

Sudoku Grid Solver

Massi-Nissa ABBOUD

Fall 2024

Summary

- Introduction
- Project Description
- Implementation
- Decision Rules
- Design Patterns

Introduction

The **Sudoku Grid Solver** project is a software application designed to simulate and solve the popular Sudoku puzzle game. Sudoku is a classic logic-based, combinatorial number-placement puzzle that consists of a 9x9 grid divided into nine 3x3 sub-grids. Each puzzle begins with some cells pre-filled with numbers, and the objective is to fill in the remaining cells so that every row, column, and sub-grid contains each digit from 1 to 9 exactly once.

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

Figure 1: Example of a Sudoku Puzzle Grid

In this project, we developed a Sudoku game that not only allows users to load, display, and interact with a puzzle grid, but also includes an automated solving feature based on decision-making algorithms. These algorithms simulate logical deduction similar to how a human might solve Sudoku puzzles, gradually filling in cells by analyzing potential values for each empty cell and eliminating invalid options.

Project Specifications

0.1 Language specifications

- The project has been developed using python.
- The code does not rely on any external library making it portable and easy to setup.
- The development of the project have been focused on respecting common design principles. We focus on 4 design patterns that we will describe later in another section.

0.2 Technical functionalities:

- The application takes as input a file that follows the following format The file should contain 9 lines, each representing a row in the grid. - Each line should have 9 numbers separated by commas, with values between '1' and '9' for filled cells, and '-1' or '0' for empty cells.
- The application implements a set of deduction rules that are applied on the grid to try to fill the empty cells.
- The decision rules are applied repeatedly until filling all the empty cells or until being stuck without being able to derive any other decision.
- At that point, the application allows the user to manually fill the value of a cell.
- The application allows the user to restart the solving in the case of any inconsistency.
- The application allows the evaluation of a grid based on its difficulty to solve. (bonus)
- The user interacts with the application through a simple console interface.

1 Rules of Sudoku

Sudoku is a logic-based puzzle game played on a 9x9 grid, subdivided into nine 3x3 sub-grids, also known as "boxes" or "regions." The objective of the game is to fill each cell in the grid with a digit from 1 to 9, following specific rules that ensure a unique solution for each puzzle. The rules of Sudoku are simple yet require careful reasoning and deduction to solve the puzzle successfully:

- **Unique Numbers in Rows:** Each of the nine rows in the grid must contain every number from 1 to 9 without repetition. No number should appear more than once in any row.

- **Unique Numbers in Columns:** Similarly, each of the nine columns must also contain every number from 1 to 9, with no duplicates in any column.
- **Unique Numbers in 3x3 Sub-Grids:** Each 3x3 sub-grid, or region, should also include the numbers 1 through 9 exactly once, with no repeated numbers within the same sub-grid.
- **Initial Clues:** Sudoku puzzles start with a set of numbers already placed in some cells. These pre-filled cells, known as “clues,” provide a foundation for solving the puzzle by narrowing down possible values for the remaining empty cells. The difficulty of a puzzle depends on both the number and the arrangement of these initial clues.
- **Solving Logic:** Players use logical deduction to determine which numbers go into empty cells, eliminating possibilities based on the numbers already present in the same row, column, and sub-grid. Through a process of elimination and pattern recognition, players work towards completing the grid according to the rules.

A successfully completed Sudoku puzzle will contain each number from 1 to 9 exactly once in every row, column, and 3x3 sub-grid. The challenge lies in using deductive reasoning to fill in the grid, as no guessing is required in a standard Sudoku puzzle with a unique solution.

2 Implementation

Below are brief descriptions of the classes implemented in the application:

- **GridZone:** Represents a zone in the Sudoku grid (such as a row, column, or square). It maintains a list of cells within the zone and tracks which numbers are present in the zone through a tracker set. It provides methods to add cells to the zone, check if a value is already in the zone, get remaining possible values, and check if the zone is completed. Additionally, it has methods for updating zones when a cell’s value is modified. This class is used as an abstraction to the GridZones described below.
- **Row:** Inherits from GridZone and represents a horizontal row in the Sudoku grid. It adds functionality to handle the row-specific logic but otherwise behaves similarly to a general GridZone.
- **Column:** Inherits from GridZone and represents a vertical column in the Sudoku grid. Like Row, it handles column-specific logic while leveraging the general functionality of GridZone.
- **Square:** Inherits from GridZone and represents one of the nine 3x3 sub-grids in the Sudoku grid. Each square holds the cells that belong to that sub-grid, and it shares the functionality of GridZone to manage the values and integrity of the square.

- **Cell:** Represents a single cell in the Sudoku grid, with a unique id and a val (value). Cells can be empty (denoted by -1) and have associated notes, which are sets of possible values for the cell. The Cell class also manages the interaction between cells and zones, ensuring that values are valid and notifying zones when a value is set or notes are updated. It ensures that cells' values are consistent with the rules of Sudoku and facilitates the backtracking process through its set_value and set_notes methods.
- **Note:** Represents a collection of possible values (notes) for a Cell. This class encapsulates the logic for adding, removing, and managing notes for a cell. It is used by the Cell class to track possible values when the cell's value is unknown, and it is updated dynamically as the grid evolves.
- **Grid:** Represents the entire Sudoku grid. It contains instances of Row, Column, and Square to organize the grid's structure. The Grid class is responsible for managing the state of the Sudoku puzzle, loading the grid from a file, setting values, checking if the puzzle is complete, and displaying the grid. It provides methods to interact with individual cells, ensuring that constraints are respected across all zones. The grid is initialized with empty or predefined values, and it supports functionality for solving the puzzle through decision rules.

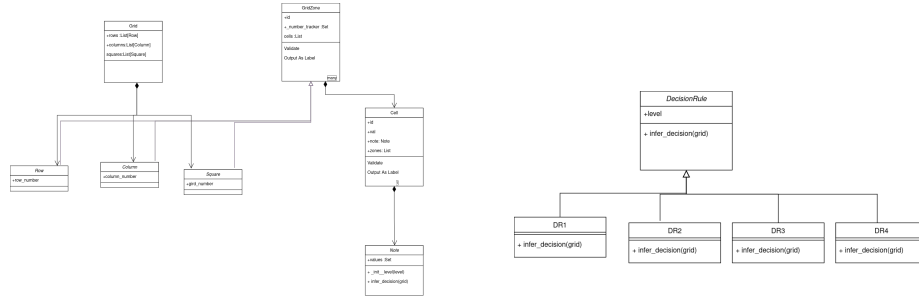


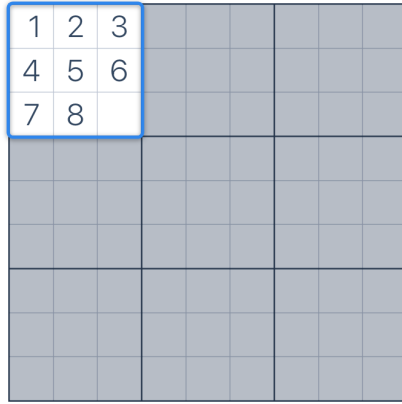
Figure 2: Classes used in the project

3 Decision Rules

The Decision Rules are key techniques applied within the Sudoku solver to progressively infer values for empty cells based on logical constraints. These techniques range from fundamental to advanced strategies, which helps in making the puzzle-solving process efficient. The decision rules implemented in this project include:

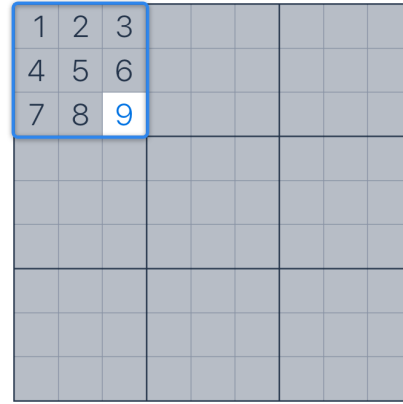
3.1 Decision Rule 1 (DR1): Last Free Cell Technique

- The "Last Free Cell" technique is a basic, straightforward approach for identifying missing values in the grid. This method relies on the Sudoku rule that each row, column, and 3x3 block must contain every number from 1 to 9 exactly once.
- Using this technique, when there is only one free cell remaining within a 3x3 block, vertical column, or horizontal row, the missing number for that cell can be easily identified. DR1 simply finds this missing number and assigns it to the empty cell, completing the row, column, or block as needed.



1	2	3						
4	5	6						
7	8							

(a) Sudoku grid before applying DR1



1	2	3						
4	5	6						
7	8	9						

(b) Sudoku grid after applying DR1

Figure 3: Applying DR1 on a sudoku grid

3.2 Decision Rule 2 (DR2): Last Possible Number Technique

- The "Last Possible Number" technique is another simple but effective strategy for determining the value of an empty cell. This method involves examining the numbers that are already present within the related row, column, and 3x3 block of the target cell.
- Since there should be no repetition of numbers within any row, column, or 3x3 block, the value of the target cell cannot be any of the numbers already present in its related grid zones. By process of elimination, if only one possible number remains that does not conflict with existing numbers, then this value is assigned to the cell.

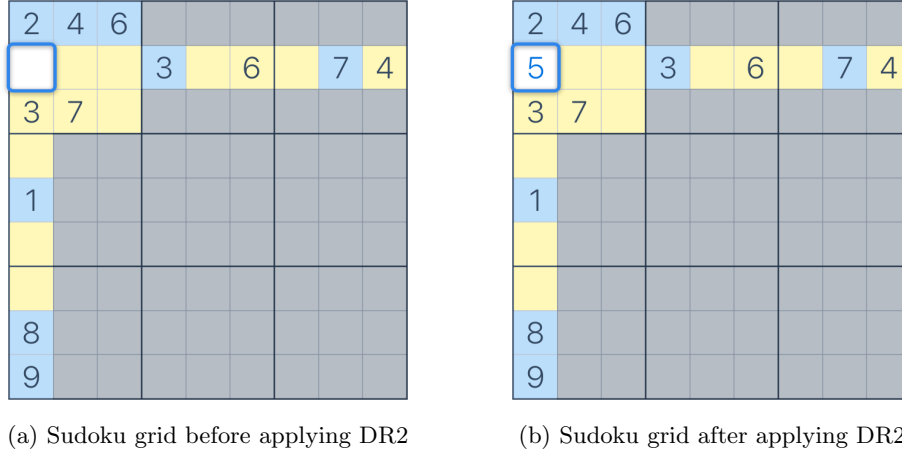


Figure 4: Applying DR2 on a sudoku grid

3.3 Decision Rule 3 (DR3): Obvious Pairs and Obvious Triples

- The "Obvious Pairs and Triples" technique is an advanced method that uses the notes (possible values) within each cell. For this approach, DR3 checks the notes of each cell within a row, column, or 3x3 block for pairs or triples that appear exclusively in two or three specific cells, respectively.
- If a pair (or triple) of numbers only appears in two (or three) cells within a grid zone, these numbers can only logically belong to those cells. Consequently, these numbers are removed from the notes of all other cells within the same grid zone. This technique effectively reduces the options for other cells, streamlining the solving process.

3.4 Decision Rule 4 (DR4): Hidden Singles

- The "Hidden Singles" technique is another powerful rule that leverages notes. For each cell, this technique checks whether a particular note (potential value) appears exclusively in that cell within a specific grid zone (i.e., row, column, or 3x3 block).
- If a note is found to be unique to a single cell within a grid zone, it means that this value must be the correct one for that cell. DR4 then sets this value for the cell, which narrows down the options in neighboring cells.

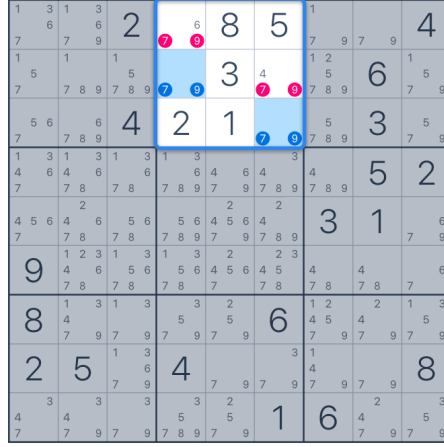
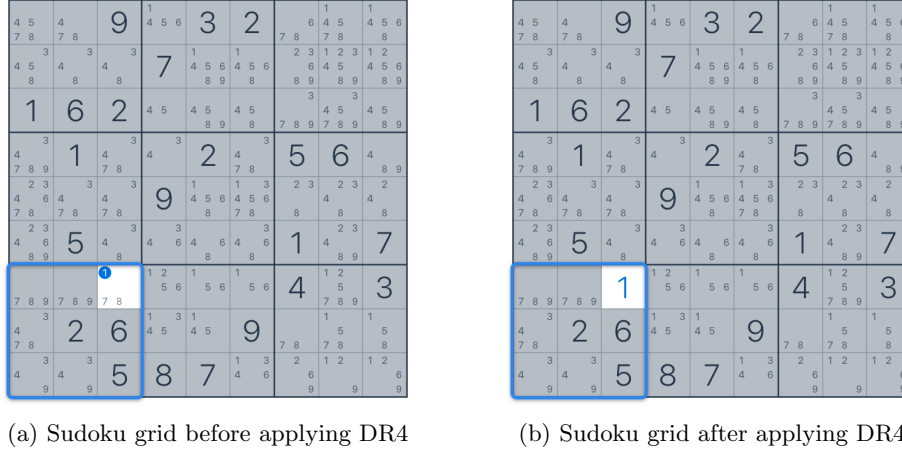


Figure 5: Applying DR3 on a sudoku grid



(a) Sudoku grid before applying DR4

(b) Sudoku grid after applying DR4

Figure 6: Applying DR4 on a sudoku grid

4 Design Patterns

In our project, we employed several well-known design patterns to enhance flexibility, modularity, and code reusability. Below, I detail the four primary patterns used, including a brief explanation of each and how we applied them in our code.

4.1 Factory Method Pattern

- **Explanation:** The Factory Method pattern provides an interface for creating objects but allows subclasses to alter the type of objects that will

be created. This design pattern is commonly used when the exact types and dependencies of objects are determined at runtime, allowing for more flexibility in object creation without hardcoding specific classes.

- **Application:** In the ‘Grid’ class (within ‘sudoku.py’), we applied the Factory Method pattern to create different types of ‘GridZone’ objects, specifically ‘Row’, ‘Column’, and ‘Square’. These zones represent different sections of the Sudoku grid, each containing cells. Instead of directly instantiating each zone type, the ‘Grid’ class dynamically initializes them according to the context (such as the row, column, or square) without needing to know the specific subclass details. This approach enables a flexible way of managing various grid zones and facilitates any future modifications to zone creation.

4.2 Observer Pattern

- **Explanation:** The Observer pattern defines a one-to-many relationship between objects, where changes in one object (the subject) are automatically communicated to its dependent objects (observers). This pattern promotes loose coupling, as the subject and observers remain independent but stay synchronized.
- **Application:** We implemented the Observer pattern in the interaction between ‘Cell’ and ‘GridZone’ objects. Each ‘GridZone’ (Row, Column, or Square) acts as an observer for its cells. When a cell’s value changes, it notifies the zones it belongs to by calling their ‘onCellUpdate’ method. This notification allows the ‘GridZone’ to update its internal tracking of numbers and adjust notes for other empty cells within the zone. This pattern ensures that changes in a cell’s value are automatically propagated to its related zones, helping maintain the integrity of the grid’s state.

4.3 Memento Pattern

- **Explanation:** The Memento pattern captures and stores an object’s internal state, allowing the object to be restored to this state later. This pattern is particularly useful in applications requiring undo or rollback capabilities, as it enables easy restoration of previous states without violating encapsulation.
- **Application:** We used the Memento pattern in the ‘Game’ class (in ‘game.py’) to manage a history of grid states, allowing users to reset the Sudoku puzzle to a previous state if needed. Each grid state is stored as a deep copy in the ‘self.history’ list every time a significant action is taken. When the ‘reset’ method is called, the game retrieves a prior grid state from ‘history’, effectively reverting to that specific state. This feature is essential for enabling the undo functionality, giving users flexibility in exploring and reverting moves.

4.4 Command Pattern

- **Explanation:** The Command pattern encapsulates a request as an object, allowing for parameterization and queuing of requests. It decouples the object that invokes the operation from the one that knows how to perform it. This pattern is especially useful in applications requiring flexible and reusable command handling.
- **Application:** In our ‘Game’ class, the ‘run_command_loop’ method implements a Command pattern-like structure to interpret and execute user commands. Commands like ‘load’, ‘print’, ‘set’, and ‘solve’ are parsed from user input and executed as individual operations on the ‘Game’ object. Each command is processed independently, which enables flexible handling of commands and simplifies the extension of the command set, should new functionalities be added. This approach decouples the command input from its execution, making it easier to manage command logic within the game.

Each of these design patterns played a vital role in structuring our application, helping achieve a cleaner, more maintainable codebase while allowing for easier future modifications and feature additions.