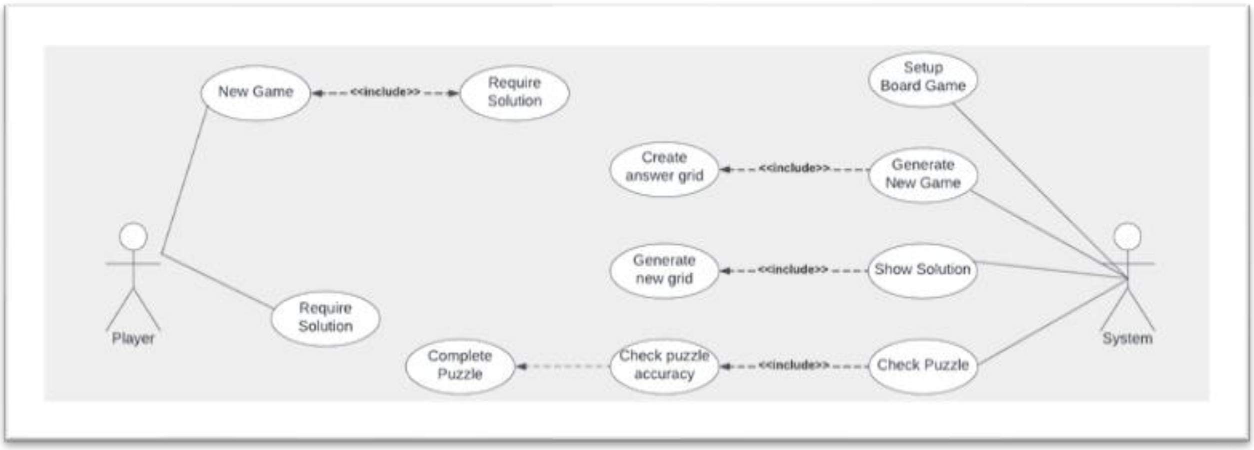


Figure 19: *Use Case diagram for Sudoku Game*

A player can start a new game by choosing to play the game of the last saved puzzle or by uploading their own puzzle. The player also has the choice to access a help option by pressing ***Space*** and the puzzle is self-solving. The system should be able to setup the Game Board and let the user to generate a new game by creating the answer grid.

7.4. Functional Requirements

The functional requirements for this project are displayed in table form, with descriptions of each requirement, and a prioritization based on the ***MoSCoW*** scheme. The requirements specified as fundamental features are extracted from the aims and objectives of the project.

7.4.1. Table 1: Functional requirements for the Sudoku Solver GUI

The requirements' names are an abbreviation of Sudoku Solver Application (SSA).

Requirement	Description	Priority	Priority Justification
All1.	The ability of the system to generate a GUI	M	Fundamental feature – the project requires data exchange between the user and the system.
SSA1.	Provide a medium-sized, clear Sudoku puzzle.	M	Fundamental feature
SSA2.	A system that reads image files from a Sudoku grid.	M	Fundamental component
SSA3.	Provide clear and	M	Fundamental feature

	accurate help.		
SSA4.	Help tools are accessible and easy to find.	S	Accessible way to use Augmented Reality and Automatically-Solve sudoku puzzle.
SSA5.	The system should generate a solution for the uploaded/current sudoku grid.	M	Fundamental feature
SSA6.	The system allows the user to upload a new grid.	M	Fundamental feature
SSA7.	The user is able to solve the puzzle.	M	Fundamental feature
SSA8.	The system allows the user to scan a grid using Augmented Reality Option.	M	Fundamental feature
SSA9.	The ability for the user to graphically see the solution in Augmented Reality.	M	There is a visual representation of the Sudoku puzzle.

7.5. Acceptance Tests

The acceptance tests will also be provided in a table form with their ID, name of the tested requirement, description of the test, and the expected outcome.

7.5.1. Table 2: Acceptance Test for the Sudoku Solver GUI

Test ID	**Tested Requirement**	**Test Description**	**Expected Outcome**
AT1.	SSA1 and All1.	Test if the system can successfully generate a GUI with 2 options – “Play Game” and “Augmented Reality”	The system displays displays the graphical model on the modelling GUI
AT2.	SSA2.	Test if the system can read and upload a Sudoku grid from an image file.	User to be able to add a new grid to the GUI.
AT3.	SSA3.	The system displays clear and accurate help options – Solve	User is able to use “Solve” and “Augmented

		and Augmented Reality.	Reality" option to see the solution for a given grid.
AT4.	SSA3 and SSA4.	The help tools are accessible and easy to find.	User is able to see and use the help tools.
AT5.	SSA5 and SSA6.	The performance test should be acceptable for loading, solving and validating Sudoku grids.	Acceptable speed for all 3 processes – polynomial time.
AT6.	SSA7 and SSA8.	Test if the system can solve and generate a solution for the uploaded file.	User should be able to visualise the solution of the uploaded grid.
AT7.	SSA6 and SSA7.	Test if the user can write numbers from range [1,9] to the grid.	User to be able to add numbers to the grid to solve the Sudoku.
AT8.	SSA5 and SSA8.	Test if the user can upload a grid using the Augmented Reality option.	User to be able to visualise the grid scanned by the camera.
AT9.	SSA6 and SSA9.	Test if the user can exit from the Augment Reality Option using 'Q' key.	User should be able to exit from the Augmented Reality option and is guided to the main menu.
AT10.	SSA7 and SSA8.	The user should be warned when when an image file is loaded - blurry image or dimension too small/large	User is able to see the message in order to correct the mistake - upload a clearer image or another grid.
AT11.	SSA7 and SSA8.	Check if an error message is displayed when the uploaded image in Augmented Reality is blurred or too far from the camera.	User is able to see the message in order to correct the mistake.
AT12.	SSA8.	Check if the solution is generated correctly in the grid when uploading a Sudoku puzzle image.	The user should be able to see the solution displayed after an image was uploaded.
AT13.	SSA9.	Check if the solution, in	The user should be able to see the image

		Augmented Reality mode, is displayed correctly on the screen.	containing the solution with the missing numbers in a different font/color.

7.6. Sudoku constraints

The technique for reducing a grid to an exact cover issue is one of the most essential elements of this thesis. The Dancing Links method cannot be used to solve a grid until this reduction is understood. In the introduction, the rules of Sudoku were presented, and each rule may be characterised as a constraint:

- Cell – each cell can only contain one number from [1, 9].
- Row – each row can only contain nine distinct numbers from [1, 9].
- Column – each column can only include nine distinct numbers from [1, 9].
- Box – each box may only have nine distinct numbers from [1, 9].

Considering a grid *G*, which has at least 17 clues and a unique solution. Because a clue cannot be relocated/modified, it will have an influence on all four restrictions. The relationship between the clues will then determine where the remaining integers may be placed in the row, column, and box. In other words, the clue defines who will fill the remaining cells on the same row, column, and box.

The restrictions in a binary matrix *M* (**Figure 14**) must be preserved in the reduction from the grid *G*. There must therefore be a set of rows in *M* such that a union between them covers all columns, else there is no solution. At a first look, this reduction does not appear realistic because a binary matrix can only hold 1s and 0s and a grid can only contain numbers between 1 and 9. It also does not appear feasible to maintain the limits for all cells. This reduction is achievable because each row in *M* describes all four constraints for each *G* cell. When a row from *M* is chosen as part of the solution, it is actually an integer that satisfies all four constraints for the cell in *G* that the row specifies. Each cell in *G* generates rows in the following sequence, while the constraints might appear in any order as long as they are consistent across all rows:

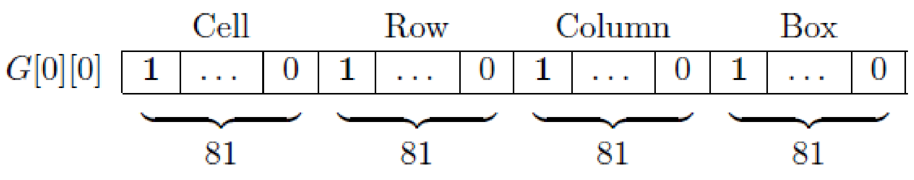


Figure 20: *Generated rows for Sudoku with constraints*

7.6.1 Cell constraint

Because each cell in *G* includes nine candidates, there must be nine rows in *M* for each cell. Each cell's solution must include one of these nine rows. Since the solution must include all columns, each of the nine rows contains a "1" in its own column (Hanson, Robert M, 2022). For the first nine rows, the first cell has 1s in the first column, the second cell has 1s in the second column, and so on for the remaining cells. This forces the algorithm to include at least one of the rows for each cell in order to cover all columns. Because there are 81 cells in total, the cell constraint demands 81 columns, and each cell requires nine rows, *M* must have space for its 9·81= 729 rows in total.

$$\begin{array}{r|c|c|c|c|}
 & 1 & 2 & & 81 \\
G[0][0]\ 1 & 1 & 0 & \dots & 0 \\
\vdots & & & & \\
G[0][1]\ 1 & 0 & 1 & \dots & 0 \\
\vdots & & & & \\
G[8][8]\ 9 & 0 & 0 & \dots & 1
\end{array}$$

Figure 21: *Cell constraints included in matrix M and each cell from G*

7.6.2 Row constraint

In *G*, each row may only contain numbers between 1 and 9. To keep this limitation in *M*, the 1s are arranged differently than in the cell constraint. To meet a row limitation of 9 cells, we must arrange the 1s for one row in *G* over 9 rows in *M*.

$$\begin{array}{r|c|c|c|c|c|}
 & 1 & 2 & & 80 & 81 \\
G[0][0]\ 1 & 1 & 0 & \dots & 0 & 0 \\
G[0][0]\ 2 & 0 & 1 & \dots & 0 & 0 \\
\vdots & & & & & \\
G[0][1]\ 1 & 1 & 0 & \dots & 0 & 0 \\
G[0][1]\ 2 & 0 & 1 & \dots & 0 & 0 \\
\vdots & & & & & \\
G[8][8]\ 8 & 0 & 0 & \dots & 1 & 0 \\
G[8][8]\ 9 & 0 & 0 & \dots & 0 & 1
\end{array}$$

Figure 22: *The row constraints depicted in the grid and matrix M*

For each of the nine rows in *M*, the 1s are arranged in a new column for one cell in *G*. This is repeated for the first nine cells in *G* or the first 81 rows in M in the same columns. The following row restriction begins at the 10th cell or on row 82, while the 1s begin after the final column utilised by the first row in *M* (Hanson, Robert M., 2022). Since the columns with 1s span the same columns for the cells on the same row, the algorithm is forced to select a unique integer value for each cell in the row, yet only one column can be in the final solution.

7.6.3 Column constraint

The columns in G, like rows, can only have a set of numbers between 1 and 9. To keep this limitation from G in the matrix, the 1s are arranged in a different pattern.

The 1s are inserted in a new column for each of the nine rows in M for every cell in G. This is done for the second cell, except that instead of utilising the same columns as for the row restriction, the 1s begin where the last cell ended (Hanson, Robert M, 2022). This is the case for the first nine cells. The pattern resumes in the first column for the next nine cells. This will match each cell in a column in M with a cell in a column in G. This forces the algorithm to select a single unique integer value for each cell in the G column.

	1	2	3	4	5	6	7	8	9	10	11		81
$G[0][0]$ 1	1	0	0	0	0	0	0	0	0	0	0	...	0
$G[0][0]$ 2	0	1	0	0	0	0	0	0	0	0	0	...	0
⋮													
$G[0][1]$ 1	0	0	0	0	0	0	0	0	0	1	0	...	0
$G[0][1]$ 2	0	0	0	0	0	0	0	0	0	0	1	...	0
⋮													
$G[1][0]$ 1	1	0	0	0	0	0	0	0	0	0	0	...	0
$G[1][0]$ 2	0	1	0	0	0	0	0	0	0	0	0	...	0
⋮													

Figure 23: *The column constraints depicted using the grid*

7.6.4 Box constraint

The box constraint, like the other three constraints, will follow a pattern. Each box has three rows and three columns, resulting in what seems to be an irregular pattern at first.

The first three cells in G will have the same columns of 1s as the first three cells in M. The following three cells in G will have the same columns of 1s as the previous three cells in that row, and the last three cells in that row will similarly have the same columns of 1s. This is due to the fact that the nine cells on the first row of G are divided into three separate boxes. The pattern is repeated for all of the rows in G (Hanson, Robert M, 2022).

This will impose a limit on three full boxes. If their columns are in the same box in G, they are equivalent in M. Columns are set in the same order as in the row constraint in M for one cell in G, first a 1 in the first column, then a 1 in the second column for the second row in M, and so on until the next cell in G. This is due to the fact that the numbers in the box must contain nine distinct integers ranging from 1 to 9. This forces the algorithm to choose rows based on the grid restriction once again.

This was the final restriction. Because each constraint requires 81 columns, the total number of columns necessary in M is $81 \cdot 4 = 324$.

$$
\begin{array}{l|c|c|c|c|c|c|c|c|c|c|}
 & 1 & 2 & & 10 & 11 & & 19 & 20 & & 81 \\
\hline
G[0][0]\ 1 & 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\
G[0][0]\ 2 & 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\
\hline
\vdots & & & & & & & & & & \\
\hline
G[0][1]\ 1 & 0 & 0 & \dots & 1 & 0 & \dots & 0 & 0 & \dots & 0 \\
G[0][1]\ 2 & 0 & 0 & \dots & 0 & 1 & \dots & 0 & 0 & \dots & 0 \\
\hline
\vdots & & & & & & & & & & \\
\hline
G[0][2]\ 1 & 0 & 0 & \dots & 0 & 0 & \dots & 1 & 0 & \dots & 0 \\
G[0][2]\ 2 & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 0 \\
\hline
\vdots & & & & & & & & & & \\
\hline
G[1][0]\ 1 & 1 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\
G[1][0]\ 2 & 0 & 1 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\
\hline
\vdots & & & & & & & & & &
\end{array}
$$

Figure 24: *Box constraints depicted using the grid*

7.7. Obtaining a solution

Obtaining the grid solution is simple, but needs a slight adjustment to DLX. Each data object must additionally have a field for recording the row location in the original binary matrix *M*, since *M* is transformed by DLX into the links, there is no way to tell which data item is for which row in M from the obtained solution.

Once a solution has been found, the list of data items must be sorted in descending order by their row number. The solution should consist of 81 rows, one for each cell in the grid being solved (D. Knuth, 2000). Because the domain ranges from 1 to 9, we will use modulus 9 for each row. Because the implementation uses zero-based indices whereas the solution does not, the row number must be incremented with one before applying modulus.

As a result, the solution discovered by DLX may be quickly turned back into the original grid, but without any unknown cells. The grid may then be checked to see if it satisfies the constraints.

8. Methodology

8.1. Chapter Introduction

This section of the report will assess which approach should be used throughout the project. Following that, the implementation of the chosen technique will be outlined.

8.2. The chosen methodology

The Agile development technique is being used for this project. Agile methodology emphasizes responding to change, such as new user feedback, changes in scope, rather than following a predetermined plan (Software Development, 2001), as is the case with Waterfall development. The Agile manifesto specifies four values:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

The ability to react quickly and effectively to changing circumstances is a common trait shared by all of these values. Agile is a methodology, not a set of instructions, and its major goal is to help development teams achieve their goals, not how. The details of 'how' to achieve this are determined by the specific Agile methodology used – which might include Scrum, Feature-driven development, and others.

The Scrum framework was selected as the Agile framework for this project. Scrum is a project management paradigm that emphasises cooperation, responsibility, and incremental progress toward a well-defined objective (Search Software Quality, n.d.).

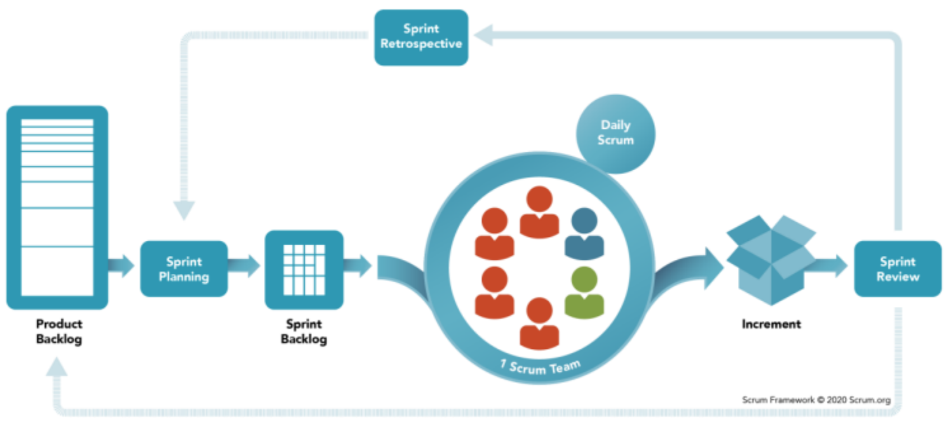


Figure 25: *Scrum flow (Scrum.org, 2022)*

8.3. Sprint Planning

Scrum divides tasks in the product backlog into “sprints”. During a sprint, a team focuses on completing a specific set of tasks from the product backlog, which is a list of all the features and functionality that the product must have. Scrum allows for a product owner to determine the priority of these tasks, but in this project, I will take on this role myself. Once each sprint has been completed, I will present a demonstration of the software to my project supervisor. Each sprint need is stated in **Table 1** in the *Requirements* section.

8.4. Tests

Following each sprint, a series of tests are performed on the app to ensure that it works as expected.

8.4.1 Sprint one – User Interface

The goal of this sprint is to create the system’s interface. The following requirements must be met throughout this sprint:

- All1
- SSA1

8.4.2 Sprint two – Algorithm X and the ability to upload a grid

The focus of this sprint is on Algorithm X and the ability of the user to upload a grid. The requirements to be implemented in this sprint include:

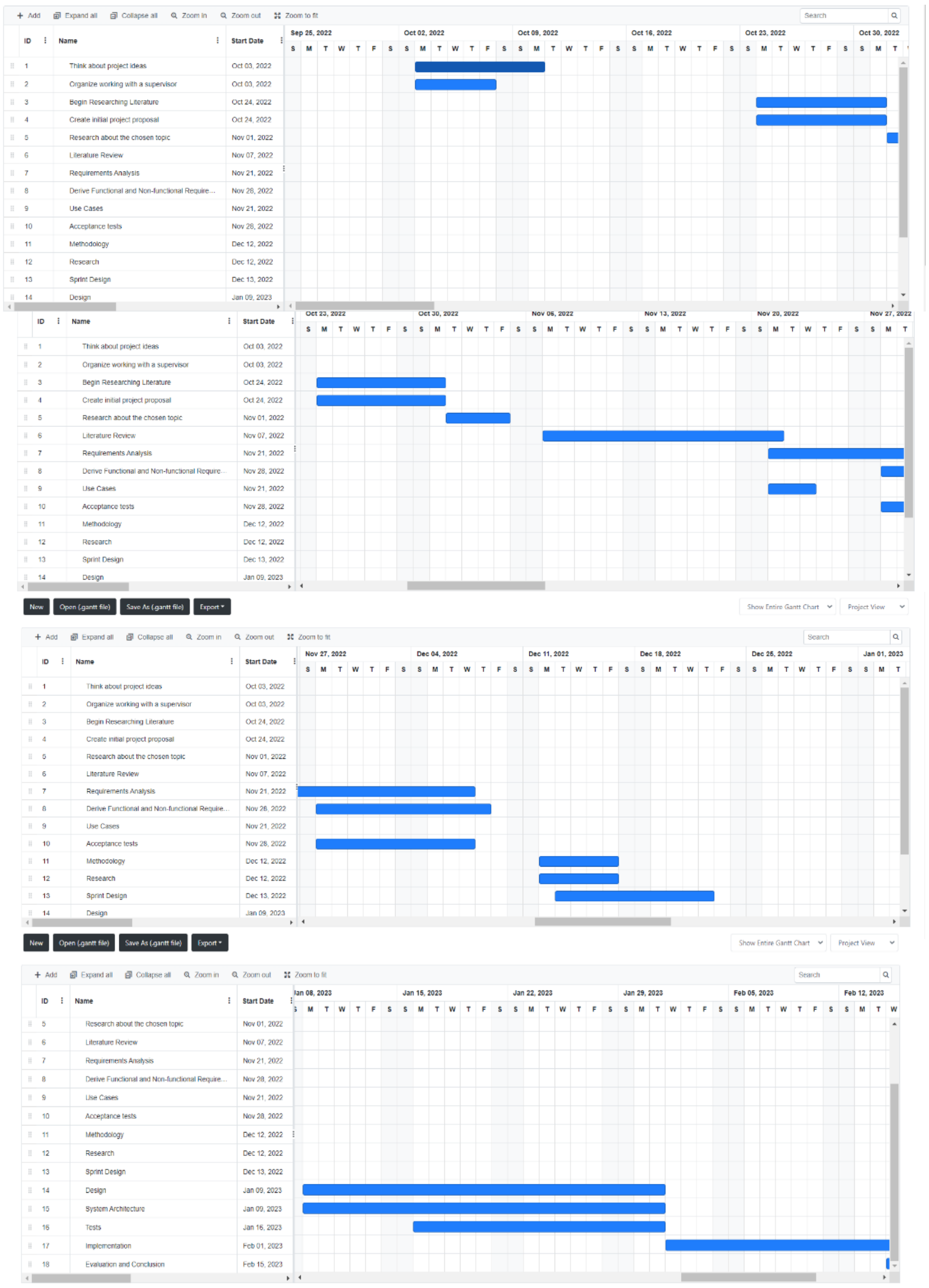
- SSA2
- SSA3
- SSA4
- SSA7
- SSA8
- SSA9

8.4.3 Sprint three – Augmented Reality and speed performance

The focus of this sprint is on Augmented Reality and the speed performance of the application. The requirements to be implemented in this sprint include:

- SSA2
- SSA3
- SSA4
- SSA5
- SSA6
- SSA7
- SSA8
- SSA9

8.5. Project Timeline



ID	Name	Start Date
1	Think about project ideas	Oct 03, 2022
2	Organize working with a supervisor	Oct 03, 2022
3	Begin Researching Literature	Oct 24, 2022
4	Create initial project proposal	Oct 24, 2022
5	Research about the chosen topic	Nov 01, 2022
6	Literature Review	Nov 07, 2022
7	Requirements Analysis	Nov 21, 2022
8	Derive Functional and Non-functional Require...	Nov 28, 2022
9	Use Cases	Nov 21, 2022
10	Acceptance tests	Nov 28, 2022
11	Methodology	Dec 12, 2022
12	Research	Dec 12, 2022
13	Sprint Design	Dec 13, 2022
14	Design	Jan 09, 2023
15	System Architecture	Jan 09, 2023
16	Tests	Jan 16, 2023
17	Implementation	Feb 01, 2023
18	Evaluation and Conclusion	Feb 15, 2023

Figure 26: *Project Gantt Chart*

9. Design

This section covers the overall design and architecture of the Sudoku solver system. The criteria described in the Requirements section influenced the choices and decisions made in the system's design. This section starts with a high-level thorough study of the Frontend and Backend before moving on to a more extensive explanation.

9.1 High Level Design

9.1.1 Architecture Diagram

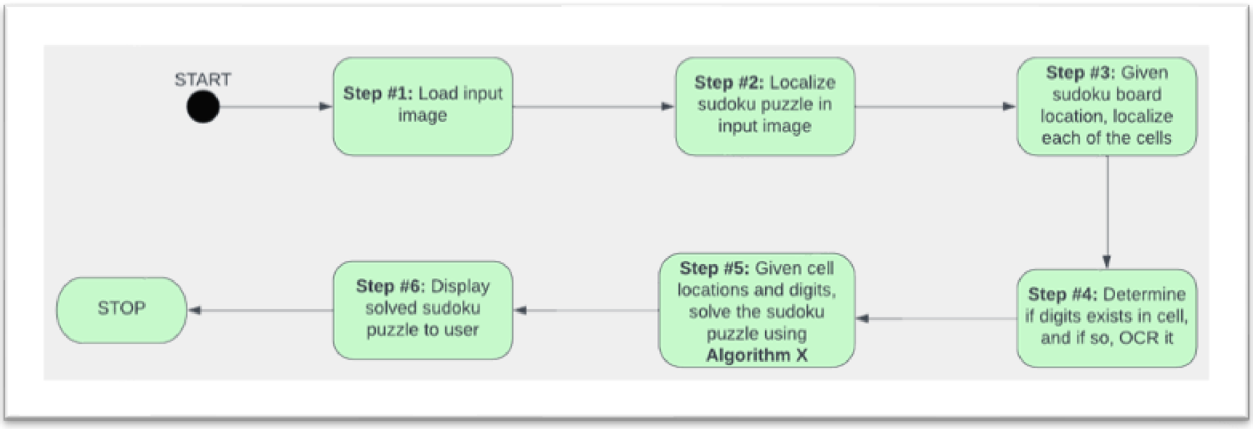

START
Step #1: Load input image
Step #2: Localize sudoku puzzle in input image
Step #3: Given sudoku board location, localize each of the cells
Step #4: Determine if digits exists in cell, and if so, OCR it
Step #5: Given cell locations and digits, solve the sudoku puzzle using Algorithm X
Step #6: Display solved sudoku puzzle to user
STOP

Figure 27: *Architecture diagram of the Solver*

9.1.2 Use Case Diagram for Interface

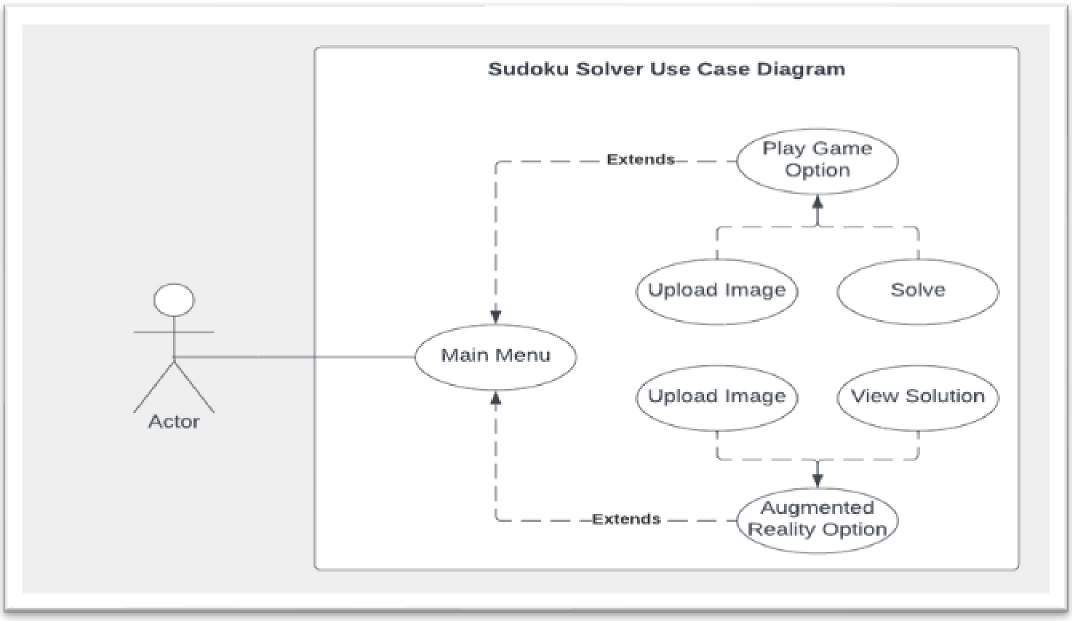


Figure 28: *Interface Use-Case Diagram*

Figure 28 illustrates how a particular user interacts with the GUI and how they may use the two options. Users in this example comprise both game players and ordinary users who do not own the game but want to see the answer to a certain grid.

9.1.3 Use Case Diagram for Play Game Option

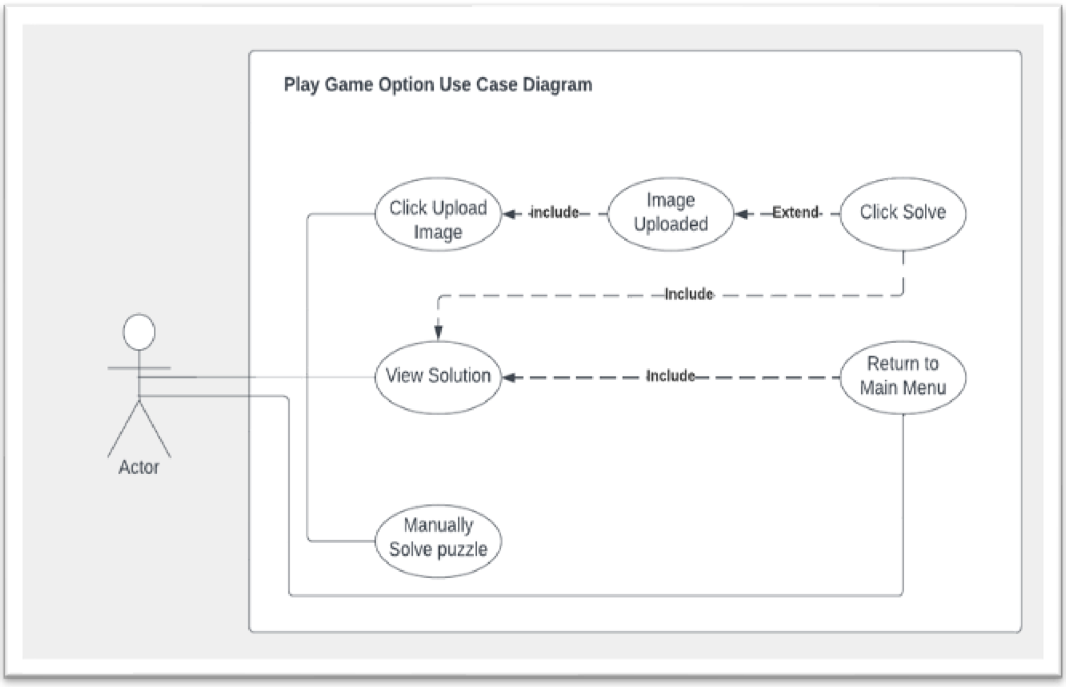


Figure 29: *Play Game – Use Case Diagram*

This above diagram depicts the interaction of the player with the Play game option. The purpose of this is to view the solution of an input puzzle by solving it manually or using Algorithm X and returning to the main menu after the chosen puzzle has been solved.

9.1.4 Use Case Diagram for Augmented Reality Option

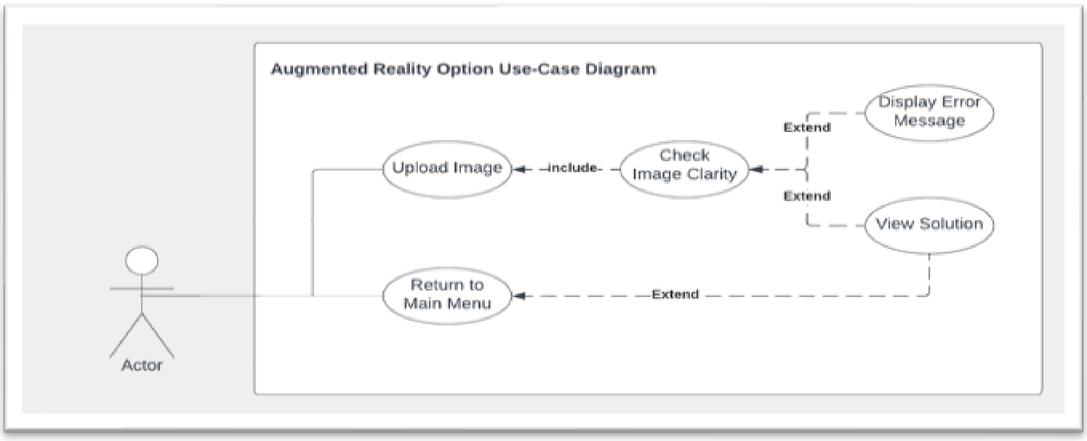


Figure 30: *AR Use-Case Diagram*

Figure 30 depicts the interaction of the player within the AR option. The purpose of this is to view the solution of an input puzzle in AR and returning to the main menu after the solution was displayed.

9.2 Low Level Design

9.2.1 Frontend component

The frontend component is responsible for displaying the view to an user and responding to actions performed by them. It handles displaying two options, “Play Game” and “Augmented Reality”.

The functional requirements associated with this component include:

- All1
- SSA1
- SSA4
- SSA7
- SSA8
- SSA9

9.2.1.1 User Interface

The principal medium of interaction with the application is the user interface. The frontend was created in Python.

Figure 31 depicts the user interface design. The goal is to provide a general outline for how the UI will be put up and where certain critical features will be located.

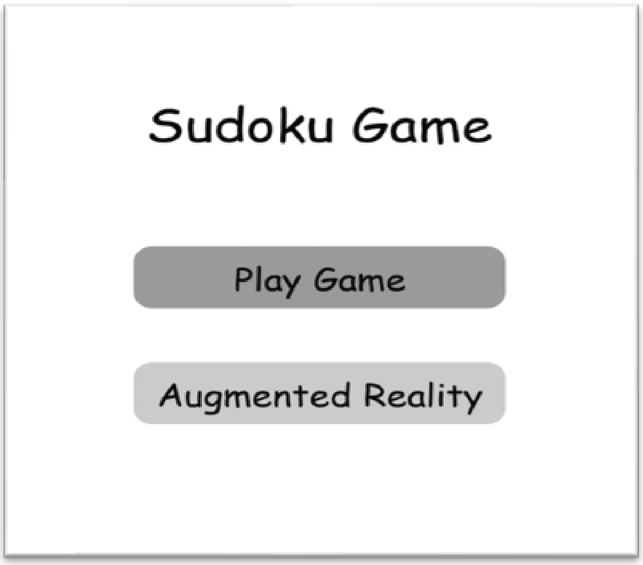


Figure 31: *GUI design of Sudoku solver*

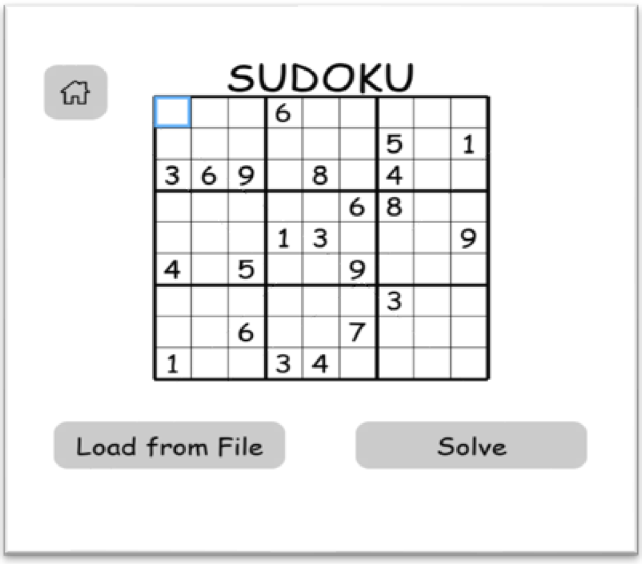


Figure 32: *Play Game option – a loaded grid*

9.2.1.2 Play Game Option

Figure 32 depicts the application interface when the user chooses the "Play Game" option. The user has the possibility to load a new grid–image using the "Load from File" or they can click on "Solve" to display the solution using the *Dancing Links* technique implemented with *Algorithm X*.

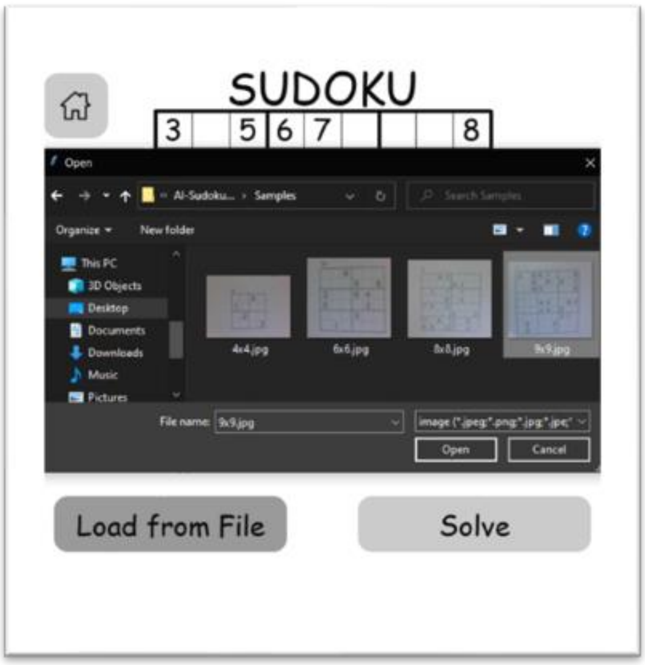
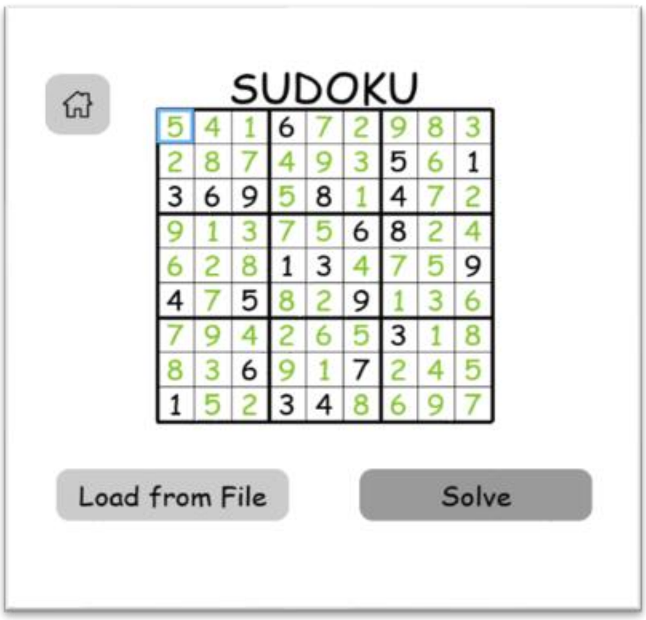


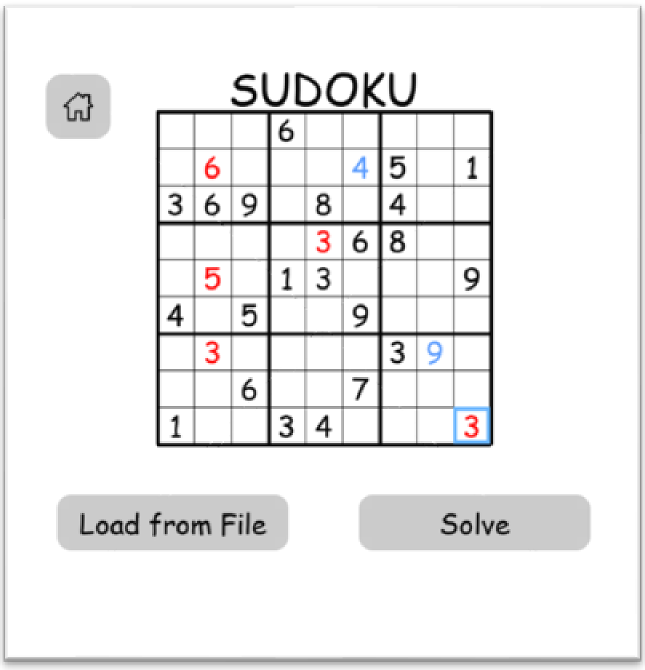
Figure 33: *Load a file option*



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

Figure 34*: Solve option*

Therefore, the users can try to solve the puzzle by themselves – they are warned when they enter a number in a cell that does not respect the Sudoku rules – by changing the font colour from **black** to **red**. In addition, the candidate number is represented by the **blue** colour as shown in **Figure 35**.



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

Figure 35: *Candidates which do not meet the Sudoku conditions; possible candidates*

Moreover, when the "Solve" option is selected, as well as when the user properly solves the puzzle, a message similar to the one depicted in **Figure 36** is displayed.

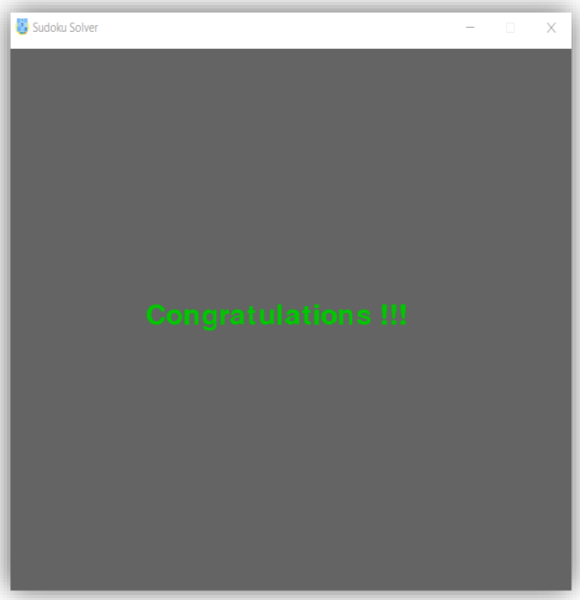
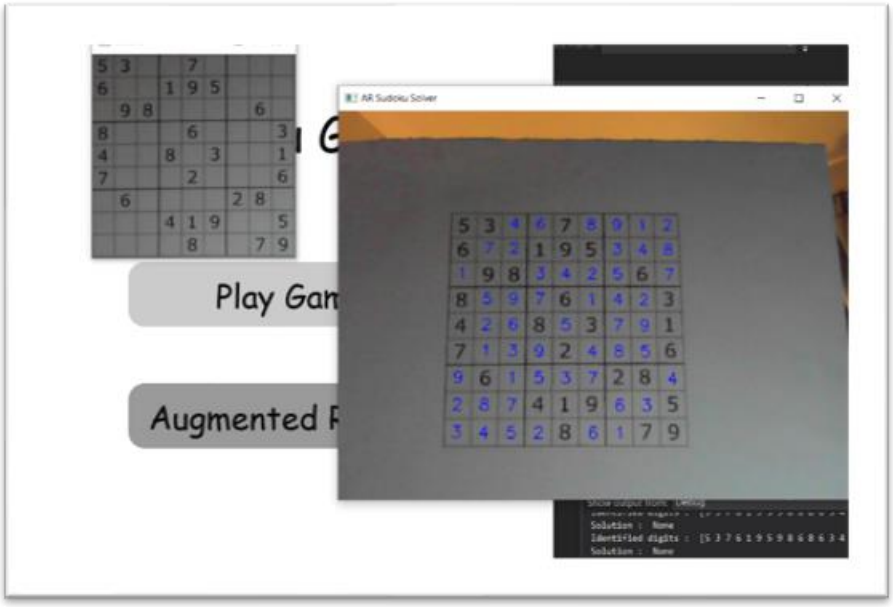


Figure 36: *A message's displayed after the puzzle has been correctly solved*

9.2.1.3 Augmented Reality Option

Figure 37 depicts when the AR option was selected. The user presents a puzzle printed on paper, which they bring in front of the camera, and the solution of the puzzle is displayed on the screen, but also in the console in the form of a list.



```
Show output from: Debug
Solution : None
Identified digits : [5 3 7 6 1 9 5 9 8 6 8 6 3 4
Solution : None
```

Figure 37: *Displaying solution in AR*

The users can be notified by a message that the puzzle they are trying to present to the camera does not comply with certain conditions – low light, the puzzle being too far away from the camera as depicted in **Figure 38**.

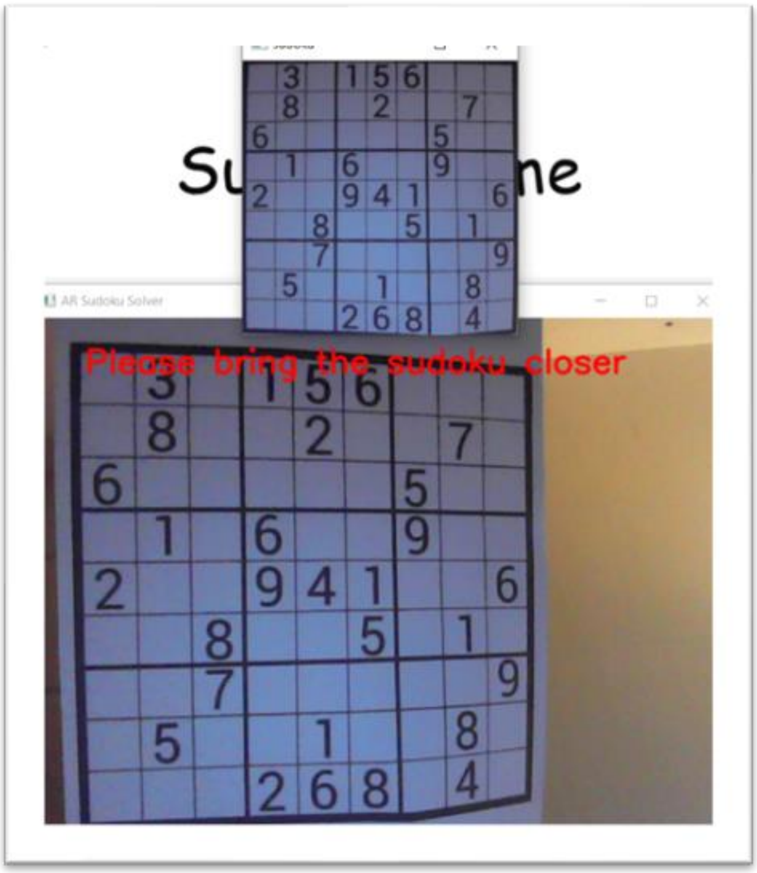


Figure 38: *A warning message – the puzzle is too far from camera*

9.2.2 Backend component

The backend component for communication between the user and the solver. It responds to commands from the frontend. The functional requirements associated with this component include:

- All1
- SSA2
- SSA3
- SSA5
- SSA6
- SSA7
- SSA8
- SSA9

9.2.2.1 Sequence Diagram of the solver

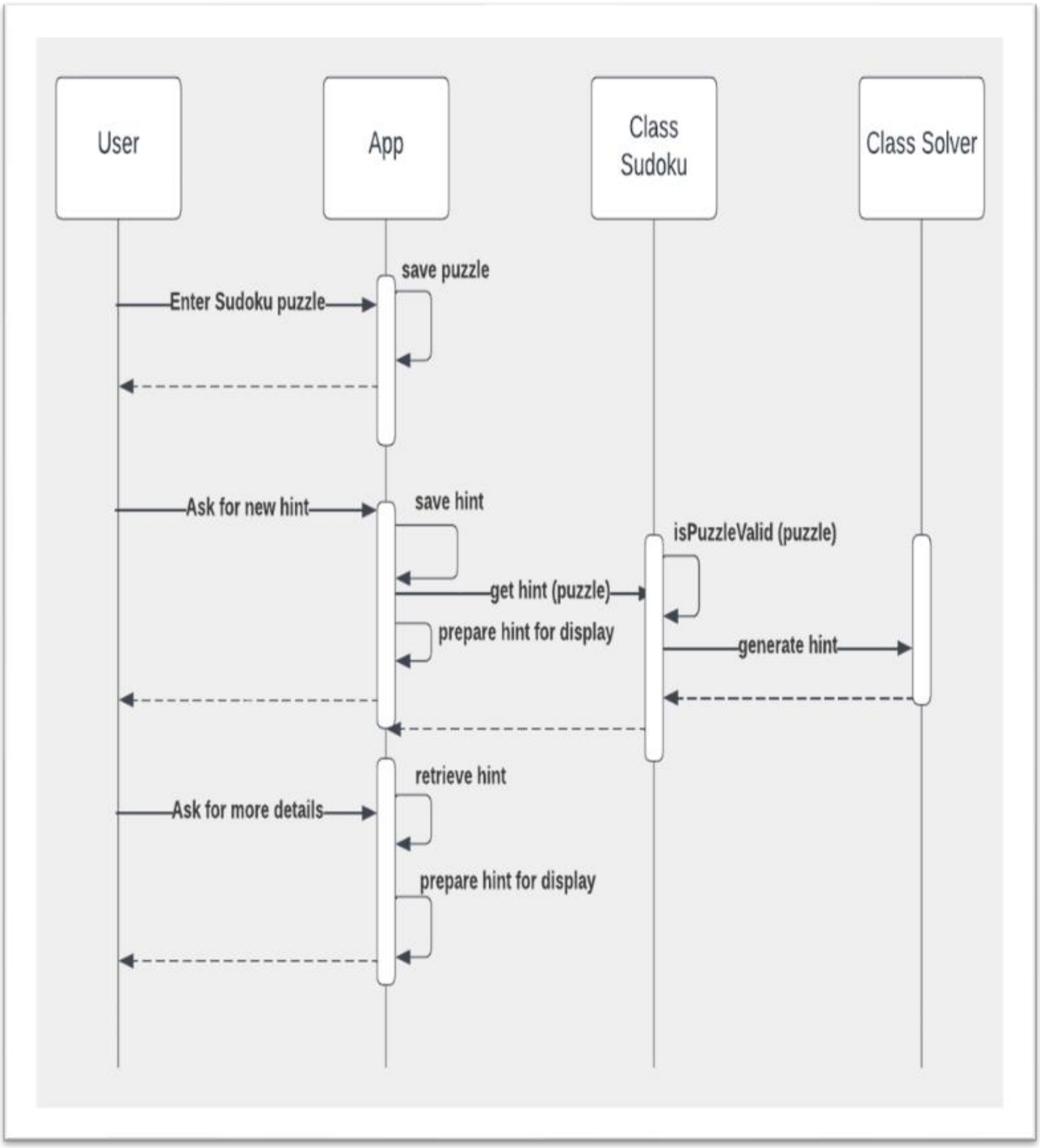


Figure 39: *Sequence Diagram*

Figure 39 depicts the sequence of messages passed between the objects in connection to the solver.

9.2.2.2 Functional Diagram of the solver

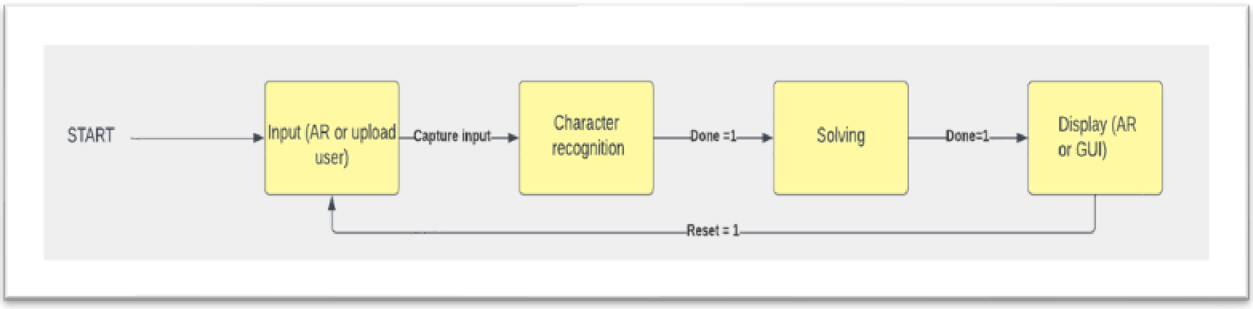


Figure 40: *Functional Diagram – process of displaying the solution*

The previous diagram shows the process of displaying the solution of a puzzle – the input is taken from AR/Play Game Option, the 9×9 grid is populated using Character Recognition, then it is solved using Algorithm X and the solution is displayed according to the option selected in step 1.

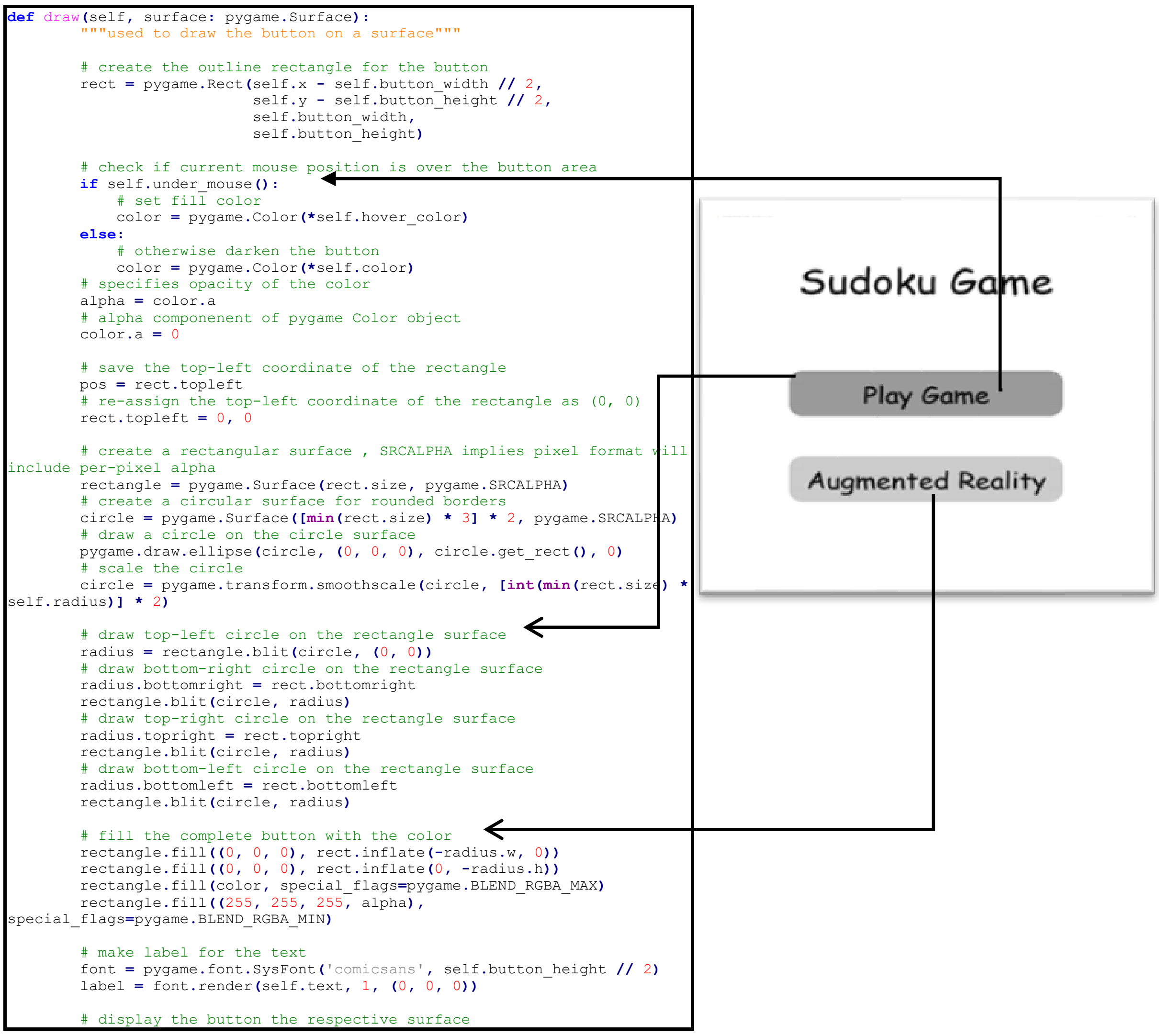
10. Implementation

This section aims to describe how the project was implemented. This chapter is divided into three sprints, each addressing a different component of the application. After each sprint, the results and feedback from the tests are evaluated.

10.1. Sprint 1 – User Interface

This sprint is allocated to the development of the system's frontend.

10.1.1 Buttons and methods



```
def draw(self, surface: pygame.Surface):
        """used to draw the button on a surface"""

        # create the outline rectangle for the button
        rect = pygame.Rect(self.x - self.button_width // 2,
                           self.y - self.button_height // 2,
                           self.button_width,
                           self.button_height)

        # check if current mouse position is over the button area
        if self.under_mouse():
            # set fill color
            color = pygame.Color(*self.hover_color)
        else:
            # otherwise darken the button
            color = pygame.Color(*self.color)
        # specifies opacity of the color
        alpha = color.a
        # alpha componenent of pygame Color object
        color.a = 0

        # save the top-left coordinate of the rectangle
        pos = rect.topleft
        # re-assign the top-left coordinate of the rectangle as (0, 0)
        rect.topleft = 0, 0

        # create a rectangular surface , SRCALPHA implies pixel format will
include per-pixel alpha
        rectangle = pygame.Surface(rect.size, pygame.SRCALPHA)
        # create a circular surface for rounded borders
        circle = pygame.Surface([min(rect.size) * 3] * 2, pygame.SRCALPHA)
        # draw a circle on the circle surface
        pygame.draw.ellipse(circle, (0, 0, 0), circle.get_rect(), 0)
        # scale the circle
        circle = pygame.transform.smoothscale(circle, [int(min(rect.size) *
self.radius)] * 2)

        # draw top-left circle on the rectangle surface
        radius = rectangle.blit(circle, (0, 0))
        # draw bottom-right circle on the rectangle surface
        radius.bottomright = rect.bottomright
        rectangle.blit(circle, radius)
        # draw top-right circle on the rectangle surface
        radius.topright = rect.topright
        rectangle.blit(circle, radius)
        # draw bottom-left circle on the rectangle surface
        radius.bottomleft = rect.bottomleft
        rectangle.blit(circle, radius)

        # fill the complete button with the color
        rectangle.fill((0, 0, 0), rect.inflate(-radius.w, 0))
        rectangle.fill((0, 0, 0), rect.inflate(0, -radius.h))
        rectangle.fill(color, special_flags=pygame.BLEND_RGBA_MAX)
        rectangle.fill((255, 255, 255, alpha),
special_flags=pygame.BLEND_RGBA_MIN)

        # make label for the text
        font = pygame.font.SysFont('comicsans', self.button_height // 2)
        label = font.render(self.text, 1, (0, 0, 0))

        # display the button the respective surface
```

Figure 41: *Draw button method*

To create a button in pygame, we draw it over a 2D image called a "pygame.surface" object. The button is defined as a rectangular shape, and its color changes when the mouse is over it. Circular outlines are added to the corners, and the button is filled with the chosen color while the outlines and text are filled with black. Finally, the button is displayed on the surface.

10.1.2 Click Methods

```
def clicked(self, event):
        """used to check if the button is clicked"""

        # if LEFT mouse button is clicked
        if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
            # if the mouse was over the button during click
            return self.under_mouse()
```

Figure 42: *Method Clicked*

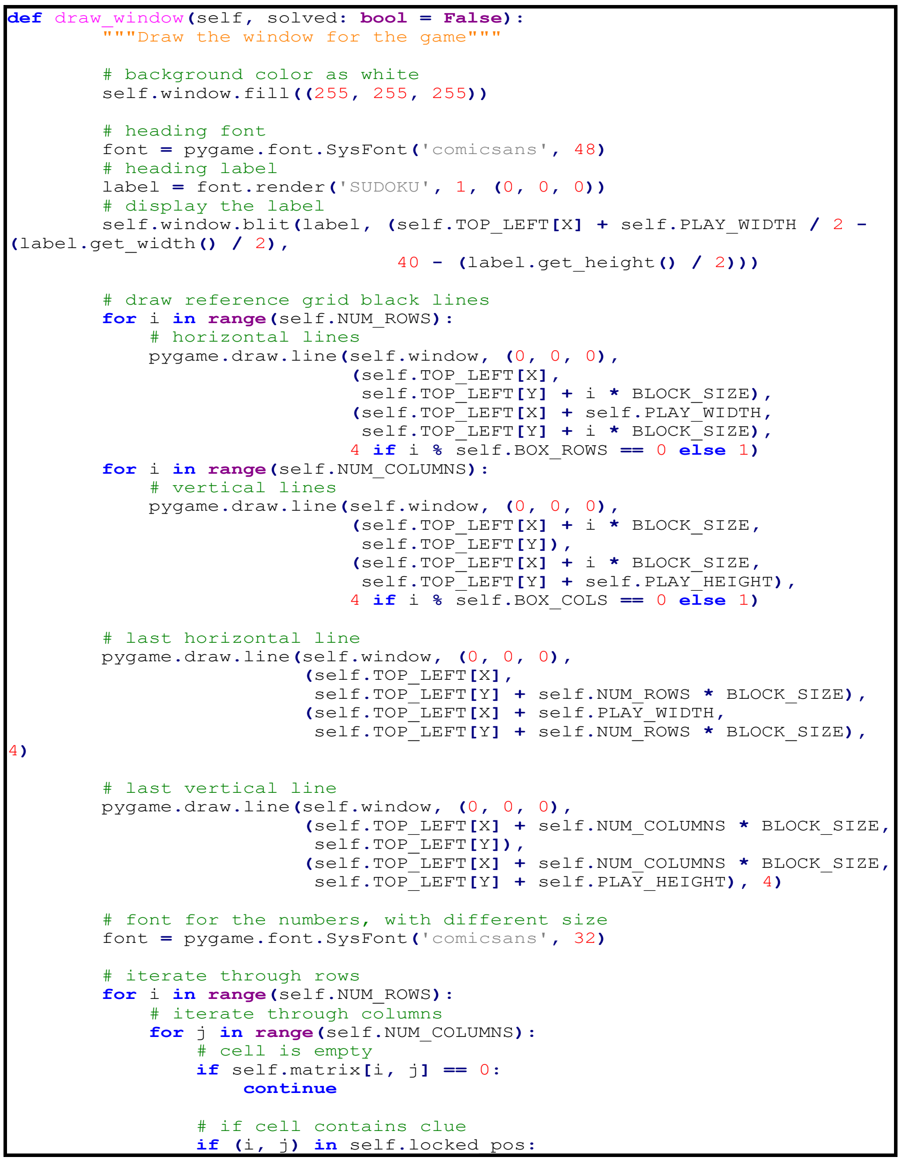
If the mouse button is clicked while the mouse is over the button, this function returns True; otherwise, it returns False.

```
def under_mouse(self):
        """find if the current mouse coordinates"""

        # get the current coordinates of the mouse
        mouse_x, mouse_y = pygame.mouse.get_pos()
        # if mouse coordinates during the click are in the range of the
button coordinate
        if mouse_x in range(self.x - self.button_width // 2, self.x +
self.button_width // 2) and \
                mouse_y in range(self.y - self.button_height // 2, self.y +
self.button_height // 2):
            # return true
            return True
        # otherwise return false
        return False
```

Figure 43: *Check if the Button is Under the Mouse*

This method obtains the mouse coordinates and determines if the coordinates of the button are inside the region of the button space. If the x-coordinate is within the width of the button, and the y-coordinate is within the height, it returns True; otherwise, it returns False.

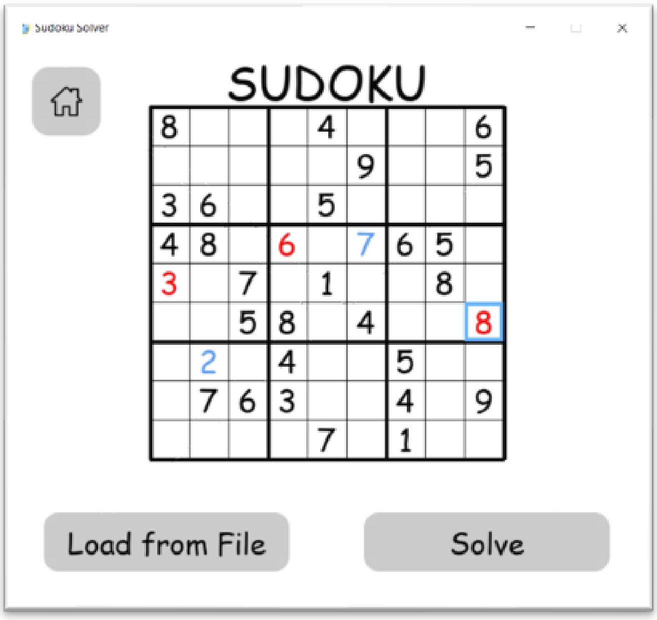
10.1.3 Game window and Mouse Clicker events



```
def draw_window(self, solved: bool = False):
        """Draw the window for the game"""

        # background color as white
        self.window.fill((255, 255, 255))

        # heading font
        font = pygame.font.SysFont('comicsans', 48)
        # heading label
        label = font.render('SUDOKU', 1, (0, 0, 0))
        # display the label
        self.window.blit(label, (self.TOP_LEFT[X] + self.PLAY_WIDTH / 2 -
(label.get_width() / 2),
                                 40 - (label.get_height() / 2)))

        # draw reference grid black lines
        for i in range(self.NUM_ROWS):
            # horizontal lines
            pygame.draw.line(self.window, (0, 0, 0),
                             (self.TOP_LEFT[X],
                              self.TOP_LEFT[Y] + i * BLOCK_SIZE),
                             (self.TOP_LEFT[X] + self.PLAY_WIDTH,
                              self.TOP_LEFT[Y] + i * BLOCK_SIZE),
                             4 if i % self.BOX_ROWS == 0 else 1)
        for i in range(self.NUM_COLUMNS):
            # vertical lines
            pygame.draw.line(self.window, (0, 0, 0),
                             (self.TOP_LEFT[X] + i * BLOCK_SIZE,
                              self.TOP_LEFT[Y]),
                             (self.TOP_LEFT[X] + i * BLOCK_SIZE,
                              self.TOP_LEFT[Y] + self.PLAY_HEIGHT),
                             4 if i % self.BOX_COLS == 0 else 1)

        # last horizontal line
        pygame.draw.line(self.window, (0, 0, 0),
                         (self.TOP_LEFT[X],
                          self.TOP_LEFT[Y] + self.NUM_ROWS * BLOCK_SIZE),
                         (self.TOP_LEFT[X] + self.PLAY_WIDTH,
                          self.TOP_LEFT[Y] + self.NUM_ROWS * BLOCK_SIZE),
4)

        # last vertical line
        pygame.draw.line(self.window, (0, 0, 0),
                         (self.TOP_LEFT[X] + self.NUM_COLUMNS * BLOCK_SIZE,
                          self.TOP_LEFT[Y]),
                         (self.TOP_LEFT[X] + self.NUM_COLUMNS * BLOCK_SIZE,
                          self.TOP_LEFT[Y] + self.PLAY_HEIGHT), 4)

        # font for the numbers, with different size
        font = pygame.font.SysFont('comicsans', 32)

        # iterate through rows
        for i in range(self.NUM_ROWS):
            # iterate through columns
            for j in range(self.NUM_COLUMNS):
                # cell is empty
                if self.matrix[i, j] == 0:
                    continue

                # if cell contains clue
                if (i, j) in self.locked_pos:
```

Figure 44: *Draw Game Window method*

We begin by filling the window with white and displaying the Sudoku title. The horizontal and vertical lines are then created by iterating over the number of rows and columns. Finally, we cycle through the puzzle matrix, displaying the number in the appropriate cell if the element is a not zero. If the location is of a locked cell, we display the number in black; otherwise, we verify if the number is a possible number for that cell and display the text in blue or red as seen in **Figure 45**.



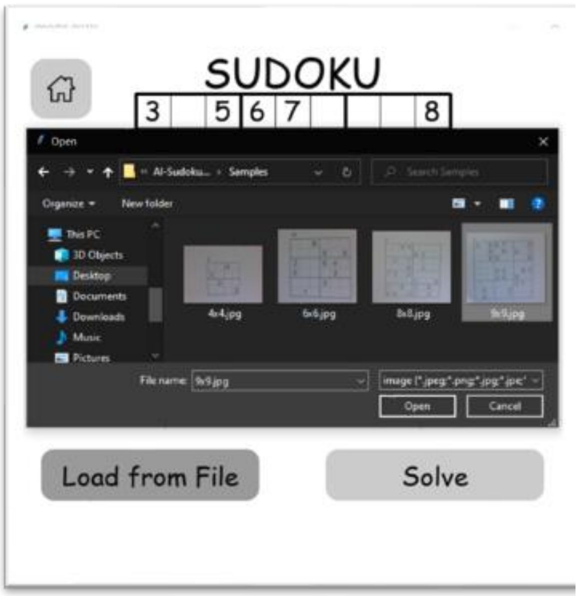
8				4				6
					9			5
3	6			5				
4	8		6		7	6	5	
3		7		1			8	
		5	8		4			8
	2		4			5		
	7	6	3			4		9
				7		1		

Figure 45: *Game window*

```
def handle_click(self, event):
        """This method helps handle leff mouse clicks"""
        # check if load from file button is clicked
        if self.button_load_image.clicked(event):
            # open the dialog box to select file
            path = askopenfilename(filetypes=[("image", ".jpeg"),
                                              ("image", ".png"),
                                              ("image", ".jpg"),
                                              ("image", ".jpe"),
                                              ("image", ".bmp")])

            # if file was selected
            if len(path) != 0:
                # load the file for image processing
                sip = SudokuImageProcessing(fname=path)
                # detect puzzle matrix
                mat = sip.get_matrix()
                # if matrix not detected
                if mat is None:
                    # show error
                    tkinter.messagebox.showerror(title="Error",
                                                 message="Unable to load
file, try with different image.")
                    return
                # get the dimensions of the puzzle
                _, (box_rows, box_cols) = sip.get_dimensions()
                # re-initialize the object with the new data
                self.__init__(mat, box_rows, box_cols)

                # save the current puzzle loaded
                np.save('last_loaded.npy', self.matrix)
                # save the dimensions of the current puzzle
                np.save('last_loaded_dim.npy', np.array([self.BOX_ROWS,
self.BOX_COLS]))
                # restart game
                self.play_game()
        # check if load from camera button is clicked
        if self.button_cam_image.clicked(event):
            # open the camera feed window
            with CameraWindow(capture_image=True) as feed:
                # while button is not pressed
                while True:
                    # frame is None until the button is clicked
                    frame = feed.draw_window()
                    if frame is not None:
                        break
            # intenstional delay to avoid accidental mouse clicks
            sleep(0.5)
            try:
                # load the image for image processing
                sip = SudokuImageProcessing(image=frame)
                # detect puzzle matrix
                mat = sip.get_matrix()
                # if matrix not detected
                if mat is None:
                    # raise exception
                    raise RuntimeError()
                # get the dimensions of the puzzle
                _, (box_rows, box_cols) = sip.get_dimensions()
                # re-initialize the object with the new data
                self.__init__(mat, box_rows, box_cols)
                # save the current puzzle loaded
                np.save('last_loaded.npy', self.matrix)
                # save the dimensions of the current puzzle
                np.save('last_loaded_dim.npy', np.array([self.BOX_ROWS,
self.BOX_COLS]))
                # restart game
                self.play_game()
            except RuntimeError:
                # show error
                tkinter.messagebox.showerror(title="Error", message="Image not
clear, please try again.")
                # reset screen size back to default game screen size
                self.window = pygame.display.set_mode((SCREEN_WIDTH,
SCREEN_HEIGHT))
                return
        # check if solve button is clicked
        if self.button_solve.clicked(event):
            # reset the matrix to inital state, i.e. remove all current entries
            self.matrix = self.init_matrix.copy()
            # while not solved
            while 0 in self.matrix:
                # find positions of empty cells
                rows, cols = np.where(self.matrix == 0)
                # choose a random coordinate of an empty cell
                coords = choice(list(zip(rows, cols)))
                # fill the cell with the solution value
                self.matrix[coords] = self.solution[coords]
                # delay to better visualize
                sleep(0.1)
                # draw the entries with green color
                self.draw_window(solved=True)
            # status variable
            hold_screen = True
            # while not action performed stay on screen
            while hold_screen:
                # iterate through events
                for event in pygame.event.get():
                    # if event is of type key press or button click
                    if event.type in (pygame.KEYDOWN, pygame.QUIT,
pygame.MOUSEBUTTONDOWN):
                        # break outer while loop
                        hold_screen = False
        # check if home button is clicked
        if self.button_home.clicked(event):
            # go back to main menu
            return False
        # if LEFT mouse button is clicked
        if event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
            # get the current coordinates of the mouse
            mouse_x, mouse_y = pygame.mouse.get_pos()
            # convert to integer
            mouse_x, mouse_y = int(mouse_x), int(mouse_y)
            # if mouse coordinates during the click are in the range of the game
area
            if mouse_x in range(self.TOP_LEFT[X], self.TOP_LEFT[X] +
self.NUM_COLUMNS * BLOCK_SIZE) and \
                    mouse_y in range(self.TOP_LEFT[Y], self.TOP_LEFT[Y] +
self.NUM_ROWS * BLOCK_SIZE):
                # select the box in which the mouse is clicked
                self.selected_box = ((mouse_x - self.TOP_LEFT[X]) // BLOCK_SIZE,
                                     (mouse_y - self.TOP_LEFT[Y]) // BLOCK_SIZE)
        # default return is True
        return True
```

Figure 46: *Mouse Click Handler*

The system initially starts by determining if the "Load Image from File" button is pressed. If this is the case, we open the image with the Tkinter library's "askopenfilename" function. The "SudokuImageProcessoring" object is then used to load the picture and retrieve the matrix. If the image is valid, the puzzle is displayed on the screen and the matrix and its dimensions are recorded; otherwise, an error dialogue box with the message "Unable to load image" appears.



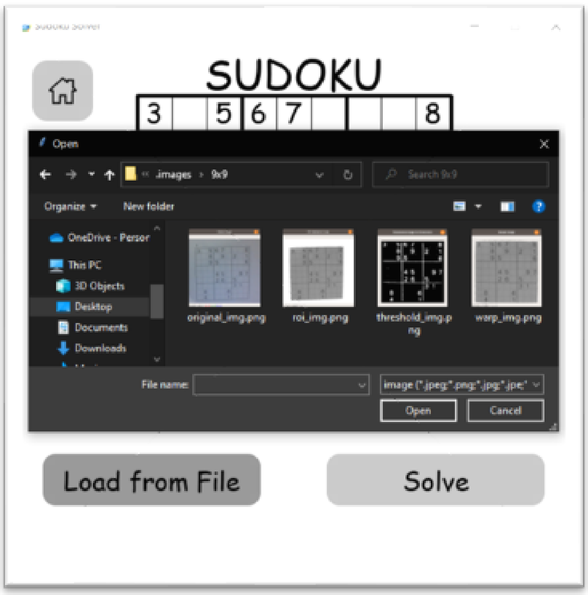


Figure 47: *Load from file option*

10.1.4 Test results

The tests for this sprint can be found in **Table 3**, and the results can be found in **Table 6** in the Appendix chapter.

10.2. Sprint 2 – Play Game Option

This sprint is allocated to backend component – Algorithm X implementation.

10.2.1 Game Play

```
def play_game(self):
        """This method handles the input part of the Game"""

        # display the game GUI
        self.draw_window()
        # iterate through events
        for event in pygame.event.get():

            # check for any click, return True if home is pressed
            if not self.handle_click(event):
                # return to the main menu
                return False

            # kill game, if window is closed or escape key is pressed
            if event.type == pygame.QUIT or (event.type == pygame.KEYDOWN and
event.key == pygame.K_ESCAPE):
                # save and exit
                self.graceful_exit()

            # if a key is pressed
            if event.type == pygame.KEYDOWN:
                # get coordinates of selected box
                box_i, box_j = self.selected_box
                # if up arrow key
                if event.key == pygame.K_UP:
                    # shift the selected box up
                    box_j -= 1

                # if down arrow key
                if event.key == pygame.K_DOWN:
                    # shift the selected box down
                    box_j += 1

                # if right arrow key
                if event.key == pygame.K_RIGHT:
                    # shift the selected box right
                    box_i += 1

                # if left arrow key
                if event.key == pygame.K_LEFT:
                    # shift the selected box left
                    box_i -= 1

                # update the selected box
                self.selected_box = (box_i % self.NUM_ROWS, box_j %
self.NUM_COLUMNS)

        # check for keys pressed
        keys = pygame.key.get_pressed()
        # if escape key is pressed
        if keys[pygame.K_ESCAPE]:
            # save and exit
            self.graceful_exit()

        # get coordinates of selected box
        box_i, box_j = self.selected_box
        # iterate throught the number keys
        for i in range(pygame.K_0, pygame.K_0 + self.NUM_ROWS + 1):
            # if key is pressed and the box does not contain a clue
            if keys[i] and (box_j, box_i) not in self.locked_pos:
                # fill the numeric value of the key pressed
                self.matrix[(box_j, box_i)] = i - pygame.K_0

        # iterate throught the numpad keys
        for i in range(pygame.K_KP0, pygame.K_KP0 + self.NUM_ROWS + 1):
            # if key is pressed and the box does not contain a clue
            if keys[i] and (box_j, box_i) not in self.locked_pos:
                # fill the numeric value of the key pressed
                self.matrix[(box_j, box_i)] = i - pygame.K_KP0

        # if delete key is pressed and the box does not contain a clue
        if keys[pygame.K_DELETE] and (box_j, box_i) not in self.locked_pos:
            # remove the value
            self.matrix[(box_j, box_i)] = 0

        # if the current matrix is matches solution
        if np.array_equal(self.matrix, self.solution):
            # draw the keys as green
            self.draw_window(solved=True)

            # status variable
            hold_screen = True
            # while not action performed stay on screen
            while hold_screen:
                # iterate through events
                for event in pygame.event.get():
                    # if event is of type key press or button click
                    if event.type in (pygame.KEYDOWN, pygame.QUIT,
pygame.MOUSEBUTTONDOWN):
                        # break outer while loop
                        hold_screen = False

            # fill backgroud with gray
            self.window.fill((100, 100, 100))
            # font for Heading
            font = pygame.font.SysFont('comicsans', 48)
            # label for heading
            label = font.render('Congratulations !!!', 1, (0, 200, 0))
            # display heading
            self.window.blit(label,
                                 ((SCREEN_WIDTH - label.get_width()) / 2,
                                  (SCREEN_HEIGHT - label.get_height()) / 2))
            # update the pygame display screen
            pygame.display.update()
            # save and exit
            self.graceful_exit()

        # True means game not over
        return True
```

Figure 48: *Play Game method*

This function processes input and checks the solution's correctness. We check events for mouse clicks, close actions, numerical input, key deletion, or arrow input. For numerical input, we display the value in the selected box as blue or red depending on whether the input is possible. If it was a delete key, we would remove the current entry and reset it to zero if the box was not locked. If the key is an arrow key, we move the box in the opposite way.

10.2.2 Game Window Class

```
class SudokuGUI:
    """Template Class for the GUI in the game"""

    # type hints for the class variables
    BOX_ROWS: int
    BOX_COLS: int
    NUM_ROWS: int
    NUM_COLUMNS: int
    PLAY_WIDTH: int
    PLAY_HEIGHT: int
    TOP_LEFT: Tuple[int, int]
    matrix: np.ndarray
    init_matrix: np.ndarray
    solution_list: List[np.ndarray]
    solution: np.ndarray
    window: pygame.Surface
    selected_box: Tuple[int, int]
    locked_pos: List[Tuple[int, int]]
    home_icon: pygame.Surface
    button_home: Button
    button_load_image: Button
    button_cam_image: Button
    button_solve: Button
    button_play_game: Button
    button_AR: Button

    def __init__(self, matrix: np.ndarray, box_rows: int = 3, box_cols: int = 3):
        """default initialization"""
        # ========================== GUI Parameters ============================
        self.BOX_ROWS = box_rows
        self.BOX_COLS = box_cols
        self.NUM_ROWS = self.BOX_ROWS * self.BOX_COLS
        self.NUM_COLUMNS = self.BOX_ROWS * self.BOX_COLS
        self.PLAY_WIDTH = BLOCK_SIZE * self.NUM_COLUMNS
        self.PLAY_HEIGHT = BLOCK_SIZE * self.NUM_ROWS
        self.TOP_LEFT = (int((SCREEN_WIDTH - self.PLAY_WIDTH) / 2),
                         int((SCREEN_HEIGHT - self.PLAY_HEIGHT) / 2 - 80))
        self.matrix = matrix
        self.init_matrix = self.matrix.copy()
        try:
            # find the solutions for the given matrix
            self.solution_list = Sudoku(matrix.copy(), box_row=self.BOX_ROWS,
box_col=self.BOX_COLS).get_solution()
            # take the first solution
            self.solution = self.solution_list[0]
        except Exception:
            # incase no solution show error
            tkinter.messagebox.showerror(title="Error",
                                         message="Solution does not exist or Image
not clear, Try Again.")
        self.window = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
        self.selected_box = (0, 0)
        self.locked_pos = self.get_locked_pos()
        self.home_icon = pygame.image.load('.images/home_icon.png')
        self.button_home = Button(
            60, 60, 70, 70, (200, 200, 200), '  ')
        self.button_load_image = Button(
            162, 510, 250, 60, (200, 200, 200), "Load from File")
        self.button_solve = Button(325, 590, 250, 60, (200, 200, 200), "Solve")
        self.button_play_game = Button(
            325, 300, 400, 80, (200, 200, 200), "Play Game")
        self.button_AR = Button(
            325, 450, 400, 80, (200, 200, 200), "Augmented Reality")
```

Figure 49: *Game Window class*

The GUI class defines the project's primary components. We begin by initialising the class using the variables:

- **Box Rows:** The number of rows in a puzzle's sub box.
- **Box Columns:** The number of columns in a puzzle' s sub box.
- **Num of rows:** The total number of rows and columns in the puzzle.
- **Width of the Play Area:** The total pixel width of the game area.
- **Height of the Play Area:** The total pixel height of the game area.
- **Top Left Area Coordinates:** The top left coordinates of where the puzzle is placed in the window.
- **Sudoku Puzzle Matrix:** This is a sudoku puzzle matrix.
- **Initial Copy of Puzzle Matrix:** A backup copy of the initial matrix.
- **Solution List:** List of solutions for the current puzzle.
- **Selected Solution:** Solution that is selected to be displayed on clicking the solve button.
- **Game Window:** This is game window screen object.
- **Selected Box:** This is the presently highlighted box where we input the numerical value.
- **Locked Boxes:** List of non-zero positions of the puzzle which cannot be modified.
- **Home Icon:** Icon for the home button to navigate to the main menu.
- **Home Button:** Button to navigate to the main menu.
- **Load Image Button:** Button to load an image from a file.
- **Load from Camera Button:** Button to click an image from the camera.
- **Solve Puzzle Button:** Button used to reveal the puzzle solution.
- **Play Game Button:** Button in the main menu used to play the game.
- **Augmented Reality Button:** Button in the main menu used to load the Augmented Reality.

10.2.3 Locked Box

```python
def get_locked_pos(self):
        """Get list of coordinates of the clues in the given puzzle"""

        # initialize empty list
        locked_pos = []
        # iterate through rows
        for i in range(self.NUM_ROWS):
            # iterate through columns
            for j in range(self.NUM_COLUMNS):
                # if clue i.e. non zero
                if self.matrix[i, j] != 0:
                    # then add to locked_pos
                    locked_pos.append((i, j))
        # return the list
        return locked_pos
```

Figure 50: *Get Locked Box positions*

This method returns the non-zero positions from the puzzle matrix.

10.2.4 Test results

The tests for this sprint can be found in **Table 4**. The results can be found in **Table 7** in the Appendix chapter.

10.3. Sprint 3 – Augmented Reality Option

This chapter covers the implementation of the ***AR*** option within the Solver.

10.3.1 AR Option

```
def load_AR(self):
        """This method handles the augmented reality solver"""

        # open the camera feed window
        with CameraWindow() as feed:
            # initialize frame count
            i = 0
            # initialize solution
            solution = None
            # infinite loop
            while True:
                # supress all kinds of exception
                with suppress(Exception):
                    # get frame from the camera
                    frame = feed.get_frame()
                    # every 10th frame
                    if i == 0:
                        # load the image for image processing
                        sip = SudokuImageProcessing(image=frame)
                        # detect puzzle matrix
                        detected_matrix = sip.get_matrix()
                        # if matrix detected
                        if detected_matrix is not None:
                            # get dimensions of the puzzle
                            game_size, (box_rows, box_cols) = sip.get_dimensions()
                            # minimum number of clues required to get a unqiue
solution
                            min_clues_required = {4: 4, 6: 8, 8: 14, 9: 17}
                            # check for the number of clues given
                            if np.count_nonzero(detected_matrix) >=
min_clues_required[game_size]:
                                # get list of solutions
                                solution_list = Sudoku(detected_matrix.copy(),
                                                       box_row=box_rows,

box_col=box_cols).get_solution()
                                # if atleast one solution is available
                                if len(solution_list) != 0:
                                    # store the solution
                                    solution = solution_list[0]
                    # for every frame, if a solution is found
                    if solution is not None:
                        # plot the solution on the frame
                        frame = sip.plot_on_image(frame, detected_matrix, solution,
 (box_rows, box_cols))
                # if home button is pressed
                if feed.draw_window(frame) is False:
                    # return to main menu
                    return False
                # increment frame count
                i = (i + 1) % 10
```

Figure 51: *AR class implementation*

This code opens the Camera Window and sets the solution and frame count to None and 0, respectively. Next, it uses the "suppress(Exception)" context to prevent any exceptions from occurring. The picture is then processed every ten frames to obtain the matrix, its dimensions, and the solution. The answer is then plotted across the picture for ten frames.

10.3.2 Graceful exit

```
def graceful_exit(self):
        """Helper method, it saves the last loaded puzzle and its dimensions before
quitting"""
        # save the current puzzle loaded
        np.save('last_loaded.npy', self.init_matrix)
        # save the dimensions of the current puzzle
        np.save('last_loaded_dim.npy', np.array([self.BOX_ROWS, self.BOX_COLS]))
        # exit pygame runtime
        pygame.quit()
        # exit program
        quit()
```

Figure 52: *Exit from the application and save the last loaded puzzle*

This function ensures that the most recently played matrix and its dimensions are stored before exiting the GUI.

10.3.3 Test results

The tests for this sprint can be found in **Table 5**. The results can be found in **Table 8** in the Appendix chapter.

10.4. Solving Steps

The matrix in **Figure 14** represents an exact cover problem. We assume we have a root column object *h* that describes this matrix – *search*(*h*,*0*,*s*), DLX being invoked; because h ≠ R[h], column *A* is chosen as the next column object to be covered.

Figure 14 shows how to remove the first two rows of A. Because these two column objects have data objects on this row, this will affect the data objects in D and G. **Figure 14** also shows how the new links are formed.

We know from DLX in **Figure 15** that *r* points to the first data object in A. As shown in the loop on row 11, the next columns to be covered are D and G. The result is shown in **Figure 41**, where the thick lines represent the new links.

The branch then employs *search(h, 1, s)* to look for all column objects that were covered in the first row of A. This will cover column B but leave no 1s in column E, leaving *search(h, 2, s)* without a solution. This returns the algorithm to **Figure 54**, and it moves on to the first row in column object A.

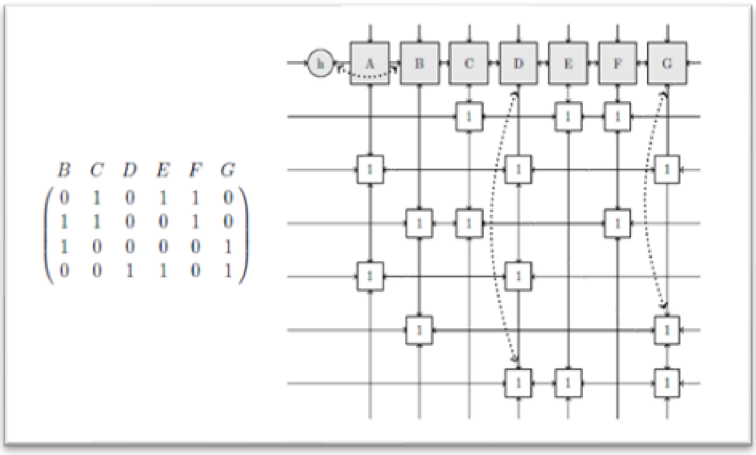

$$\begin{array}{cccccc} B & C & D & E & F & G \\ \left(\begin{matrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{matrix}\right) \end{array}$$

Figure 53: *Column A from **Figure 14** has been covered*

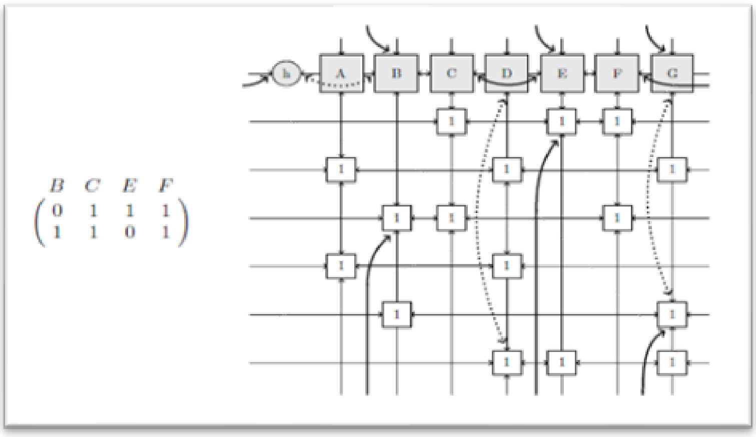

$$\begin{array}{cccc} B & C & E & F \\ \left(\begin{matrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{matrix}\right) \end{array}$$

Figure 54: *Columns D and G in **Figure 26** have been covered (rows 1 and 3)*

Eventually, a solution is found, which can vary in various ways. Given the solution s, Knuth prints the rows containing s_0 , s_1 , … , s_{k-1} where each s_k for $k \in \{0, 1, 2, \dots, n-1\}$ is printed using the algorithm represented in **Figure 43**.

10.5. Difficulty Rating

During my research, I found that there are a variety of ways to rate the difficulty of puzzles. Some measures suggest that puzzles with fewer clues are harder, while others suggest that puzzles with more clues are harder.

However, the ratings are discussed in various Sudoku forums, some people believe that the difficulty of Sudoku doesn't depend on the number of clues, but rather on the number of different techniques and decisions required, as well as the difficulty of these techniques.

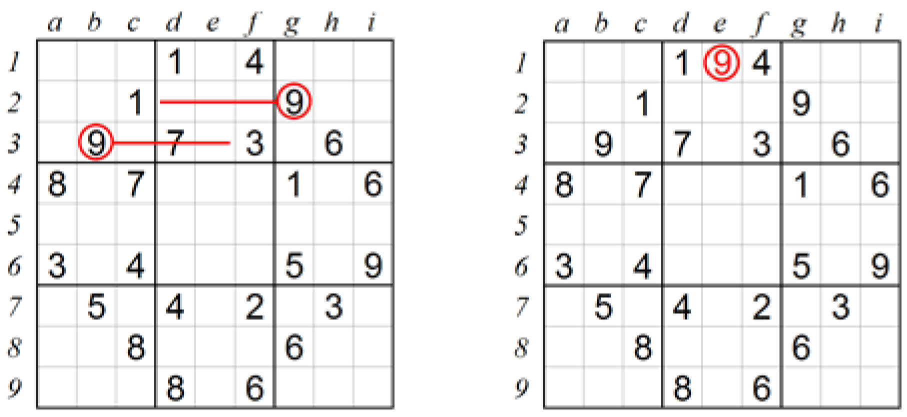


Figure 55: *Scanning in one direction*

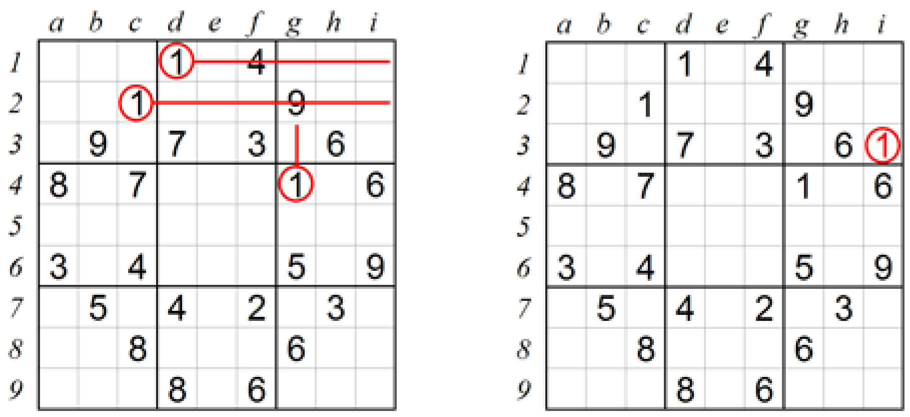


Figure 56: *Scanning in two directions*

11. Project Evaluation

This review will provide a comprehensive overview of the project's challenges and solutions that were implemented. These changes may be helpful in improving the product.

11.1 Research

The research found that there are many ways to implement a new system, but that some areas were not explored as deeply as they could have been. Additionally, the research failed to look at some potential solutions because of bias towards a particular programming language – Python. If the search criteria had been widened, better solutions might have been found.

11.2 Requirements

The main benefit of using "MoSCoW" methodology for prioritizing requirements was that it made it easier to plan the development of features, as it allowed critical requirements to be identified before development began. Additionally, no features were added that were not necessary, resulting in a product that is fully playable. However, there are some problems with this methodology, most notably that some "should have" requirements were not completed because of time constraints.

11.3 Methodology

The approach chosen was ***Agile Scrum***, as stated in the methodology section. Although the approach is typically applied in groups, it was observed that it worked very well with individual initiatives as well during this project. Setting time boundaries for sprints with specific work to perform within them made the vast amount of work that needed to be done simpler to handle and manage within those sprints.

11.4 Design

In the case of this project, time management was critical, hence the design section was critical since not being prepared in this area while implementing may have resulted in time being wasted due to poor planning. As a result of the schematics, the implementation went effectively, making this segment a success. The flow charts in this section, in particular, proved extremely valuable in making some of the more complex algorithms in the project considerably faster to implement.

11.5 Implementation and Testing

During the game's development, feedback from those who played it was obtained. One of the most significant challenges that users encountered was that if they were absolutely new to AR, it was quite impossible for them to play the game without assistance from someone with more knowledge.

One of the significant findings when users were allowed to test the game was that they were not as familiar with the game as the developers were. For example, users might not know what they need to do in order to win the game, while developers know exactly what they need to do.

Although this may not appear to be a problem at first, it led to the discovery that there are bugs and vulnerabilities in the game that they may have missed since when we test, we are checking to see if a certain feature works.

The testing strategy was flawed because it was only tested against project requirements, and not against the game itself. A better strategy would have been to have external testers test the game after each sprint, to catch any unexpected bugs. This would have made the game better-polished.

12. Further Work and Conclusions

12.1 Further work

Although due to time constraints, just "***MoSCoW***" – "***Musts***" were implemented due to time constraints, other features could be included with more time. For example, if development continues on this project, new puzzles and a timer should be added to the game. Other functions, such as a website login system that allows users to update their existing results. In addition, creating a version of the game that runs on different operating systems – Android/iOs is another option that could be considered.

12.2 Conclusions

This report describes the successful completion of a project that aimed to create an **AI Sudoku Solver** game with **Algorithm X** and **Augmented Reality**. All objectives stated at the outset of the project were met. Extending this would be simple in the future because the game is designed in such a manner that adding additional problems would be simple.

This paper provides an in-depth look at the emerging technology of augmented reality and presents one method in which it may be applied. Other potential applications for augmented reality were identified throughout the report's research. Future work on this topic would allow for a better understanding of augmented reality's capabilities and limitations.

13. Glossary

Term	Definition
Agile Development	The ability to adapt to changing circumstances is a key skill for success in an unpredictable environment.
Android	A mobile operating system designed specifically for touchscreen devices that is based on a modified version of the Linux kernel and other open-source software.
Backtracking Technique	An algorithmic technique where the goal is to get all solutions to a problem using the brute force approach.
Boolean variable	A data type that has one of two possible values (usually denoted ***true***/***false***) which is intended to represent the two truth values of logic.
Brute Force	A method of computation in which the computer is made to try all permutations of a problem until one is found that provides a solution, in contrast to the implementation of more intelligent algorithms.
Buddy Cell	Any of the 20 cells that are connected to a single cell through a set.
Candidate	A possible number that could go into a particular cell.
Cell	One of the 81 squares that make up the Sudoku grid.
Circular Doubly Linked List	A more complexed type of data structure in which a node contain pointers to its previous node as well as the next node.
Clone	The number of times a single number appears within the puzzle at that time. When a puzzle is complete, each number will have 9 clones.
Column	A collection of 9 cells running vertically down the grid.
Complexity	The amount of resources required to run a program.
Cover Problem	Computational problems that ask whether a certain combinatorial structure 'covers' another, or how large the structure has to be to do that.
Cross Hatching	Drawing a line through a row and column containing a particular number.
Dancing Links	A technique for adding/deleting a node from a circular doubly linked list.
Final Value	When a cell is left with only one candidate, this candidate is the cell's value.
Given Number	A pre-assigned number on the Sudoku grid.
Grid	A 9×9 square of 9 cells that makes up the playing area of a Sudoku puzzle.
Likely Number	A number which has a higher probability of being placed in a cell.
Number	Any integer between 1 – 9 that can appear withing the puzzle.
Row	A group of 9 cells running horizontally across the grid.
Set	A collection of 9 cells in the form of a column, row or sub-grid.
Sub-grid	A collection of cells forming a smaller 3×3 grid.

14. Table of Abbreviations

Abbreviation	Explanation
AI	Artificial Intelligence
AR	Augmented Reality
AT	Acceptance Test
DLX	Dancing Links for Algorithm X
GUI	Graphical User Interface
ID	Identifier
iOs	iPhone Operating System
MoSCoW	Four categories of initiatives: **M**ust-have, **S**hould-have, **C**ould-have, **W**on't-have/**W**ill not have right now
Ms	Milliseconds
PNG	Portable Network Graphics
ROI	Region of Interest
SSA	Sudoku Solver Application
UI	User Interface

15. References / Bibliography

- Bäck T., Fogel, D. B. (2000) *Handbook of Evolutionary Computation Routledge &amp; CRC Press*. Available at: https://www.routledge.com/Handbook-of-Evolutionary-Computation/Baeck-Fogel-Michalewicz/p/book/9780750308953 [Accessed: 27 February 2023].

- Dechter, R. (2013) *Constraint processing*, *ScienceDirect* [online]. Available from: https://www.sciencedirect.com/book/9781558608900/constraint-processing [Accessed: 27 February 2023]

- Ercsey-Ravasz, M., Toroczkai, Z. *The Chaos Within Sudoku* [online]. Scientific Report 2, 725 (2012). Available from: https://doi.org/10.1038/srep00725 [Accessed: 17 November 2022]

- Hanson, R.M. (2022) *Introduction to exact cover problem and Algorithm X, Exact Cover Matrix*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/introduction-to-exact-cover-problem-and-algorithm-x/ [Accessed: 30 November 2022]

- Johnson, A. (2005) *Solving Sudoku*, *Solving sudoku*. Available at: http://www.angusj.com/sudoku/hints.php [Accessed: 2 December 2022]

- Johnson, A. (2005) *A Simple Sudoku*, *Simple sudoku - freeware puzzle maker and solver*. Available at: http://angusj.com/sudoku/ [Accessed: 6 December 2022]

- Knuth, D.E. (2000) *Dancing links*, *arXiv.org*. Donald Knuth. Available from: https://arxiv.org/abs/cs/0011047 [Accessed: 1 November 2022].

- Knuth, D.E. (2000) *Dancing links. Millenial Perspectives in Computer Science* [online]. 1 (1) *pages 187-214* [Accessed: 2 November 2022].

- Kapanowski, A. (2010) *Python for education: The exact cover problem*, *https://arxiv.org/abs/1010.5890*. Available from: https://arxiv.org/abs/1010.5890 [Accessed: 7 October 2022].

- Lewis, R. (2007) *Metaheuristics Can Solve Sudoku Puzzles Journal of Heuristics* [online]. 13 (4) *pages 387-401* [Accessed: 20 November 2022].

- Lewis, R. (2022) *Guide to graph colouring: Algorithms and applications*. 2nd edn. S.l., US: SPRINGER NATURE (1).

- Norvig, P. (2016) *Solving Every Sudoku Puzzle*, *Sudoku puzzles*. Available at: https://norvig.com/sudoku.html [Accessed: 18 November 2022].

- Perez, M., Marwala, T. (2008*) Stochastic Optimization Approaches for Solving Sudoku* [online]. 1 (1) *pages 1-13* [Accessed: 28 November 2022].

- Person, T, D.B. and Baeck, F. (2018) Evolutionary computation 1: Basic algorithms and operators: Thomas B, Taylor &amp; Francis. Taylor &amp; Francis. Available at: https://www.taylorfrancis.com/books/edit/10.1201/9781482268713/evolutionary-computation-1-thomas-baeck-fogel-michalewicz (Accessed: February 27, 2023).

- Thomas, N.A. (2006) *Advanced sudoku and kakuro*. 1st edn. Blacklick, OH, London: McGraw-Hill Companies, Inc. (1).

- Werf, R. (2007) *Sudoku Solving Guide*, *SudoCue*. Available at: http://www.sudocue.net/guide.php [Accessed: 17 December 2022]

- Werf, R. (2007) *Sudo Cue Downloads*, *SudoCue Sudoku*. Available at: http://www.sudocue.net/download.php [Accessed: 4 January 2023]

16. Appendix A

16.1 Sprint one tests

Table 3

Test ID	Test Type	Test Description	Test Steps	Expected result	Requirements Tested
T-01	Widget Test	Displaying the main interface.	Run the application in VS Code.	The main menu is displayed on the screen.	All1, SSA1
T-02	Widget Test	Change the colour of a button.	Hoover over a button and check if its colour changed.	The colour changes to dark gray.	SSA1, SSA2
T-03	Widget Test	Click on a button.	Hoover over a button and click on it.	Colour of the button changes and the user is redirected to next page.	SSA1
T-04	Widget Test	Click on “Load from File” button	Hoover over the button and click on it. Check if the upload option is displayed.	The upload option is displayed and the user is able to select a file.	SSA2
T-05	Widget Test	Upload a valid puzzle	Repeat steps at T-04 and upload a valid puzzle.	The puzzle is validated and displayed in the grid.	SSA2, SSA3
T-06	Widget Test	Upload an invalid puzzle	Repeat steps at T-04 and upload an invalid puzzle.	The puzzle is rejected and an error message is displayed.	SSA2, SSA3
T-07	Widget Test	Functional home button	Click on “Play Game” option and then click on the home button.	The user is redirected to the main menu.	SSA3

16.2 Sprint two tests

Table 4

Test ID	Test Type	Test Description	Test Steps	Expected result	Requirements Tested
T-08	Unit Test	Load a jpg/png/jpeg/jpe/bmp file	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a valid Sudoku in .jpg/.png/.jpeg/.jpe/.bmp format	The photo is validated and the grid is displayed on the GUI.	SSA2, SSA3, SSA4
T-09	Unit Test	Load a file with a different extension than the one listed in T-08	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a file with a different extension than the one listed in T-08	The file is rejected and an error message is displayed.	SSA2, SSA3, SSA4
T-10	Unit Test	Load a 4×4 valid grid.	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a 4×4 valid grid.	The file is validated and the grid is displayed on the GUI.	SSA3, SSA4, SSA5, SSA6
T-11	Unit Test	Load a 6×6 valid grid.	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a 6×6 valid grid.	The file is validated and the grid is displayed on the GUI.	SSA3, SSA4, SSA5, SSA6
T-12	Unit Test	Load a 8×8 valid grid.	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a 8×8 valid grid.	The file is validated and the grid is displayed on the GUI.	SSA3, SSA4, SSA5, SSA6
T-13	Unit Test	Load a 9×9 valid grid.	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a 9×9 valid grid.	The file is validated and the grid is displayed on the GUI.	SSA3, SSA4, SSA5, SSA6

T-14	Unit Test	Load and solve a puzzle with a unique solution. Display the solution.	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a sudoku with a unique solution. After the grid is uploaded, click on “Solve”.	The file is validated and the grid is displayed on the GUI. After the “Solve” button is clicked, the solution is displayed.	SSA6, SSA7, SSA8
T-15	Unit Test	Load and solve a puzzle with no solution. Display error	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a sudoku with a unique solution. After the grid is uploaded, click on “Solve”.	The file is validated and the grid is displayed on the GUI. After the “Solve” button is clicked, the error-message “No solution” is displayed.	SSA6, SSA7, SSA8
T-16	Unit Test	Load and solve a puzzle with multiple solutions. Display one of the solutions.	Run the application and click “Play Game Option”. Click on “Upload from File” and upload a sudoku with a unique solution. After the grid is uploaded, click on “Solve”.	The file is validated and the grid is displayed on the GUI. After the “Solve” button is clicked, one of the solutions is displayed.	SSA6, SSA7, SSA8
T-17	Unit Test	Load and manually solve a puzzle.	Run the application and click “Play Game Option”. Click on “Upload from File” and	The user is able to insert digits	SSA6, SSA8, SSA9

			upload a sudoku. Manually solve the puzzle.	from [1,9] range in the cells.	
T-18	Unit Test	Change the colour of the digits, depending of the constraints – red for mistake.	Run the application and click "Play Game Option". Click on "Upload from File" and upload a sudoku. Manually solve the puzzle. In an empty cell, insert a digit which does not respect the Sudoku's constraints.	The user is able to view the colour of the input in red.	SSA6, SSA8, SSA9
T-19	Unit Test	Change the colour of the digits, depending of the constraints – light blue for potential candidate – cell.	Run the application and click "Play Game Option". Click on "Upload from File" and upload a sudoku. Manually solve the puzzle. In an empty cell, insert a digit which does respect the Sudoku's constraints.	The user is able to view the colour of the input in light blue.	SSA6, SSA8, SSA9
T-20	Unit Test	Acceptable performance speed for loading and validating puzzles.	Run the application and click "Play Game Option". Click on "Upload from File" and upload a sudoku. After the grid is uploaded, click on "Solve".	The time required for loading and validating puzzles is of the order of ms.	SSA6
T-21	Unit Test	Acceptable performance speed for validating and solving puzzles.	Run the application and click "Play Game Option". Click on "Upload from File" and upload a sudoku. After the grid is uploaded, click on "Solve".	The time required for validating and solving puzzles is of the order of ms.	SSA6

16.3 Sprint three tests

Table 5:

Test ID	**Test Type**	**Test Description**	**Test Steps**	**Expected result**	**Requirements Tested**

T-22	Unit Test	Open the AR option	Run the application and click "Augmented Reality".	The window camera is opened as expected	SSA6
T-23	Unit Test	Upload a valid grid using AR	Run the application and click "Augmented Reality". Present in front of the camera a valid grid	The application is able to read the data from the grid.	SSA7, SSA8
T-24	Unit Test	Solve a grid using AR	Run the application and click "Augmented Reality". Present in front of the camera a valid grid	The application is displaying in the camera window the solution.	SSA7, SSA8
T-25	Unit Test	Upload an invalid grid in AR	Run the application and click "Augmented Reality". Present in front of the camera an invalid grid	The application reads the grid and displays "Grid does not have a solution" message	SSA8
T-26	Unit Test	Upload a grid in AR in which the image is not clear/too far from the camera	Run the application and click "Augmented Reality". Present in front of the camera an valid/invalid grid	The application is displaying "Image is too far from the camera" message	SSA9
T-27	Unit Test	Exit the AR option	Run the application and click "Augmented Reality". After the camera window is displayed, click "Q"	The user is redirected back to the main menu.	SSA8

16.4 Sprint one test results

Table 6:

Test ID	Result	Comments
T-01	**Pass**	Tests passed as expected.

T-02	**Pass**	Tests passed as expected.
T-03	**Pass**	Tests passed as expected.
T-04	**Pass**	Tests passed as expected.
T-05	**Pass**	Tests passed as expected.
T-06	**Pass**	Tests passed as expected.
T-07	**Pass**	Tests passed as expected.

16.5 Sprint two test results

Table 7:

Test ID	**Result**	**Comments**
T-08	**Pass**	Tests passed as expected.
T-09	**Pass**	Tests passed as expected.
T-10	**Pass**	Tests passed as expected.
T-11	**Pass**	Tests passed as expected.
T-12	**Pass**	Tests passed as expected.
T-13	**Pass**	Tests passed as expected.
T-14	**Pass**	Tests passed as expected.
T-15	**Pass**	Tests passed as expected.
T-16	**Pass**	Tests passed as expected.
T-17	**Pass**	Tests passed as expected.
T-18	**Pass**	Tests passed as expected.
T-19	**Pass**	Tests passed as expected.
T-20	**Pass**	Tests passed as expected.
T-21	**Pass**	Tests passed as expected.

16.6 Sprint three tests results

Table 8:

Test ID	**Result**	**Comments**
T-22	**Pass**	Tests passed as expected.
T-23	**Pass**	Tests passed as expected.
T-24	**Pass**	Tests passed as expected.
T-25	**Pass**	Tests passed as expected.
T-26	**Pass**	Tests passed as expected.
T-27	**Pass**	Tests passed as expected.

17. Appendix B

GitHub Link: https://github.com/andrei2timo/Dissertation---AI-Sudoku-Solver-using-Algorithm-X-and-Augmented-Reality