Final Year Project Report

Full Unit - Final Report

Playing Games and Solving Puzzles using AI

Dimitar Seraffimov

A report submitted in part fulfilment of the degree of

BSc in Computer Science with Software Engineering

Supervisor: Magnus Wahlstrom

Department of Computer Science
Royal Holloway, University of London

April 12, 2024

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been properly acknowledged.

Word Count: 14,226

Student Name: Dimitar Seraffimov

Date of Submission: April 12, 2024

Signature: Dimitar Seraffimov

Table of Contents

Abstract

Sudoku is a logic-based, combinatorial number-placement game that became popular in the last decade. The Japanese puzzle became so sensational and influential that it managed to quickly spread around the world and capture the attention of millions. The simple structure and clear rules made it a burning passion for millions of people at some point in their life.

The game does not require strong mathematical skills, but it is essential to have solid problem-solving and logical mindset to solve difficult puzzles in a timely manner. Due to the puzzle's popularity, different creative variants emerged quickly, changing the size of the grid, or adding additional rules are typical examples of the new game features. As interesting they can get, the new modifications still did not manage to provide sufficient excitement in order to completely replace the original NxN puzzle grid game.

This project implements and researches a backtracking algorithm, specifically fine-tuned to satisfy the Sudoku game's constraints. To create an additional level of complexity, I have chosen to extend the algorithm to a Sudoku mobile application game written in React Native (JavaScript based mobile app framework).

Chapter 1: **Introduction**

There are many different 'human' tactics and software approaches to solving the Sudoku problem, but this project focuses on researching, developing, and implementing an algorithmic solution to the original Sudoku puzzle constraints. Constraint Satisfaction [Apt03] is a leading approach for solving search problems and in this project, it is researched and implemented following the rules of the Sudoku game. The fundamental technique chosen to successfully solve the problem is to incorporate a backtracking algorithm with strategic backjumping.

Following the completed Proof Of Concept programs, it is obvious to me that thanks to its time complexity, the chosen methodology is a solid foundation for future improvements of the application and with some additional fine-tuning it can achieve a good final application. The planned extension of the project to android mobile application [Eis15] is an ideal stepping stone and gives an enormous variety of possible functionalities to be researched and developed. Detailed documentation and explanation of the mentioned functionalities and features can be found in the Aims and Objectives chapter.

Throughout the development process, I had to dedicate significant time to expand my knowledge in React Native, JavaScript, and Expo mobile development framework. These skills were necessary in order to achieve the planned extension to mobile application, which currently operates on both IOS and Android platforms. This said, my focus during the second academic term was developing a user-friendly application that proves software development concepts and provides professional and minimalistic look of the application.

1.1 Sudoku Puzzle Constraints

A Sudoku puzzle can be characterised as a partly finished N×N grid. A general Sudoku puzzle has 9x9 rows and columns, and 3x3 minigrids which account to nine cells each. The full grid has 81 cells in total and 9 separate minigrids.

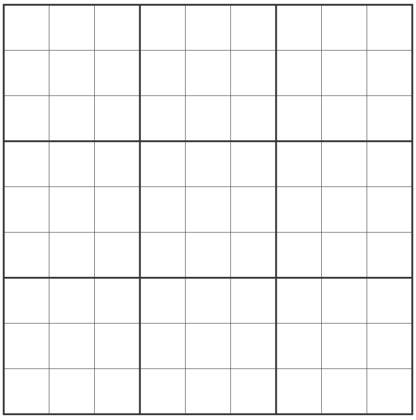

Figure 1.1: An empty Sudoku Grid

The simplest way to describe the game's constraints is that each digit shows up once in every unit (row, column, minigrid). The mathematical approach of the Sudoku configuration can be summed up to an N×N rows, columns, and grid divided into N-areas or minigrids, where each of the N-rows, columns, and minigrids have N-cells and each of the N-digits occurs precisely once in every row, column, and minigrid.

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

Figure 1.2: Partially completed Sudoku Grid

Despite the simple rules of the game, there exists no generalised method to determine the minimal number of pre-filled cells needed at the start of the game to guarantee a unique solution. This is the reason for the vast number (6.67×10^{21}) of potential distinct Sudoku puzzle configurations. To ensure a single, unique solution, a Sudoku grid needs minimum 17 clues. This implies that a puzzle with less than 17 clues will inevitably have two or more solutions. However, it is not excluded for a valid Sudoku grid with 17 or more clues to have multiple solutions.

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

Figure 1.3: Sudoku puzzle with only 17 clues

1.2 Problem Statement

The primary challenge in developing a Sudoku solving game, lies in the integration of different additional functionalities with the end goal to create a user-friendly and efficient application. The direction of this project is to create a beautiful and seamless user experience, with the focus not only on solving and generating new puzzles with unique solutions, but also exploring other features in order to enhance the player's involvement.

Writing the final report, I've reflected on the challenges faced during the second term of the year, specifically implementing key features that enhance user engagement and interactions with the application. Among these challenges were the generation and categorisation of different difficulty levels with the goal to attract players with various experience, the integration of a validation system, step back button (which involves storing some history of the previous moves made) and time tracking. A significant problem during development, involved researching and correctly implementing local history and game state management, responsible for resuming a previous game, tracking user inputs and archiving the solved game.

In the beginning of the year, I have identified these features as crucial to achieve the desired functionality and minimalistic user experience. Some of them presented considerable challenges in terms of visual and logical implementation within the application. I had to research asynchronous storage management (ASYNC) in React Native [Com24] and understand how to rewrite some of the functions from Python to JavaScript, in order to keep the implementation logic concise and responsive. I have carefully explained each one of the implemented features in the *Aims and Objective* section of the report.

1.2.1 Algorithmic Complexity and Integration

The developed Proof of Concept (PoC) programs research and implement a constraint satisfactory [BPS99] backtracking algorithm, which solves Sudoku puzzles and prints in the command-line interface (CLI) the starting grid and below - the solved one. The backtracking algorithm inside the Proof of Concept - Program 3 [Civ13] is fine-tuned for efficiency and accuracy, ensuring that in a timely manner it can handle a wide range of puzzle complexity. For each PoC program, the starting state of the Sudoku grids are defined as 2D lists (not being generated by a specific program), containing the specific number arrangement and empty cells (denoted by 0).

One of the challenges I faced was developing an algorithm dedicated to creating puzzles with unique solution, that not only follow the standard Sudoku rules but also categorises them into different difficulty levels. To generate a Sudoku puzzle, the criteria for determining the level of difficulty needs to be considered. The first factor affecting the criteria is the number of empty cells in the initial puzzle - it is generally safe to assume that the more empty cells there are, the higher the level of difficulty. It is important not to forget that to ensure a unique solution, the grid needs a minimum of 17 clues (Sudoku Puzzle Constraints), if there are less than 17 starting clues - the puzzle has multiple solutions. The second influential factor in determining the puzzle's difficulty is the placement of the initial numbers on the grid. From observation, simpler puzzles usually have their initial number of clues evenly spaced apart, compared to the more difficult ones which have their clues clustered in small groups.

In the next sections, I describe in detail my implementation approach, specifically for the important 'sudokuGenerator.js' file.

From a developer's standpoint, to correctly evaluate the difficulty level of a Sudoku puzzle, the computational effort and tools used by the program to solve the puzzle need to be analysed.

This is an aspect on which if I decide to focus, it will shift the *Aims and Objective* of the whole project. The reason behind this, is that I will have to compare different solving techniques and some of them do not align with my primary ambition and goal of this project. Which simply put, is to research and implement a more advanced backtracking algorithm to solve and generate different Sudoku puzzles, and create a minimalistic, yet fully functional mobile application.

The techniques discussed and developed in the book *Programming Sudoku* [Lee06] primarily fall under the category of pattern-based solving strategies and brute-force elimination methods. The listed approaches are Column, Row, and Minigrid Elimination (CRME), Lone Rangers, Twins / Triplets in minigrids/columns/rows, and Brute-force elimination which are undeniably valuable for a human approach to solving Sudoku puzzles, as they help the player discover patterns and strategically eliminate possibilities. However, the project does not focus on comparing human-like methodologies and would inevitably divert my resources and time away from exploring more sophisticated algorithmic solutions. The approach I will implement is discussed in the Generating Sudoku Puzzles future Proof of Concept programs in combination with additional backtracking proof of the puzzle uniqueness.

In my Interim Report, I had discussed an intriguing, but challenging extension of the project - implementing 'Machine Vision' that allows the user to scan Sudoku puzzles using the phone's camera and have them solved by the application. The described feature is really advanced and requires in-depth research and the development of new skills, particularly in image processing and pattern recognition. Furthermore, the described extension demands substantial research and understanding in image processing and pattern recognition technologies, which are distinct from the conventional logic-based Sudoku solving methods. Even though I found this possible extension interesting, I started researching it too late, and it was impossible for me to properly implement it.

1.2.2 User Interface and Interaction Design

When discussing the User Interface and Interaction Design [RA03], the main challenge lies in crafting an interface that is not only visually appealing but also intuitive and efficient for the user. My focus is on mitigating and tackling the risk of focusing too much effort and time on the front end of the application. The methodology to achieve this, consists of minimalist design, single colour scheme and easy to navigate single-page main menu. This approach is used with the intent to streamline the user interaction with the system, reducing unnecessary complexity and enhancing the overall user experience.

In the initial phase of the project, I followed the User Stories and User Diagram files - these can be found in the code base inside the folder /Blueprints-UI. There, in concise matter, all user interactions of the mobile application are described. I have chosen React Native as the UI code framework on which to develop the front end of the application. The decision is backed with the 'native' state of the produced code, which allows the mobile game to work as a native application on both Android and IOS devices. I dedicated around three weeks to complete a beginner-friendly course on Udemy in React Native, explaining the basics needed to create a functional and professional mobile application [Sch24].

To ensure all essential application requirements are covered, I systematically followed User Centred Design (UCD) methodologies. This organised approach benefited me in gaining a better understanding and identifying the expectations and needs of the end-user. The techniques utilised to guide me in the development process are thoroughly discussed in the User Centered Design subsection of the report.

Chapter 2: **Aims and Objectives**

When describing the fundamental features and implemented extensions of the Aims and Objectives for this project, important to me is achieving a harmonic combination between seamless user experience and professionally implemented, and documented backend software[FH+01]. This chapter of the report and its sections are dedicated to present all backend and frontend functionalities of the final application. In my 'Interim Report', I have introduced an additional extension, which I managed to research but not to implement. As I have made some research on the topic, I will give my reasons on why I have not implemented it in the section Extension: Machine Vision.

My idea for the application follows minimalist, yet effective software development principles and processes. The process of starting a new Sudoku game will present the user with an option to choose from different difficulty levels. When started, a dedicated program will generate a Sudoku puzzle with single unique solution and will present the game screen with different in-game functionalities. On leaving this screen, the program will automatically save the current status of the game and the user will be able to continue from the last state of the previous not finished attempt. Expanding further on this, the application will keep local track of the completed previous attempts, giving insight to the date of completion, previous difficulty level and completion time.

When describing the in-game user experience, the implementation of five important functionalities is essential – grid correctness validation, complete puzzle solution, one step back, grid reset and puzzle completion time. Each one of the forementioned functionalities significantly enhances user engagement, by offering a chance to the user to assess their strategy and introducing a competitive element, encouraging them to improve their problem-solving skills. All functionalities mentioned above, and their implementation, will be discussed in detail in the following sections of this chapter.

Apart from extending this project to a mobile application, I have explored another possible extension of the project - machine vision [BW12] to enter and automatically solve puzzles for the user. The discussed feature in my 'Interim Report' was researched by me, but I did not have enough time and confidence in my understanding of the subject to start implementing it into the code base, as it requires a lot of additional research and more complex code development [WH14].

2.1 Algorithmic Approach and Puzzle Generation

The primary objective of the project's algorithmic approach is to ensure each generated and solved Sudoku puzzle has only one, unique solution. To achieve this, a backtracking algorithm is being developed to best satisfy the constraints in the Sudoku grid. This technique, which in its essence is a depth-first search, when applied to the Sudoku game's strictly defined configuration is highly effective in iterating through all possible solutions.

The algorithm developed inside *'backtracking_algo.py'* in the third Proof of Concept program is the backbone of the whole project, as it takes the responsibility for finding the unique solution to the Sudoku puzzles. Solving Sudoku problems involves a form of exhaustive search of all possible configurations. However, exploiting constraints to rule out certain possibilities for specific positions is the key method utilised to improve the solving time of the Sudoku puzzles. This approach takes advantage of the game's rules by eliminating possible options

and simplifies the process of finding the correct solution. The fine-tuned algorithm for this project will be covered in detail in the Completed Proof of Concept Programs section of the report.

Building on the fundamental backtracking algorithm researched and discussed in my third Proof of Concept program, in the second term I had to 'translate' and fine-tune the algorithm into a JavaScript code environment. The decision was made after I realised that the React Native (JavaScript based) framework, which acts as the backbone of this mobile game application, will be best implemented when only one 'main' programming language is used. There are many reasons behind this, but keeping the code base consistent and focused on efficient implementation of the algorithm inside the React Native's architecture, are the main ones. By 'translating' the Python code, I aim for easier future code maintenance and a wider integration rate considering different mobile platforms without sacrificing performance or user experience.

The final JavaScript version of the backtracking algorithm is implemented and used from the 'sudokuSolver.js' file. The researched Python implementation in the Completed Proof of Concept Programs section of the report is not simply copied and 'translated' into JavaScript, but also improved by using the specific language's features and React Native's advantages to make solving puzzles more efficient. The algorithm speeds up solving by marking and sorting empty cells, focusing on those with fewer options first, with the idea to quickly narrow down the correct number for each place. It efficiently tracks and updates the grid, reducing the steps to find the solution.

In the developed 'sudokuGenerator.js' file, a Sudoku puzzle generator is implemented with focus on creating grids with a unique solution. As already outlined in the Algorithmic Complexity and Integration subsection, there are two main factors that affect the difficulty of the project. To evaluate the difficulty level of the generated project, I will use a simple yet effective technique - the puzzles will be generated as fully completed and randomised Sudoku grids, after which a number of cells will be randomly removed to create a puzzle. A key part of this process is ensuring that each puzzle has only one, unique solution - this will be achieved by leaving a minimum of 17 clues in the standard 9x9 grid [Lee06] and by using backtracking to test for the puzzle's uniqueness, restoring any removed numbers (and jumping back to that state) if it detects potential multiple solutions. The choice of how many cells to remove will depend entirely on the level of difficulty chosen by the user. To ensure a different puzzle is generated each time, the list of possible values is randomised and a shuffling method for the numbers before placement ensures that each puzzle is new. The application will have four levels of difficulty - Easy, Medium, Hard and Expert.

Level	Empty Cells
1 (Easy)	40 to 43
2 (Medium)	44 to 47
3 (Hard)	48 to 51
4 (Expert)	52 to 55

Table 2.1: Number of Empty Cells for Each Difficulty Level

Note: It is important to specify, that I will only be judging the level of difficulty based on the number of empty cells in the generated puzzle. I will not be considering the positioning of the empty cells as a factor affecting the difficulty level, as well as how much computational 'effort' is needed to solve it.

2.2 In-Game Functionalities

In the project specification provided by the Department of Computer Science, there are multiple extensions discussed, and particular emphasis is placed on enhancing the in-game user experience [Dep23]. I have identified and decided to implement the following critical functionalities. Each one of them is carefully designed and developed to not only enhance the user engagement but to provide strategy assessment and competitive element to the application. The detailed objectives of the features are discussed in this section of the report.

Before starting a new game, the user is presented with the option to learn more about the Sudoku game and this application's functionalities inside a separate /Instructions screen. In this screen, an overview of the app's features and gameplay tips, are structurally presented.

The game offers four levels of difficulty - Easy, Medium, Hard and Expert, ensuring a progressive challenge for players of all skill sets. Once a game is initiated on the /Game screen and based on the level chosen, the user encounters a pre-generated 'initial' Sudoku grid. Upon starting the new game, a timer for tracking the completion time is initialised and visually presented above the grid. This, from the first glance unimportant tracking feature, provides a competitive element for future progress comparison which can be observed later in the /History screen.

Below the grid, there is a number pad with digits from 1 to 9 and a 'X' for deleting incorrect inputs. The gameplay is enhanced with highlighting number cells based on player's interaction with the grid:

- cells part of the initial puzzle are **locked** and **marked in a light grey colour**;
- on selection, the selected cell is highlighted in **green**, providing clear visual feedback;
- on number input, all the same numbers in the grid are also highlighted in **green** for a better visual tracking.

There are four **hint buttons**, which are initially disabled and highlighted in **red**. After the player's first input, the buttons are enabled and highlighted in **white** and thus the player gains access to:

Grid reset option:

- This feature clears all user inputs, reverting the puzzle grid back to its initial setup. It provides an essential tool for correcting multiple errors simultaneously, adding a straightforward and effective user experience.

Step back option:

- The step back feature allows undoing the last user interaction with the grid - adding, resetting the grid or removing a single number. It helps in managing individual mistakes efficiently and ensures the gameplay remains seamless and user-friendly.

Validate grid option:

- The validation option allows the user to compare the current progress to the unique correct puzzle solution - it indicates the existing (or lack of) mistakes. If mistakes are detected, they are highlighted in **red**. This serves as a real-time assessment tool and will benefit the user in improving or comparing their problem-solving technique.

Computer puzzle completion option:

– The computer puzzle completion feature directly provides the user with the solution grid. If there are mistakes in the grid, it won't solve it, but it prompts the user to utilise the 'validate grid option' to identify the errors. If there is a solution, the solved grid will be displayed for a few seconds and the user will be transitioned to the home screen. This mechanism supports error correction and learning by directing the user's attention to 'unregistered' mistakes - as in my opinion, this option will be used only when the player has already given up and does not want to use the validation option.

The player can exit the current game state at any point (accounting both closing the application or simply leaving the in-game screen) and the progress made will be automatically saved, allowing them to resume later without losing any progress. The saved puzzle state for resuming includes several internal props for tracking: '*initialGrid*', '*currentGrid*', '*solutionGrid*', '*level*', '*puzzleId*', and '*timeTrack*'. In the home screen, the player is presented with the option '*Resume*', which brings him to the in-game screen and not losing any progress.

Upon completing a game, the player can navigate to the /History screen to view statistics of their previously completed games: '*puzzleId*', '*level*', '*timeTrack*', date of completion and '*solutionGrid*'. Clicking on a game entry displays a popup with a mini version of the completed puzzle grid, offering a recap of the game. The immediate feedback allows players to assess their performance in the specific game they have completed and offer insights into the problem-solving methods they have used. My idea with this 'archival' feature is to not only add more depth to the game but also create a sense of continuous learning, progress and achievement. The application's variety of features, supports both novice and experienced players, ensuring a dynamic and engaging puzzle-solving experience.

2.3 Graphical User Interface

The graphical user interface (GUI) of the application is focused on a minimalist approach, emphasizing functionality over overcomplicated visual effects. The aim is to create a simple and easy-to-navigate design that focuses on the user interactions without unnecessary complexity. By prioritizing essential functionalities and user-friendly design elements, the interface is tailored to provide an efficient and engaging experience for the player. This approach ensures intuitive and accessible application interface, addressing users' needs while maintaining a clean and straightforward mobile interface.

I have chosen a stark black background for the application, enhancing visibility and intuitive navigation for other elements. All additional components such as buttons, headers, and text are styled in green and white, creating a sharp contrast that focuses on user navigation and readability. Additionally, there are specific cell and button indicators:

– Cells that are part of the initial puzzle setup are **locked** and **marked in a light grey colour**. This acts as a visual cue for a direct and clear contrast between cells open to user input.

– Upon selection, the selected cell is highlighted in **green**, providing clear visual feedback, drawing the user's focus to their current interaction.

– When a number is entered, all instances of that number across the grid are also highlighted in **green**.This consistent use of colour not only strengthens visual tracking but also helps in assessing the grid's layout more quickly and effectively.

– The validation option compares the current grid to the solution grid, detected mistakes are highlighted in **red**.

– The four hint buttons are initially disabled and highlighted in **red**, upon activation - their colour is changed to white.

2.4 Researched Extension: Machine Vision

Apart from the above discussed functionalities, probably the most challenging (possible) extension to me is using machine vision (the phone's camera) to allow the user to input a completely new puzzle to the system. I have a good idea and understanding of the functionalities which will become the fundamentals of the final mobile application, but the implementation of Machine Vision requires detailed research and acquiring a new set of skills.

My primary motivation and interest in incorporating Machine Vision into the Sudoku application was the implementation of an additional level of user interaction and gain practical knowledge in advance image processing. The purely theoretical idea for this feature, would have allowed the user to scan Sudoku puzzles using their phone's camera, which the application would interpret and input into the solving algorithm. If implemented, this would have also been the only way for the user to input custom Sudoku grids, increasing the user's overall gameplay experience.

I focused on reading and understanding the theoretical aspects, I researched sophisticated pattern recognition algorithms [BW12], capable of effectively distinguishing numbers and the Sudoku grid lines in various lightning conditions, angles and backgrounds. If I had time to implement this feature, the goal would have been to achieve high success rate in correctly scanning and interpreting puzzles, also comparing different approaches and algorithms to solve this problem [SUW18]. It would have represented a significant step in advancing the technical complexity [WH14] of the application and the real-world use cases for the user.

Even though I considered the integration of the feature, I prioritised the fundamentals of the game and in-game functionalities. While further exploring this possible extension, it became clear to me that adding it would significantly enhance the application by providing a unique user feature. However, I did not manage to implement the extension, as it required advanced knowledge in pattern recognition algorithms that needed way more time and research than was available within the second term.

Chapter 3: **Theoretical Analysis**

In this chapter of the report, my focus is to showcase the researched theories and explain the important, logical parts of them. For better readability and understatement of the discussed topics, I have logically divided the chapter into Algorithms Analysis and User Interface Theory subsections.

In the Algorithms Analysis section, a more theoretical approach is taken to explain the strategies for finding the best solution and puzzle generation. This includes a brief theory explanation of the solution to the N-Queens puzzle (my first Proof of Concept program), the backtracking algorithm's approach [MG17] (my third Proof of Concept program) and the challenge behind puzzle difficulty generation (my final 'sudokuGenerator.js' file) [Lee06].

In the User Interface Theory section, my focus shifts to the structured design principles and interaction models applied to shape the application's front end. This includes a short explanation on the Unified Modelling Language (UML) diagram and the practical user stories, which outline the framework and key interactions inside the application. In the Software Engineering chapter, I will go in detail about the mobile application's file structure, feature implementation methods and technology frameworks used.

3.1 Algorithms Analysis

This section gives a closer look at the technical aspects that make the software effective and accurate. The algorithms are broken down, with the goal of achieving a detailed explanation of the theory behind them, and understanding their role in the application's functionalities.

3.1.1 N-Queens Problem

This is a classic constraint satisfactory problem, in which N queens are placed on an N×N chessboard in such a way that no two queens threaten each other[RVW06]. The developed program inside the 'First Proof of Concept Program' finds all possible solutions for a given number of queens. The '*n_queens_solver.py*' program uses a recursive backtracking algorithm to find all possible solutions to the puzzle. This algorithm places queens on the board and iterates through the possible positions, backtracking when a conflict arises and therefore exploring all potential configurations.

Each solution is represented as a list of N tuples, where each tuple represents the position of a queen on the board (row, column). My solution is configurable to different NxN board sizes, it provides a clear demonstration of the recursive backtracking power in solving problems that require exploring multiple possibilities and making sequential decisions.

3.1.2 Backtracking Algorithm

Based on my research and understanding, the backtracking recursion algorithm is a systematic method to iterate through all possible configurations of a search space. It is a general algorithm which must be customised for each problem and set of constraint satisfactions [Civ13]. In the 'Third Proof of Concept Programs' I explore the application of recursive backtracking to solve simple and complex Sudoku puzzles [RVW06]. The program's docu-

mentation discusses two recursion types: basic recursion and backtracking recursion. The basic recursion can be described as a 'self-calling' function that solves smaller segments of a larger problem until a base case is reached, and the backtracking recursion is a more systematic approach involving depth-first search of all potential configurations. Backtracking is the key to implementing exhaustive search programs correctly and efficiently [Lee06] [DHS10]. Based on my personal understanding and the simplest way for me to explain it, backtracking is a special approach to problem-solving that involves:

- depth-first search on an implicit graph of configurations;
- exploring all possible solutions to a problem;
- eliminating those solutions that fail to satisfy the constraints of the problem;
- 'backtracking' to a previous step and trying alternative solutions.

I have identified three main categories of problems that can be solved by using backtracking algorithmic recursion:

- **enumeration problem**: the goal is to generate all possible solutions to a problem or count the total number of possible solutions to a problem;
- **decision problem**: the goal is to find one specific solution to a problem or prove that one exists;
- **optimisation problem**: the goal is to find the best possible solution to a given problem.

In the case of this project, the focus will be to solve any given Sudoku puzzle by the one and only possible solution. Therefore, the backtracking algorithm will be used and fine-tuned to solve the above described 'decision problem'. I have used my understanding of the backtracking application, specifically for Sudoku puzzles, to develop fine-tuned functions in the 'sudokuGenerator.js' file.

3.1.3 Puzzle Generation

The Sudoku puzzle generation involves multiple logical checks to ensure each puzzle is both challenging, unique and solvable. In the '*sudokuGenerator.js*' file, the algorithm dynamically creates Sudoku puzzles based on predefined difficulty levels, ensuring the generated puzzle maintains a unique solution. This section provides a theoretical analysis of the puzzle generation process inside the '*SudokuGenerator*' class.

Initialisation and Grid Setup

The '*SudokuGenerator*' class begins by initialising a 9x9 grid filled with zeros, which represents an empty Sudoku puzzle board. The constructor accepts a difficulty level, which adjusts the complexity of the puzzle based on the number of clues provided. The ranges are determined by introducing a random element in the number of cells removed after the generation of a complete and verified for correctness grid:

```
// different difficulty levels with number of removed clues for each level
Easy: 40 + Math.floor(Math.random() * 4), // 40 to 43, Easy level
Medium: 44 + Math.floor(Math.random() * 4), // 44 to 47, Medium level
Hard: 48 + Math.floor(Math.random() * 4), // 48 to 51, Hard level
Expert: 52 + Math.floor(Math.random() * 4), // 52 to 55, Expert level
```

Valid Placement Check

The '*isValidPlacement*' function is crucial for following of Sudoku rules. It checks whether placing a specific number in a designated cell is in conflict with the puzzle's basic constraints:

- this number must not already exist in the same row or column;
- this number must not already exist within the same 3x3 subgrid.

This function is called during both - the grid generation and the solution verification phases. This ensures, all pre-generated numbers placed in the Sudoku grid are valid under the puzzle's constraints.

Recursive Grid Filling

The foundation of the Sudoku puzzle generation is the '*recursiveFillGrid*' method. This function uses a backtracking algorithm that attempts to place numbers in empty cells:

- iterates through each cell in the grid;
- for each empty cell:
  - generates a random sequence of numbers 1 through 9;
  - tries to place each number using the '*isValidPlacement*' check;
- if a number is valid, the function recursively attempts to fill in the next cells;
- if no numbers fit, the function backtracks by resetting the cell to 0 and tries the next possibility in the previous cells.

This recursive approach is designed to explore all potential configurations systematically, until the entire grid is validly filled, or no solution is possible under the current configuration, at which point it backtracks to a previous 'safe' stage.

Puzzle Generation and Uniqueness Check

The process of generating a complete grid was already discussed: the '*generateCompleteGrid*' function initiates the process by calling '*initialiseGrid*' and '*recursiveFillGrid*'. After successful generation of a complete solution, the '*removeNumbersFromGrid*' function methodically removes numbers from the filled grid to create the final puzzle. The removal is based on the target count, linked to the selected difficulty level. To ensure the puzzle maintains a unique solution, which is critical for a valid Sudoku game, the '*hasUniqueSolution*' function checks the puzzle's uniqueness by:

- attempting to solve the puzzle, starting from the modified grid;
- using a backtracking solver, similar to '*recursiveFillGrid*' but designed to count all possible solutions, from the current progress perspective;
- the puzzle is considered to have a unique solution only when exactly one solution exists.

The '*hasUniqueSolution*' function was changed after my supervisor pointed out a possible logic flaw in ensuring puzzle uniqueness. It was then, when I added the additional backtracking solver, which keeps track of all possible solutions and ensure, that each puzzle is ready to be 'sent' to '*removeNumbersFromGrid*' function. This function is crucial for creating an initial grid to be displayed to the user, as it removes numbers (based on the chosen difficulty level) from a completely filled grid and creates the final initial grid:

– starts by determining the number of cells to clear, based on the puzzle's difficulty level;
– randomly selects grid cells and temporarily removes their numbers;
– before each removal, the function backs up the current grid state;
– after removal, it verifies uniqueness by calling the *hasUniqueSolution* method;
– if uniqueness verification fails, the grid is restored to its backup state, preserving integrity;

3.2 User Interface Theory

In the development of the user interface, I have approached and utilised two key theoretical principles, I learned from and used in my last year's 'CS2800: Software Engineering' module - UML (Unified Modelling Language) diagrams and detailed User Stories. During my research on the best approach to develop the interface, I have searched for inspiration and found motivation in the following book 'Systems analysis and design: An object-oriented approach with UML' by A. Dennis [DWT15].

The UML diagram serves as a visual and schematic representation of the system's architecture, providing a structured framework that visualises various components and their interactions. This methodology, rooted in object-oriented theory, offers a clear and systematic approach to visualize and document the design of the system, ensuring coherence and clarity in the development process. On the other hand, the user stories offer a human-centric approach, focusing on the practical needs and interactions of the end-users. Originating from Agile methodologies [FH+01], user stories encapsulate the requirements and functionalities of the system from the user's perspective, guiding the development process towards creating features that are directly aligned with the user's needs and preferences.

Chapter 4: **Development Progress**

In this chapter of the report, I provide an overview of the technical implementation required for bringing the application to life. The following sections are intended to offer a clear understanding of the 'first term stage' of development and provide detail into the final application's code and file structure. I will be discussing the Programming Languages & Frameworks in the linked section.

4.1 Completed Proof of Concept Programs

This section includes a detailed explanation of the developed algorithms, focusing on the implementation of the '*n_queens_solver.py*' First Proof of Concept Program and Third Proof of Concept Program in '*backtracking_algo.py*'. Additionally, the progress made on the front end side of the development, highlighting the advancements in the Sudoku GUI as part of the Second Proof of Concept Program.

4.1.1 First Proof of Concept Program

The First Proof of Concept Program focuses on the N-Queens problem, a classic constraint-satisfaction puzzle with the objective to place N number of queens on an NxN chessboard, such that no two queens threaten each other (the queen can be moved any number of unoccupied squares in a straight line vertically, horizontally, or diagonally [Tur88]). This PoC showcases the technical integration and application of a recursive backtracking algorithm to systematically find all valid solutions.

The program begins with setting the '*numQueens*' variable, which determines the number of queens and the size of the chessboard, set by default to 8. There are two key functions inside the program '$isSafe$' and '*placeQueen*'. The '$isSafe$' function verifies if it is possible to place a queen on a given row and column without any conflicts. It iterates through all previously placed queens to check for conflicts and incorporates a logical assessment of any potential threats from other queens (columns and diagonals):

```
# iterate through all previously placed queens to check for conflicts
for row in range(0, testRow):
    # check for another queen in the same column
    if testCol == currentSolution[row]:
        return False
    # check for another queen on the same diagonal
    if abs(testRow - row) == abs(testCol - currentSolution[row]):
        return False
return True  # if no conflicts, it's safe to place the queen
```

The '*placeQueen*' is a recursive function that attempts to place queens on the board, iterating over columns and rows, and backtracking when necessary. This function updates two lists: '*currentSolution*', which keeps track of the current state of the chessboard, and '*solutions*', which collects all the successful arrangements of queens found.

```
for col in range(numQueens):
```

```
# try placing a queen in each column of the current row
if not isSafe(row, col):  # check if it's safe to place the queen
    continue  # if not, skip to the next column
else:
    currentSolution[row] = col  # place the queen
    if row == numQueens - 1:  # if all queens are placed
        # add the solution to the list
        solutions.append(currentSolution.copy())
    else:
        placeQueen(row + 1)  # recursively try to place the next queen
```

The output provided by the program is the total number of solutions found, each represented as a list where the index signifies the row and the value at that index indicates the queen's column position.

4.1.2 Second Proof of Concept Program

The focus in the Second Proof of Concept Program is entirely on developing the main Graphical User Interface (GUI) for the Sudoku game, structured around the user stories and guided by the UML diagram.

The interface provides a logical path for the player to start a new game, choose from four difficulty levels and customise the application's theme. Additionally, it allows easy interaction with the Sudoku grid, allowing for number entry, cell highlighting, erasing mistakes and thereby enhancing the puzzle-solving experience.

Features added to the system, include a 'verify' button for error checking, hints for assistance, and a mistake counter to help players track their progress and improve their problem-solving abilities. It also includes options to pause the game and monitor playtime for added flexibility. Upon puzzle completion, players receive feedback on their performance and completion time, with time bonuses or penalty for those who solve puzzles without using the 'verify' button, encouraging skill improvement.

4.1.3 Third Proof of Concept Program

The backtracking algorithm in the center of this project, the program and documentation for it, are designed and created to answer the following question: '*How can I use recursive backtracking to find the best possible solution to very challenging Sudoku puzzles?*'

The backtracking algorithm involves a depth-first search through all possible puzzle configurations, systematically eliminating solutions that fail to meet the Sudoku's constraints and backtracking to the previous position, when such an 'elimination' is reached. However, exploiting constraints to rule out certain possibilities for specific positions, prunes the search to the point where that Sudoku puzzle can be solved even by hand. Backtracking is the key to implementing exhaustive search correctly and efficiently. Simply explained, backtracking is a special approach to problem-solving that involves:

– depth-first search on an implicit list of configurations;

– exploring all possible solutions to a problem;

– eliminating those solutions that fail to satisfy the constraints of the problem;

– 'backtracking' to a previous step and trying alternative solutions.

In this specific case, the technical implementation of this program is carried out in Python version 3.10.4. An example Sudoku puzzle is defined at the beginning as a 2D list with zeros representing empty cells. The algorithm's effectiveness is demonstrated by its ability to find the unique solution to this and other test puzzles. The program's output includes printing the original puzzle and the solved grid, showcasing the algorithm's successful implementation.

It initialises dictionaries to keep track of empty cells and their count in each row and column, which in the Sudoku puzzle is crucial for optimizing the backtracking process. By identifying the most constrained parts of the grid, the algorithm can start with these areas, making the process more efficient.

```
# sorting empty cell positions to prioritize those
# in rows or columns with fewer empty cells
# helps in prioritizing cells that are in rows or
# columns with fewer empty cells

sorted_positions = sorted
    (position.keys(), key=lambda x: remaining[x[0]] + remaining[x[1] + 9])
```

The '*valid*' function plays a pivotal role in this process by ensuring that each number, placed on the grid, follows the Sudoku constraints. The function conduct three critical checks - row, column and 3x3 minigrid check. In the case, when all checks pass without finding a conflict, the function returns '*True*', signifying a valid placement. The most interesting of the three is the 3x3 minigrid check:

```
# checks the 3x3 box, in which the number is inserted
box_x = pos[1] // 3 # gives the x position of the 3x3 box
box_y = pos[0] // 3 # gives the y position of the 3x3 box

# iterates through the rows of the box
for row in range(box_y * 3, box_y * 3 + 3):

    # iterates through the columns of the box
    for column in range(box_x * 3, box_x * 3 + 3):
        # checks if the current number inserted is already in the box
        if grid[row][column] == num and (row, column) != pos:
            return False
```

The program's core function '*solve_sudoku*' is the heart of the puzzle-solving process, fills the grid by using iteration, placing numbers in valid positions and backtracking when a dead end is reached. This approach ensures that every possible configuration is considered, leading to the discovery of the puzzle's solution. The function takes the grid and a list of sorted positions of empty cells '*sorted_positions*'. The sorting is based on the strategy to prioritize cells in rows or columns with fewer empty cells, optimizing the backtracking process. The function iterates through numbers 1 to 9 and checks if each of these numbers can be validly placed in the first empty cell (determined by '*sorted_positions*'). When a valid number is found, it is placed in the cell, and the function is recursively called for the rest of the empty cells. This function showcases the essence of backtracking, methodically exploring and retracting steps to find a valid solution to the Sudoku puzzle.

```
for num in range(1, 10):
    # checks if the current number is valid for the current position
    if valid(grid, num, (row, col)):
        grid[row][col] = num    # if the number is valid, insert it

        # recursively call the function to solve the grid
        if solve_sudoku(grid, sorted_positions[1:]):
            return True # the puzzle can be solved with this number

        # the puzzle can not be solved with this number, backtrack
        grid[row][col] = 0
```

The Third Proof of Concept Program represents a significant step in the development process, showing the integration of a complex algorithm into a functional code. It also proves my progress in applying theoretical computer science concepts to practical problem-solving scenarios. The implementation highlights the power of backtracking recursion in solving similar to the Sudoku puzzles, requiring a methodical and exhaustive exploration of possible solutions.

4.2 Final Application & Code Structure

In this section, a direct QR code is presented, in order to download the Android .APK from Expo Dev. This is the direct link to download the application: `https://expo.dev/artifacts/eas/2Nr9pawmLtQdvR3KYBiPUK.apk`

Figure 4.1: QR to download the Android APK

Following the discussed Aims and Objectives, the following subsections are dedicated to provide overview of the application's file structure and explain in detail code from some critical files. The idea with this section is to provide explanation to the architecture of the application and showcase the role and connection between different components, hooks, screens and utility files. The goal is after reading this section of the report to gain an overall understanding of:

- How each connection between files collectively creates the functionality of the application?
- Why the code is integrated in such a structure? Why this approach is chosen?

4.2.1 Files and Code Structure

The mobile application is created using the React Native framework Programming Languages & Frameworks, which supports a modular architecture that 'promotes' separation of concerns and logic. This approach improves the maintainability and scalability of the application, by organising the codebase into logical units that safely manage different aspects of the application's functionalities and internal properties.

Components

Components are the building blocks of the application's user interface. Each component is designed with the intention to be reusable and capable of handling specific UI elements, such as:

- **'*numberPad.js*':**
  controls number input for the Sudoku grid - its logic is integrated with different hooks to manage state changes related to number selection, reflecting these changes directly in the gameplay interface.
- **'*sudokuCells.js*':**
  manages the display and interaction of individual Sudoku cells - each cell style is adjusted, based on its state, such as: being selected, locked, or incorrect, thus enhancing the user's visual and interactive experience.
- **'*sudokuGridLogic.js*':**
  this component acts as a controller, all the logic and state management necessary for the player's interaction with the puzzle is defined here. It integrates multiple additional hooks and the above-mentioned components, in order to abstract the complex state management and manage detailed user interactions, from number input and cell selection to complex game validations and error handling.
- **Pop-up Messages ('*confirmationPopup.js*' and '*customPopContinue.js*'):**
  the pop-ups provide feedback and interactive dialogues for user actions, like navigation confirmations and errors, they are essential for guiding user interactions and double-confirming user actions within the application.

These components are used across different screens to reduce code redundancy, while ensuring UI consistency and seamless user interactions.

Hooks

I have used hooks a lot in this application, allowing me to encapsulate and manage state logic that supports different game mechanics. With the modular use of hooks, my intent is to simplify the component's logic, as state management and side effects are methodically abstracted away from UI code, making the components clean and focused entirely on rendering. I had logically divided the hooks into five folders. Below, compressed information about each hook's functionality can be found:

/inGameHooks folder

- **'*useCellSelection.js*':**
  manages the selection state of Sudoku cells across the grid, allowing for simple control of the logic behind features, such as highlighting and error indication.

– **'*useNumberPad.js*':**
handles logic for the number pad interaction, allowing users to select numbers and place them on the Sudoku grid. The hook ensures that number inputs are handled correctly and integrated smoothly with other gameplay mechanics.

– **'*useNumberCounts.js*':**
tracks the count of each number remaining to be placed on the grid, using memorisation to optimise performance, reducing unnecessary re-renders.

– **'*useSudokuSolver.js*':**
convert the grid into a 9x9 array expected by the '*SudokuSolver*', necessary for checking and providing solutions dynamically, important for the game's solving feature, more detail in the In-Game Functionalities section.

– **'*useHintPad.js*':**
manages the functionality of hint buttons, specifically the restart and step back options. It holds the logic for the puzzle reset and to revert to a previous state features, more detail in the In-Game Functionalities section. Below is a code snippet from the file, explaining the '*handleStepBack*' function:

```
// function for the step back button
const handleStepBack = useCallback(() => {

// if more then one input has been made
if (history.length >= 2) {

  const newHistory = history.slice(0, -1);
  setGrid(newHistory[newHistory.length - 1]); // update the grid state
  setHistory(newHistory); // update the history
} else {
  // in case the user goes back after the first input
  // history.length === 2, including the initial state and one user action

  setGrid([...newGrid]); // reset the grid to it's initial state
  setHistory([newGrid]); //reset to initial grid state
}
},[newGrid, setGrid, history, setHistory]);//dependencies for useCallback,
    calling the function only when these props change
```

The '*handleStepBack*' function in '*useHintPad.js*' manages the undo mechanism for the Sudoku gameplay. It checks if the history has more than one entry (since the first entry is the initial grid state) to enable a meaningful step back. If valid, it removes the most recent move from the history, reverting the grid to the previous state. If the user attempts to undo the first move, it resets both the grid and history to their initial conditions. The function is controlled by a '*useCallback*', optimising performance by preventing unnecessary re-renders and function redefinitions, ensuring the game's responsiveness and state integrity even during complex interactions.

– **'*useValidate.js*':**
offers real-time validation of the current Sudoku grid (the one including the user's input) against the puzzle's solution grid, this handles the logic for the feature 'Validate grid option', more detail in the In-Game Functionalities section. Below is a code snippet from the file, explaining the '*validateGrid*' function:

```
const validateGrid = useCallback(() => {
    let allCorrect = true; // assume all inputs are correct initially
```

```
    const newGrid = grid.map((cell, index) => {
        const userValue = String(cell.value);
        const solutionValue = String(solutionGrid[index]);

        // if the cell's value matches the solution grid's value
        const isCorrect =
        cell.locked || cell.value === "" || userValue === solutionValue;

        // if any cell is incorrect, update allCorrect
        if (!isCorrect) allCorrect = false;
            // update the cell's "correct" property
            return { ...cell, correct: isCorrect };
    });

    setIsCorrect(allCorrect);
    // show a success popup if all inputs are correct
    setPopupVisible(allCorrect);

    // return the new grid for the component to update state
    return newGrid;
}, [grid, solutionGrid]);
```

Each cell is checked against the solution, cells are marked incorrect only if they do not match the solution, the rest of the cells are unchanged. The grid is updated to reflect which cells are incorrect, providing a visual clue for the player in identifying errors, a pop-up is displayed if the entire grid is correct.

/useGamesArchive folder

– **'*useManuallySolved.js*':**
checks if the grid is correctly solved by the user, if the puzzle is filled and matches the solution grid, the game state is saved as solved; otherwise, a pop-up message shows an error feedback, which states that the puzzle is incorrect.

– **'*useSolvedArchive.js*':**
manages the storage and retrieval of solved games states (AsyncStorage), it fetches and updates the list of solved games, providing historical gameplay data to the user in the '*historyScreen.js*' screen.

/useInputHistory folder

– **'*usePuzzleHistory.js*':**
controls the 'history' prop of the game state, important for the functionality of the step-back button, itt saves and retrieves game history from AsyncStorage, allowing users to undo their actions. Below is a code snippet from the file, explaining the '*saveHistory*' function:

```
// save history to AsyncStorage
const saveHistory = useCallback(async (history) => {
    try {
        const serializedHistory = JSON.stringify(history);
        await AsyncStorage.setItem
```

```
            (`puzzleHistory_${puzzleId}`, serializedHistory);
    } catch (error) {
        console.error("Failed to save puzzle history", error);
    }
}, [puzzleId]);
```

This function saves the current state of the puzzle's history to AsyncStorage, allowing the game's progress to be saved and accessible in the case of a step back. Uses AsyncStorage (the history array is converted into a string using JSON.stringify for storage) to save the serialised history string. This makes it possible to resume or step back in the puzzle from each saved game state, at any time during the game progress.

/useResumeGame folder

– **'*useResumePrevious.js*':**
controls the resume game session feature by managing the loading and clearing of the game state from AsyncStorage, the hook ensures that players can resume their previous games exactly where they left off. The saved game state includes the following props: '*initialGrid*', '*currentGrid*', '*solutionGrid*', '*level*', '*puzzleId*', and '*timeTrack*', more detail in the In-Game Functionalities section. Below is a code snippet from the file, explaining the '*loadGameState*' function:

```
// function to load the game state from AsyncStorage
const loadGameState = useCallback(async () => {
setIsLoading(true);
try {
  const serialisedState =
    await AsyncStorage.getItem("gameState");
  if (serialisedState) {
    const gameState = JSON.parse(serialisedState);
    setPreviousGameState(gameState);
    setHasPreviousGame(true);
  } else {
    setHasPreviousGame(false);
  }
} catch (error) {
  console.error
    ("Error loading game state:", error);
} finally {
  setIsLoading(false);
}
}, []);
```

The function fetches the game state from AsyncStorage and parses it from a JSON string back into an object. It updates the component's state to reflect whether a previous game is available to be resumed.

/useTimeTrackHooks folder

– **'*useTimeTrack.js*':**
tracks the time passed since the start of the puzzle, updating every second, it is an interesting feature for providing competitive metrics or personal improvement.

Each hook is designed to perform one, specific task that contribute to the game's functionality, improving the modular architecture of the application and ensuring that components remain simple and maintainable. These hooks facilitate complex interactions within the game's user interface, enhancing the user experience by trying to make the game intuitive and responsive.

Screens

The application has four main screens, each corresponding to a different view and set of features within the app:

- **Home Screen - '*homeScreen.js*':**
  the landing and navigation screen of the application, from which the player can start new games, resume previous games, or navigate to the game history.
- **Game Screen - '*gameScreen.js*':**
  the interactive screen of the application where the user solves the Sudoku puzzle. The screen displays the Sudoku grid, the number pad and the hint pad (for the hint features in the In-Game Functionalities section) along with necessary hooks that directly manage the game's state.
- **History Screen - '*historyScreen.js*':**
  displays a log of completed games, allowing the player to review their past game statistics and solutions, more details in the In-Game Functionalities section.

Each screen is designed to handle specific user interactions and to present information in a user-friendly manner. The connection between these screens and their components is achieved by React Navigation, which provides a smooth and intuitive navigation experience for the user.

Styles

The styles directory within the Sudoku app project contains CSS-in-JS files that define the visual aspects of the application, ensuring that the UI is both functional and appealing. Each file targets specific components or screens:

- **'*cellHighlightStyle.js*':**
  styles related to the highlighting of cells based on user interactions or game logic, such as selection or error indications.
- **'*homeScreenStyles.js*':**
  styles for layout and elements found on the home screen, such as button layouts and background colors.
- **'*popupStyles.js*':**
  styles for popup modal elements, styling the messages as clearly visible and aesthetically pleasing.
- **'*styleHistoryScreen.js*':**
  styles for the history screen, styling the list of past games and the additional data presentations.
- **'*styles.js*':**
  central stylesheet, includes common styles that are reused across multiple components or screens.

– **'*stylesInstrScreen.js*':**
styles the instructions screen, customising layouts and fonts to enhance readability and user engagement.

– **'*sudokuGridStyle.js*':**
styles the Sudoku grid itself, styling cells, grids, and number pads to align with the game's functional needs and visual consistency.

The style sheets are fundamental for maintaining a consistent appearance throughout the app, following design principles that ensure a streamlined and user experience.

Utils Functions

The files inside this folder are essential for handling side effects and ensuring data persistence, packed in utility functions that support the core gameplay functionalities of the Sudoku application.

– **'*gameStateUtils.js*':**
manages the serialisation and storage of the game state, including grids and tracking details, into persistent storage using AsyncStorage, it allows the game to be paused and resumed later without any loss of progress.

– **'*puzzleIdUtils.js*':**
generates unique identifiers for each puzzle session, using AsyncStorage to track and increment the last used puzzle ID, the system saves each game and references it directly, creating effective game state management.

– **'*saveSolvedGames.js*':**
records games that users have solved by storing details, such as the '*puzzleId*', '*level*', '*solutionGrid*', '*timeTrack*' and the exact completion date into a list of solved games in AsyncStorage. The saved state from this utils function allows the detailed display of information inside the /History screen.

– **'*timeUtils.js*':**
provides essential time-related functions that convert time between seconds, in a more readable format - 'MM:SS'.

Each utility file addresses specific requirements that enhance the application's functionality, from managing the game state and IDs of each puzzle, to formatting and recording the time track. These utilities ensure the game's data integrity and usability, important for providing a professional and reliable gaming experience.

Overall, the structured approach in using React Native's component-based architecture allows the application to maintain high performance and responsiveness while providing a clear pathway for future enhancements. The implemented architecture is robust enough to integrate in the future additional features, such as the discussed 'Machine Vision' extension in other sections of the report. This modularity ensures that updates and improvements can be made with minimal disruptions to the existing codebase.

4.2.2 Algorithm for Generating Sudoku Puzzles

Fundamental part of the development of this project, focused on automated puzzle generation and difficulty categorisation. The approach chosen is to first create a fully completed Sudoku puzzle, from which a specific number of cells will be randomly removed to form the puzzle grid [Lee06] in combination with a small backtracking algorithm that ensures a unique solution throughout this process. The method ensures that each puzzle has only one, therefore unique, solution. The difficulty of these puzzles will be categorized into four levels – Easy, Medium, Hard, and Expert – based on the number of cells left empty. This categorisation is critical for appealing to a broad spectrum of players, from beginners to advanced. Furthermore, using a randomization technique in the puzzle generation, my goal is to make each gameplay unique, thereby maintaining player's interest and engagement over time.

The '*SudokuGenerator*' class holds the entire logic required for generating and validating Sudoku puzzles of varying difficulty levels. The class is designed to ensure that each generated puzzle is both solvable and unique. A simple explanation of how the class operates can be found below:

- **Grid Initialisation:**
  sets up a 9x9 grid filled with zeros, representing an empty Sudoku board. The difficulty level is chosen by the player - Easy, Medium, Hard, or Expert, influencing the range number of cells left empty in the puzzle.
- **Puzzle Generation:**
  '*generateCompleteGrid*' function fills the grid completely with a Sudoku puzzle, using a backtracking algorithm implemented in the '*recursiveFillGrid*' function, which calls the '*isValidPlacement*' function that checks if each placement satisfies the Sudoku puzzle constraints.
- **Clue Reduction:**
  after generating a solution, the '*removeNumbersFromGrid*' function strategically removes numbers from the game grid, based on the set difficulty level, ensuring the puzzle remains solvable with a unique solution by calling the '*hasUniqueSolution*' function.

I would like to discuss two key functions inside the '*SudokuGenerator*' class, which are responsible for proving that each Sudoku grid presented to the user has only one, unique solution.

- **Function '*removeNumbersFromGrid*()':**
  The function is used to create the actual Sudoku puzzle by removing numbers from a completely solved grid based on the selected difficulty level. Provided below, is a well-commented code snippet of the function.

```
// function to remove numbers from the grid, creating a unique puzzle
removeNumbersFromGrid() {
 // numbers to remove based on selected difficulty level
 const removalsBasedOnLevel = this.difficultyLevels[this.level];
 let numbersRemoved = 0;

 // remove numbers until the target based on the level is reached
 while (numbersRemoved < removalsBasedOnLevel) {
   // select a random row and column
   let row = Math.floor(Math.random() * 9);
   let col = Math.floor(Math.random() * 9);
```

```
    // check if cell is empty
    if (this.grid[row][col] !== 0) {
      // creates a backup of the current grid state before removing
      const backupGrid = JSON.parse(JSON.stringify(this.grid));
      this.grid[row][col] = 0; // attempt to remove a number

      // check if current grid config has unique solution
      if (!this.hasUniqueSolution(JSON.parse(JSON.stringify(this.grid)))){
        this.grid = backupGrid; // restore if uniqueness is compromised
      } else {
        // add to counter, number successfully removed
        numbersRemoved++;
      }
     }
  }
}
```

The method attempts to remove numbers randomly from the filled grid while ensuring the resulting grid maintains only one valid solution. It uses the '*hasUniqueSolution*(*grid*)' function to validate each removal, reverting changes if the removal leads to multiple solutions.

– **Function '*hasUniqueSolution*(*grid*)':**
This function checks whether the given Sudoku grid configuration has a unique solution. It is critical for validating the puzzle's integrity after numbers have been removed to set the difficulty. The function uses a recursive backtracking approach to explore all possible number placements in the grid. It counts solutions and stops exploring as soon as more than one solution is found, ensuring efficiency. Provided below, is a well-commented code snippet of the function.

```
// function to check if the generated puzzle,
// after removing numbers has unique solution
hasUniqueSolution(grid) {
  let solutions = 0; //  counter for solutions

  // recursive function to solve the puzzle
  const solve = (grid, currentPos = 0) => {
    // stop if more than one solution is found
    if (solutions > 1) return;
      // base case: if all cells are processed and a solution is found
    if (currentPos === 81) {
      solutions++;
      return;
    }

  // calculate current row and column based on the current position
  const row = Math.floor(currentPos / 9);
  const col = currentPos % 9;

  // move to the next cell if current cell is not empty
  if (grid[row][col] !== 0) {
    solve(grid, currentPos + 1);
    return;
  }
  // try placing numbers 1-9 in the current empty cell
```

```
    for (let num = 1; num <= 9 && solutions <= 1; num++) {
      // stop trying numbers if solution is found
      if (this.isValidPlacement(grid, num, [row, col])) {
        grid[row][col] = num; // place num in the grid
        solve(grid, currentPos + 1); // move to solve the next cell

        // backtrack by removing the number if it doesn't lead to a solution
        grid[row][col] = 0;
      }
    }
  };
  // create a deep copy of the grid to solve without changing original grid
  solve(JSON.parse(JSON.stringify(grid)));
  // true if only one solution is found
  return solutions === 1;
}
```

This algorithm class holds the most important logic in the entire application. It efficiently combines Sudoku solution generation with strategic clue reduction to produce challenging puzzles. These functionalities ensure that the puzzles not only provide an engaging challenge suited to the player's preference, but also adhere to the classic Sudoku requirement of having a unique solution. This approach highlights the application of algorithmic concepts and advanced programming techniques in creating enjoyable and interactive game experiences.

Chapter 5: **Software Engineering Methodology**

This chapter delves deeply into the programming languages and software methodologies used and followed in the project, explaining the reason behind their selection. The focus is on the explanation of the programming languages and frameworks used, the design methodologies adopted, and the documentation approach.

5.1 Programming Languages & Frameworks

Python (current version: 3.10.4) was chosen as the main language for writing all the Proof of Concept programs. Its simplicity, readability, and wide support for scientific computing and algorithm development[MG17]. Python's extensive libraries and community support make it ideal for implementing complex algorithms like backtracking, which is the foundation of the Sudoku solver.

For the front end, I selected React Native as the primary JavaScript-based framework [Eis15], as I already have some experience in developing with it. React Native offers a detailed and fairly recent documentation for developing mobile applications, allowing for a single codebase to be deployed on both iOS and Android (even though in this project I will focus on creating only an Android application). Although this project is primarily targeted at Android applications, the ability to extend seamlessly to iOS is a valuable asset. Its compatibility and resemblance with JavaScript, efficiency in rendering, and vast component library make it a preferred to me choice for creating a responsive and user-friendly interface for the Sudoku application.

The final code for the application is based on JavaScript as I wanted to maintain a single codebase for both Android and iOS, not only simplifies development but also ensures consistency across platforms. Additionally, React Native's architecture, coupled with JavaScript, allows for optimal rendering of user interfaces. Its use of the virtual DOM to manage UI components dynamically helps in minimising performance bottlenecks, especially important in applications requiring frequent UI updates based on user interactions, such as digit entry or puzzle validation in a Sudoku game. JavaScript's asynchronous processing capabilities make it ideal for handling multiple processes running in the background. This non-blocking nature ensures that the application remains responsive, enhancing the user experience.

The 'Expo' is an open-source framework that is designed to aid in the development of cross-platform mobile applications using React Native [MF17]. The use of 'Expo' within this project framework simplifies the process of building, deploying, and quickly iterating on iOS, Android, and web apps from the same codebase. 'Expo' provides a rich set of tools and services, including access to native functionalities such as the camera, which is ideal for the possible Machine Vision extension of the project. The framework is particularly beneficial for projects like this, as it allows for rapid development and easy updates, ensuring a smooth and consistent user experience across various platforms.

5.2 UML design & User Stories

The use of Unified Modelling Language (UML) and User Stories in the project serve as a navigational tool for me, as a developer [DWT15]. UML is utilized for its effectiveness in visually representing the system's architecture, providing a clear and structured overview of all components and their interactions. This is a great support for me in maintaining clarity and coherence throughout the development process.

User Stories, on the other hand, are used to keep the development process user-focused and not forget the features I find exciting. Trying to imagine the requirements from the user's perspective, guide my development process towards features that align closely with the user's expectations. Following, this ensures me that my final year project and application are theoretically and technologically complex, as well as user-friendly.

5.3 Application Testing

The testing framework used for the application includes a combination of unit tests and integration tests to ensure each component functions correctly both individually and when integrated with other parts of the application. The primary tools used for testing are Jest and the React Native Testing Library, which facilitate the creation of tests that simulate user interactions and check the application's state management logic. The tests are organised into several categories within the '*tests*' directory, each targeting specific components or functionality within the application:

- **Test Algorithms:**
  they verify the correctness and efficiency of the algorithmic solutions used for generating and solving Sudoku puzzles. This includes tests for the '*SudokuGenerator*' and '*SudokuSolver*' algorithm classes, ensuring they perform correctly under various conditions and constraints.
- **Test Hooks:**
  the foundational use of React hooks for state management and side effects in the application, required a series of tests dedicated to validating these hooks.
- **Test Utils:**
  the utility functions are tested to ensure their reliability, especially their interactions with AsyncStorage for data persistence.

The structured approach to testing the Sudoku application not only increases confidence in its stability and performance but also ensures that the user experience is seamless across different devices and operating systems. By covering a broad spectrum of components—from backend logic with algorithms to frontend usability with hooks and utilities, the testing strategy plays a pivotal role in the application's development lifecycle.

5.4 Future Risk Assessment

Thinking for potential future enhancements of the project, I have outlined three main concerns. I will not go into much detail, as I have already discussed every single feature and the problems that arise are logically connected to them.

1. Complexity in Machine Vision:
   The possible implementation of Machine Vision in the application, could bring complexities in the accuracy of the processing and interpreting diverse puzzle images (consider different lighting, backgrounds and angles). These possible problems will require extensive testing and refinement, again explaining why I did not pursue this extension for the Final Year Project code integration.

2. Performance Optimisation:
   As more features are integrated, there's a risk of decreased performance. Ensuring that the application remains responsive and efficient, especially on lower-end devices, is crucial. This risk is also directly connected to the Machine Vision extension.

3. User Interface Consistency:
   Maintaining a consistent and intuitive user interface as the application evolves, especially with the addition of the Machine Vision extension feature, will be essential to ensure a positive user experience and also satisfy the developer (me).

Chapter 6: **Conclusion**

In this final chapter, I am discussing the Professional Issues encountered during the mobile development phase and my personal self reflection on the whole project. I will not go into too much detail when speaking about the personal, self assessment, as my personal diary can be found in the /Appendix section, where I go into more 'personal' details and the progress is easy to be observed.

6.1 Professional Issues

While developing the mobile application, I have deliberately chosen to use React Native, as it provides native framework to the dominating mobile platforms - Android and IOS. Each platform has its unique characteristics, development environments and case-specific challenges. In this section, I explore the implications of developing mobile applications for these platforms, focusing particularly on the constraints posed by Apple's iOS development environment compared to Android's open ecosystem.

iOS development is characterised by its closed system, where Apple controls all aspects of the ecosystem, including the hardware, software, and the distribution process through the App Store. The primary tools for iOS development are Xcode, Swift, and Objective-C. Additionally, deploying apps to a device for testing requires a special 'developer account', which costs $99 per year. Moreover, testing iOS apps can feel more restrictive due to the need for app provisioning profiles. If the developer eventually gets to the point where they are ready to upload their application to the 'App Store', there is a rigorous app review process, often taking weeks to approve an app. The review process is known for its strict adherence to design and security guidelines, strictly managed by Apple. Apps can be rejected for minor design issues or functionalities that Apple deems inappropriate, which can delay the launch or update of an app significantly.

Additionally, developers must have a substantial initial investment in - macOS hardware, pay for a developer account, and often invest more in design and development to meet Apple's requirements. The closed nature of the iOS platform means that there are significant restrictions on what developers can do. This limits the use of third-party tools, customisation of elements, and access to certain hardware or software features. While Apple's closed development environment ensures a high standard of app quality and security, it comes with increased costs and barriers to entry for developers. Overall, Apple has closed their development ecosystem, enforcing directly their rules on developers and indirectly on their users/clients.

In contrast, Android's ecosystem offers a more open development environment. Google provides fewer restrictions on the hardware and software that can run Android, which is based on the Android Open Source Project (AOSP) [AOS24]. Developers can use Android Studio along with Java and Kotlin to create applications. The Google Play Store and other third-party app stores offer various options for app distribution. Developers can install apps on any Android device easily, using the .APK file, without any cost. This ease of testing accelerates the development process and user connection. Google Play has a more lenient app review process, usually completed within hours (it is worth noting, that this can lead to a higher presence of low-quality apps). Additionally, Android provides more flexibility in app design and functionality due to its open nature, allowing for more creative and innovative app experiences.

The open nature of the platform encourages experimentation and innovation. Developers can customise app behaviours and integrate with a multitude of third-party services and hardware, creating a more innovative app ecosystem. Android developers can start app development with a wider range of hardware and software, and without any initial and specific hardware or software 'investment'. The availability of Android Studio on multiple operating systems further reduces the cost and barriers to entry.

In conclusion, the choice between these platforms will depend on a developer's specific needs, resources, and target audience. I am happy that for this project I used and gained more knowledge in React Native's mobile development framework. It allowed me to create an application, which can be packed and distributed on both IOS and Android ecosystems. Unfortunately, I do not have Apple's 'developer account' and therefore I am able to only package and distribute the finished application to Android devices, packaged as an .APK file.

6.2 Self Assessment

This project, took place over the first and second term of my final year, each term challenged different aspects of my analytical ability and skill set. Initially, the project tested my theory analysis ability, specifically in the sphere of backtracking algorithms. While theory was not my primary interest, gaining deeper understanding into the practical implementation of these complicated algorithms in puzzle / constraint satisfactory games, turned out to be interesting and engaging. I have learned new approaches to algorithm development, and have definitely built a solid foundation and understanding in the fundamental backtracking techniques.

The focus on mobile development during the second term presented unexpected challenges to me. I felt insecure about my skills, therefore I undertook a three-week intensive course on React Native [Sch24]. This course was crucial as it provided me with the necessary skills to confidently handle different challenges in native mobile app development. I learned to construct and manage components and properties efficiently, and gained valuable knowledge into managing asynchronous storage — a critical aspect of maintaining state consistency across local user sessions. These skills helped me create a smooth user interaction with the Sudoku application, across different devices and operating systems.

In conclusion, the final year project has been an interesting journey that challenged and expanded both my theoretical knowledge and practical skills in software development. From mastering backtracking algorithms to navigating the case-specific difficulties of mobile app development with React Native, I have significantly enhanced my technical capabilities. The acquired skills will undoubtedly support my future endeavours, as I look forward to my career in the field of software engineering.

Bibliography

[AOS24] Android Open Source Project - AOSP. "Android Open Source Project - AOSP". In: (2024). URL: https://cs.android.com/android/platform/superproject/main.

[Apt03] Krzysztof Apt. *Principles of constraint programming*. Cambridge university press, 2003.

[BPS99] Sally C Brailsford, Chris N Potts, and Barbara M Smith. "Constraint satisfaction problems: Algorithms and applications". In: *European journal of operational research* 119.3 (1999), pp. 557–581.

[BW12] Bruce Batchelor and Frederick Waltz. *Intelligent machine vision: techniques, implementations and applications*. Springer Science & Business Media, 2012.

[Civ13] Pinar Civicioglu. "Backtracking search optimization algorithm for numerical optimization problems". In: *Applied Mathematics and computation* 219.15 (2013), pp. 8121–8144.

[Com24] React Native Community. "Async Storage". In: *URL https://react-native-async-storage.github.io/async-storage/docs/install/* (2024).

[Dep23] Department of Computer Science, Royal Holloway, University of London. *Full unit project list*. 2023.

[DHS10] Sean Davis, Matthew Henderson, and Andrew Smith. "Modeling Sudoku Puzzles with Python". In: 7 (2010), p. 3.

[DWT15] Alan Dennis, Barbara Wixom, and David Tegarden. *Systems analysis and design: An object-oriented approach with UML*. John wiley & sons, 2015.

[Eis15] Bonnie Eisenman. *Learning React Native: Building native mobile apps with JavaScript*. " O'Reilly Media, Inc.", 2015.

[FH+01] Martin Fowler, Jim Highsmith, et al. "The agile manifesto". In: *Software development* 9.8 (2001), pp. 28–35.

[Lee06] Wei-Meng Lee. *Programming Sudoku*. Apress, 2006.

[MF17] Eric Masiello and Jacob Friedmann. *Mastering React Native*. Packt Publishing Ltd, 2017.

[MG17] Andreas C Muller and Sarah Guido. *Introduction to machine learning with Python*. O'Reilly, 2017.

[RA03] Andrew Rollings and Ernest Adams. *Andrew Rollings and Ernest Adams on game design*. New Riders, 2003.

[RVW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[Sch24] Maximilian Schwarzmüller. "React Native - The Practical Guide". In: (2024). URL: https://www.udemy.com/course/react-native-the-practical-guide/.

[SUW18] Carsten Steger, Markus Ulrich, and Christian Wiedemann. *Machine vision algorithms and applications*. John Wiley & Sons, 2018.

[Tur88] Alan M. Turing. "Chess". In: *Computer Chess Compendium*. Ed. by David Levy. New York, NY: Springer New York, 1988, pp. 14–17. ISBN: 978-1-4757-1968-0. DOI: 10.1007/978-1-4757-1968-0_2. URL: https://doi.org/10.1007/978-1-4757-1968-0_2.

[WH14] Baptiste Wicht and Jean Hennebert. "Camera-based sudoku recognition with deep belief network". In: IEEE. 2014, pp. 83–88.

Appendix A: **Personal Diary**

NOTE:
The 'references' in the Personal Diary are completely different from the ones in the Interim Report. The diary was regularly updated over the past term, any changes to it can be seen in the GitLab history.
Most of the 'references' refer back to the plan, submited in October. I will have to change that.

/Sprint 1

- During the first sprint I had the important task to write my Project Plan and present it to my supervisor. The plan is ready, as well as I am ready in terms of fully understanding the project requirements, objectives and the development path that needs to be followed.
- Started reading the 'Introduction to machine learning with Python' [1] which is part of the reading list for the 'Machine Language' model I am taking. The book will give me the foundations need to understand the code I will be writing for my project and the knowledge needed in my 'Machine Language' model.
- Finished reading the 'Modeling Sudoku Puzzles with Python' [2] article in which the modeling of Sudoku puzzles in a variety of different mathematical domains was discussed and simplified.

References:

- [1] Muller, A.C. and Guido, S., 2017. Introduction to machine learning with Python. O'Reilly.

- [2] Davis, S., Henderson, M. and Smith, A., 2010. Modeling Sudoku Puzzles with Python. In practice, 7, p.3.

/Sprint 2

- During Sprint 2, I focused on exploring in-depth the foundational references in the project plan, intently spending time on understanding the algorithms and technologies that were explained and presented. From the discussions on the Backtracking search optimization algorithm [1] to the complex nuances of constraint satisfaction problems discussed by Brailsford [2], each one of the recources provided me with valuable knowledge into problem-solving methodologies and computational techniques.
- Specifically, the frameworks presented in the 'Solving the set cover problem and the problem of exact cover by 3-sets' [3], the elegance of Knuth's Dancing Links [4] also gave me a comprehensive understanding of solving puzzle-type problems. I also familiarized myself with the dynamics of stochastic optimization methods by K. Marti [5] and I had a quick overview of the 'Agile software development models TDD, FDD, DSDM, and Crystal Methods: A survey' where I was happy to learn more about the various agile software development models and confirmed my decision to utilize the Agile development in the project.
- Furthermore, I briefly delved into the challenges of Sudoku puzzle modeling and its computational ratings by studying the works of R. Pelánek [7], T. Yato and T. Seta [8]. The practical approach with which some essential algorithms were illustrated by R. Stephens [9] using Python equipped me with a few practical strategies on how to approach the next steps in the project - writing the Proof of Concept Programs. Out of curiocity, I had some time to review the Eisenman's take on React Native [11], and I was happy to find out that there are more than valuable insights on how to successfully set the foundational stage for the GUI.
- Taken the time to read and learn something new before the start of the development phase, I've also started crafting user stories and am in the initial phase of sketching a user flow dia-

gram, giving a clear direction to the visual aspect of the application. The following /Sprint 3 will bring forward the culmination of these preparatory steps into real work. My immediate goals include finalizing the user stories, completing the user flow diagrams, and initiating coding for the first Proof of Concept program. Concurrently, I will begin laying the groundwork for the GUI, ensuring it aligns seamlessly with the technologies that I will use and with the user stories.

References:

- The references used are from my Project Plan, I am not going to add them to this file as they are simply reused here.

I have some questions which I will address with my supervisor:
- Do I want the application to give hints for next steps?
- How do I test the application's algorithms and their implementation while playing the game? Do I have to do this? Do I have to display some information?

/Sprint 3

01/11/2023, 11:10am - meeting with my supervisor, what I have taken from the meeting: Do I want the application to give hints for next steps?
- Yes, it should not be that hard and it is an important feature.
How do I test the application's algorithms and their implementation while playing the game? Do I have to do this? Do I have to display some information?
- Test the implementation of those algorithms manually during the development process. Focus on the outcome and implementation (technologies used, time under dev etc.) in the final report.
- It is a good idea to send my presentation and my report to my Supervisor before submitting it for grading.
- Start with the implementation of the 8 Queens problem, continue with implementing the Backtracking algorithm.

- I worked on developing and documenting the first Proof of Concept (PoC) program, a classic computational problem. The program is written in Python and utilizes a recursive backtracking algorithm to find all possible solutions for placing N queens on an N×N chessboard without any two queens threatening each other. Each solution is represented as a list of N tuples, indicating the precise positions of the queens on the board.
- In the process, I created a comprehensive report detailing the program's functionality, usage, and configuration. Documented explanation of the algorithm can be found here [1]. The report serves as a user guide, providing clear instructions on how to run the script, set up the necessary configurations, and understand the output.
- Reflecting on today's work, I am satisfied with the progress made and the depth of analysis provided in the report. The documentation is detailed and well-structured, ensuring that any future users or developers can easily understand and utilize the program. I will continue with the development of the second PoC program, which will look into the backtracking algorithm's application in solving the Sudoku puzzle.

References: - [1] file '*doc_n_queens_solver.txt*' in the 'PoC - N-Queens puzzle' folder.

/Sprint 4

- Started working on the second Proof of Concept (PoC) program, which will focus on the application's GUI. The initial plan to create a simple, intuitive, and user-friendly GUI will be

followed so that the frontend will be mostly completed and will not require a lot of changes in a later stage of development. The user interface will be written entirely using the React Native framework, and the application will be tested on Android devices.

There is more information regarding the GUI in the 'PoC - Program2' folder, more specifically in the '*doc_simple_GUI.txt*' file. Here is a list of what I will strive to achieve in the following week:

Home screen
- slide bar to choose from different difficulty levels - DONE
- button to choose different theme
- button to start a new game - DONE
- button to continue from previous game
- button to view the history of games, redirect to History - DONE

Game screen
- the grid of 9x9 cells and 1-9 numbers displayed as buttons under the grid - DONE
- button to validate the solution - for now the button will be inactive
- button to clear the selected cell - DONE
- button to clear the entire grid
- button to return to Home screen - DONE

About screen
- displays information about the application
- button to return to Home screen

Instructions screen
- displays instructions on how to solve a Sudoku game and how to interact with the application
- button to return to Home screen

- I also reconsidered the user stories and I have made the required changes. Changes were needed as the previous file was not structured properly and it was not focused only on the user stories for the application.

I fell quite ill on Friday, 11th November 2023, I was supposed to complete the described components in the previous week, but I was not able to do so. I will try to complete them in the following week.

I am still working on the GUI but decided it is time to implement the backtracking algorithm and therefore I will start working on the third Proof of Concept (PoC) program in a separate /thirdProgram branch.

The work over the next two weeks (20-26th and 27-30th November) will be considered as part of Sprint 4 and it will be focused on the following:
- Implementing and documenting the backtracking algorithm in the third PoC program.
- Completing the GUI pages for the second PoC program.
- Creating a presentation and sending it over for review to my supervisor - before the 24th November.

/Nov 21, 2023

After careful research on the backtracking algorithm I came up with implementation which unfortunately does not provide the desired results. I will have to do more research and try to find a solution to the problem. For now, I will keep this program as a stepping stone for the next - corrected implementation of the algorithm. The *first_backtracking_algo.py* is in the 'PoC - Program3' folder and it is documented in the '*doc_first_backtracking_algo.txt*' file. I will continue working on the GUI for the second PoC program.

I also realised that the README.md file is not up to date and I will have to update it with the instructions on how to run each PoC program. I will do this in the following week.

/ Nov 23, 2023

As I mentioned on the 21st of November, I will keep the *first_backtracking_algo.py* as a stepping stone for the next - corrected implementation of the algorithm. Today, I believe to have found the solution to the problem and I have implemented it in the *backtracking_algo.txt* file in the 'PoC - Program3' folder. I have thoroughly documented the algorithm in the '*doc_first_backtracking_algo.txt*' file. I will continue working on the GUI for the second PoC program.

Notes: In the following weeks, I will write a program to generate Sudoku puzzles inside the future application. The program will generate a random Sudoku board where all numbers are filled in and then it will remove some of them to create an unique puzzle. This will ensure that the puzzle always has a solution. BUT! Making sure that the generated puzzle has exactly one solution will be a bit more challenging as I must leave at least 17 numbers for a 9x9 Sudoku (Game theory).

/ Dec 07, 2023

I have completed a simple 'demo' GUI for the future application. The GUI is written in React Native and it is tested on Android devices. The GUI is not fully functional, but it is a good starting point for the future development of the application. I have also updated the README.md file with instructions on how to run each PoC program.

/ 22 Jan 2024

While I was thinking about the design and implementation of the important features, I realised that I need more knowledge on React Native, I decided to complete this React Native course on Udemy: https://www.udemy.com/course/react-native-the-practical-guide/ In the past few weeks I have not worked directly on the project, but I am halfway with the completion of the course and I am confident that I will be able to implement all features I have in mind.

DEADLINE 22 March 2024

The ideal scenario for me is to be ready with the code by the end of February, so that I leave time for bug fixing and writing the final report + presentation. I am writing here with the intent to create a structured plan for the last nine weeks of the project:
- 22 Jan - 28 Jan: complete the React Native course, read more on database implementation and think of introducing account creation.
- 29 Jan - 04 Feb: GUI creation - complete the layout as I have it in mind(use Figma for the design), have every element in place, but not functional.
- 05 Feb - 11 Feb: GUI creation - complete the layout as I have it in mind(use Figma for the design), have every element in place, but not functional.
- 12 Feb - 18 Feb: make the GUI functional, implement the backtracking algorithm and the Sudoku puzzle generator.
- 19 Feb - 25 Feb: feature implementation - develop and integrate each feature, start with

simpler ones like validation and proceed to hint generation and time tracking.
- 26 Feb - 03 Mar: implement 'one step back' and completion statistics + start fixing bugs.
- 04 Mar - 10 Mar: fix bugs and start writing the final report.
- 11 Mar - 17 Mar: finalise the report and presentation.

29/01/2024 and 08/02/2024

Notes on where to begin with puzzle generation and backtracking algorithm implementation:
- write tests for the sudoku generator + write the sudoku generator - Done = PoC - Puzzle-Generation
- connect the sudoku generator with the already created backtracking algorithm in PoC - Program3 - Done = PoC - PuzzleGeneration
- connect the sudoku generator and 'solve' option with the GUI

18/02/2024
I used Python to research and develop the initial versions of the backtracking algorithms and the Sudoku puzzle generator.
In order to integrate these algorithms into the React Native application, I decided to duplicate the logic and rewrite the files in JavaScript. This approach is better in the following ways:
- use unified language for the entire application;
- direct integration with the React Native framework, without additional backend server to run the Python code;
- ensure that the algorithms are compatible with the React Native framework;
- better performance for the end-user, as the logic will be executed on the user's side;

Today I created the tests for the generator and the solver, and 'translated' the code from /PoC - PuzzleGeneration from Python to JavaScript.
Added these files in the /algorithms folder and the next step will be to integrate them with the frontend.

21/02/2024

In the past week, I am focused on implementing puzzle generation logic and GUI.
The puzzle is successfully generated based on the selected difficulty level, and the Sudoku-Generator function creates a new puzzle with unique solution.
There are clues of how many numbers there are left for the user to input. Added 'Solve it' button which functionality/error message will be fixed in the future.
I have implemented different highlighting, depending on the number 'type' and also made the code and files more readable by:
- renaiming files, renaiming functions and extracting some functions in new files.

The progress is going as planned, the application generates an unique puzzle, there is a tracker for the remaining numbers and it can be solved. I will continue introducing new features in the next commits.

23/02/2024

Decided to change the initial approach to the plan made on 22 Jan. I am developing each feature's front and backend simultaneously.
In this branch I will focus on developing the 'Hint' option, which will give the user the correct number for the currently selected cell.
After that, functionality to the 'X' button will be added - it will delete the number from the currently selected cell (of course if the cell is not locked).
Added the 'Step back' button functionality, will need to find some icons to replace the text buttons.

26/02/2024

Restructured the sudokuGrid.js file, I will keep each (group) feature in a different file, so that the code is more readable and easier to maintain.
I need to make some more structural changes to the code, but I am happy with the programs so far.

27/02/2024 Thinking about the so called 'hint' feature, I decided to change/rename it to 'validate' feature.
The user will be able to validate its input at any time and the application will:
- highlight any mistakes in red;
- give a popup message if at this point all input is correct.

Implemented this feature today. I created a new popup message component and added the functionality to the 'validate' button.
I was trying to create proper tests, because I feel like I might be missing some edge cases, but I couldn't manage to configure the testing environment properly.
Will do that soon. For now, everything seems to work fine, even though it is not the best approach.

28/02/2024

Decoupled the sudokuGridLogic.js file into separate hooks files:
-useCellSelection.js
-useHintpad.js
-useNumberCounts.js
-useNumbarPad.js
-useSudokuSolver.js
-useValidate.js.

In these smaller files, the code is more readable and easier to understand and maintain. Managed to fix a couple of 'bugs' in the progress, they were mainly related to the logic implementation. Will confirm that everything works properly by testing the application on a real device.

29/02/2024

Added comments and checked for any bugs. Fixed a bug with the step back feature.

01/03/2024

I am starting to implement 'Resume Previous' button functionality. The button will be used to resume the last game the user was playing.

03/03/2024

Added the 'Resume Previous' button functionality. The button is used to resume the last game the user played.
I had to learn how to use the AsyncStorage in React Native and also change the storing logic of the grid state.

Starting IOS and Android testing today.

13/03/2024

In the past week I had to focus on other university assignments and I was sick for a couple of days. I was thinking a lot about the discussed issue in the puzzle generation logic, specifically

possible two valid options for some numbers, a very niche case.
I identified the problem in the SudokuGenerator.js file - removeNumbersFromGrid() and hasUniqueSolution() functions.
They did not perform enough checks and I will have to rewrite them. The original implementation could generate puzzles with multiple valid solutions because of insufficient uniqueness checks post-number removal.

After the changes, the hasUniqueSolution method:
- implemented an early termination strategy, terminates the search as soon as a second solution is identified - reduces computational effort by avoiding exhaustive searches when unnecessary.

After the changes, the removeNumbersFromGrid method:
- implemented systematic trial-and-error method, removing numbers and checking for uniqueness - restoring to original state if the removal compromises puzzle's uniqueness.

I added additional tests to confirm the new implementation is correct. Made some small changes to the app icons and some comments.

14/03/2024

Added more tests to confirm the new implementation is correct, I needed to double check.

15/03/2024

Added crucial functionality to the application, the history is saved and controlled from an additional hook.
This allows on resuming the game:
- the user can go back to the initialGrid; before the history was handled only inside the sudokuGridLogic.js file, not 'globally' - bad initial design

This required me to introduce a new hook and to change the logic in the sudokuGridLogic.js and gameScreen.js files.

I am happy with the result, although I am not sure how to test this properly. This feature requires of me to test how history is stored and controlled in the entire application. how am I supposed to test that?

16/03/2024

I have to fix the saving logic for the specific case when there is no user input - as of now the time and the whole game is not properly saved on leaving the screen rather only on making a grid change. As the whole game state, including the time, currently is saved when a new game is started or when there is user input.

I will have to change the logic in the sudokuGridLogic.js file so that the game is saved AND before leaving the game screen.

18/03/2024

All sudoku properties are properly saved on user input and on leaving the game screen. At the moment, all planned functionality is working correct, I am happy with the result.

I used the useFocusEffect function, a hook from the @react-navigation/native package to save the game state on leaving the game screen. Added some small additional styling.

20/03/2024

Fully implemented the history screen, where the user can see the history of the games solved. The history is saved in the AsyncStorage. The user can see the date, the time spent on solving the game, the difficulty level and the puzzle number. On click, the user can see the grid of the game and the solution.

21/03/2024

Added styling and content inside the instructions screen.
The user can 'learn' how to play the game and how to interact with the application. Restructured the application's structure - organised files in new folders. Added some additional comments and fixed some small bugs. I am preparing the final version of the application for the submission. I will continue with testing the application on real devices (Andoroid and IOS) and make sure everything works as expected.

21/03/2024

Supervisor Meeting questions:
code:
- how should I test GUI related functionalities? I am not sure how to test the AsyncStorage and the navigation.
- is the file structure okay?

report:
15k words?
keep the Interim Report structure + add more details on the implemented features, new sections and the challenges I faced?

for now, my plan is to discuss:
– the project's structure:
+++ components, hooks, styles, navigation, AsyncStorage, testing
+++ application packaging and hosting (Expo)
+++ more explanation regarding the updating generate sudoku algorithm

– the implemented features:
+++ game reset to initial grid
+++ step back feature
+++ validate grid feature
+++ automatically solve grid feature
+++ history feature = thinking of adding final change, delete previous games feature
+++ resume previous game feature

– the challenges I faced:
+++ fixing the puzzle generation algorithm
+++ AsyncStorage and navigation testing
+++ the application is not perfect, I can handle time tracking better

From the meeting with my supervisor:
- I have to write additional tests for each small component, this should take about a day as I have covered most of the application with console.log() tests. Therefore, I do not expect any new bugs or errors.
- I have to add a section in the report regarding: the testing of the application / the application's structure / the implemented features / the challenges I faced / professional issues / possible future improvements
- For the demo day, all I need to prepare is a working version of the application, either on

my phone or on the emulator. I will also prepare a poster with the application's features and the technologies used.

For now, I am doing good with writing the report, I will finish testing today/tomorrow and all will be good for the submission.

I will be 'mocking' functionalities. Completed testing all important functionalities and utils functions.
This ensures the application's stability, and I am ready for the submission.
I might have missed some corner cases in the tests, but I am confident in my work and the thorough (console.log) testing during the development process.

I am happy with the result!
I will give myself time in the next few days to go over the report again and make sure everything is in place.

I have uploaded the final version of the application on the Expo Dev server and this is the Android APK: `https://expo.dev/artifacts/eas/2Nr9pawmLtQdvR3KYBiPUK.apk`