University of Nottingham

School of Computer Science

Solving Advanced Puzzles with AI Methods

Interim Report

December 30, 2024

Submitted by: Masud Sheikh
Student ID:20390915

Supervisor: Radu Muschevici

Table of Contents

1 Introduction 3
2 Motivation 4
3 Image Preprocessing 5
3.1 Grid Detection 7
3.2 OCR Integration 7
3.2.1 Cell Extraction: 8
3.2.2 Text Extraction Using ML Kit OCR: 8
3.2.3 Post-Processing: 8
3.2.4 Error Handling and Confidence Scores 8
4 Solving the puzzles 8
4.1 Solving Sudoku using backtracking 9
4.2 Solving Nonogram using constraint propagation 9
4.2.1 Initial Setup 10
4.2.2 Propagation of Row and Column Constraints 10
4.2.3 Overlap and Intersection of Constraints 10
4.2.4 Deduction of Certain Cells 10
4.2.5 Iterative Process 11
4.3 Solving Kakuro using Candidate Filtering 11
4.3.1 Initialization 11
4.3.2 Candidate Filtering Process 11
4.3.3 Backtracking Process 11
4.3.4 Example of Constraints 12
5 Design 12
6 Implementation 14
6.0.1 Puzzle Scanning and Recognition 14
6.0.2 Solving Algorithms 14
6.0.3 Storage and Authentication 14
6.0.4 User Interface 15
6.0.5 Testing and Deployment 15
7 Progress 16
7.1 Project Management 16
7.1 Contributions and Reflections 17
8 References 18

1 Introduction

Puzzles such as Sudoku, Nonogram, and Kakuro are well known puzzles for their reliance on logical reasoning and problem-solving skills. Each of these puzzles presents a unique challenge within a grid-based structure, requiring solvers to deduce solutions by satisfying a series of constraints:

Sudoku: The puzzle consists of a 9x9 grid, further divided into nine 3x3 sub grids. The objective is to fill the grid with the numbers 1-9 such that no number repeats within any row, column, or 3x3 sub grid. Sudoku puzzles are typically provided with some pre-filled numbers as clues. Solving them involves deducing the placement of the remaining numbers based on logical reasoning. Variations of Sudoku exist, such as larger grid sizes or additional constraints, but the fundamental challenge remains the same: satisfying the placement rules without conflicts (Delahaye, 2006).

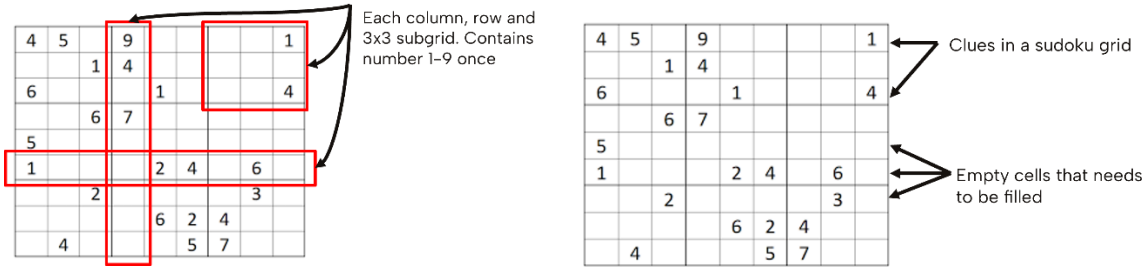

Figure 1.1 Sudoku Grid

Nonograms: These are also known as "Picross" or "Griddlers" and combine logic with creativity. The puzzle consists of a blank grid with numerical clues provided alongside each row and column. These clues indicate the lengths of contiguous groups of shaded cells in that row or column. For instance, a clue of "3 1" means there's a group of three shaded cells followed by at least one unshaded cell, then a single shaded cell. The goal is to shade the appropriate cells to reveal a hidden picture while adhering to the constraints. The challenge often lies in interpreting and reconciling the clues from both rows and columns simultaneously (Oosterman, 2017).

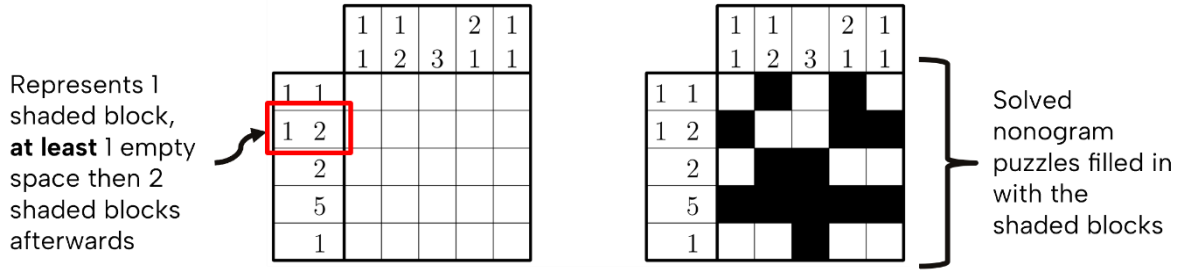

Figure 1.2 Nonogram grid

Kakuro: Sometimes referred to as "cross-sum puzzles," Kakuro blends arithmetic and logical reasoning. The grid resembles a crossword puzzle, but instead of letters, it is filled with numbers. Some grid cells contain clues—numbers that represent the total sum of the digits to be placed in the connected empty cells. For example, a clue of "9" in a row indicates that the digits in the adjacent empty cells must sum to 9, with no digit repeating. The same rule applies to columns. Solving Kakuro requires both mathematical calculations and logical deduction, as solvers must figure out which combinations of digits meet the sum constraints without duplication (Ruepp O. and Holzer M., 2010).

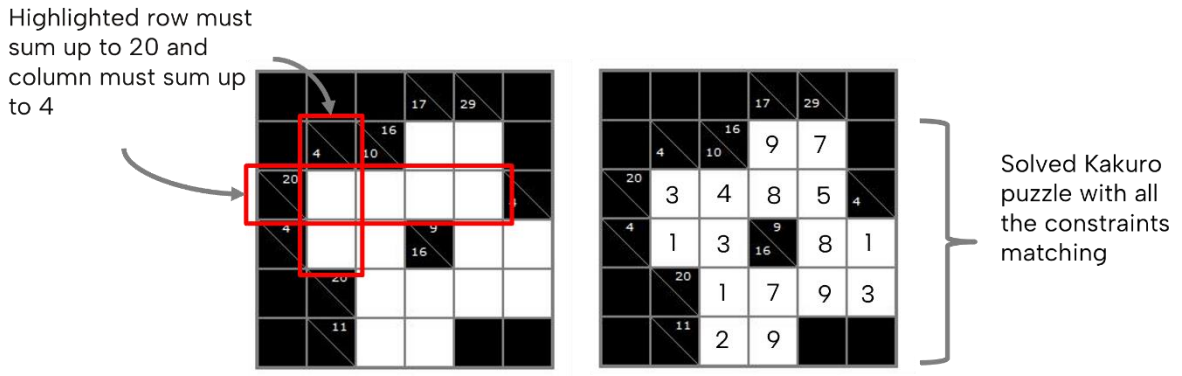

Figure 1.3 Kakuro Grid

These puzzles belong to a class of problems known as **constraint satisfaction problems (CSPs)**, a class of problems that require assigning values to variables while satisfying specific constraints. Many CSPs, including these puzzles, fall into the category of **NP-complete problems** in computational complexity theory. An NP-complete problem is one for which a solution can be verified in polynomial time but finding the solution may require exponential time in the worst case. This classification highlights the inherent difficulty of these puzzles, especially as their size or complexity increases.

For example, Sudoku has been proven to be NP-complete for generalized versions where the grid size NxN is arbitrary, although standard 9x9 Sudoku puzzles can often be solved efficiently due to their constraints and the availability of clues. Similarly, Nonograms and Kakuro are also NP-complete when generalized to larger or more complex grids. This complexity arises from the combinatorial explosion of possible solutions as the grid size and the number of constraints grow.

Solving these puzzles involves fundamentally different approaches depending on the solver. Human solvers often rely on intuition, pattern recognition, and heuristic reasoning to narrow down the search space, making progress through logical deduction. In contrast, computers can systematically explore all possible solutions using brute-force methods. While brute force is effective for smaller puzzles, it becomes impractical for larger or more complex instances due to the exponential growth in possibilities. To handle this challenge, efficient algorithms such as backtracking and constraint propagation are employed to address the computational demands of solving NP-complete puzzles.

To make solving these puzzles more accessible, this project involves developing a **Kotlin-based mobile app** capable of recognizing puzzle grids through image recognition and solving them using artificial intelligence. The app leverages the device's camera to capture puzzle images, processes the visual data to extract the grid, and applies tailored algorithms to solve the puzzle efficiently.

2 Motivation

Puzzles like Sudoku, Nonogram, and Kakuro have long been cherished for their ability to engage the mind and improve problem-solving skills. These puzzles are not only entertaining but also serve as valuable exercises in logic, reasoning, and arithmetic. While numerous digital tools and apps exist to solve individual puzzles, such as Sudoku solvers or Nonogram interpreters, there is a notable gap in solutions that integrate all three puzzle types into a single application.

This project aims to address this gap by developing an app that supports solving Sudoku, Nonograms, and Kakuro within the same platform. By combining these distinct puzzle types, the app offers users a unified and convenient experience, eliminating the need to switch between multiple apps to solve different puzzles.

In addition to solving these puzzles, the project seeks to analyse the computational approaches required for each type. Although all three puzzles fall into the category of NP-complete problems, they vary significantly in terms of structure, constraints, and solving strategies. Exploring these differences provides an opportunity to evaluate how computational complexity and solving methodologies affect the speed and efficiency of solutions.

The app's AI-driven solving mechanisms will employ advanced techniques like constraint propagation, backtracking, and optimization heuristics to solve each puzzle type efficiently. Exploring how AI adapts to the unique constraints of Sudoku, Nonograms, and Kakuro will provide valuable insights into the interplay between puzzle structure, computational complexity, and solution speed.

This motivation stems from the desire to create not only a functional and innovative tool for puzzle enthusiasts but also to contribute to the understanding of how different constraint satisfaction problems can be tackled using modern technologies like image recognition and artificial intelligence. By incorporating image recognition into the app, users can capture puzzles from physical sources, such as magazines or books, and have them solved digitally. This feature further enhances accessibility and convenience for users.

Ultimately, the project combines practical utility with theoretical exploration, aiming to deliver a versatile puzzle-solving app while deepening insights into the nature of these popular puzzles and their computational demands.

3 Image Preprocessing

The initial stage of the project is the preprocessing phase which transforms the raw input image into a suitable format for subsequent grid detection. The main steps are:

1. **Grayscale Conversion:** The input image is converted to grayscale to reduce computational complexity and enhance edge detection. Grayscale conversion simplifies the image by retaining only intensity information, discarding unnecessary colour data.

$$I_{gray}(x,y) = 0.2989 \cdot R(x,y) + 0.5870 \cdot G(x,y) + 0.1140 \cdot B(x,y)$$

where R(x,y),G(x,y),B(x,y) are the red, green, and blue intensities, respectively.

Fig 3.1: On the left is a picture without grayscale and on the right is a picture with grayscale

2. **Noise Reduction:** To improve edge detection and grid recognition, noise is suppressed using Gaussian blur. The Gaussian filter smoothens the image, reducing high-frequency noise:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

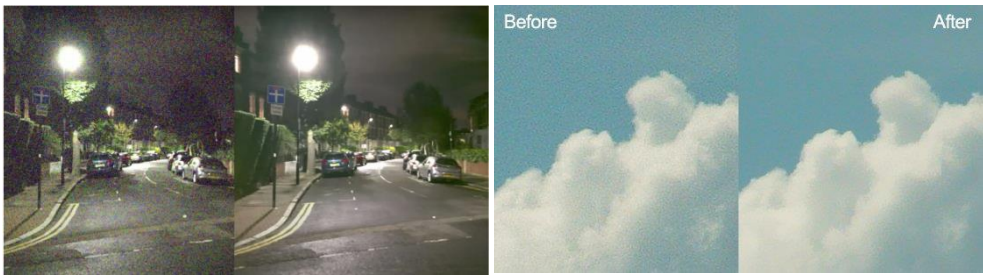

Fig 3.2 On the left of both images is a picture with a lot of noise and on the right a picture with much less noise

3. **Edge Detection:** The Canny edge detection algorithm is applied to identify prominent edges in the image. The algorithm uses gradient-based intensity analysis and non-maximum suppression to detect edges:

    - Compute gradients:
    - Apply hysteresis thresholding to segment edges.

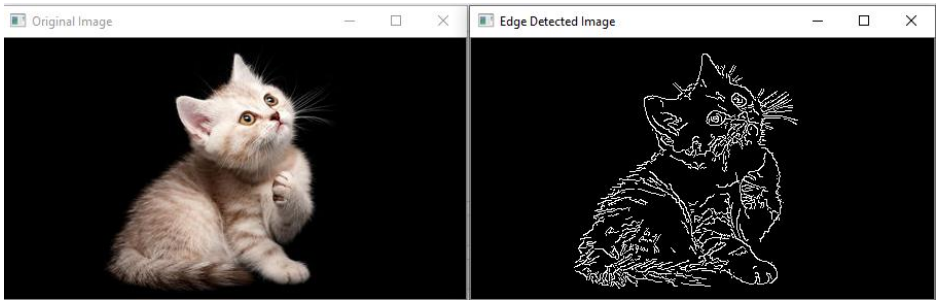

Fig 3.3 On the left is a normal image and on the right is the edge detected image

4. **Perspective Correction:** After detecting edges, the perspective of the image is corrected to align the grid with the camera's plane. This is achieved using the Hough Line Transform to identify the boundary lines of the grid, followed by warping the image into a top-down perspective using a homography matrix.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where (x′,y′) are the corrected coordinates.

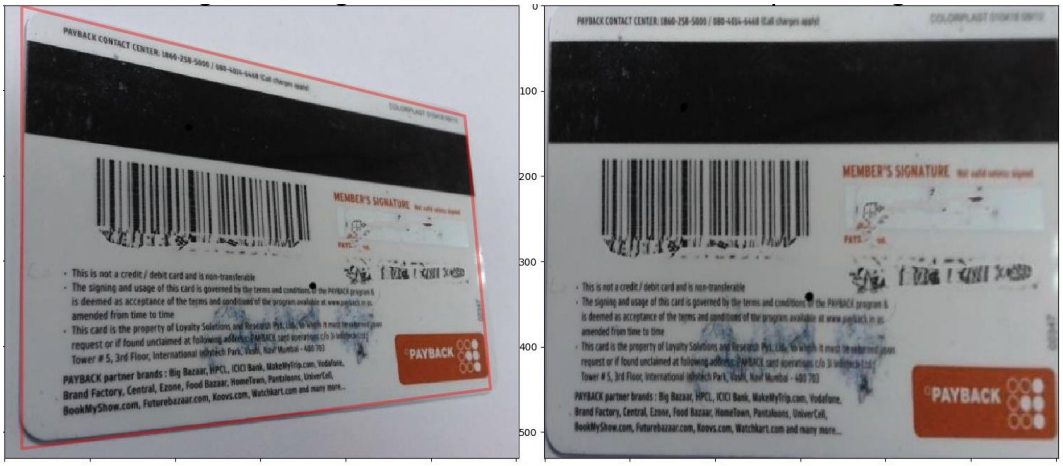

Fig 3.4 The image before and after its perspective is corrected

3.1 Grid Detection

The grid lines, now highlighted, are detected using the Hough Line Transform. This method identifies straight lines by transforming points in the Cartesian plane to sinusoidal curves in Hough space. A line in the image is represented by:

$$\rho = x \cos\theta + y \sin\theta$$

where ρ is the perpendicular distance to the origin, and θ is the angle of the normal. Peaks in the Hough accumulator correspond to grid lines. The intersection points of these lines define the grid cells.

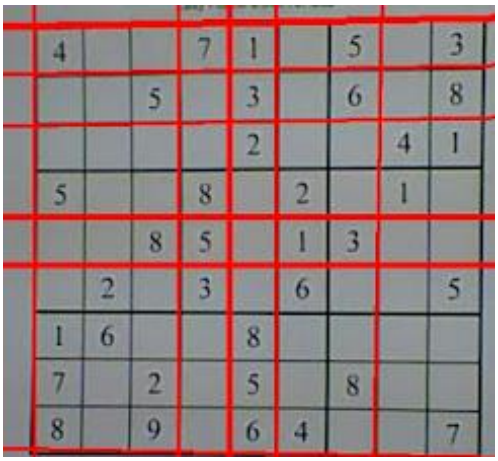

Fig 3.5 Image with grid lines detected

3.2 OCR Integration

Once the grid cells are extracted, Optical Character Recognition (OCR) is used to interpret the contents of each cell. For this purpose, Google's ML Kit OCR API is integrated, which employs machine learning models to recognize text and numbers.

3.2.1 Cell Extraction:

Each cell is cropped from the aligned grid. To improve OCR accuracy, the following preprocessing steps are applied:

- **Resizing**: Cells are resized to a uniform size (N×N),like 28×28 or 64×64 pixels, to match the expected input size for OCR.
- **Binarization**: Further thresholding ensures a clear contrast between text and background.
- **Morphological Operations**: Erosion and dilation are applied to enhance the readability of digits or clues.

3.2.2 Text Extraction Using ML Kit OCR:

The pre-processed cells are passed through the ML Kit OCR API. This involves:

- **Character Segmentation**: Individual characters within the cell are segmented using bounding box detection.
- **Recognition**: Each segmented character is classified using convolutional neural networks (CNNs) trained on a diverse dataset of handwritten and printed digits.

3.2.3 Post-Processing:

The OCR output is validated and corrected using domain-specific constraints for:

- Sudoku, recognized digits must lie in the range {1,2,...,9}.
- Kakuro, digits must satisfy the sum constraints.
- Nonograms, numeric clues are cross-referenced with the puzzle grid layout.

3.2.4 Error Handling and Confidence Scores

Confidence scores for each recognized character are generated. If the score is below a predefined threshold, the cell is flagged for manual intervention or reprocessing. Additionally, heuristic rules are applied to correct misrecognized characters, leveraging the logical constraints of the puzzle.

4 Solving the puzzles

To solve puzzles like Sudoku, Kakuro, and Nonogram, several techniques can be applied to navigate the constraints and narrow down the possible solutions. The problem is essentially a **constraint satisfaction problem** (CSP). These methods share common principles but are adapted to suit the unique characteristics of each puzzle type. The methods discussed below—backtracking, constraint propagation, and candidate filtering—are central to solving these puzzles and can be used in combination or independently. There may also be overlap in their application, depending on the puzzle's complexity and structure.

Techniques such as plain brute force, which involves enumerating all 9^{81} possible configurations, is computationally infeasible due to the exponential growth of possibilities (Felgenhauer B. and Jarvis F., 2006). Backtracking, on the other hand, uses the constraints to prune invalid branches early, significantly reducing the search space. This logical approach ensures efficiency and is particularly effective for standard Sudoku puzzles with sufficient clues(S. S. Skiena, 2008).

Constraint propagation further enhances efficiency by iteratively enforcing constraints to simplify the problem. Often triggering cascading effects that resolve additional variables. This process reduces the problem's complexity in stages, minimizing the need for exhaustive search (Bessiere C., 2006).

Candidate filtering maintain and refine a list of possible values for each variable based on the constraints. For puzzles like Kakuro, constraints such as sum requirements or alignment rules are used to eliminate invalid options early, streamlining the solving process.

Backtracking, constraint propagation, and candidate filtering form a powerful toolkit for solving these puzzles efficiently, leveraging logical deductions to manage the exponential complexity inherent in constraint satisfaction problems.

4.1 Solving Sudoku using backtracking

The algorithm starts from the first empty cell and attempts to assign a number from 1 to 9**.** Exploration - It moves sequentially from one empty cell to the next, assigning values that satisfy the Sudoku constraints. If a number cannot be assigned to a cell (i.e. all values lead to conflicts), the algorithm backtracks to the previous cell and tries the next available candidate (Indriyono B. 2023).

The Sudoku puzzle is essentially a constraint satisfaction problem. The constraints ensure that the solution is valid:

Row Constraint: A number must not repeat in any row. Mathematically:

$$x_{ij} \neq x_{ij'}$$

for all $j' \neq j$, where i is fixed.

Column Constraint: A number must not repeat in any column. Mathematically:

$$x_{ij} \neq x_{i'j}$$

for all $i' \neq i$, where j is fixed.

Sub grid Constraint: A number must not repeat in the same 3x3 sub grid. Mathematically:

$$x_{ij} \neq x_{i'j'}$$

where i' and j' are indices of cells within the same 3x3 subgrid as i, j.

These constraints reduce the number of valid candidates for each cell, significantly limiting the search space.

For a standard Sudoku puzzle with a reasonable number of given clues, backtracking can solve the puzzle in a matter of milliseconds.

4.2 Solving Nonogram using constraint propagation

The Nonogram puzzle solving algorithm works by iteratively narrowing down the possibilities for each cell in the grid using the row and column clues. These clues specify the lengths and positions of contiguous blocks of shaded cells, and the goal is to place these blocks while avoiding contradictions in the rows and columns (Wu, 2013).

4.2.1 Initial Setup

The grid consists of cells that can either be unshaded (empty) or shaded (filled). Each row and column in the grid is associated with a set of clues, which define the length and order of contiguous shaded blocks. The solver begins by analysing these clues to deduce possible placements of shaded and empty cells.

Let's define the variables $\mathbf{x_{ij}}$ to represent the state of a cell in the grid. The grid has i rows and j columns, where:

- $\mathbf{x_{ij}}$ = 1 if the cell is shaded.
- $\mathbf{x_{ij}}$ = 0 if the cell is unshaded (empty).

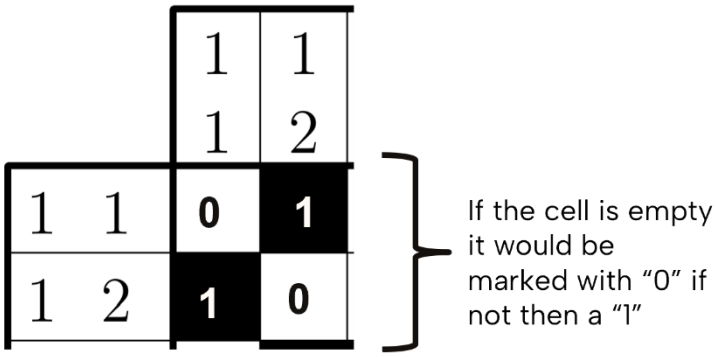

Fig 4.1 Nonogram grid 'marked'

4.2.2 Propagation of Row and Column Constraints

The core idea of constraint propagation in Nonograms is to iteratively apply row and column clues to narrow down the possible values for $\mathbf{x_{ij}}$ in each cell.

For each row i, the clues specify one or more blocks of shaded cells. The solver starts by placing blocks of shaded cells $\mathbf{x_{ij}}$ =1 and unshaded cells $\mathbf{x_{ij}}$ =0 based on the row clues. For example:

- If the row has the clue "3 1", the solver places a block of three shaded cells followed by at least one unshaded cell, then places a shaded cell. This placement eliminates impossible configurations for $\mathbf{x_{ij}}$ in that row.

Similarly, for each column j, the clues specify the lengths and positions of shaded blocks. The solver applies these clues to deduce possible placements for shaded cells $\mathbf{x_{ij}}$ =1 and unshaded cells $\mathbf{x_{ij}}$ =0 in the column, ensuring consistency with the row constraints.

4.2.3 Overlap and Intersection of Constraints

The key insight in solving Nonograms is that row and column clues intersect, and constraints from one direction can help inform the other. As the solver processes a row or column, it propagates the constraints to the intersecting rows and columns.

- For instance, when filling a row i, if a clue specifies that a block of shaded cells must occupy certain positions, the solver can use this information to adjust the constraints on the intersecting column j. If a column has a partial fill (i.e., certain cells are known to be shaded or empty), this can further refine the possible values of $\mathbf{x_{ij}}$.

4.2.4 Deduction of Certain Cells

When a block of shaded cells is placed in a row or column, certain cells in the grid become definitively shaded or unshaded. These cells can be marked as $\mathbf{x_{ij}}$ =1 (shaded) or $\mathbf{x_{ij}}$ =0 (unshaded),

and their state is propagated to intersecting rows and columns. This process reduces the number of possible configurations for the intersecting areas and helps the solver progress.

4.2.5 Iterative Process

Constraint propagation is an iterative process:

1. The solver places a block of shaded cells in a row or column based on the clues.
2. The solver revisits intersecting rows and columns to apply the updated information.
3. This leads to more cells becoming definitively filled or empty, providing further constraints for the remaining cells.

This process continues iteratively until the puzzle is solved or a contradiction is found. The Nonogram solver narrows down the possibilities for each cell $\mathbf{x_{ij}}$ and fills the grid in a logical, efficient manner. This method ensures that no contradictions arise between rows and columns while filling out the puzzle, leading to faster and more reliable solutions.

4.3 Solving Kakuro using Candidate Filtering

4.3.1 Initialization

Start by identifying all empty cells, $\mathbf{x_{ij}}$, in the Kakuro grid and the corresponding sum constraints for each row and column.

4.3.2 Candidate Filtering Process

1. **Row and Column Sum Constraints**: For each row **i** and column **j**, the algorithm first calculates all valid combinations of numbers from 1 to 9 that can fit the sum requirement for the given number of cells. For example, if row **i** has 4 cells (x_{i1}, x_{i2}, x_{i3}, x_{i4}) and a sum clue of 22, the algorithm generates all valid combinations of 4 distinct digits from 1 to 9 that sum up to 22.
2. **Uniqueness Constraint**: Once valid combinations are generated for the row **i** and column **j**, the uniqueness constraint is applied. This means that no number can appear more than once in any row or column, reducing the valid candidates for each empty cell $\mathbf{x_{ij}}$. If a number is already used in the same row or column, it is eliminated as a possible candidate for other cells $\mathbf{x_{ij}}$ in that row or column.
3. **Filtering Process**: For each empty cell $\mathbf{x_{ij}}$:
    - **Eliminate candidates that violate the row sum constraint**: The sum of the numbers in row **i**, including the current candidate $\mathbf{c_{ij}}$, must match the row's sum constraint.
    - **Eliminate candidates that violate the column sum constraint**: The sum of the numbers in column **j**, including the current candidate $\mathbf{c_{ij}}$, must match the column's sum constraint.
    - **Eliminate candidates that violate the uniqueness constraint**: If a number is already present in the same row or column, it is removed as a possible candidate for $\mathbf{x_{ij}}$.

4.3.3 Backtracking Process

Once the candidate filtering is complete, the algorithm attempts to fill the grid:

- **Assignment**: For each empty cell $\mathbf{x_{ij}}$, pick a candidate $\mathbf{c_{ij}}$ from the filtered list.
- **Propagation**: After assigning a value $\mathbf{c_{ij}}$, update the row and column constraints and remove this value $\mathbf{c_{ij}}$ from the possible candidates for other cells in the same row and column.

- **Conflict Detection**: If no valid candidates are available for a given cell x_{ij}, the algorithm backtracks to the previous cell x_{i-1j-1} and tries the next available candidate.
- **Termination**: The algorithm continues this process until all cells x_{ij} are filled, or it reaches a dead end where no valid assignments are possible (in which case, it backtracks further).

4.3.4 Example of Constraints

Row Constraint: For a row with three empty cells x_{i1}, x_{i2}, x_{i3} and a sum clue of 23, valid candidate combinations for those three cells must sum to 23 and consist of distinct digits between 1 and 9. For example, the combination [6, 8, 9] might be valid, but [7, 7, 9] would not, due to repetition.

Column Constraint: Similarly, a column with a sum clue of 17 for three empty cells x_{i1}, x_{i2}, x_{i3} must have distinct values from 1 to 9 that sum to 17.

Uniqueness Constraint: If the number 8 is placed in cell x_{ij}, it cannot appear again in the same row or column, thus eliminating it from the candidate list for other cells x_{ij} in the same row and column.

5 Design

Puzzles such as Sudoku, Minesweeper, Nonograms, and Kakuro are inherently well-suited for mobile platforms because they are engaging and enjoyable activities that people can solve during their free time. A mobile app provides a convenient and accessible medium for users to interact with these puzzles on the go, offering advantages in portability and immediacy compared to a web-based application.

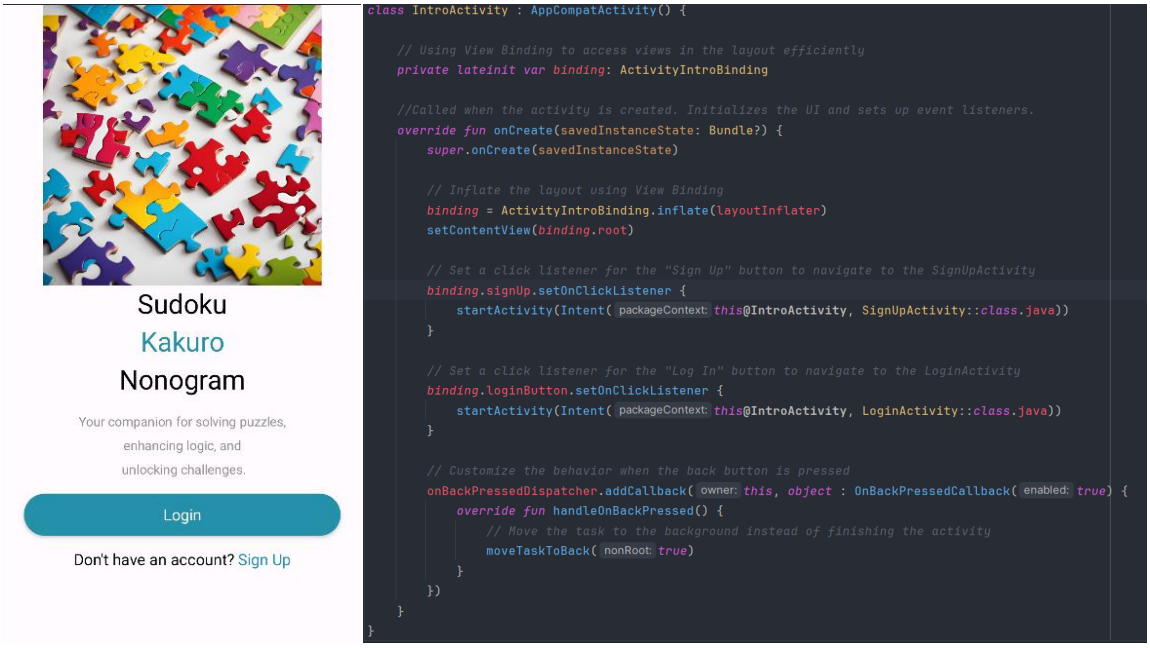

Fig 5.1 Splash screen

The app incorporates a data storage system that allows users to keep a record of all puzzles they have scanned and solved. By maintaining a history, users can revisit past puzzles, analyse their problem-solving journey, and track their progress over time. To enable this, the app requires a dedicated login system and individual sessions for each user. This is implemented using Firebase Authentication which is a secure industry-standard solution. Additionally, Firebase's cloud-based storage system

ensures that user data is synchronized across devices, providing access to puzzle history even if the app is used on multiple devices or reinstalled.

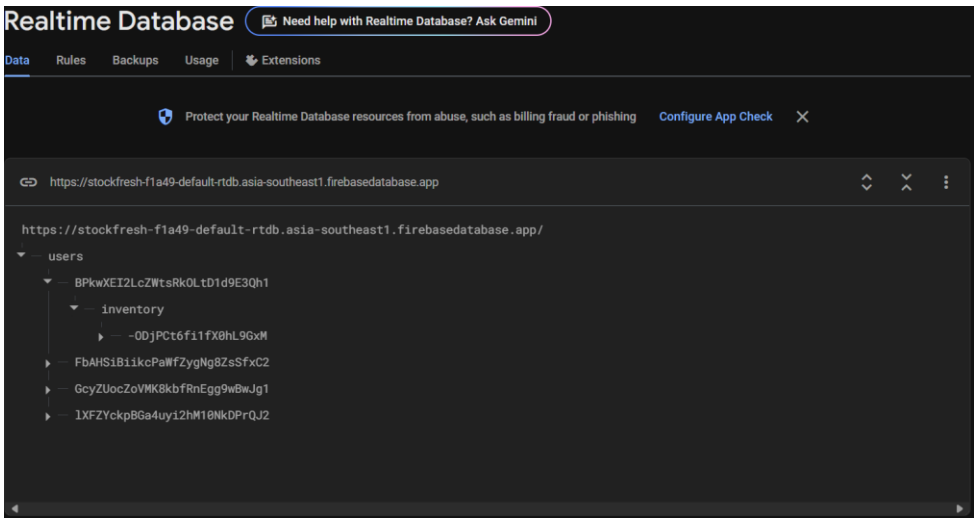

Fig 5.2 Database Dashboard

The app's user interface is designed with simplicity and functionality in mind. The layout ensures that users can easily navigate through different puzzle options—Sudoku, Minesweeper, Nonograms, and Kakuro—from a unified home screen. Consistent visual elements, including typography, colour schemes, and button styles, are maintained throughout the app to deliver a cohesive and user-friendly experience.

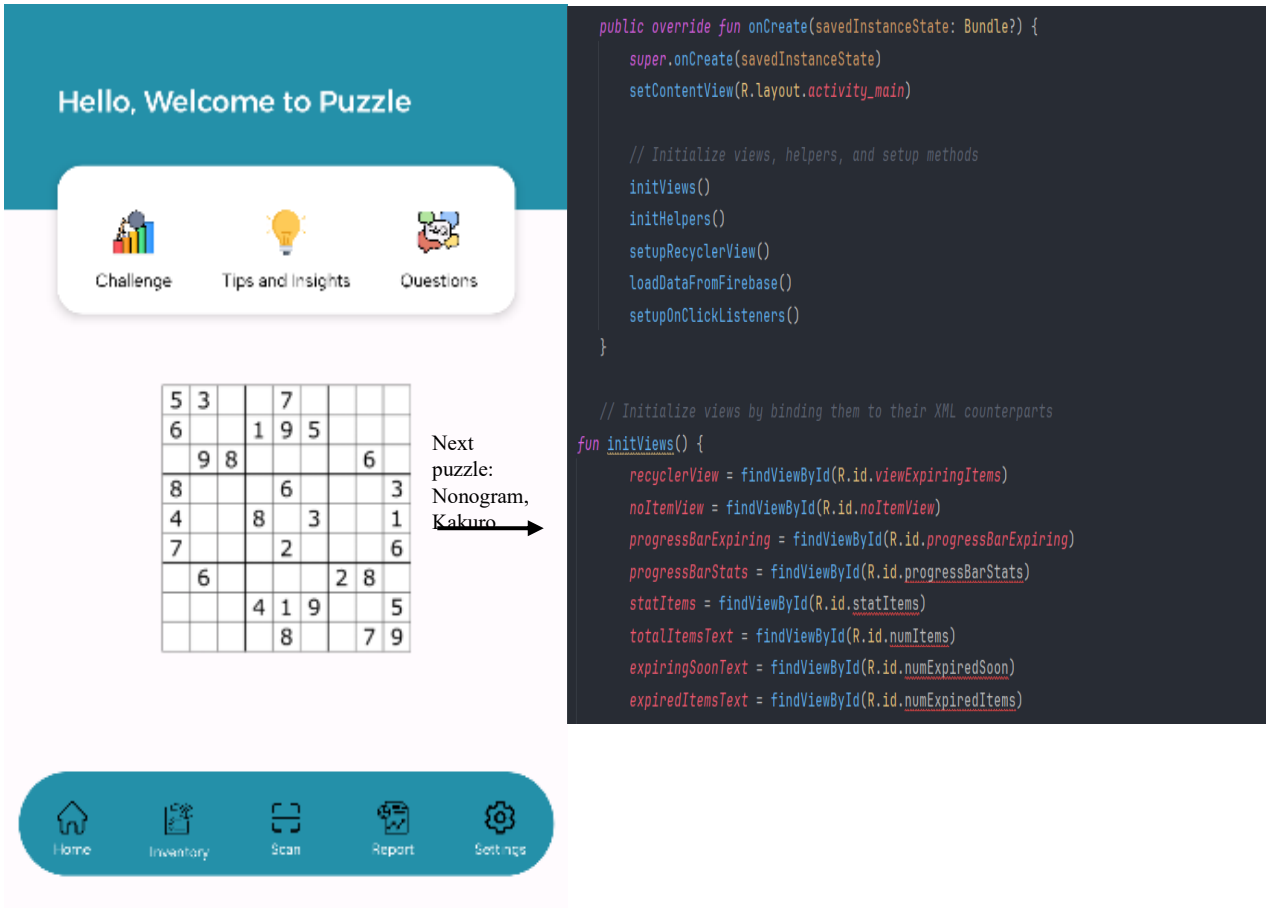

Fig 5.3 Dashboard screen

Kotlin was chosen as the development language for its numerous advantages over older languages like Java. It offers a modern syntax, enhanced null safety, and more concise code, enabling faster development and reducing the likelihood of errors. While Java has been the traditional choice for Android development, its verbosity and slower runtime are addressed effectively by Kotlin. Furthermore, Kotlin maintains full compatibility with Android's extensive ecosystem, making it an ideal choice for mobile app development on the Android platform, which powers over 80% of smartphones worldwide.

6 Implementation

The implementation of the puzzle-solving app revolves around several key components: puzzle scanning and recognition, solving algorithms, data storage, user authentication, and the user interface. Each of these is developed to ensure smooth functionality and a seamless user experience.

6.0.1 Puzzle Scanning and Recognition

The app leverages advanced image processing libraries such as OpenCV to enable the scanning of puzzles directly from user uploaded images. Images captured by the user are processed to enhance clarity and detect puzzle boundaries. This includes steps like grayscale conversion, noise reduction, edge detection and perspective correction. Google's ML Kit OCR API are integrated to extract numbers or puzzle elements from the image. The scanned puzzle is divided into cells to determine which numbers are already filled in and identify empty cells.

6.0.2 Solving Algorithms

Each puzzle type has a dedicated solving algorithm For Sudoku, a backtracking algorithm is implemented to recursively fill empty cells while adhering to the puzzle's constraints.

Nonograms, the app employs constraint propagation. This approach works by iteratively applying the known row and column constraints to eliminate impossible values, progressively narrowing down the possibilities.

For Kakuro, the app utilizes candidate filtering, which is a technique that filters out impossible candidate numbers for each cell based on the rules of the game. The app begins by populating each cell with a list of potential candidate numbers that could satisfy the constraints of the row and column.

6.0.3 Storage and Authentication

User data, including puzzle history and personal preferences, is securely stored in Firebase's cloud databases. Each user's data is organized in a structured format, ensuring it is easily accessible while maintaining a high level of security. Puzzle records, such as solved puzzles, their timestamps, and solutions, are stored for future access. This allows users to revisit past puzzles, analyse their progress, and track their puzzle-solving journey. Additionally, the cloud-based storage ensures that users' data is automatically synchronized across multiple devices. Whether a user switches devices or reinstalls the app, they can seamlessly access their puzzle history and preferences from anywhere.

To ensure secure access and personalised user experiences, Firebase Authentication is integrated into the app. Users can create accounts using their email addresses or log in with their existing credentials. For added convenience, Google Sign-In is also supported, allowing users to log in using their Google accounts. This provides flexibility while maintaining security. Secure session management is achieved using authentication tokens, ensuring that users have a smooth and secure login and logout experience.

6.0.4 User Interface

The UI is implemented using Jetpack Compose, Kotlin's modern toolkit for building native UIs. Key aspects include:

- **Home Screen**: A dashboard displaying all available puzzle options, with easy-to-access buttons and clear navigation.
- **Puzzle Scanner**: An integrated camera interface for capturing puzzle images. Real-time feedback guides users during scanning.
- **Puzzle Solver Interface**: Displays the original puzzle and its solution side-by-side, with options to step through the solving process for better understanding.
- **History Screen**: A list of previously solved puzzles, allowing users to revisit or analyse their past attempts.

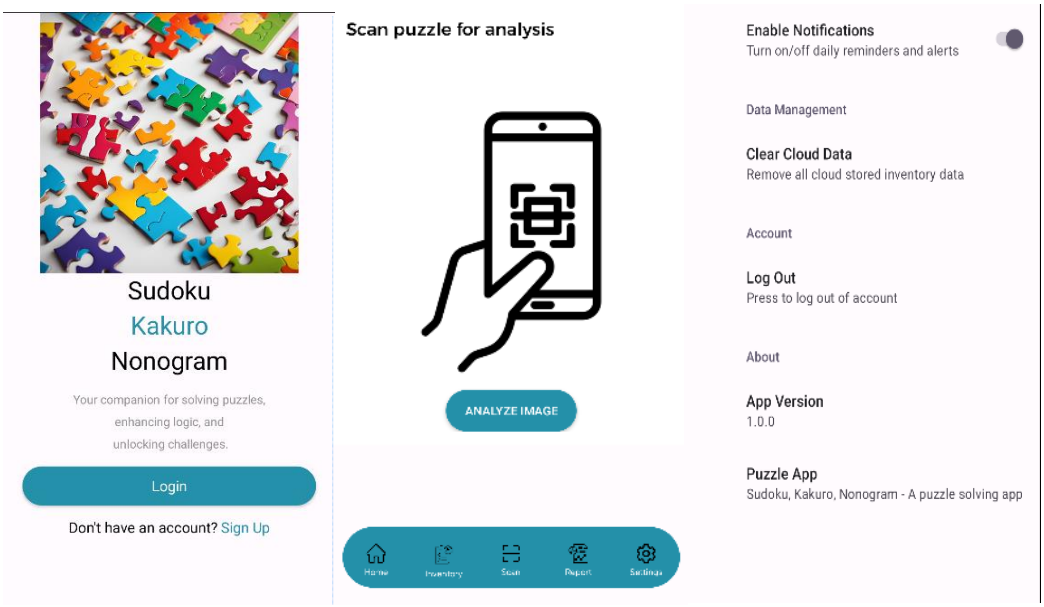

Fig 5.4 Snippets of the puzzle app

6.0.5 Testing and Deployment

Testing and deployment are crucial steps in ensuring the app functions as expected and provides a seamless user experience. Unit testing is conducted using JUnit to verify the accuracy and robustness of the core algorithms and key functionalities, ensuring they perform correctly under various conditions. Integration testing is also carried out to evaluate the app's integration with Firebase, image recognition capabilities, and puzzle-solving workflows. This ensures that all components work together seamlessly across different devices. Once testing is complete, the app is packaged as an APK and uploaded to the Google Play Store for public distribution. Additionally, regular updates and bug fixes are planned based on user feedback, ensuring that the app remains stable, user-friendly, and continuously improved.

7 Progress

7.1 Project Management

For this project, I have been following an agile development methodology, which includes the use of sprints and kanban boards to manage tasks and track progress. “To do,” “In progress,” and “Done.” Each column is filled with visual cards that represent individual tasks. Essential details such as the task title, description, due date and urgency.

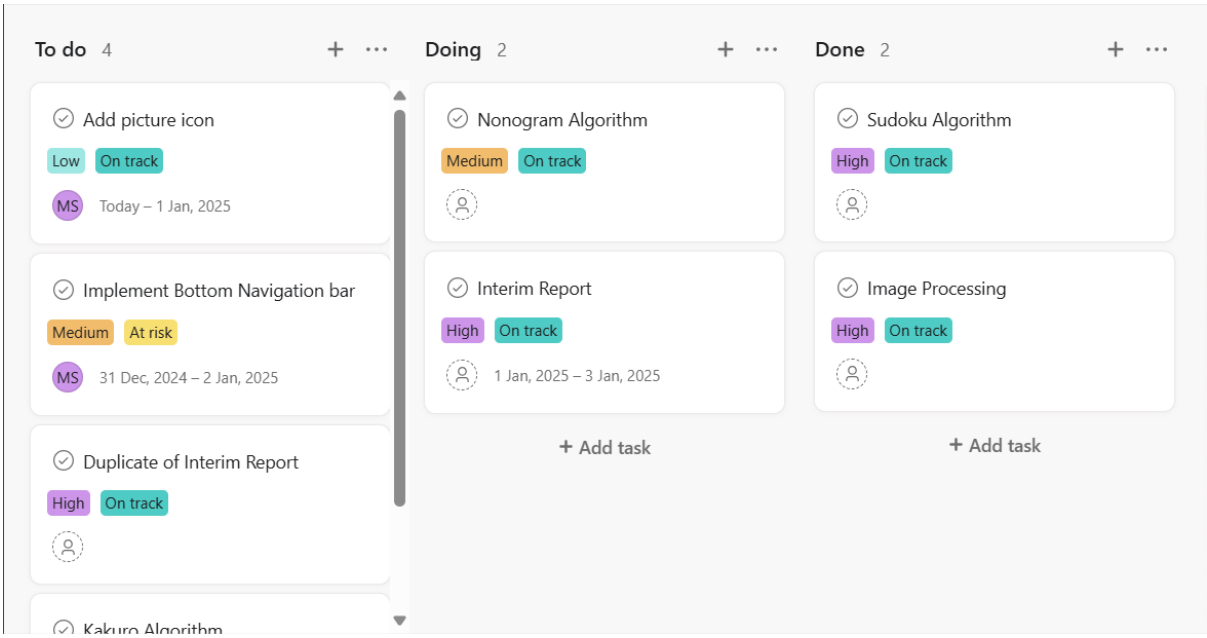

Fig 7.1 Kanban board with “To do”, “In progress” and “Done”

The project was structured into multiple **sprints**, each spanning a period of two weeks. This approach provided a clear timeline for achieving milestones while ensuring flexibility to adapt to unforeseen challenges. Each sprint was assigned specific goals, categorized based on project phases, such as image recognition, algorithm development, integration, and testing.

At the conclusion of each sprint, I evaluated the following aspects:

- **What went well?** Highlighting successes, such as the implementation of algorithms or advancements in user interface design.
- **What could be improved?** Addressing challenges, such as delays or technical issues that may have impacted progress.
- **Actions for the next sprint**: Outlining focus areas, such as increasing testing efforts or breaking tasks into smaller, more manageable units for improved efficiency.

The initial phase of the project focused on developing the image recognition component to ensure that the fundamental functionality of recognizing and processing puzzle images was achievable. This stage was crucial as it allowed me to test and validate the app's ability to interpret puzzle images accurately. By prioritizing image recognition, I was able to address potential challenges related to varying puzzle formats and image quality early on. This approach proved effective as it provided valuable insights into the technical difficulties I might encounter later in the implementation phase. By tackling the image recognition first, I was able to refine my approach for integrating the puzzle-solving algorithms later on, ensuring a smoother development process for the overall app.

This transition allowed me to build on the foundation established by the image recognition system, ensuring that the app not only identified puzzles but could also solve them correctly.

The algorithm development followed a structured approach, starting with simpler puzzle types such as Sudoku would allow me to progressively move to more complex ones like Kakuro and Nonogram in the near future.
However, moving forward what I need to work on is shifting my focus toward working on the comparison between puzzles first rather than polishing the user interface (UI) first, as initially suggested. Comparing the puzzles in terms of speed to solve, strategy utilised would allow me to draw meaningful conclusions for the final product.

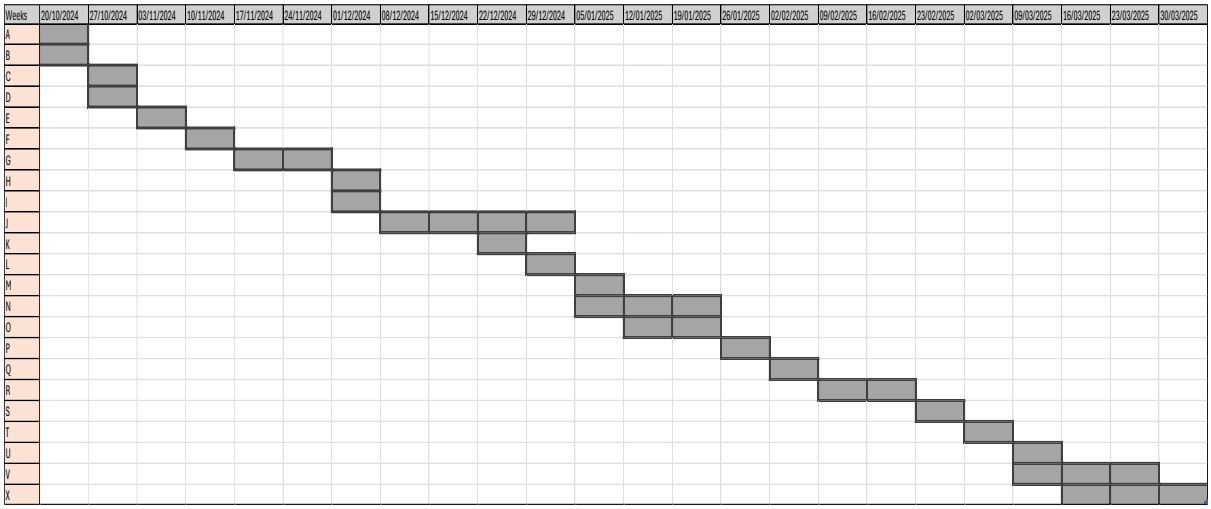

Fig 7.2 Gantt Chart

This Gantt chart provides a comprehensive timeline for the development of the puzzle-solving app, with tasks spread across different stages of the project, from research and design to implementation and testing. The tasks are logically organized to ensure the project progresses smoothly, with each phase building on the previous one. This is reiterated in the project proposal as well.

- **Research and Planning (Tasks A–D)**
- **Data and Image Processing (Tasks E–I)**
- **Algorithm Development (Tasks K–M)**
- **Integration and Testing (Tasks N–P)**
- **Comparison between puzzles(Tasks Q–R)**
- **Final Testing and Evaluation (Tasks S–V)**

7.1 Contributions and Reflections

There is limited material available that explores the intersection of the three puzzles—Sudoku, Kakuro and Nonogram— together making this project quite unique. Creating a hub to solve all three puzzles, the app offers users the opportunity to solve each puzzle we then use that data to compare them and explore the different strategies and challenges that each puzzle presents.

This approach takes a look into to a relatively underexplored area of puzzle-solving, reviving interest in a sector that has somewhat stagnated in recent years. Puzzle games, especially traditional ones, have remained relatively unchanged, and while new variations appear periodically, there hasn't been any significant innovation that brings these classic puzzles together in a meaningful way.

While the project started off with some uncertainty, particularly in selecting the right topic for my final year project. Once I explored this idea, things quickly gained momentum. This shift allowed me to dive into a subject that is not only engaging but also presents ample opportunities for exploration and growth in my skills personally and contribute to society.

The foundation and roadmap for the project has been established, and now it is up to me to stay committed to the plan to deliver a polished final year project by the time of submission. Producing a project that not only meets the requirements but also exceeds expectations. The hard work and consistency over the coming weeks will be crucial in transforming the initial idea into a fully realized app that is both functional and engaging.

8 References

[1] S. S. Skiena, The Algorithm Design Manual, Springer-Verlag, 2nd ed., 2008.

[2] Bessiere, C., 2006. Constraint propagation. In *Foundations of Artificial Intelligence* (Vol. 2, pp. 29-83). Elsevier.

[3] Felgenhauer, B. and Jarvis, F., 2006. Mathematics of sudoku I. *Mathematical Spectrum*, *39*(1), pp.15-22.

[4] Wu, I.C., Sun, D.J., Chen, L.P., Chen, K.Y., Kuo, C.H., Kang, H.H. and Lin, H.H., 2013. An efficient approach to solving nonograms. *IEEE Transactions on Computational Intelligence and AI in Games*, *5*(3), pp.251-264.

[5] Delahaye, J.P., 2006. The science behind Sudoku. *Scientific American*, *294*(6), pp.80-87.

[6] Oosterman, R.A., 2017. *Complexity and solvability of Nonogram puzzles* (Doctoral dissertation, Faculty of Science and Engineering).

[7] Ruepp, O. and Holzer, M., 2010, June. The computational complexity of the Kakuro puzzle, revisited. In *International Conference on Fun with Algorithms* (pp. 319-330). Berlin, Heidelberg: Springer Berlin Heidelberg.

[8] Indriyono, B., Pamungkas, N., Pratama, Z., Mintorini, E., Dimentieva, I. and Mellati, P., 2023. Comparative analysis of the performance testing results of the backtracking and genetics algorithm in solving Sudoku games. *International Journal of Artificial Intelligence & Robotics (IJAIR)*, *5*(1), pp.29-35.