C1: Research Computing

Max Talberg

December 16, 2023

Contents

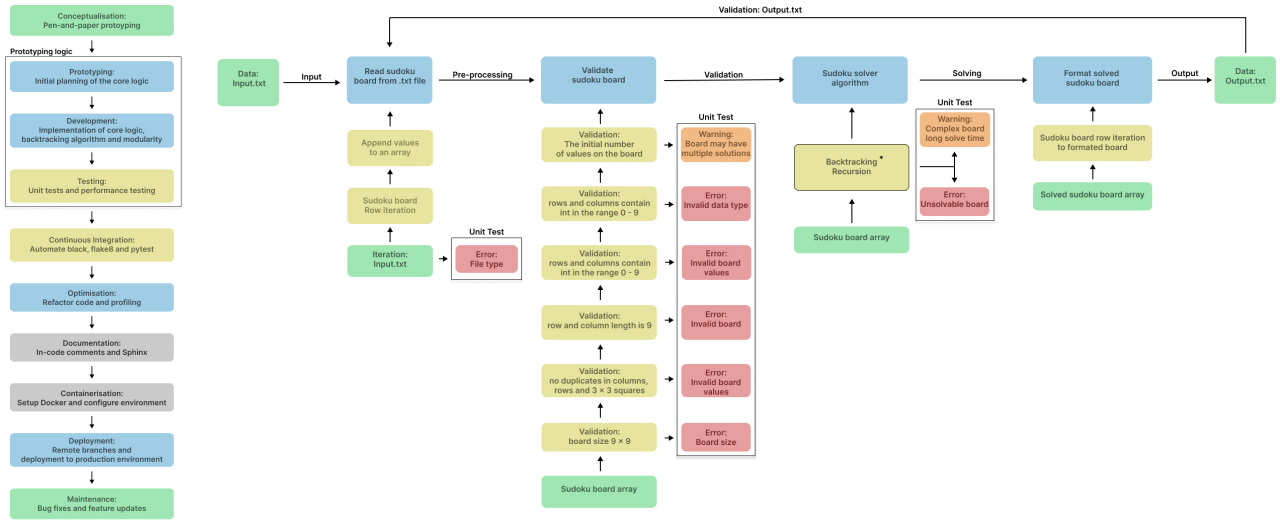
0.1 Introduction 2
0.2 Algorithm Design and Prototyping 2
0.3 Development and Unit Testing 3
0.4 Advanced Experimentation and Refinement 4
0.5 Packaging and Usability 7
0.6 Summary and Future Work 8

0.1 Introduction

This short report presents the development process of a Sudoku solver project, developed in Python. The emphasis of this project is to implement robust software development practices in conjunction with the development of sophisticated coding techniques. Throughout the report, we will detail these processes and the challenges faced in their implementation. We will also discuss prototyping, unit testing, version control, optimisation, documentation and the use of containerisation tools, providing insight into the software development.

0.2 Algorithm Design and Prototyping

The initial phase of the Sudoku project involved conceptualising the entire development process. This prototyping step outlined a high-level workflow, which we refer to as the larger software development cycle (LSDC). The LSDC encompasses all of the phases involved in good software development practices, from the initial planning to the final deployment. Within this high-level workflow we focused on the prototyping logic specific to the algorithm design which is to be coded. Figure 1 illustrates the high-level LSDC and the detailed prototyping logic.



(a) LSDC (b) Prototyping logic

Figure 1: (a) High-level workflow of the entire Sudoku solver project, detailing the integration of appropriate software development practices with the core algorithmic logic. (b) A closer look at the detailed prototyping of the core algorithmic logic to be coded.

The first step when implementing the prototyping logic is to understand the constraints and laws of Sudoku. This project is built on the following classic rules of Sudoku[1]: The objective is to fill a 9 x 9 gird with digits so that each column, row and 3 x 3 square contains all the digits from 1 to 9, without repetition. A well constructed Sudoku has a single solution, with a minimum number of 17 clues provided[2].

The coding process began in a singular Python file, containing functions for reading the puzzle, solving it and printing the output. This initial approach evolved into a more sophisticated structure with error handling and class abstraction. Ultimately to improve modality we transitioned to four separate classes: ***SudokuReader***, ***SudokuBoard***, ***SudokuAlgorithm*** and ***SudokuFormat***. Generating these four classes in individual files greatly improved the readability and facilitated targeted optimisation.

SudokuReader Class

This class is responsible for the reading of a Sudoku board from a text file. It parses the file's contents, transforming the Sudoku grid into a structured list of lists for further processing. This initialisation is key as it ensures the board is in an appropriate format for the validation and solving algorithms.

SudokuBoard Class

This class is the central component of the puzzle processing and handles all things related to the Sudoku board. It serves two fundamental roles: first, it transforms the initial representation of the Sudoku board from a list of string values into a 2D list of integers, in preparation for validation. Second, it undertakes the validation process, ensuring the board adheres to the classic Sudoku rules, as well as some additional edge cases that are discussed in more detail later. Upon confirmation that all the conditions are satisfied the class returns 'True' status which signals a valid Sudoku board and allows the solving process to begin.

SudokuAlgorithm Class

This class is the core of the Sudoku solver, executing a backtracking algorithm to solve puzzles. After receiving a validated 2D list of integers, it employs backtracking a method similar to a brute force that systematically traverses the Sudoku grid in search of a solution [11]. Backtracking is a recursive algorithm, in this context the algorithm cycles through possible values (1 to 9) for each cell, advancing when a valid number is found or backtracking when a dead end is reached. This iteration process continues until the algorithm finds a solution or exhausts all possible values for the grid and determines the puzzle to be unsolvable.

Although backtracking is straightforward and effective for simpler puzzles, it becomes less efficient with increasing difficulty. Recognising this, we considered implementing constraint propagation on top of the backtracking algorithm. Constraint propagation uses the rules of Sudoku to prune the Sudoku gird and guide the solver to areas where solutions are more likely to be found [10]. Much later on this led to the implementation of the Minimum Remaining Value (MRV) heuristic [14]. The MRV heuristic is an addition to the backtracking algorithm, it chooses squares with the fewest legal moves. By prioritising the most solvable squares the MRV heuristic improves the efficiency of the backtracking algorithm.

This was the only other algorithm we had time to implement. The ***SudokuAlgorithm*** was initially developed with the idea that it could contain multiple solving algorithms, to later launch an optimisation investigation for solving harder puzzles.

SudokuFormat Class

This class handles the formatting of the solved Sudoku board. It takes the solved 2D list and iterative transforms it into the specified output format to be printed on the console.

0.3 Development and Unit Testing

This section will take a closer look at the development process and unit test practises adopted during the project.

SudokuRead Class

In the early stages, error handling was integrated during the reading phase. Invalid characters and incorrect row lengths would be flagged as the data was iteratively read in. However to maintain modularity and due to the low computational cost, validation checks were isolated from the reading function. The final class only reads text files and returns a 2D list of strings, with error handling for not existent files, incorrect file types or empty files set as runtime errors and captured in unit tests (`tests/test_sudoku_read.py`).

SudokuBoard Class

This class began by handling column-related errors but was refactored to perform all the validation tasks. The class checks the rules of Sudoku are being followed: It ensures board values are integers in the appropriate range, with no duplicates in any row, column or 3 x 3 square. The class also handles edge cases: Including invalid board sizes, invalid board format and text files containing normal text. These errors are set as value errors and captured in unit tests (`tests/test_sudoku_board.py`). We also implemented warnings for scenarios likely to lead to multiple solutions, such as a puzzle with fewer than 17 clues. This was informed by the characteristics of a well-constructed puzzle [2] and an investigation into the solver's time complexity, concerning the number of initial empty squares. A more detailed analysis of these findings and their implications is explored in the Advanced Experimentation and Refinement section.

SudokuAlgorithm Class

With the assumption of a validated board the algorithmic part of the code has few error handling instances. Unit tests confirm the solver's performance and solution accuracy (`tests/test_sudoku_algorithm.py`), as well as the edge case of an unsolvable puzzle. During this development we utilised a combination of mocking and data file imports for testing. Importing files improves readability and replicates the end application use. Whereas the mocking board ensures testing isolates components, as we are not relying on reading and validating a puzzle before it is solved, we have just isolated the expected solution.

SudokuFormat Class

This class was initially part of the validation function, although was separated for modularity, following the initial prototype logic. It converts the solved board as a 2D list of initegers into a formatted string for output, with unit tests to ensure correct conversion (`tests/test_sudoku_format.py`). This test used mocking to isolate the unit test to just the formatting class.

Continuous Integration

The use of Continuous Integration (CI) early on in the coding process proved to be incredibly useful. CI was integrated in the project using pre-commit hooks with Black, Flake8 and Pytest, which enforced the PEP8 styling and conducted continuous testing. CI meant that each at commit all of my unit tests would run and the code would be formatted and styled. This ensured early bug detection and consistent code quality. This was particularly useful when transitioning from a singular class project into a more complex four class system.

0.4 Advanced Experimentation and Refinement

Exploring Edge Cases and Complex Scenarios

The design of our unit tests initially focused on ensuring compliance with Sudoku rules, with code identifying and specifying the location of errors. Through research and brainstorming [3], we identified several edge cases including: Invalid board sizes, invalid board format, inputs that didn't represent a board at all and unsolvable puzzles. After developing unit tests and error codes for these edge cases it became clear there are scenarios where the solver should issue warnings without stopping execution. An example of this is when scenarios produce multiple solutions, even when following classical Sudoku rules. This can typically occur when there are less than 17 clues, this led to some experimentation to understand the solvers performance at various numbers of clues.

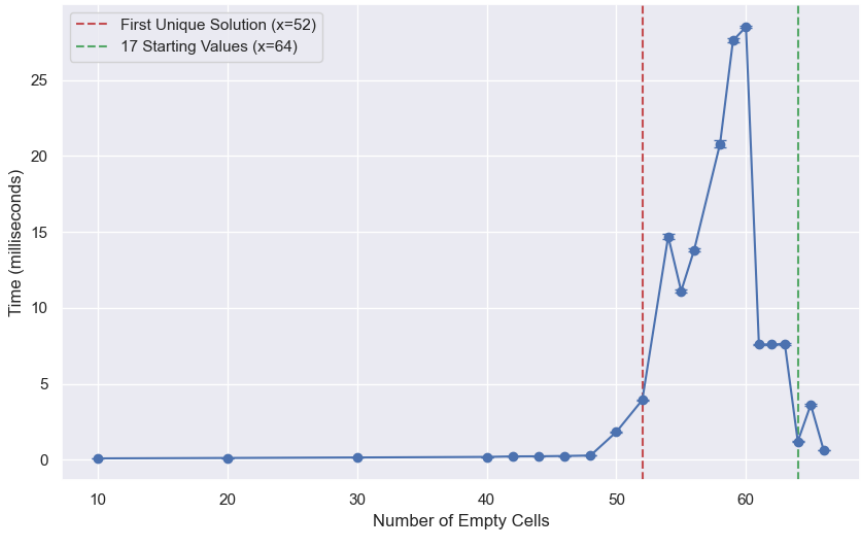


Figure 2: Time complexity of the backtracking algorithm

Our investigation into the time complexity of the solving process involved measuring how long our solver took to solve a Sudoku puzzle while systematically varying the number of initial empty squares. This study revealed a significant increase in solve time and non-uniqueness of solutions when the empty cells exceeded a certain threshold (52 empty cells). There is clearly less variability at this puzzle complexity forcing the backtracking algorithm to deeper levels of recursion. After 61 empty cells the time taken significantly drops, likely due to the increased accessibility of the numerous potential solutions.

To address this problem, warnings were implemented for boards with fewer than 17 clues or empty boards. The code still runs, it will just produce a warning that there is a possibility of multiple solutions.

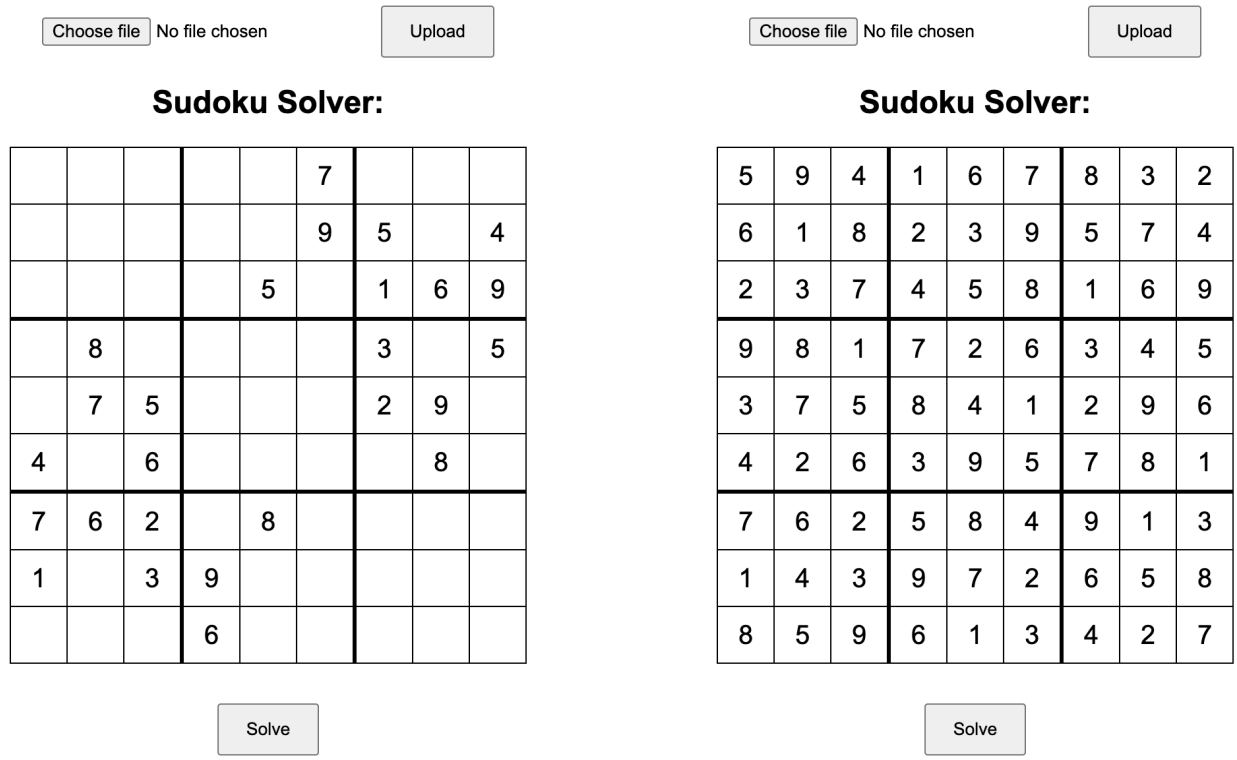
Although this was only performed on one puzzle [9], figure 2 gave practical insight on the performance characteristics of Sudoku boards with a high number of empty cells.

Iterative Development and Branch Management

Utilising Git alongside CI maintained a high quality of code throughout development. Branching allowed for safe experimentation and feature development without risking the functionality of existing work. The evolution of the project is evident in version control: the first release branch (V1.0.0) contained a single class in one file, while the third release branch (V3.0.0) four seperate files for source and testing. Version facilitated continuous development while preserving previous work done.

Flask

An exploratory extension we decided to keep involved developing a user interface (UI) for the Sudoku solver using Flask [12]. Flask is a lightweight Python module for developing web applications. The layout of the interface was developed in HTML, this simple layout consisted of an option to choose or drag a file, an upload button and a solve button. The upload button is linked ***SudokuRead*** class, once clicked the read-in board will fill up the oncscreen board unless an error is thrown. The solve button is linked to the ***SudokuBoard*** and ***SodukuAlgorithm*** classes, once clicked the board is validated then solved unless an error occurs. In that case an error message will pop-up on the screen. Time permitting, future enhancements could include an animation of the backtracking process or other algorithms.



(a) Uploaded puzzle

(b) Solved puzzle

Figure 3: Figures show the UI before and after puzzle resolution, illustrating the solver's user interaction flow.

Performance Profiling and Optimisation

Profiling and optimisation are essential processes in enhancing the efficiency of code. After releasing the V1.0.0, we applied the `%timeit` to measure the time complexities of the code. This is expanded upon in the Time Complexity Investigation below.

In the initial development of the validation code, checks for duplicates in the rows, columns and 3 x 3 squares was performed in three separate iterations. Optimising the code led to these checks being consolidated into a single iteration. Additionally, more extensive use of list comprehensions was implemented. Although solving a Sudoku isn't computationally intensive, such optimisations is crucial in more complex projects that require scaling up.

Time complexity Investigation

puzzles_kaggle	**puzzles/sec**
rc_mt942v3.0.0	31.3
rc_mt942v3.1.0	10.27
fsss [4]	1,477,910.9
jczsolve [5]	597,074.2
sk_bforce2 [6]	1,234,240.0

Table 1: Performance comparison of different Sudoku solvers on a Kaggle dataset of 1,000,000 puzzles.

Table 1 presents a performance benchmark using Kaggle data set of 1,000,000 Sudoku puzzles [7]. It compares puzzles solved per sceond between our solver, rc_mt942v3.0.0 and several leading solvers. While rc_mt942v3.0.0 successfully solved all puzzles, its rate (0.17 puzzles/sec) is significantly lower than others like fsss2, jczsolve3, and sk_bforce2, which solve tens of thousands per second (Table 1).

puzzles_magic_tourtop1465	puzzles/sec
rc_mt942v3.0.0	0.17
rc_mt942v3.1.0	1.13
fsss2	69,752.5
jczsolve3	77,888.4
sk_bforce2	86,535.4

Table 2: Performance comparison of different Sudoku solvers.

In Table 2, we assess rc_mt942v3.0.0 against the Magic Tour Top 1465 dataset, a challenging set of puzzles [8]. Our solver's performance, although successful in solving all puzzles, remains slower in comparison.

Analysis revealed the competitor solvers use an advanced and optimised brute-force approach with bitwise operations [5][13]. Notably these solvers are implemented in C. The bitwise operations allow the solver to handle multiple candidates for multiple cells simultaneously, when executed in C it is clear why these solvers have such a performance advantage.

Utilising the `%timeit` function to pinpoint performance, revealed that the clear bottleneck of our code is the backtracking algorithm. The process of reading, validating and formatting all take nanoseconds, while the algorithm class requires microseconds.

Based on this bottleneck we implemented the minimum related value (MRV) heuristic into our Sudoku solver (rc_mt942v3.1.0) to enhance the backtracking algorithm. The MRV heuristic chooses the square that has the fewest legal values remaining, this guides the backtracking algorithm to this square. To investigate the performance we ran the solver through the benchmark tests. As expected the solver was much fast solving the Magic Tour Top dataset, almost an order of magnitude faster, demonstrating MRV's effectiveness on difficult puzzles. However, on the simpler Kaggle dataset, the solver showed a slowdown, suggesting an overhead in MRV calculations for easier puzzles. Despite the decrease in speed the MRV is fast for hard Sudoku puzzles and unless the user is iterating through a million puzzles the additional MRV heuristic has a better net performance than just the backtracking algorithm.

A potential optimisation, given more time, would be integrating the Degree Heuristic alongside the MRV heuristic during backtracking [15]. The Degree Heuristic assigns a value to the variable involved in the largest number of constraints. If paired with MRV the pair provides a balance between searching for the most constrained variables (MRV) and those that have the greatest impact on the rest of the puzzle (Degree).

Furthermore, the adoption of Cython or a complete transition to C could prove beneficial. Cython enables the use of the speed benefits of C while maintaining the readability and structure of our Python code. Alternatively, rewriting the code in C would provide the performance benefits of a compiled language compared to an interpreted language, although sacrifices readability and ease of maintenance.

Further research could focus on comparative analysis of various algorithms on these tasks.

0.5 Packaging and Usability

The project commenced with the creation of a dedicated Conda environment. This isolates the current project allowing control over the dependencies used.

Dockerfile and Environment Setup

Deploying a project via Docker ensures consistent performance across different hardware configurations. Our setup includes a Dockerfile, which contains a series of commands to build an image of the project. This includes specifications for working directories, an exposed port for the Flask application and the project's environment. The Dockerfile references an `environment.yml` file that lists all the necessary environmental dependencies. These dependencies are installed during the image-build, ensuring the project runs reliably in the Docker container.

Documentation

Sphinx was chosen for documentation due to its aesthetic appeal compared to Doxygen. Sphinx is integrated with the code base, automatically updating after each run. Implementing a Continuous Development (CD)

pipeline would streamline this process and automating documentation updates and making them available on GitLab. This is however only possible in a public directory. Our project follows NumPy style docstrings, by implementing `sphinx.ext.napoleon` in the `conf.py` file ensured Sphinx correctly interprets this notation.

Documenting code is key for simplifying code maintenance and debugging. Using Sphinx, we on can centralise the documentation for each release branch or commit. Facilitating easier access and comprehension of the code throughout the project.

0.6 Summary and Future Work

In this project we developed a functional Sudoku solver using the backtracking algorithm compatible with the classical rules of Sudoku.

The project began by prototyping the high-level workflow with the LDSC and the core algorithmic logic, which was then turned into code. This initial version of code was later refactored into four distinct classes: ***SudokuReader***, ***SudokuBoard***, ***SudokuAlgorithm*** and ***SudokuFormat***. Collaboratively these classes solve Sudoku puzzles.

Further complexity was introduced with advanced error handling within these classes. Addressing issues such as breaches of the classical rules, handling of incorrect inputs and warnings for less than 17 starting values. These errors were integrated into our unit tests to ensure the robustness of the code.

The incorporation of CI tools like Black, Flake8 and Pytest ensured automatic testing with every commit. This greatly aided the development process by ensuring early bug detection and a consistently clean codebase.

Once the core logic was established and functioning alongside unit tests, the project entered an advanced stage of experimentation and refinement. This involved optimising the validation process and assessing the models performance on a variety of benchmark datasets, alongside renowned Sudoku solvers. Additionally this experimentation stage resulted in the development of simple UI using Flask.

The final stages of the LSDC consisted of refining the projects docstrings for documentation via Sphinx. We also developed an Docker image, ensuring a self-contained environment for easy deployment across different devices.

Potential Areas for Future Improvement

The most significant area for improvement in Sudoku performance lies in optimising the backtracking algorithm, which currently is the largest bottleneck. Future work should focus on implementing the Degree heuristic to enhance the algorithms efficiency. Additionally, integrating Cython into the project or developing a different algorithm altogether in Cython or C, could offer faster solutions due to performance advantage of these languages. Alternatively, implementing an advanced and optimised brute-force approach in C, similarly to the leading solver, might be the fastest approach.

Another area for development is the sophistication of the error handling system, particularly for puzzles with multiple solutions. Given more time, we aim to identify puzzles with multiple solutions and presents all of the solutions.

The current front-end of the solver is quite basic. Future plans would aim to visualise the solving process by integrating an animation of the backtracking process or an alternative method to visually understand these algorithms.

In streamline the projects maintenance implementing CD with the project would enable automatic deployment of the projects documentation. Additionally, CI with Docker would facilitate continuous deployment of the project.

Bibliography

[1] https://en.wikipedia.org/wiki/Sudoku

[2] https://arxiv.org/abs/1201.0749

[3] http://sudopedia.enjoysudoku.com/Invalid_Test_Cases.html

[4] https://github.com/dobrichev/fsss2

[5] http://forum.enjoysudoku.com/3-77us-solver-2-8g-cpu-testcase-17sodoku-t30470-210.html#p249309

[6] https://github.com/GPenet/SK/_BFORCE2

[7] https://www.kaggle.com/datasets/bryanpark/sudoku

[8] http://magictour.free.fr/sudoku.htm

[9] https://gitlab.developers.cam.ac.uk/phy/data-intensive-science-mphil/c1_assessment/mt942/-/blob/main/Instructions.md?ref_type=heads

[10] https://classes.engineering.wustl.edu/cse231/core/index.php?title=Sudoku_Constraint_Propagation

[11] https://en.wikipedia.org/wiki/Backtracking

[12] https://flask.palletsprojects.com/en/3.0.x/

[13] https://en.cppreference.com/w/c/language/operator_arithmetic

[14] https://steven.codes/blog/constraint-satisfaction-with-sudoku/

[15] https://codepal.ai/code-generator/query/fB2PHcbt/java-sudoku-solver-ac3-algorithm: :text=The