

Development of a Sudoku Solver: C1 Research Computing Coursework

Vishal Jain

December 14, 2023

Contents

1 Introduction

"Sudoku is a denial of service attack on human intellect" - Ben Laurie

This report details the development of a Sudoku solver inline with the requirements of the C1 Research Computing coursework. The programme takes as input an incomplete grid in the form of a text file with a 9x9 grid of numbers with zero representing unknown values and '—','+', '-' separating cells and , i.e.:

```
$ cat input.txt
000|007|000
000|009|504
000|050|169
----+----+----
080|000|305
075|000|290
406|000|080
----+----+----
762|080|000
103|900|000
000|600|000
```

and outputs the completed grid in the same form.

1.1 Background and motivation

Sudoku, a logic-based combinatorial number-placement puzzle, presents a paradigmatic example of a constraint satisfaction problem (CSP), a class of problems fundamental to the field of computer science. The puzzle's structure, consisting of a 9x9 grid divided into subgrids, adheres to stringent placement rules, thereby embodying the essence of CSPs where the objective is to find a solution that satisfies all given constraints.

This project, centered on the development of a Sudoku solver, is primarily motivated by the pedagogical value inherent in addressing such a well-defined and constrained problem space. Sudoku solvers exemplify the application of algorithmic strategies to a finite, yet non-trivial problem domain. This aligns with the core objectives of the C1 Research Computing coursework, which emphasizes the development of computational solutions that are both efficient and effective.

2 Problem Decomposition

To architect the Sudoku solver program, an initial flowchart was constructed to map out the high-level logical sequence. Each step of the flowchart was assigned a color based on its independence; steps that could be altered without impacting preceding steps received a unique color, whereas related, interdependent steps shared the same color. This method helped identify distinct, modular components within the program's workflow. The resulting color-coded flowchart is presented in Figure ??.

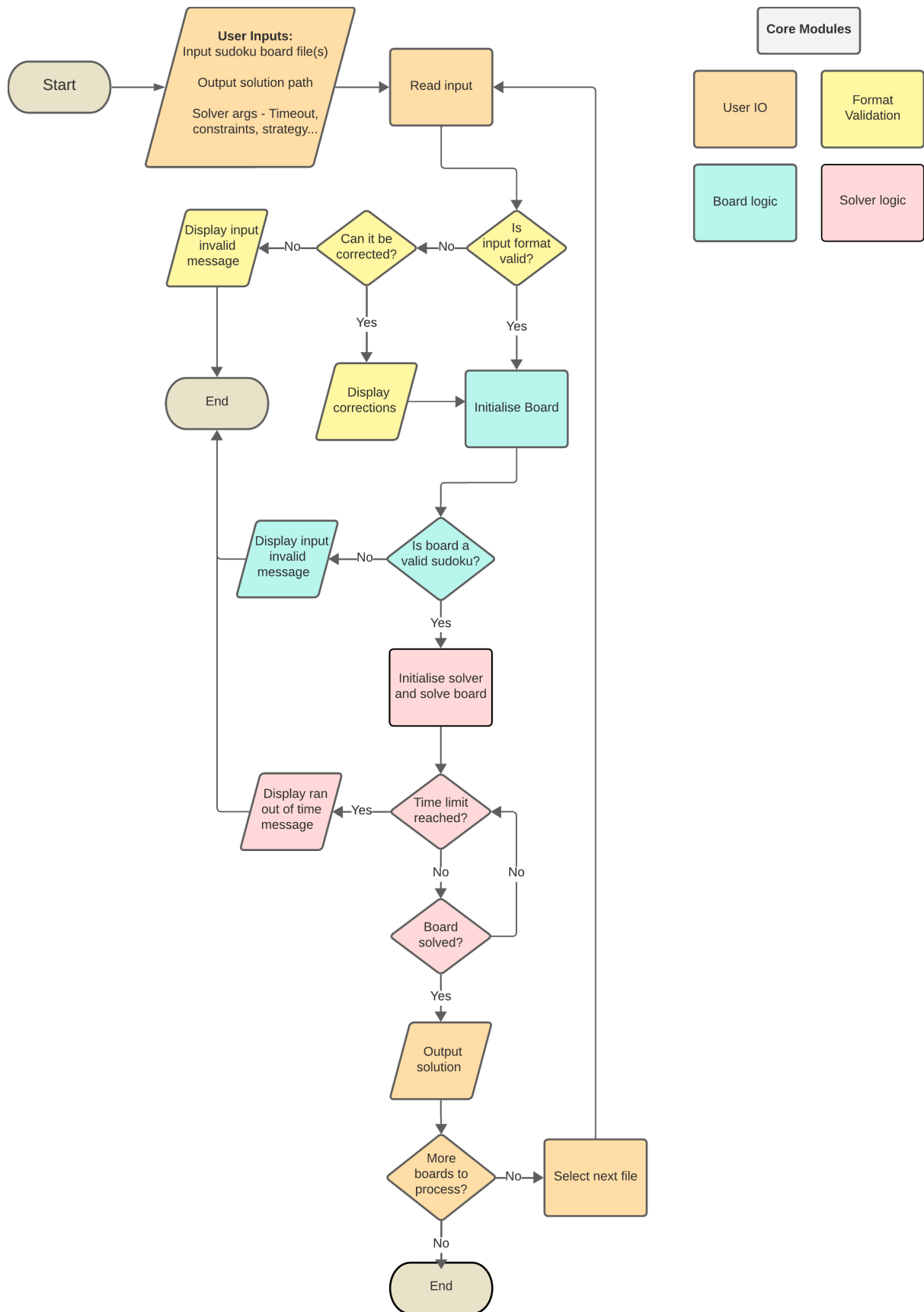


Figure 1: High-level flowchart of the Sudoku solver program, illustrating the initial conceptual design. Functionally related modules are color-coded: orange for user interaction, yellow for format validation, cyan for board logic, and pink for solver logic components.

The analysis led to the identification of the following key modular components in the Sudoku solver:

- **User IO:** Responsible for user interaction and input/output processing.
- **Format Validation:** Ensures the correctness of the input format.
- **Board Logic:** Manages the internal representation and manipulation of the Sudoku board.
- **Solver Logic:** Implements the algorithms to solve the Sudoku puzzle.

2.1 Developmental Journey

Before delving into the final structure of the Sudoku solver, it is insightful to explore the initial prototyping phase. This phase laid the groundwork for the project and provided key insights that shaped the final design.

2.1.1 Early Prototyping

In the initial stages, the envisioned usage of the program was conceptualised as a straightforward sequence of operations involving board validation, board representation, and solving. The logical flow was as follows:

1. **Board Validation:** The user provides a file containing the Sudoku puzzle, which is then read, validated, and corrected if necessary. This process was handled by a utility function, `validate_board`, located in the `utils` module. The function's purpose was to ensure the input adhered to Sudoku format standards and to correct any discrepancies. The code snippet for this step:

```
board_array = utils.validate_board(board_file)
```

This function returns a 9x9 array representing the initial state of the Sudoku board.

2. **Board Representation:** The returned board array is then used to initialise an instance of the `'SudokuBoard'` class. This class encapsulates the board's representation and manipulation logic, providing common game board methods like `'reset'` and `'check_valid'`. The initialisation step is shown below:

```
sudoku_board = SudokuBoard(board_array)
```

3. **Solving the Puzzle:** Finally, the `'SudokuBoard'` instance is passed to the `'solve'` method of a `'BacktrackingSolver'` class instance. The solver applies the backtracking algorithm to find a solution to the puzzle, returning a new `'SudokuBoard'` instance representing the solved board:

```
solver = BacktrackingSolver().solve(board)
```

2.1.2 Early Hurdles and Insights

The initial testing of the program, particularly on diverse Sudoku puzzles, including a Kaggle dataset of a million boards, brought to light several key issues and limitations with the initial design. These challenges played a crucial role in shaping the subsequent development of the solver.

Modularity Challenges: One of the first hurdles encountered was the lack of modularity in handling board validation, correction, and parsing. The initial approach used a single utility function, `validate_board`, which combined these processes. This lack of separation hindered flexibility and made it difficult to handle multiple input formats or to separate validation from parsing:

Supporting Multiple Input Formats: The project faced a significant challenge when dealing with input formats different from the expected grid-based format. Specifically, the Kaggle dataset was in a single line format, necessitating a conversion process to and from the grid format. This experience underscored the need for native support for multiple input formats and highlighted the inefficiency of the existing approach where saving and loading logic were tied to a specific format.

Evolution of SudokuBoard: The `SudokuBoard` class underwent significant refinement. Initially, it included some output logic, but it became clear that this class should focus solely on representing the Sudoku board and interfacing with solvers through methods like `get_empty_cells`, `check_valid_move`, and `reset_board`. This realisation led to a cleaner and more focused class. It also took on some of the validation logic from the utility function that related to checking the input board followed the rules of Sudoku.

Solver Extensibility: Another insight gained was the need for a more flexible system to integrate new solver algorithms. The initial setup did not provide an easy way to select or initialise different solvers that may not share initialisation parameters, leading to potential difficulties in adding complex algorithms with unique initialisation requirements.

Enhancing Input Flexibility and User Options in `main.py`: The final insight was on making `main.py` more adaptable and user-friendly. The current script was designed for single file-based inputs however it needed to be able to accept both single puzzle file paths and directories containing multiple puzzles. Additionally, a desirable feature that was initially not thought to be significant was the incorporation of string input capability. This greatly facilitated quick puzzle testing from online sources. This feature allowed for direct copy-pasting of puzzles into the terminal, streamlining the validation process. Moreover, the script was adapted to enable dynamic selection and initialisation of various solvers and format handlers without the need for code modifications, thereby increasing the programs flexibility while maintaining modularity.

2.1.3 High-Level Overview of the Final Implementation

The final design of the Sudoku solver program encapsulates a modular and flexible architecture, as outlined below:

```
# Initialise the Format Handler and Solver
format_handler = SudokuFormatHandler()
solver = SudokuSolver()

# Parse the Sudoku board from the given input in the desired format
board = format_handler.parse(board_file, format_type)

# Employ a specific solver to find a solution for the parsed board
solved_board = solver.solve(board, solver_backend)

# Save the solved board in the desired format and output path
format_handler.save(solved_board, format_type, output_path)
```

In this design, `SudokuFormatHandler` and `SudokuSolver` act as wrapper classes that have methods which allow the user to specify which format and solver backend they want to use, respectively. These classes utilise standardised methods to process and solve the Sudoku puzzle, irrespective of the specific format or solving algorithm used. This approach significantly enhances the extensibility of the program, allowing for the easy integration of new formats and solving methods without requiring changes to the `main.py` script. Additionally, the flexibility to save the output in either the native format of the input or any other specified format is achieved by adjusting the `format_type` argument, further increasing the program's versatility and user-friendliness. The next section will explore the design and implementation of the various modules in greater detail.

3 Programme Modules

3.1 Class `SudokuFormatHandler`

3.1.1 Scope

The scope of the `FormatHandler` classes implemented in `src/sudoku_format_handler.py` is defined by its role in bridging the gap between external representations of Sudoku boards and standardised internal representation. Specifically, the scope of this class encompasses:


- **Parsing:** Handling the conversion of the input, whether it comes as a string or as a file path, into a standardised `SudokuBoard` object.
- **Saving:** Conversely, the class also manages the conversion of `SudokuBoard` objects back into user-readable formats.
- **Format Validation and Correction:** This class is also where the developer defines any methods for validation checks and corrections to ensure that the input is in the expected format before parsing.

3.1.2 Design

The design of the `SudokuFormatHandler` class is centered around principles of modularity and extensibility, driven by the use of an abstract base class and a handler dictionary. Key design aspects include:

- **Modularity:** The class leverages an abstract base class, `FormatHandler`, which outlines essential methods – `parse` and `save`. This structure ensures consistency across different format handlers while allowing for flexibility in their specific implementations.
- **Extensibility:** New formats can be easily integrated into the system by creating a subclass of `FormatHandler` and adding it to `SudokuFormatHandler`'s `handler_dict` class attribute. This approach simplifies the process of extending the system to accommodate new Sudoku board formats.
- **Abstraction:** The abstract methods in `FormatHandler` enforce a contract that all subclasses must provide specific implementations for parsing and saving Sudoku boards, thus maintaining a standardised interface.

The UML class diagrams in Figure ?? provide a visual representation of the design and implementation of the `FormatHandler` abstract base class and the `SudokuFormatHandler` class, respectively.



figs/UML_sudoku_handler.png

Figure 2: Class diagram for the `SudokuFormatHandler` module, illustrating the design and implementation of the `FormatHandler` abstract base class and the `SudokuFormatHandler` class. Along with the composition of the `SudokuFormatHandler` class with the implemented `Format` classes (`GridFormatHandler` and `FlatFormatHandler`).

3.1.3 Implementation

The implementation of `SudokuFormatHandler` involves several key components that collectively enable dynamic handling of various board formats:

- **Handler Dictionary:** The `handler_dict` serves as a central repository linking format types (e.g., 'grid', 'flat') to their corresponding handler objects, facilitating the flexible handling of different formats.
- **Handler Retrieval:** The method `_get_handler` takes a format type as input and retrieves the appropriate handler from `handler_dict`. It raises a `KeyError` with available format options in case of an unsupported format type.

- **Parse Method:** The `parse` method interfaces with the selected handler to transform the input into a `SudokuBoard` object. It delegates the specifics of parsing to the corresponding format handler.
- **Save Method:** Similarly, the `save` method utilizes the appropriate handler to convert a `SudokuBoard` object back into a file, ensuring consistency in the output process across different input formats.

3.1.4 Extensibility

The `SudokuFormatHandler` is designed with extensibility in mind. To support a new format, a developer simply needs to create a new handler class extending `FormatHandler` and implement the `parse` and `save` methods with the expected inputs and outputs. The new handler can then be added to the `handler_dict`. This design makes it straightforward to extend the solver's capabilities to accommodate future requirements or user preferences for different Sudoku board formats.

3.1.5 Limitations

The current design of the `SudokuFormatHandler` module, as well as other related modules, is specifically tailored to process single Sudoku boards from individual input files. This intentional design choice is due to the significant increase in complexity that would arise from trying to build a general framework which supports multiple formats that also supports accommodating multiple boards within a single file. Maintaining a single input, single board paradigm simplifies the overall program structure and aligns with our current functional objectives.

3.2 SudokuBoard Class

Future work is to make a board abc for future representations like dictionary based boards

3.2.1 Scope of the SudokuBoard Class

The `SudokuBoard` class in `src/sudoku.board.py` is integral to managing the internal representation and manipulation of the Sudoku board and providing a standardised interface for the other classes in the program. The scope of this class encompasses:

- **Board Validation:** Makes sure board state always follows Sudoku rules.
- **Board Representation:** Provides a standardised representation of the board state for the other classes.
- **Board Manipulation:** Facilitates methods for resetting the board and modifying cell values to allow for board manipulation.
- **Solving Assistance:** Provides common utility methods required by typical solvers, such as fetching empty cells and checking valid number placements.

3.2.2 Design

The design of the `SudokuBoard` class is centered around the principle of encapsulation. The other classes in the program interact with the board through a standardised interface of class methods, without needing to know the internal representation or logic. Key design aspects include:

3.3 SudokuSolver Class

3.4 main.py

4 Testing and Optimisation

4.1 Profiling

5 SWE

Unit Tests and CI set up Packaging and Usability

6 Summary