MASTER INFORMATIQUE

UNIVERSITÉ CÔTE D'AZUR

Sudoku Solver

TRISTAN PATOUT

J.-C. Régin | Software Engineering | November 2024

Table des matières

1. INTRODUCTION

This project presents a complete solution for solving Sudoku puzzles, focusing on the use of object-oriented programming principles and design patterns. The main objective was to develop a Sudoku solver capable of automatically filling grids based on a set of standard Sudoku deduction rules. This solver also needed to be able to interact with users when they got stuck, prompting them to manually enter values for unsolvable cells. Another objective was to incorporate a rating system to classify puzzles by level of difficulty.

In this project, we aimed to create a structured, modular solution capable of solving Sudoku grids of different difficulty levels by implementing deduction techniques based on common Sudoku strategies. This solution can be improved by incorporating new deduction rules.

2. PROBLEM ANALYSIS

Initial Challenges

Understanding the main Sudoku rules

The primary challenge was translating the complex logic of Sudoku-solving rules into efficient algorithms. To understand the problem, we reviewed various online resources, including the great beginner friendly sudoku.com's rule guide, to get a foundational understanding of Sudoku solving techniques. However, as this guide was somewhat limited, we later cross-referenced SudokuWiki.org for a more detailed explanation of the rules and solving strategies (we also used it for testing purposes – see ...).

Defining the input format

A secondary early challenge was to handle grid inputs in the specific format outlined by the project requirements. The need for accurate grid representation required defining a clear structure for managing cells and grid properties.

```
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0
```

This is the input format we defined
(0 when no values else numbers between 1 and 9 included).

Specific Issues with Deduction Rules and Solver Design

From a technical standpoint, the most challenging aspect was translating Sudoku-solving rules into precise algorithms. The solver required a class hierarchy that allowed each rule to operate independently while also interacting effectively within a sequence of rules. This was especially important to ensure flexibility and reusability of code.

By approaching the problem using well-structured classes (*SudokuSolver*, *SudokuGrid*, *Cell*, and *DeductionRule*) we effectively separated concerns from the start. This approach allowed for each deduction rule to be implemented as a modular unit, simplifying both the code's readability and the process of adding new rules. This foundational setup made it possible to manage complexity as the project evolved, facilitating the addition of advanced deduction techniques without the need for major revisions.

In the following sections, we will examine the structure of the project and highlight the key solutions and design choices made to create a robust and flexible Sudoku solver.

3. FIRST SOLUTIONS AND APPROACHES

Class Structure

The project was structured around several core classes to enable a clear separation of responsibilities:

- **SudokuSolver**: The main class responsible for solving the Sudoku puzzle by applying various deduction rules sequentially. This class controls the solving flow and handles interactions with the user when required.
- **SudokuGrid**: Represents the Sudoku grid and provides methods to manipulate and validate grid states and its cells (of class Cell).
- **Cell**: Represents individual cells in the grid, storing possible values (notes) and methods to update cell values from the deduction rules.
- **DeductionRule** (base class): This is the core class for all deduction rules, providing a consistent interface for rule application. By inheriting from *DeductionRule*, each specific technique ensures uniformity in its implementation, minimizing code duplication and allowing easy extension of the solver with new rules.

This structure enabled the solver to apply different deduction rules in a predefined order, attempting to solve cells until either the puzzle was completed, or no further deductions could be made. If the grid cannot be fully solved, the solver will either prompt the user for input (if enabled) or return a failure, indicating that the puzzle is unsolvable under the current constraints.

Deduction Rules and Their Interaction

The solver's deduction rules were implemented as subclasses of *DeductionRule*, with each rule designed to target specific deduction strategies, such as:

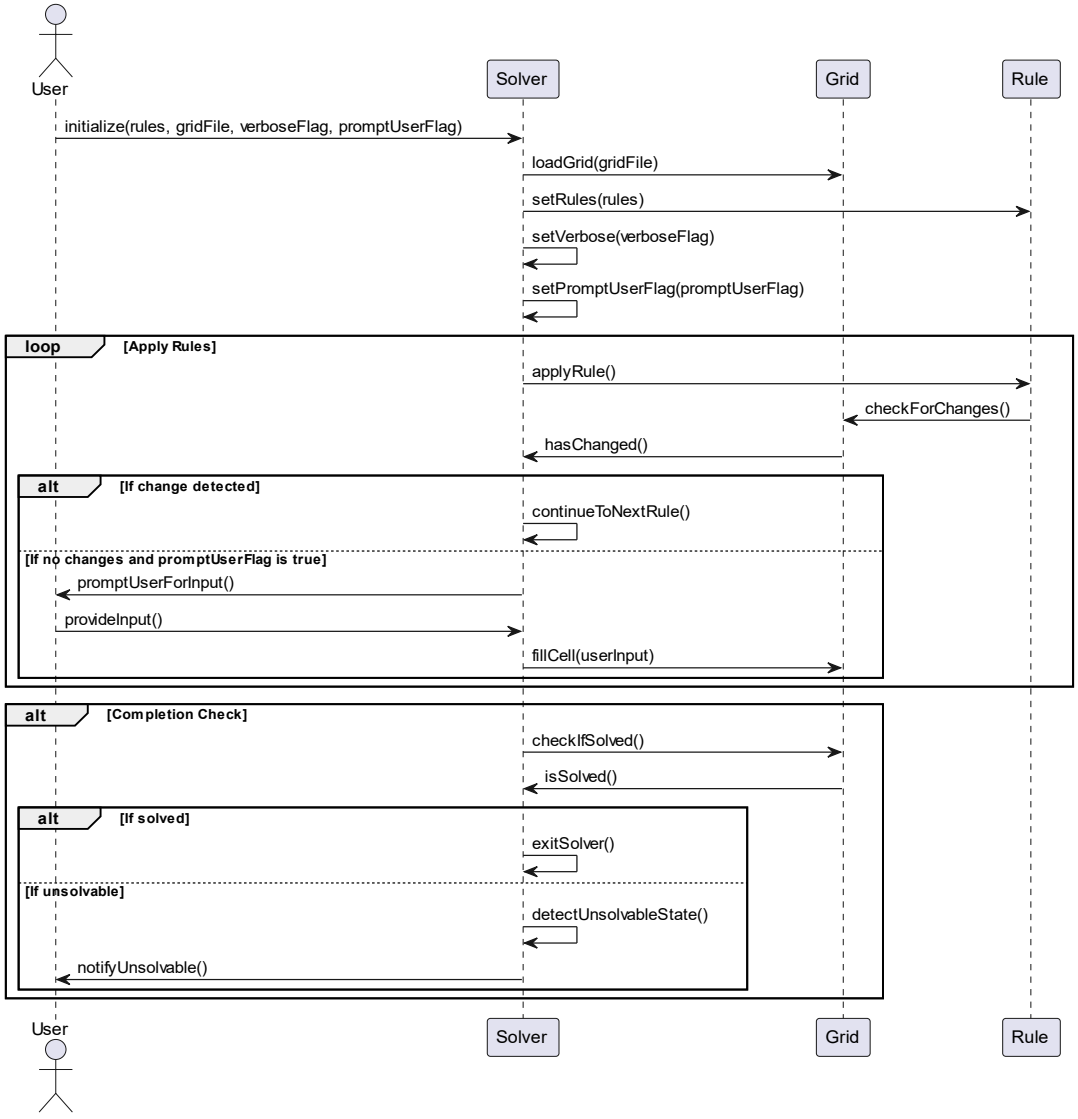
- **Naked Techniques**: Basic strategies like "Naked Singles," "Naked Pairs," and "Naked Triples". These techniques involve identifying cells with a common limited number of possibilities and reducing notes of the other cells accordingly. (see sudokuwiki.org/Naked_Candidates for a better understanding).
- **Hidden Techniques**: Like naked techniques but focused on hidden singles, pairs, and triples. In these techniques the notes are reduced in the cells having common notes. (see sudokuwiki.org/Hidden_Candidates for a better understanding).

- **Advanced Techniques (e.g., Y-Wing)**: We then tried to implement more complex technique that identifies interrelated cells to deduce values indirectly. The successful implementation proved the effectiveness of our structure, even for harder puzzles.

Sudoku Solving Flow

1. **Initialization**: The solver is initialized with a set of rules, the grid file, and a flag for verbose (if we want to print steps) and user input (prompt_user).
2. **Rule Application**: The solver applies each deduction rule sequentially, checking after each rule if changes were made (has_changed flag).
3. **User Interaction**: If no deductions can be made and prompt_user is true, the solver prompts the user to manually fill a cell to proceed.
4. **Completion Check**: The solver continues applying rules or requesting input until the grid is fully solved or an unsolvable state is detected.

Below is a simplified sequence diagram illustrating our Sudoku Solving flow.



4. DESIGN PATTERNS

To enhance our project clarity, modularity and reusability, we implemented several design patterns. Below is a description of the main ones:

- **Factory Pattern**: *DeductionRuleFactory* instantiates rules with their difficulty level. This pattern supports flexibility by enabling easy addition of new rules.
- **Chain of Responsibility**: *RuleHandler* instances are created from the rules and applied sequentially in a chain *RulesHandlerChain*, with each rule attempting deductions independently. This pattern fits well with the iterative nature of Sudoku solving. It allows to increase the difficulty based on the rules applied to solve a grid.
- **Observer Pattern**: Each cell has an observer, allowing it to rapidly access the cell's related cells (cells in the same row, column or zone as the cell) and propagate changes to related cells.
- **Template Pattern**: *DeductionRule* is a base class template for all specific rules. This standardizes rule application across the solver and supports extending the rule set with minimal changes to the overall architecture.

These design patterns structured the code into manageable components and enhanced both readability and scalability, making it easier to incorporate additional rules or change rule behavior.

5. USER INTERACTION

When deduction rules fail to make progress, the solver prompts the user to input a value directly. This functionality is managed by prompt_user_for_input, which activates only if prompt_user is set to True in the solver. When the user enters a value, the solver resumes the deduction process, integrating the user-provided input into its ongoing calculations.

To handle error cases gracefully, the solver provides feedback if an incorrect value leads to an invalid grid state, allowing the user to adjust inputs as needed. I also experimented with python decorator functionality for checking if the user input is valid.

6. DIFFICULTY EVALUATION

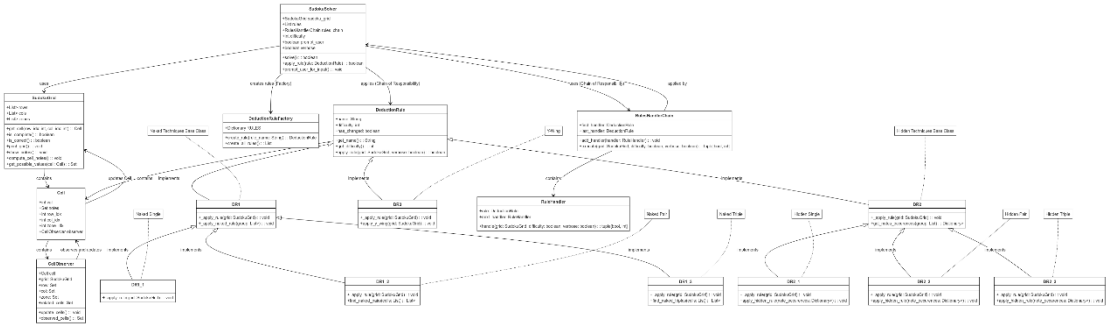
As explained a bit previously; to assess the difficulty of each grid, we assigned a difficulty level to each deduction rule within the *DeductionRuleFactory*. Rules range from simple (e.g., naked singles) to complex (e.g., Y-Wing). The solver evaluates the grid's overall difficulty based on the most advanced rule required to complete the puzzle.

Classification Scheme

- **Easy**: Solvable using only basic rules like naked singles and hidden singles.
- **Medium**: Requires hidden pairs or naked pairs/triples techniques.
- **Difficult**: Involves hidden triples and other advanced techniques, such as Y-Wing.
- **Very Difficult**: Requires the input of the user to solve the grid.

7. FINAL PROJECT CLASS DIAGRAM

Here is below a class diagram of the project (available in the markdown preview for better readability).



8. TESTING AND VALIDATION

Testing Methodology

We created *SudokuTestRunner* from *unittest.TestCase* class for systematic testing, employing basic unittest to validate the solver's functionality. Tests included solvable grids, incomplete grids, and invalid configurations to ensure robustness across various scenarios.

- **Standard Rule Tests**: We tested almost all deduction rules using sample puzzles from SudokuWiki, which served as reliable benchmarks. Thanks to SudokuWiki, we can determine in advance which rules are necessary to solve a specific grid. Additionally, the grids are embedded in the URL of each page. To streamline the process, we wrote a utility function to convert these URLs into our grid format, saving the grid to a temporary test file used for the test.
- **Failure Cases**: We tested edge cases like grids with no solution or incomplete grid and overly complex grids that required extensive user input, confirming the solver's ability to identify unsolvable states accurately.

9. CHALLENGES AND TRADE-OFFS

I thought it might be interesting to talk about this. A compromise was found between performance and clarity. The use of class-based, object-oriented design has improved code organization and maintainability. We could now ask ourselves whether an implementation using matrix operations would not have performed better.

10. FINAL THOUGHTS

I'm particularly proud of how object-oriented programming principles and design patterns structured the project into manageable and reusable parts. Translating sudoku rules into deduction rules algorithms was challenging but rewarding. This project helped me strengthen my object-oriented programming skills, allowing me to discover new things along the way. I successfully implemented diverse set manipulations and even created decorator function. Future improvements could focus on optimizing rules efficiencies and potentially incorporating even more advanced Deduction Rules strategies.

Tristan Patout