Development of a Sudoku Solver: C1 Research Computing Coursework

Vishal Jain

December 17, 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.

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 logical independence. This method helped identify distinct, modular components within the program's workflow. The resulting color-coded flowchart is presented in Figure 1.

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

- **User IO:** Handles user interaction and input/output processing.

- **Format Validation:** Ensures the correctness of the input format.

- **Board Logic:** Converts input Sudoku boards into a standardised internal format suitable for efficient manipulation.

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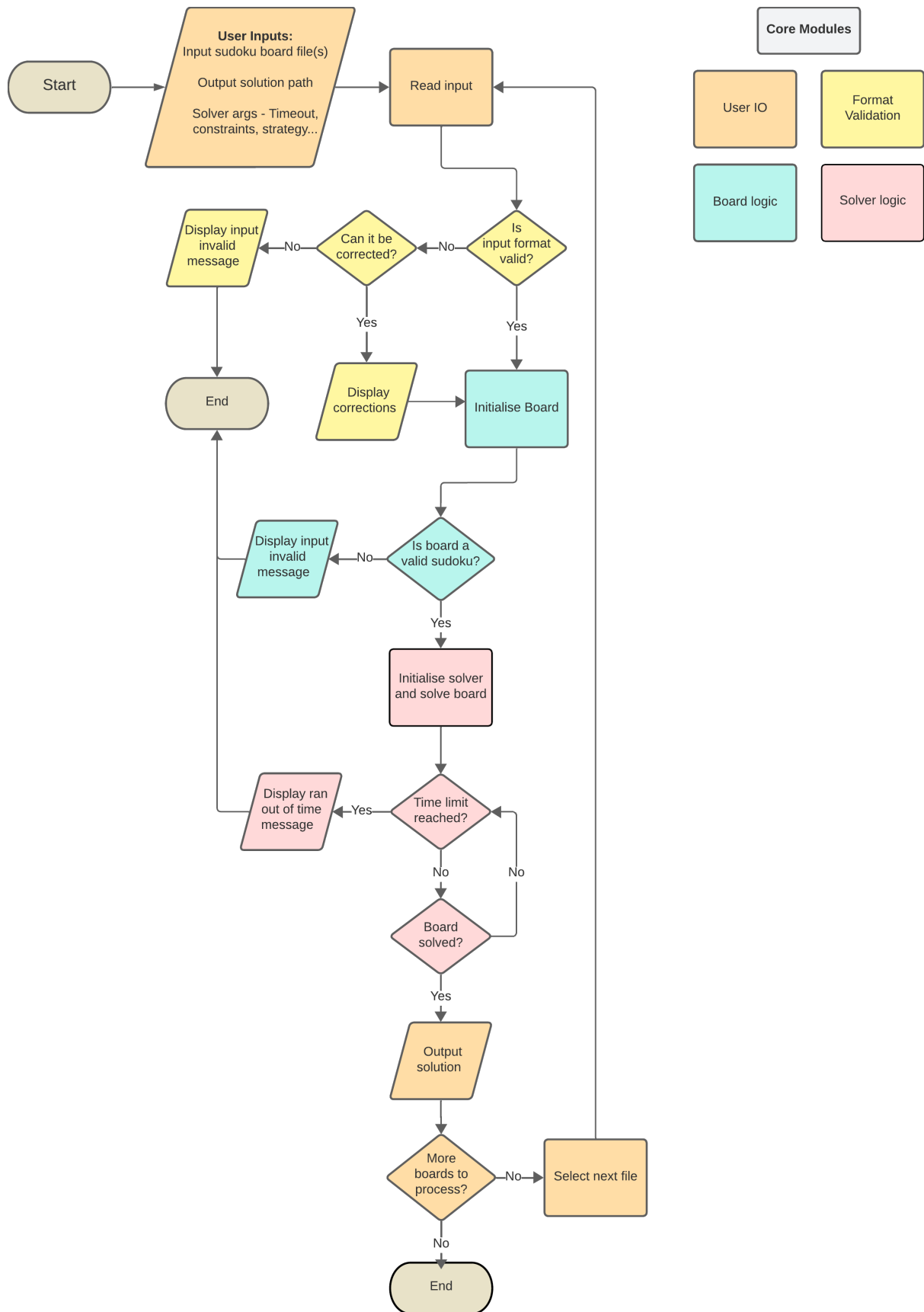
Figure 1: High-level flowchart of the Sudoku solver program, illustrating the initial conceptual design. Related pieces of logic are color-coded: orange for user interaction, yellow for format validation, cyan for board logic and pink for solver related logic.

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, solving and saving. 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 numpy 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, offering standard board operations such as 'reset' and 'check_valid'. The initialisation step is shown below:

   ```
   board = SudokuBoard(board_array)
   ```

3. **Solving the Puzzle**: The next step involved passing the `SudokuBoard` instance to the `solve` method of the solver class. This returned a new 'SudokuBoard' instance representing the solved board:

   ```
   solved_board = Solver(board).solve()
   ```

   The details of the specific solver used will be discussed in a later section.

4. **Puzzle Saving**: The final step was to save the solution in the original format. This functionality is encapsulated in the board.save() method of the `SudokuBoard` class.

   ```
   solved_board.save(save_path)
   ```

2.1.2 Early Hurdles and Insights

Benchmarking the initial prototype against a Kaggle dataset revealed several design and functionality challenges:

- **Format Handling Limitation:** The primary obstacle was accommodating different input formats. Rather than resort to makeshift conversion scripts, it became clear that a sustainable approach was to develop a comprehensive format handling framework, ensuring native support for diverse input formats and streamlining the addition of new ones.

- **Validation Logic Overload:** The `validate_board` function grew cumbersome as it handled more validation and correction logic, straying from the principles of single responsibility and modularity.

- **Misplaced Save Method:** In the context of supporting multiple formats, it was logical to transfer the `save` method from the `SudokuBoard` class to the format classes where the custom parsing methods would be defined as saving and parsing are two sides of the same coin and should live together.

- **Solver Framework Incompleteness:** The solver class lacked a clear framework which was capable of for incorporating alternative algorithms, hindering the dynamic selection and testing of various solving strategies on the same dataset.

- **Lack of Back-End Abstraction in Main:** The inflexibility of the `main.py` module became apparent, as integrating new format handlers or solvers necessitated direct modifications to the front end script, going against the principles of modularity and encapsulation.

- **Main Script Inflexibility:** The inability to process directories of board files limited batch testing capabilities. Moreover, the need to support string inputs for swift testing became apparent.

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. Their methods accept parameters which allow the user to specify the desired format and solver backend. The corresponding arguments for these parameters are collected by the `main.py` script. The next section will explore the design and implementation of the various modules in greater detail.

3 Programme Modules

3.1 SudokuFormatHandler Class

3.1.1 Scope

The scope of format handler classes, as implemented in `sudoku_format_handlers.py`, is defined by their role in bridging the external representations of Sudoku boards and their standardised internal representations. In this context, their scope includes:

- **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 and Implementation

The design of the module is such that there is a central `SudokuFormatHandler` class which calls the other implemented `FormatHandler` classes. This approach is centered on the 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 new format will automatically be made available for selection through the `main.py` script.

The UML class diagrams in Figure 2 provide a visual representation of the design and implementation of the `FormatHandler` abstract base class and the concrete grid and line format handler classes used by the `SudokuFormatHandler` class.
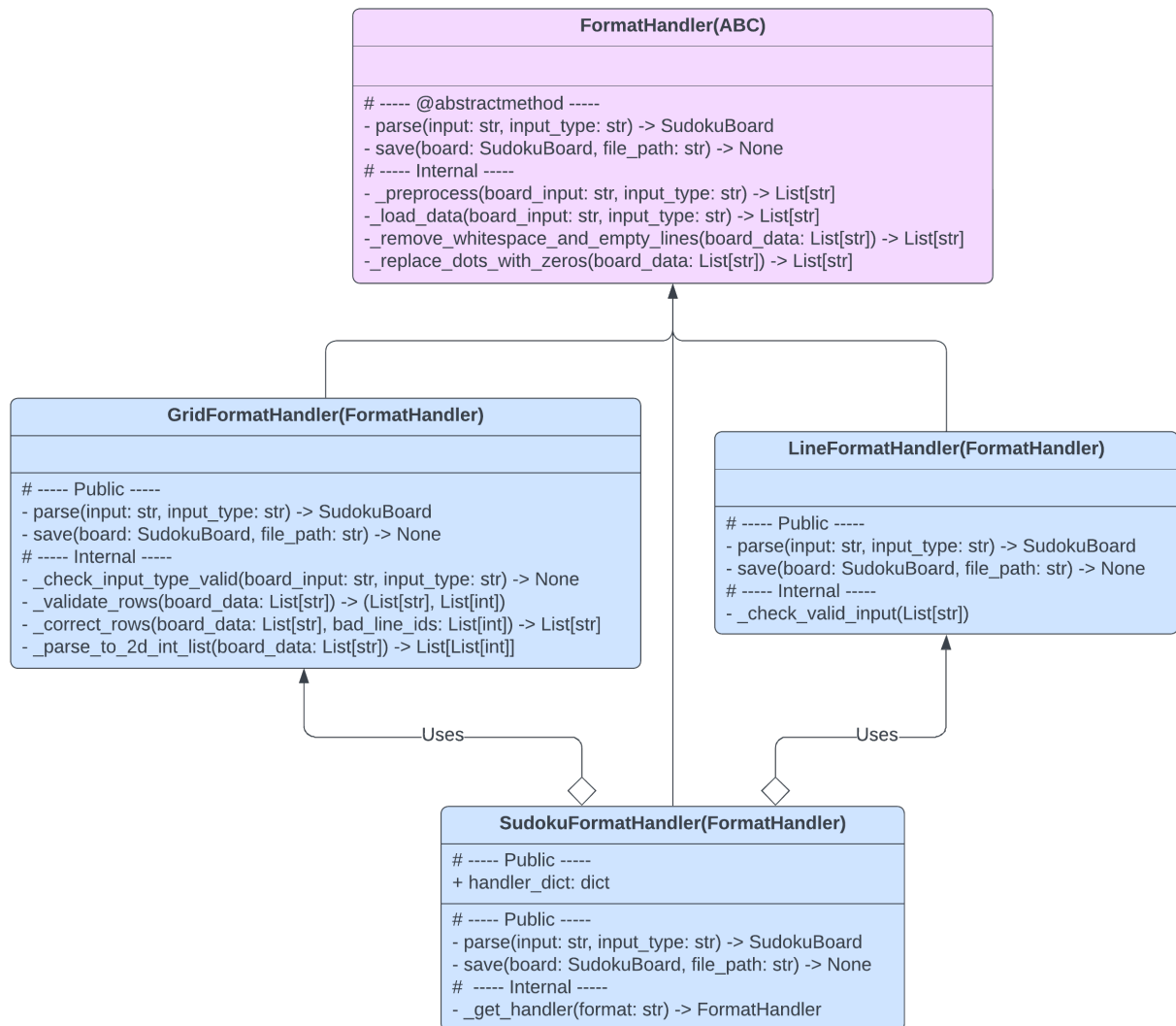

Figure 2: UML class diagram for implementation of the FormatHandler classes. Abstract classes are depicted in pink, concrete classes in blue. Composition relationships are indicated with arrows having a white diamond base and a solid head.

GridFormatHandler The GridFormatHandler is the format handler for the required format of this coursework. It is designed to handle the parsing and saving of Sudoku boards in the grid format. The parsing process involves reading the input file or string and converting it into a list of strings representing each row of the Sudoku board. The white space and empty lines are removed. Any dots are replacecd with zeros as this is commonly seen online as a placeholder for empty cells. The handler ensures there are exactly 11 rows; otherwise, it raises an error. Each row undergoes validation against a specific regex pattern. If a row deviates from the expected pattern but aligns with an alternate acceptable one, it undergoes correction; otherwise, an error is raised. Correction logic is applied selectively: separator rows lacking numbers are replaced with a standard separator row, but if they contain numbers, an error is raised to avoid digit alteration. For a row which is expected to describe a number row, if it matches the general pattern of 3 sets of numeric character triplets seperated by a non numeric character, it is corrected by inserting the expected seperator characters between the triplets.

3.1.3 Limitations

The module's primary limitation lies in its capability to process only a single Sudoku board per file, rather than supporting multiple boards in one file. This is not an oversight but a deliberate design choice. Accommodating multi-board parsing from a single file can substantially escalate the complexity of the classes. This is because the way you can specify multiple boards is unique to each format. For instance, in the grid format, multiple boards can be specified by separating them with a blank line. However, in the line format, multiple boards can be specified by separating them with a comma. Further, the complexity of the validation logic would also increase. Development of a format handling system that is capable of robustly handling such variability falls outside the scope of this coursework.

3.2 SudokuBoard Class

3.2.1 Scope

The `SudokuBoard` class defined in `sudoku_board.py` is integral to the overall programme, providing a standardised interface for the other classes. The scope of this class encompasses:

- **Board Representation**: Providing an internal representation of a Sudoku board.

- **Board Validation**: Making sure the board state always describes a valid Sudoku.

- **Board Manipulation**: Implementing methods for efficient manipulation and access of the board state.

3.2.2 Design and Implementation

The design of the `SudokuBoard` class adheres to the principle of encapsulation, ensuring that the internal state of the board is managed in a controlled manner.

- **Efficient Board Representation**: The class uses a 2D numpy array for board representation, enhanced with lists of sets to track the state of the rows, columns, and subgrids. Leveraging the speed of numpy arrays and the fast lookup capabilities of sets for validating and managing the board state.

- **Controlled and Encapsulated Board Interaction**: The class maintains encapsulation, offering array-like read-only access via the `__getitem__` method, while preventing direct board modifications. Controlled manipulation is facilitated through specific methods like `place_number` and `remove_number`, ensuring adherence to Sudoku rules and maintaining the integrity of the board's state.

- **Convenience Methods**: The class offers various utility methods such as resetting the board and finding empty cells. These methods simplify common operations required for solving Sudoku puzzles.

- **Initialisation with Integrated Validation**: The `__init__` method performs an immediate validation check of the board state, confirming adherence to Sudoku rules upon setup.

- **Board Formatting for Debugging and Visualisation**: The `__str__` method formats the board into a human-readable string, aiding in debugging and visualisation.

The UML class diagram in Figure 3 provides a visual representation of the implementation of the `SudokuBoard` class.
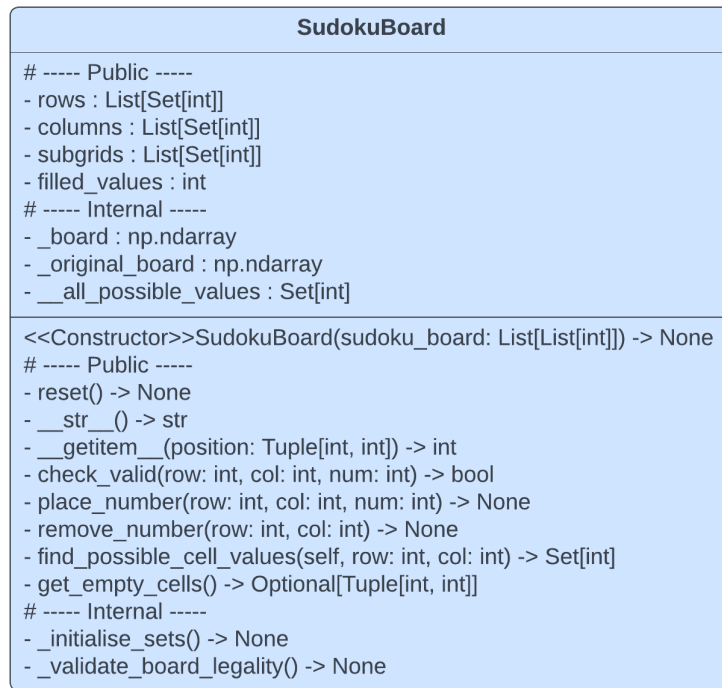
```
SudokuBoard
# ----- Public -----
- rows : List[Set[int]]
- columns : List[Set[int]]
- subgrids : List[Set[int]]
- filled_values : int
# ----- Internal -----
- _board : np.ndarray
- _original_board : np.ndarray
- __all_possible_values : Set[int]

<<Constructor>>SudokuBoard(sudoku_board: List[List[int]]) -> None
# ----- Public -----
- reset() -> None
- __str__() -> str
- __getitem__(position: Tuple[int, int]) -> int
- check_valid(row: int, col: int, num: int) -> bool
- place_number(row: int, col: int, num: int) -> None
- remove_number(row: int, col: int) -> None
- find_possible_cell_values(self, row: int, col: int) -> Set[int]
- get_empty_cells() -> Optional[Tuple[int, int]]
# ----- Internal -----
- _initialise_sets() -> None
- _validate_board_legality() -> None
```

Figure 3: UML class diagram for the SudokuBoard module.

3.2.3 Limitations

The `SudokuBoard` class, unlike `SudokuFormatHandler`, isn't inherently extensible for different board types such as dictionary representations often used in more optimised solvers. To address this, an abstract `Board` base class could be introduced, defining the common methods for solvers and format handlers with their expected inputs and outputs. `SudokuBoard` would then act as a wrapper class, instantiating the specific concrete `Board` implementations specified by the user.

3.3 SudokuSolver Class

3.3.1 Scope

The scope of solver classes, as implemented in `sudoku_solvers.py` is defined by the following:

- **Solving**: Taking in a `SudokuBoard` object and returning a solved `SudokuBoard` object.

- **Outputting Solve Status**: Returning the status of the solution.

3.3.2 Theory - Backtracking

The type of solver implemented in this coursework is based on the method of backtracking. Backtracking is a brute-force method that uses recursive depth-first search to explore all possible solutions, starting from the initial state of the Sudoku board. This method systematically tries every valid option for each empty cell and backtracks when it reaches a state where it can't make any valid moves. While this method is comprehensive, it can be inefficient on some boards due to its lack of strategic direction in cell selection. To enhance this approach, an additional version incorporating a heuristic to prioritise cells with the fewest possible values, reducing the number of possibilities to explore. Both these approaches are implemented in the Sudoku solver program as `BacktrackingSolverBasic` and `BacktrackingSolverEasiestFirst` respectively. Their performances are compared in a later section.

3.3.3 Design and implementation

- **Modularity**: The class leverages an abstract base class, `Solver`, which outlines the essential method for all solvers – `solve`. This structure ensures consistency across different solvers while allowing for flexibility in their specific implementations.

- **Extensibility**: New solvers can be easily integrated into the system by creating a subclass of `Solver`. After implementation, they can be added to `SudokuSolvers`'s `solver_dict` class attribute. They will automatically be available for selection through the `main.py` script.

The UML class diagrams in Figure 4 provide a visual representation of the design and implementation of the `Solver` abstract base class and the `SudokuSolver` class along with the implemented `BacktrackingSolverBasic` and BacktrackingSolverEasiestFirst classes.

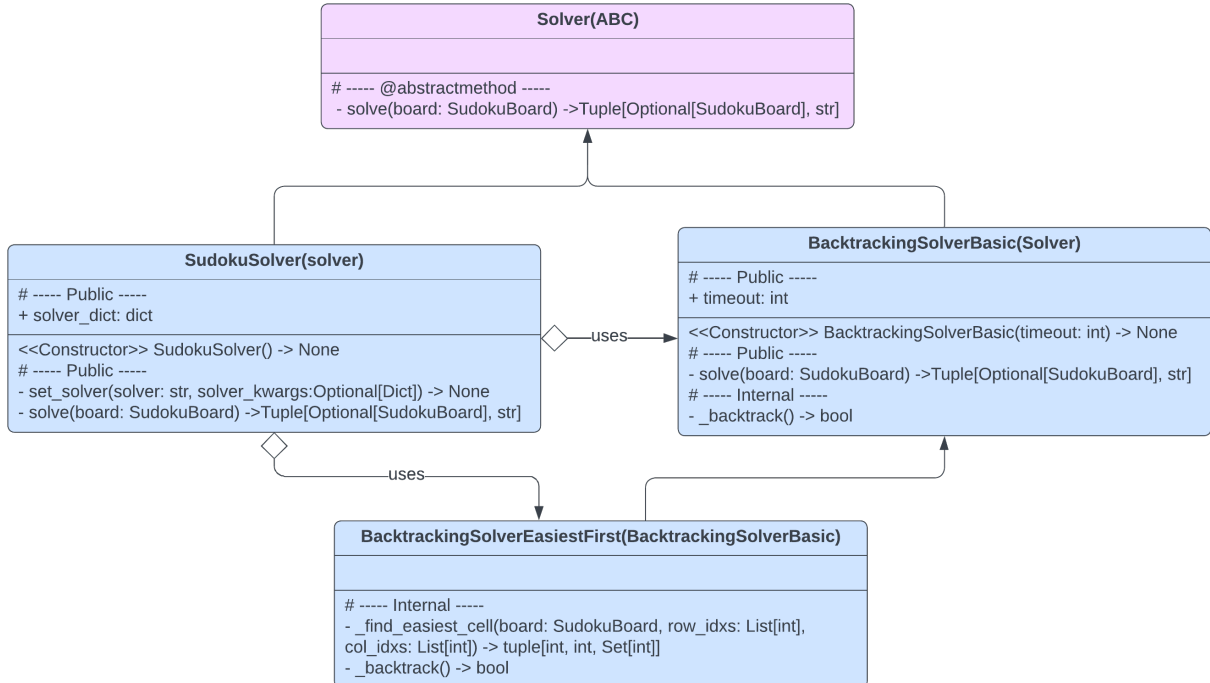Figure 4: UML class diagram for implementation of the SudokuSolver classes. Abstract classes are depicted in pink, concrete classes in blue. Composition relationships are indicated with arrows having a white diamond base and a solid head.

3.3.4 Limitations

The system currently operates on a single-threaded execution model, not utilizing parallel processing which could expedite solving by simultaneously exploring multiple solution tree branches. Additionally, while flexible within the backtracking approach, the framework might not easily support radically different solving methods.

3.4 UserIO

3.4.1 Scope

The user IO is handled by the `main.py` script. This script is defined by its role as the entry point for the Sudoku solver program. The scope of this script encompasses:

1. **Argument Parsing and Validation:** Interprets and validates command-line arguments for configuring the solver's operation.

2. **Initialisation:** Sets up core components like 'SudokuFormatHandler' and 'SudokuSolver' based on user inputs.

3. **Solving Process Management:** Manages the Sudoku solving flow, accommodating both single and batch operations.

4. **Statistics and Output Handling:** Gathers and presents solving statistics, handling outputs for solved puzzles.

3.4.2 Design and implementation

The `main.py` script in the Sudoku solver program is underpinned by the following design principles:

- **Backend Independence:** Maintains independence from the solver and format handler backends.

- **User Customisation:** Allows users to define input and output formats, enabling highly customised runs.

- **Reproducibility:** Facilitates reproducible runs by saving statistics, including the Git commit hash and run arguments.

- **Robust Error Handling:** Incorporates thorough argument validation and error handling to safeguard user interaction.

- **Intuitive Defaults with Customization Options:** Provides sensible defaults for ease of use, alongside a range of customisable arguments for advanced users.

- **Dynamic Solver and Formatter Integration:** Automatically enables selection of solvers and formatters added to the Sudoku solver and format handler.

- **Clear Output and Feedback:** Ensures outputs are easily understandable and provides clear user feedback throughout the process.

The implementation of logic is shown as a flow chart in Figure 5.

3.4.3 Extensibility and Limitations

The main way in which `main.py` is currently designed to be extensible is in the integration of more complex solvers. At present, solvers can be initialised with a singular timeout parameter. However, in anticipation of future more complex solvers which require more initialisation parameters, a space for that logic has been left in the `get_solver_kwargs` function. This function, which currently takes as input the users arguments, can easily be designed to handle the parsing of an input configuration file. This capability paves the way for the incorporation of advanced solvers without necessitating substantial modifications to the existing codebase. The main limitation of main.py is that adding more output metrics like backtracking recursions, or other solver based statistics requires some plumbing work. Further, implementing parallel processing will require refactoring the code.
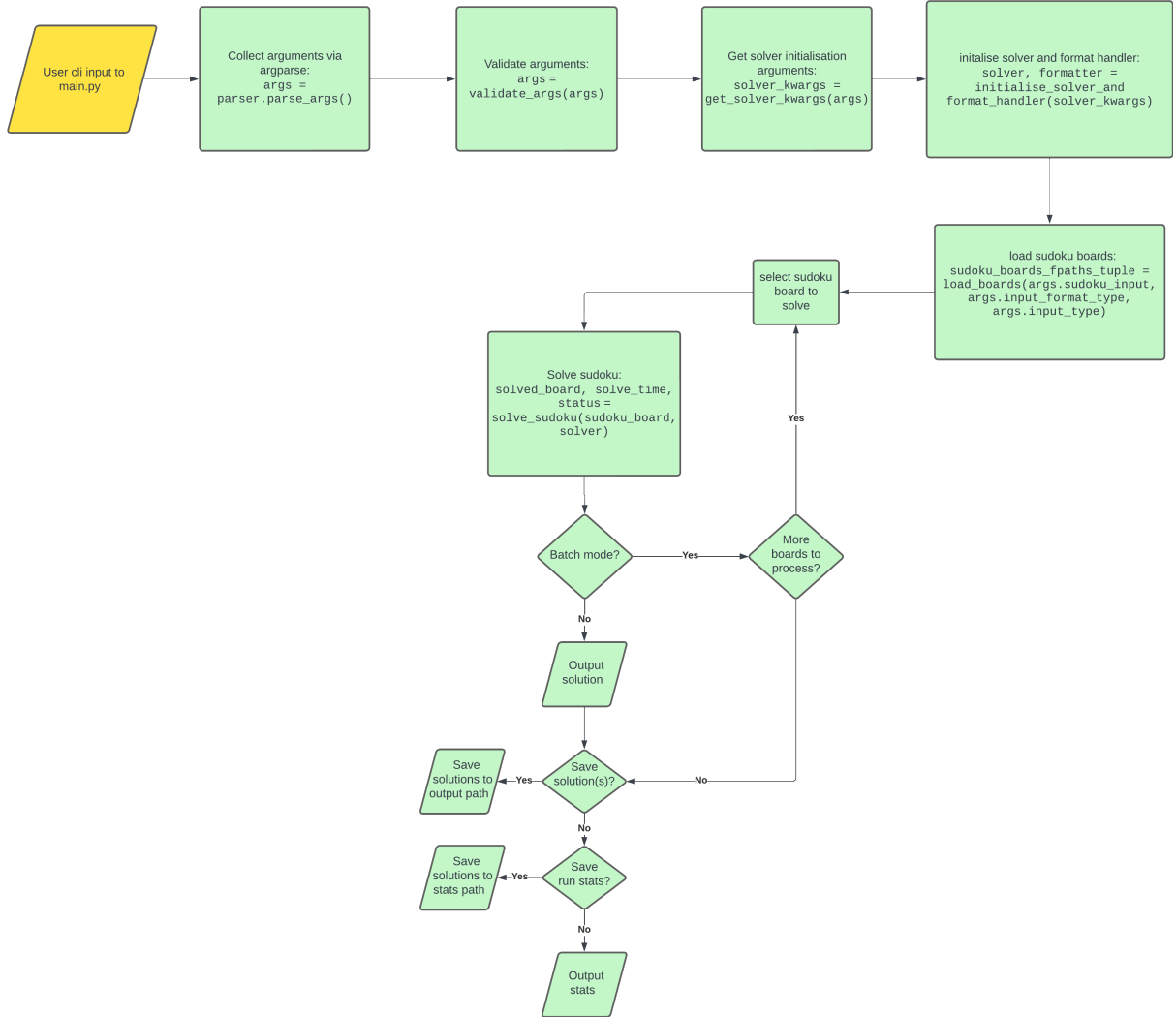

Figure 5: Flowchart of the main.py script.

4 Profiling and optimisation

Line profiling of the initial basic backtracking solver, implemented at the time as `BacktrackingSolver`, identified two critical functions: `check_valid` in `SudokuBoard` and `_backtrack` in `BacktrackingSolver`. `check_valid` emerged as a significant bottleneck, taking 242.043 seconds, largely due to frequent calls from `_backtrack`, which itself consumed 514.918 seconds, with 91.5% of this time attributed to `check_valid`. In response, the `SudokuBoard` class was restructured to store the

used values of each row, column, and subgrid in class attributes as a list of sets. This change leverages the efficiency of hash tables, inherent in set data structures, for quicker lookups, instead of iterating over the board repeatedly. This adjustment reduced the execution time of `check_valid` to just 34.9543 seconds, and the total time for `_backtrack` was brought down to 198.044 seconds. Consequently, the time percentage of `_backtrack` spent on `check_valid` decreased to 73.8%, reflecting a more evenly distributed computational effort and a significant enhancement from the original profile.

(a) `BacktrackingSolver._backtrack()` line profile results before optimisation

(b) `SudokuBoard.check_valid()` line profile results before optimisation

Figure 6: Initial line profiling results before optimisation of `SudokuBoard.check_valid` - git commit 4c1278cf

```
Total time: 198.044 s
File: /Users/vishaljain/c1_cw/src/sudoku_solvers.py
Function: _backtrack at line 8

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           @profile
     9                                           def _backtrack(self):
    10                                               """
    11                                               Solve the sudoku puzzle using a backtracking algorithm
    12
    13                                               Returns
    14                                               -------
    15                                               bool
    16                                                   True if the puzzle was solved, False otherwise
    17                                               """
    18                                               # find the next empty square
    19   9727397  18823477.0      1.9      9.5       i, j = self.board.find_empty()
    20
    21                                               # if there are no empty squares, the puzzle is solved
    22   9727397    736175.0      0.1      0.4       if i is None:
    23         1        0.0      0.0      0.0           return True
    24
    25                                               # check which numbers are valid in the empty square
    26  97273649   9429274.0      0.1      4.8       for num in range(1, 10):
    27  87546317 146076686.0      1.7     73.8           if self.board.check_valid(i, j, num):
    28                                                       # make the move and repeat recursively
    29   9727396   9154819.0      0.9      4.6               self.board.place_number(i, j, num)
    30   9727396   4126560.0      0.4      2.1               if self._backtrack():
    31                                                           # puzzle was solved
    32        64        2.0      0.0      0.0                   return True
    33
    34                                                       # if the puzzle was not solved, undo the move
    35   9727332   8921498.0      0.9      4.5               self.board.remove_number(i, j, num)
    36
    37                                               # if no numbers worked, an incorrect move was made somewhere
    38                                               # return False to backtrack
    39   9727332    775271.0      0.1      0.4       return False
```

(a) `BacktrackingSolver._backtrack()` after optimisation


```
Total time: 34.9543 s
File: /Users/vishaljain/c1_cw/src/sudoku_board.py
Function: check_valid at line 127

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   127                                           @profile
   128                                           def check_valid(self, row, col, num):
   129  87546317  15630796.0      0.2     44.7       subgrid_index = (row // 3) * 3 + (col // 3)
   130 103421351  17267512.0      0.2     49.4       return num not in self.rows[row] and num not in self.columns[col] and \
   131  15875034   2055983.0      0.1      5.9           num not in self.subgrids[subgrid_index]
```

(b) SudokuBoard.`check_valid()` after optimisation

Figure 7: Line profiling results after swapping from list based implemtation of `SudokuBoard.check_valid` to a set based implementation - git commit c3fad894.

5 Backtracking Solver comparison

The performance of the basic backtracking solvers was evaluated using two distinct datasets: a set of 1 million Sudoku puzzles sourced from Kaggle [1], and a collection of 100 puzzles known to challenge backtracking algorithms which can be found in the repository at the path `benchmark_board_sets/hard_100`. A timeout limit of 120 seconds is set to ensure completion of the solver on the hard puzzles. The results are shown in Figures 8 and 9 .

The backtracking with the easiest first heuristic is slightly slower than the basic backtracking solver on the easy puzzles due to the increased overhead of checking the number of possible values for each cell, but significantly faster on the hard puzzles. Showing the effectiveness of the heuristic in reducing the number of branches explored. Infact, two of the hard puzzles are not solved by the basic backtracking solver within the 2 minute timeout limit.

These results can be found in the `solver_output` directory in the repository.

Solve Time Metric	Basic	Easiest First
Average (ms)	0.745	1.30
Median (ms)	0.550	1.28
Min (ms)	0.273	1.08
Max (ms)	99.0	140
Std (ms)	0.725	0.261

Figure 8: Comparison of basic backtracking vs backtracking with easiest first heuristic performance metrics on the Kaggle million dataset.

Solve Time Metric	Basic	Easiest First
Average (s)	7.21	0.674
Median (s)	0.953	0.0815
Min (ms)	1.38	2.17
Max (s)	107	11.4
Std (s)	15.3	1.87
Timeout instances	2	0

Figure 9: Performance metrics comparison: basic backtracking vs easiest first heuristic on 100 puzzles designed for challenging the backtracking algorithm.

6 Software Engineering Practices

6.1 Exception Handling

The application includes an exceptions module `src/exceptions.py` with exceptions like `FormatError` and `TimeoutException` for targeted error scenarios. Common user errors are handled by the `main.py` script with try except blocks.

6.2 Version control

Git played a crucial role in version control. Branches were systematically named using feature/ and refactor/ conventions to enhance readability, while also ensuring the stability of the main branch. A release branch was used when generating the runs for the tables in figures 8 and 9 to ensure reproducibility. Commit messages focused on explaining the reason for the change as well as the change itself. A .gitignore file was used to ensure that only the necessary files were committed.

6.3 Unit Testing and Continuous Integration

The development of this programme was underpinned by a test-driven approach. Where possible, unit tests were written alongside the implementation of the corresponding functionality. As the project evolved, since the core logic remained largely unchanged, the initial unit tests remained relevant and effective. Unit tests were implemented using the pytest framework. Mocking is used where necessary in conjunction with testing on sample boards located in `test/{module_name}_test_boards`. The tests are run automatically on every commit through pre-commit hooks in found in the `.pre-commit-config.yaml`. A practice in line with CI/CD strategies to ensure code quality and functionality before integration. The pre-commit hooks also fix file endings, remove trailing whitespaces, ensure consistent line endings. It also ensures the python scripts are properly formatted with Black and checks for PEP8 compliance with Flake8.

6.4 Packaging and Accessibility

For maximum accessbility, the programme was containerised via docker which provides a lightweight and portable solution enabling consistent and reproducible deployments. The environment was created using Conda and is reproducable through the associated environment.yml file. Instructions for running the containerised application are provided in the README.md file.

7 Conclusion

The report details the Sudoku solver's development, highlighting problem decomposition, solution design and implementation, underpinned by software engineering principles like modularisation, defensive programming, single responsibility, and encapsulation. These practices not only deliver an effective solver but also set a foundation for future enhancements and adaptability.

8 Appendix

The autocomplete feature of GitHub Copilot was used whilst developing code throughout the project in the README.md and the report.

The specifications of the machine used for profiling and benchmarking are as follows: 2021 MacBook Pro with a 2.4 GHz 10-Core M1 Pro processor and 16GB unified memory. The 8 high performance cores operate at a clock speed of 3.2 GHz and the 2 low performance cores at 2 GHz. The operating system was macOS Ventura version 13.4.1 (22F82).

References

[1] 1 million sudoku puzzles. `https://www.kaggle.com/datasets/bryanpark/sudoku/`. Accessed: date-of-access.