Ivo Vladislavov Petrov
CRSID: ivp24

Research Computing Project Report

Sudoku Solver

Data-Intensive Science MPhil

December 17, 2023

Word Count: 2997

Contents

List of Figures

Chapter 1

Introduction

The **Sudoku** puzzle is among the most widespread numerical puzzles in the world. While such puzzles have existed in newspapers since the 19th century, the modern version of Sudoku gained its popularity in the late 20th century from Japan [1]. A 9×9 board is partially filled with numbers, with the objective of filling the missing cells. The rules are simple - each row, column, or 3×3 subgrid (block) has to contain all the numbers from 1 to 9 with no repetition. Puzzles of varying difficulty have been part of people's pastime for decades, spending from minutes to hours to solve one.

In this report, we detail a fully automated Sudoku solver, fully created in the Python programming language. The program was designed to mimic human approaches to solving a Sudoku, while also being vastly superior in efficiency. It is built with robustness and readability in mind, ensuring that any user with the correct setup can use the scripts. We will describe in depth the process of developing the aforementioned program, with a particular focus on good programming practices. This includes but is not limited to prototyping, maintaining a repository, code correctness through validation and experimentation.

Chapter 2

Selection of Solution Algorithm and Prototyping

2.1 Solution Algorithm

2.1.1 Backtracking

Usually, computer-based Sudoku solvers utilise some form of backtracking algorithm[2].

Definition 2.1.1 Backtracking is a programming technique, which is used to reach solutions using a brute-force approach[3]. The algorithm guesses a value for an unsolved state, accepting it as the ground truth. If at some point, the board becomes unsolvable, the algorithm returns or **backtracks** to the previous valid state.

However, backtracking is usually computationally expensive. In the case of Sudoku, a solution using only this method does not update its assumptions based on the available information until it reaches an error. At that point, the mistake might have been made many guesses ago, and the backtracking algorithm will take a lot of steps to revert to that state. In the solution, we employ the use of a backtracking algorithm "as a last resort", giving a higher priority to the logic methods, as they guarantee correct behaviour.

2.1.2 Logic

As the main component of the solution, we used standard logic identical to human-based solving techniques. While plenty of rules are obvious for a human, they might be computationally expensive for computers. Therefore, we decided to limit development to only the simplest rules. In particular, we make use of the rules **Obvious Singles**, **Hidden Singles** and **Obvious Pairs**. We also utilise a combination of the **Pointing Pairs** and **Pointing Triples** rules, which we dub **Hidden Pointers**. Short descriptions of the rules can be found below, with further details described in the SudokuRules website [4].

- **Obvious Singles** - if there is an unsolved cell with a single possibility remaining, we can solve the cell.
- **Hidden Singles** - if a certain number is valid in a single cell inside a row, column or block, the cell must contain said number.
- **Hidden Pointers** - if inside a block, a number only appears in a single row/column, all other instances of the number in the same row/column can be removed from the possibilities.

- **Obvious pairs** - if 2 numbers are the only remaining possibilities for 2 cells in the same row, column, or block - we can remove them from the possibilities of the respective row, column or block.

A visual summary of the rules can be observed in Figure 2.1.

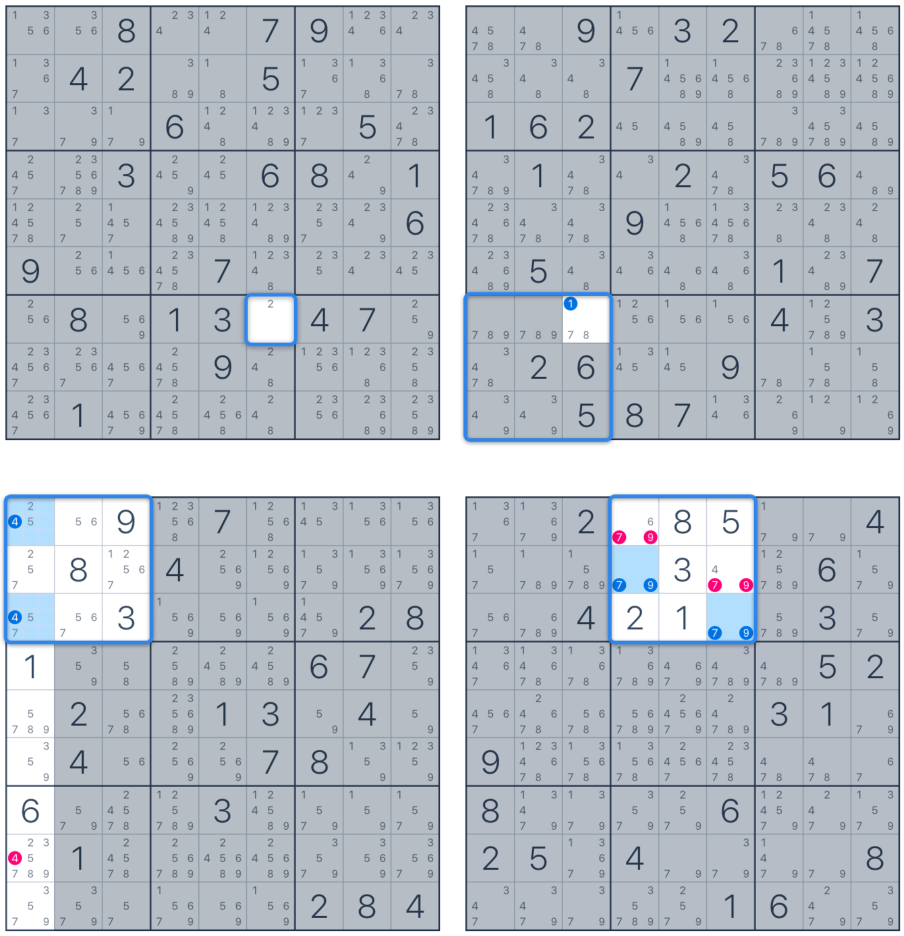


Figure 2.1: *Examples of each rule:* ***Obvious singles*** *(top-left),* ***Hidden singles*** *(top-right),* ***Hidden pointers*** *(bottom-left),* ***Obvious pairs*** *(bottom-right). Here blue cells show the relevant signal, while the red show possibilities being removed. Examples were obtained from the SudokuRules website [4].*

2.2 Prototyping

To reach a solution starting from a configuration file, we need three different components, as shown in Figure 2.2. They will all be discussed in further detail below.

2.2.1 Parser

The parsing module takes an input file, in the form of either a text file or a configuration. The latter is preferred in the case of specifying the type of output, as well as the logic components that are to be used. It can further be extended to support multiple runs by allowing custom output paths. For this purpose, the `configparser` package provides the needed utilities.

Then, to preprocess the input, we require a 9×9 or a 11×11 matrix - corresponding to a raw Sudoku board or one with boundaries between the subgrids (blocks). The parser should also be robust - being able to reshape the input in the case of small mistakes, such as trailing spaces.

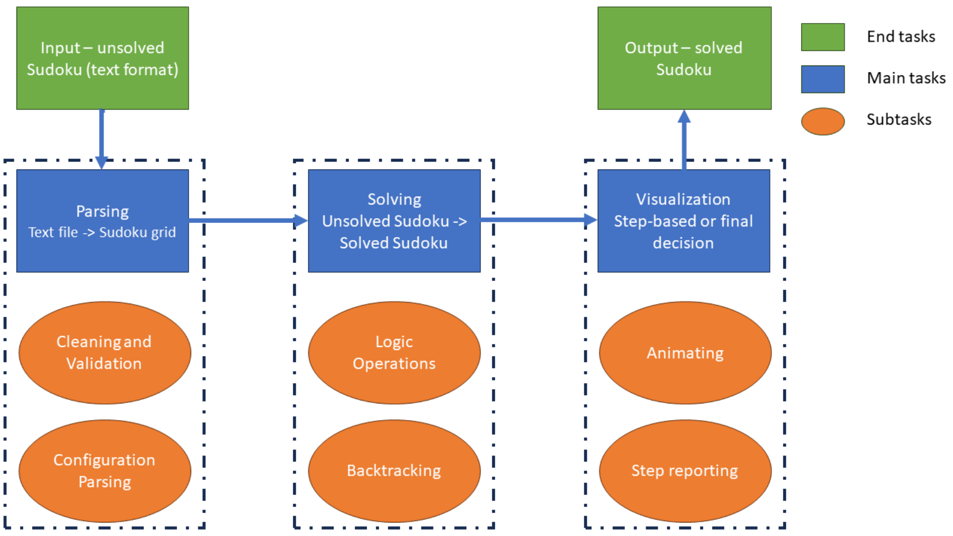

Input – unsolved Sudoku (text format)
Output – solved Sudoku
End tasks
Main tasks
Subtasks
Parsing Text file -> Sudoku grid
Solving Unsolved Sudoku -> Solved Sudoku
Visualization Step-based or final decision
Cleaning and Validation
Configuration Parsing
Logic Operations
Backtracking
Animating
Step reporting

Figure 2.2: *The three stages of obtaining a solution. Parsing takes the input and transforms it into a data structure we use for the board. The solver then fills out the empty cells. Finally, a visualization module prints out the solution in an animated or text-based format (if specified).*

In the case of a severely malformed input, the parser should report the issue.

Finally, we should convert the obtained 9×9 board into a useful data structure that can be used by the solver. We keep track of the state using a `NumPy`[5] array. It is also useful to keep track of all possible numbers in an empty cell. We can either recompute the possibilities upon request or keep track of a data structure upon updating the board. The former is computationally inefficient, as every module will require access to this information. Therefore, we will represent the possibilities using an array of sets, which have a membership check of $\mathcal{O}(1)$. The pipeline is shown in Figure 2.3.

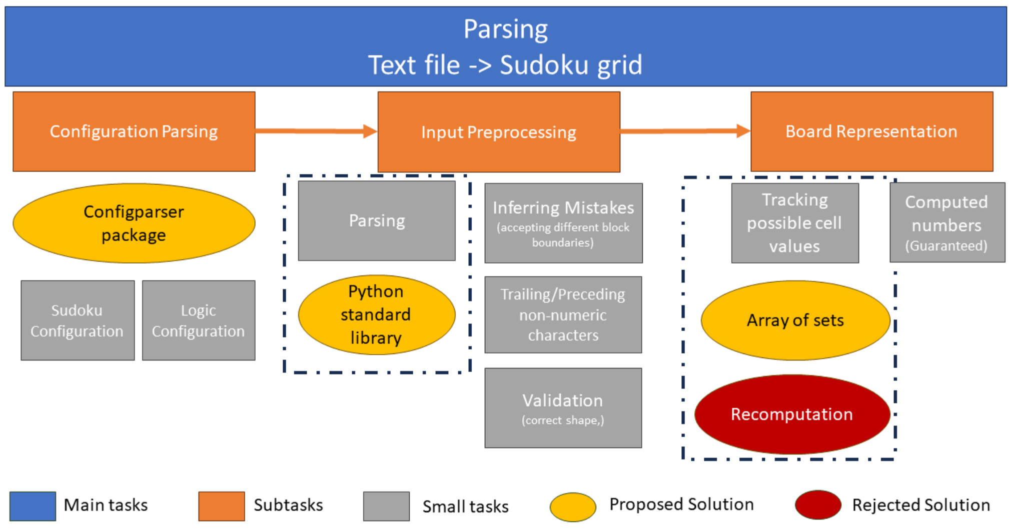


Figure 2.3: *The pipeline for the parser module. Solutions are connected with the related tasks. The rejected solution described can logically be confirmed to be inefficient but we will, nonetheless, consider it for the sake of experimentation.*

2.2.2 Solving

The solver module will use the aforementioned board structure and perform step-wise operations on it upon discovering a relevant signal. The logic rules are all preferred over the backtracking algorithm, due to their deterministic nature.

For the backtracking algorithm, we opted to develop two different versions for more optimal decision-making. The two algorithms can be described as so:

Definition 2.2.1
A **Naive Backtracker** will select the first unsolved cell, and make an arbitrary guess. The **Selective Backtracker** will instead scan the board, and select the most defined cell. This notion can be formalised as:

$$C_{i,j} = \{n | n \in \{1, 2, ..., 9\} \ \land n \text{ is a valid guess for cell in row } i\text{, column } j\}$$

The algorithm then elects the cell corresponding to $\text{argmax}_{i,j}|C_{i,j}|$

Remark While the latter seems more computationally expensive, we will show that it indeed shows a good performance improvement. The main reason behind that is the ability to quickly correct incorrect assumptions and provide more information to the logic components. For example, if we only have 2 options for a single cell, we gain more information if we guess that particular one, than in an arbitrary cell.

Recovering the previous state can be done upon reaching an unsolvable or invalid board. For this purpose, each algorithm needs a memory of what the state of the board looked like before each step. This will not be as memory-inefficient, with the algorithm using a Depth-First method[6], which has a memory complexity of $\mathcal{O}(depth)$.

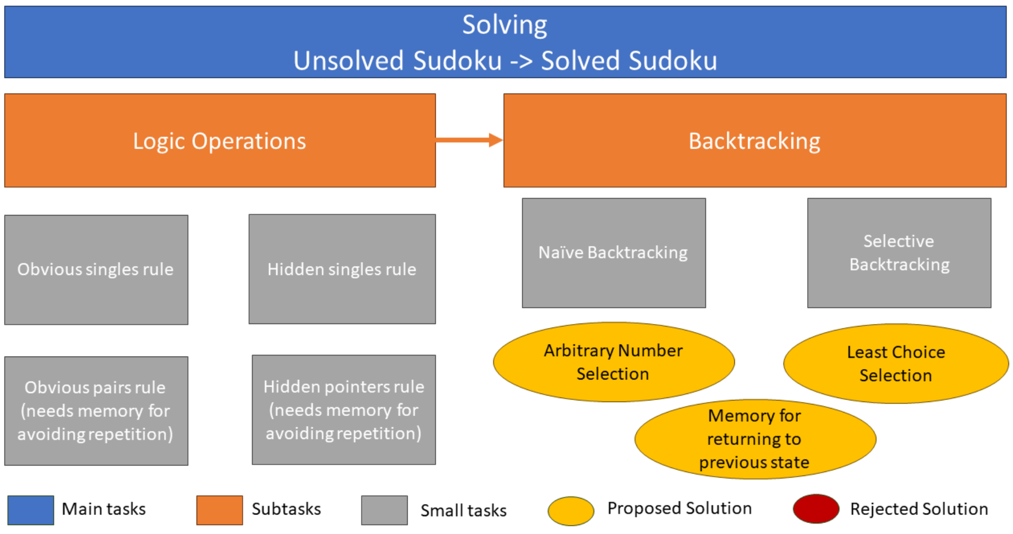

Solving
Unsolved Sudoku -> Solved Sudoku
Logic Operations
Backtracking
Obvious singles rule
Hidden singles rule
Naïve Backtracking
Selective Backtracking
Arbitrary Number Selection
Least Choice Selection
Obvious pairs rule (needs memory for avoiding repetition)
Hidden pointers rule (needs memory for avoiding repetition)
Memory for returning to previous state
Main tasks
Subtasks
Small tasks
Proposed Solution
Rejected Solution

Figure 2.4: *The pipeline for the solver module. Each solver will be made up of any non-zero number of logic operations, paired with a single backtracking algorithm. We use the backtracking algorithm as a last resort if no progress can be made using logic.*

2.2.3 Visualization

The visualization module should contain two separate modes - one displaying the animation for the solution, and one that can display the process in the command line. Using the former, one must keep track of the states and cell possibilities after each step. Then we can use `Matplotlib`'s `animation` module to show progress. If a cell is empty, we will show the value possibilities, so we will be able to display the work of the **Hidden Pointers** and **Obvious Pairs** rules. For the latter, we can display any changes if a logic/backtracking rule succeeds. We can implement this inside a rule's functionality by displaying the board state upon success.

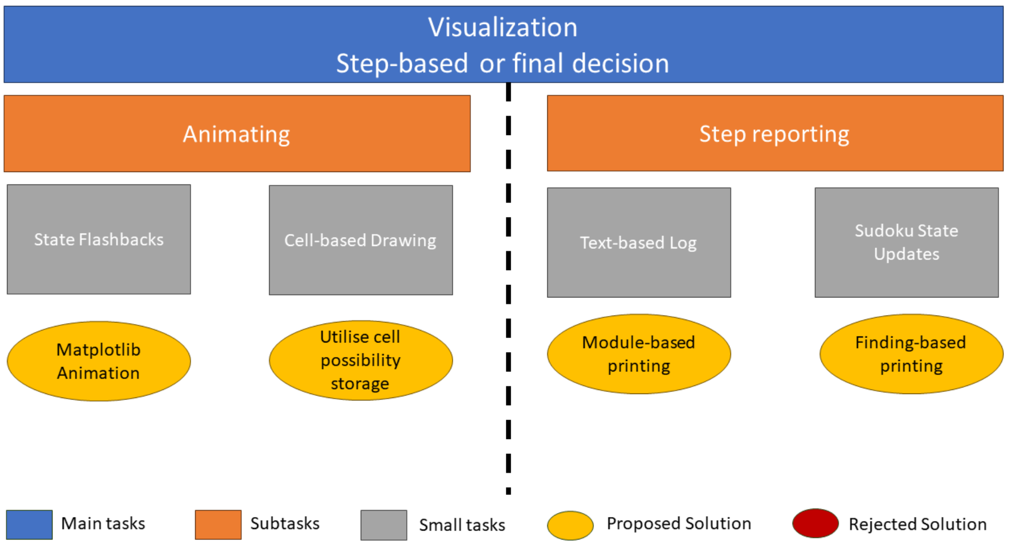


Figure 2.5: *The two visualization options for the module. The text-based explanation is mostly designed to detail how the different components of the solver affect the board. Hence, it also becomes a useful tool for debugging. Meanwhile, animation has been prototyped with the focus of understanding how the board state changes as progress is made.*

Chapter 3

Development, Experimentation and Profiling

3.1 Development

3.1.1 Git Repository

During development, we maintained an up-to-date Git repository with multiple branches. Each branch was created with a particular feature to implement in mind and the feature development was constrained to that particular branch. There were no commits pushed directly to the main branch. Furthermore, we ensured that all unit tests had been passed and the code conformed to the PEP8 standard before committing. Further details can be found in the *Continuous Integration* section. A list of all branches and their purpose can be seen below:

- **Main** - The main branch, where only correct code resides. Upon completing a feature, this is the branch it is merged to.
- **Experimentation** - A branch primarily used for optimization. Upon changing the solver pipeline, we used this branch to improve the efficiency of the algorithms.
- **Setup** - The first branch, dedicated to setting up the environment.
- **Parsing** - For parsing the sudoku board.
- **Basic Logic** - for the implementation of the **Obvious Singles** rule and the **Naive Backtracker**.
- **Animation** - for implementing an animation-based visualization.
- **Configuration setup** - for parsing a solver configuration.
- **Advanced Logic** - for additional logic rules, namely **Hidden Singles**, **Obvious Pairs** and **Hidden Pointers**.

3.1.2 Logic and Backtracking

The core components of the solver are a set of logic rules and a single backtracking algorithm. The rules are executed sequentially, and if none of them succeed, the backtracker is called to make a guess. **Class inheritance** was used to ensure certain properties of the subclasses were sustained, such as the existence of a `step` method, which attempts to work on the board.

Logic

Most logic rules do not involve any complex algorithms. Assuming the size of the board is N, with a block being of size $\sqrt{N} \times \sqrt{N}$, we can derive the complexity of each algorithm. Even though N is small, the scalability information is still generally useful, even though the constant terms will likely dominate. However, they can only be optimised through rigorous profiling.

- **Obvious Singles** - check for cells which have only a single option. The scan takes time $\mathcal{O}(N^2)$, and each check is in constant time, leading to an efficiency of $\mathcal{O}(N^2)$.
- **Hidden Singles** - for each column, row or block, check if a number only appears in a single cell. There are $\mathcal{O}(N)$ regions to check, with $\mathcal{O}(N)$ cells to check per region and $\mathcal{O}(N)$ options to scan. Therefore, the check will be of complexity $\mathcal{O}(N^3)$.
- **Hidden Pointers** - for each block, check whether a number only appears in a single row or column. There are $\mathcal{O}(N)$ blocks to check, with $\mathcal{O}(\sqrt{N})$ directions to check per block, and $\mathcal{O}(N)$ options to scan. Therefore, the check will be of complexity $\mathcal{O}(N^2\sqrt{N})$.
- **Obvious Pairs** - for each column, row or block, check if there is an isolated pair of numbers that appear alone in 2 separate cells. For $\mathcal{O}(N)$ regions to check, there are $\mathcal{O}(N^2)$ pairs of cells to compare, which can be checked for equality(and length) in $\mathcal{O}(1)$. Therefore, the overall complexity becomes $\mathcal{O}(N^3)$

Out of all the listed algorithms, only the **Obvious Singles** rule is scalable for larger sudoku puzzles. However, for the sake of this project, the constant factors matter more and we will look to optimize them in the *Optimization* section.

Backtracking

Backtrackers are developed in a similar way to logic rules, also inheriting from a base backtracker class. It ensures that backtrackers satisfy the following:

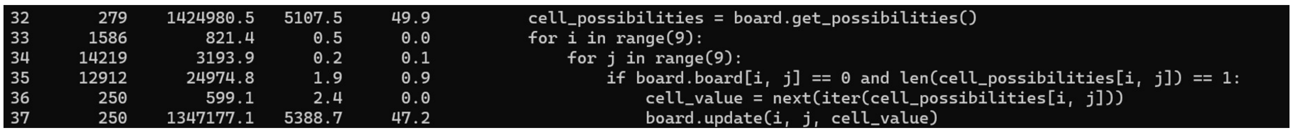
- The implementation should keep the states in memory before a guess, so the board can be restored.
- Upon reaching an unsolvable state, the backtracker should restore the previous valid state.
- After restoring the state, remove the wrong guess from the cell possibilities using built-in memory or by manipulating the board structure.

We created two different implementations - a **Naive Backtracker**, and a **Selective Backtracker**, as discussed in the *Prototyping* section. The advantages of both will be discussed in the *Optimisation* section.

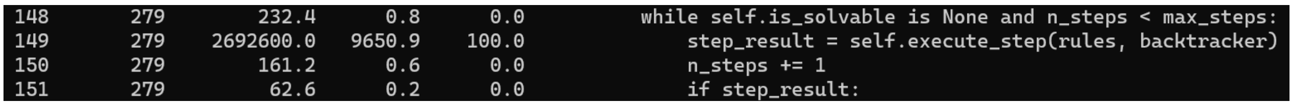
3.2 Profiling and Optimisation

We used the profiling framework provided by the `line_profiler` package. In particular, we applied profiling when running the `run_all_samples` script, which is further described in the *Validation* section. To summarize, we tested approximately 150 different Sudoku puzzles of different difficulties. This creates an average representation of the efficiency of each step.

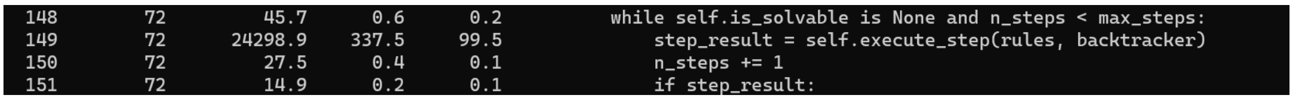
The first instance of profiling we used was primarily experimental, with us verifying the implementation efficiency of recomputing the cell possibilities against keeping an up-to-date data structure. As can be seen in Figure 3.1, the most expensive part of the code is indeed recomputing the information. Instead, we try a data structure implementation using a grid of sets keeping track of the options for each cell. This data structure is updated only when one of the rules has found a signal. As expected, we observe a great improvement of nearly 3000%.



```
32       279   1424980.5   5107.5     49.9          cell_possibilities = board.get_possibilities()
33      1586       821.4      0.5      0.0          for i in range(9):
34     14219      3193.9      0.2      0.1              for j in range(9):
35     12912     24974.8      1.9      0.9                  if board.board[i, j] == 0 and len(cell_possibilities[i, j]) == 1:
36       250       599.1      2.4      0.0                      cell_value = next(iter(cell_possibilities[i, j]))
37       250   1347177.1   5388.7     47.2                      board.update(i, j, cell_value)
```



```
148       279         232.4       0.8       0.0          while self.is_solvable is None and n_steps < max_steps:
149       279     2692600.0    9650.9     100.0              step_result = self.execute_step(rules, backtracker)
150       279         161.2       0.6       0.0              n_steps += 1
151       279          62.6       0.2       0.0              if step_result:
```



```
148        72        45.7       0.6       0.2          while self.is_solvable is None and n_steps < max_steps:
149        72     24298.9     337.5      99.5              step_result = self.execute_step(rules, backtracker)
150        72        27.5       0.4       0.1              n_steps += 1
151        72        14.9       0.2       0.1              if step_result:
```

Figure 3.1: *We can observe in the first image that the majority of the step is indeed taken by the recomputation (the* `update` *function also contains one set of recomputations). The improvement can be seen between the 2nd and 3rd images, with the* ***4th column*** *denoting time spent per call in nanoseconds. The difference in the number of calls is likely a result of the randomness of the backtracker.*

Following that, we experimented with the solver configuration to find the optimal solver structure - varying both the logic components and the backtracker. The measurements in terms of time and number of steps can be found in Table 3.1.

As expected, the more logic modules there are, the better the performance. In particular, the **Hidden Singles** rule seems to produce the best improvement from the baseline. **Hidden Pointers** and **Obvious Pairs** perform on a comparable scale, however, we can notice the former is less efficient per call. Furthermore, the **Selective Backtracker** always outperforms the **Naive** one. The improvement reduces over time, as the solver relies on backtracking less. We will then aim to optimise the operations of the best setup. We observe that the **Obvious Singles and Pairs** modules collectively take up approximately 18% of the time spent, even though the former is called most often. Hence, we focus on optimising the remaining two. The detailed profiling of the `step` functions can be found in the Appendix.

Backtracking setup	**OS**	**OS + HS**	**OS + HS + HP**	**OS + HS + OP**	**All**
Naive	2950 seconds 12.5M steps	148 seconds 431k steps	97 seconds 155k steps	116 seconds 270k steps	53 seconds 79k steps
Selective	772 seconds 4M steps	62 seconds 162k steps	45 seconds 74k steps	41 seconds 109k steps	38 seconds 47k steps

Table 3.1: *Time and step efficiency for the tested configurations. The abbreviations stand for:* ***OS*** *- Obvious Singles,* ***HS*** *- Hidden Singles,* ***HP*** *- Hidden Pointers,* ***OP*** *- Obvious Pairs.*

After identifying the expensive functions, we mostly optimised NumPy operations, such as using a boolean-specific array for existence checking. Furthermore, operations such as `count_nonzero` were more efficient than `sum` for counting boolean values. Simple optimizations such as this yielded an improvement of around 80%, with a final runtime of 21 seconds for all 150 Sudoku puzzles.

Chapter 4

Validation, Unit Tests and Continuous Integration

Ensuring the code behaves correctly and is well-structured was a key point during the development process. This was achieved through tests capturing varying parts of the pipeline, ranging from a single component to the entire process. This process allowed us to produce robust code upon pushing to the main branch.

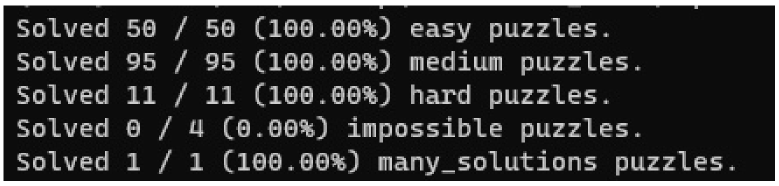
4.1 Unit and Integration Tests

During development, unit and integration tests were continuously implemented alongside the relevant features. These tests were used to validate smaller components of the solution, such as the behaviour of the parser elements and the logic steps. No changes were merged to the main branch before they passed the corresponding unit tests. A concise summary of the tests can be seen below:

1. **Parsing**:
   (a) Reading the text file.
   (b) Validation of raw input.
   (c) Cleaning the validated input.
   (d) Full pipeline test.
   (e) Configuration parsing.
2. **Logic**:
   (a) Behaviour of **Obvious Singles**.
   (b) Behaviour of **Hidden Singles**.
   (c) Behaviour of **Obvious Pairs**.
   (d) Behaviour of **Hidden Pointers**.
3. **Backtracking**:
   (a) Guessing the first cell and recovery for **Naive Backtracker**.
   (b) Guessing the most defined cell and recovery for **Selective Backtracker**.
4. **Board representation**:
   (a) Correct initialization.
   (b) Possibility computation after update.

4.2 Validation

Outside of unit testing, we created a script that could test a solver configuration on a large set of boards. The example data was extracted from Dimitri Fontaine's repository[7] and Sudopedia[8]. The repository has not been used outside of obtaining the puzzles. This framework is more powerful than the aforementioned tests because it tests how the pipeline behaves in different situations. However, it is also more computationally expensive, especially in the early development stages before any optimization. Therefore, we used it primarily before merging into the main branch to have a guarantee of correct behaviour, as seen in Figure 4.1.



```
Solved 50 / 50 (100.00%) easy puzzles.
Solved 95 / 95 (100.00%) medium puzzles.
Solved 11 / 11 (100.00%) hard puzzles.
Solved 0 / 4 (0.00%) impossible puzzles.
Solved 1 / 1 (100.00%) many_solutions puzzles.
```

Figure 4.1: *In a correct setting, the output should look like the picture above. Note that solving 0/4 impossible Sudokus is considered correct behaviour. This is because if the script has terminated, then no errors have occurred when handling unsolvable Sudoku puzzles.*

4.3 Error Trapping

During development, we made sure to anticipate any errors that could occur during the processing. That is most prevalent in parsing and when reaching the end of a solution. For every type of error, we created a custom exception, so that the try-except blocks will recognise the correct errors. As a consequence, the solver is able to report clear errors, including but not limited to:

- Incorrect input shape.
- Impossible starting board.
- Missing files.
- Incorrect configuration.

4.4 Continuous Integration setup

From the initial setup, we created an automated pre-commit configuration, to ensure that the code conforms to best practices, including the PEP8 standard. The configuration includes:

- **Pre-commit hooks** - some of the default pre-commit hooks, used for file formatting.
- **Black** - for Python code style formatting.
- **Flake8** - linter for PEP8 style conforming.
- **pytest** - for unit testing and validation.

A change that was made was to allow for longer lines for the `flake8` linting, changing the default 79 words to 120. Errors with code E203 and W503[9] were ignored, due to inconsistencies with the `black` formatter. Because the unit tests were efficient enough, they were always run upon issuing a commit.

Chapter 5

Packaging and Usability

5.1 Packaging

Code was packaged depending on the functionality of the particular modules. We decided to split the files into the following modules:

- **Parsing** - functions related to processing input, including both the board as well as the configuration parsing.
- **Logic** - the core elements of the solver, including both the logic rule and backtracking implementations.
- **Solver** - the wrapper classes for the board functionality, as well as for the actual solver. The latter uses elements from all other modules to obtain the required functionality.

Additional modules for the **animation** task, custom **exceptions** and the executable scripts were used. The file tree can be viewed in the Appendix.

5.2 Usability

To guarantee that the solver could be run on any system, we created a `Dockerfile` to build a Docker image. This includes cloning the repository, installing the required packages, as well as downloading the relevant samples. The user can then create their own configuration, which can be run using the `run_solver.py` script. A thorough description of how to do so can be found in the `README.md` file, the contents of which are present in the Appendix.

Furthermore, we have ensured that the code is compatible with Doxygen documentation[10], with us having set up the corresponding `Doxyfile`. Docstrings were handwritten to conform to the Doxygen standard.

Chapter 6

Further Work and Summary

We have described a fault-tolerant and efficient Sudoku solver. During development, we strived to follow the best development practices. The plan created during the prototyping stage was followed thoroughly, up to flexibility in implementation. The final code yielded a fully replicable repository, which can be constructed using a Docker image. Furthermore, we have provided an efficient implementation that can solve more than 150 sudoku puzzles in half a minute. The solver was both well-optimised and built for robustness, utilising error traps to provide the user with as much information as possible. Finally, correctness was ensured through thorough testing, both on a solver scale, as well as on a function scale through unit tests.

While the implementation has been highly optimized, there is also the possibility for further improvement. Rewriting the classes in **Cython** would allow for the same readability, with the efficient nature of low-level C code. Furthermore, there exist many remaining logic rules, which could further increase the effectiveness of the solver if implemented.

In conclusion, this report highlighted the importance and effectiveness of conforming to good programming practices. Good planning and optimization were key for producing a clean and robust final product, in which usability was a key focus. Finally, these practices, alongside simple Object-Oriented Programming paradigms allow us to produce this solution in a way that can be further extended with ease.

Bibliography

[1] Wikipedia. Sudoku — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Sudoku&oldid=1188770726`, 2023. [Online; accessed 17-December-2023].

[2] Wikipedia. Sudoku solving algorithms — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Sudoku%20solving%20algorithms&oldid=1185290144`, 2023. [Online; accessed 24-November-2023].

[3] Wikipedia. Backtracking — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Backtracking&oldid=1185423911`, 2023. [Online; accessed 24-November-2023].

[4] Sudoku rules - strategies, solving techniques and tricks, . URL `https://sudoku.com/sudoku-rules`.

[5] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

[6] Wikipedia. Depth-first search — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Depth-first%20search&oldid=1186499373`, 2023. [Online; accessed 14-December-2023].

[7] Dimitri Fontaine. Solving every sudoku puzzle. `https://github.com/dimitri/sudoku`, 2012.

[8] Invalid test cases - sudopedia mirror, . URL `http://sudopedia.enjoysudoku.com/Invalid_Test_Cases.html`.

[9] Ian; Stapleton Cordasco. Flake8 error / violation codes. URL `https://flake8.pycqa.org/en/latest/user/error-codes.html`.

[10] van Heesch Dimitri, 1997. URL `https://www.doxygen.nl/index.html`.

Appendix

README file

Appendix A

Sudoku Solver - ivp24

This repository contains a solver for a 9x9 Sudoku board.

A.1 Table of contents

A.2 Requirements

The user should preferably have a version of Docker installed in order to ensure the correct setup of environments. If that is not possible, the user is recommended to have Conda installed, in order to set up the requirements. If Conda is also not available, make sure that the packages described in `environment.yml` are available and installed.

A.3 Setup

We provide two different set up mechanisms using either Docker or Conda. The former is recommended, as it ensures that the environment used is identical to the one used in the development of the project.

[1] `#requirements`
[2] `#setup`
[3] `#running-the-solver`
[4] `#features`
[5] `#frameworks`
[6] `#build-status`
[7] `#credits`

A.3.1 Using Docker

To correctly setup the environment, we utilise a Docker image. To build the image before creating the container, you can run.

```
docker build -t ivp24_sudoku .
```

The setup image will also add the necessary pre-commit checks to your git repository, ensuring the commits work correctly. You need to have the repository cloned beforehand, otherwise, no files will be in the working directory.

Afterwards, any time you want to use the code, you can launch a Docker container using:

```
docker run --name <name> --rm -ti ivp24_sudoku
```

If you want to make changes to the repository, you would likely need to use your Git credentials. A safe way to load your SSH keys was to use the following command:

```
docker run --name <name> --rm -v <ssh folder on local machine>:/root/.ssh
-ti ivp24_sudoku
```

This copies your keys to the created container and you should be able to run all required git commands.

A.3.2 Using Conda

The primary concern when using Conda is to install the required packages. In this case, **make sure to specify an environment name**. Otherwise, you risk overriding your base environment. Installation and activation can be done using the commands:

```
conda env create --name <envname> -f environment.yml          conda activate
<envname>
```

A.4 Running the solver

There are 2 different ways to run the solver. If you want to run a single sudoku puzzle, you can either: - Provide only a text file containing the board. The solver will not display any steps made during the solution and will use the optimal setup. This includes all logic rules, alongside the `Selective backtracker`. An example board can be seen in `test\samples\sample_sudoku.txt`. - Provide a full configuration, as seen in `test/configs/sample_config.ini`. You can specify both the solver configuration, as well as the visualization method. Accepted visualization methods include a text-based representation and a matplotlib animation.

The command you would like to run (while in the root folder) is:

```
python run_solver.py <configuration/sudoku file>
```

A.5 Features

The scripts allow for setting up different configurations for the solver. For example, if the sudoku that is to be tried is relatively simple, fewer rules might result in a faster performance. We also allow for different types of visualizations, so that the process is better explained.

A.5.1 The solver

The solver supports different methods of solving, all specified by a set and order of logic rules and backtracking. A solver must always contain a set of logic rules, (perhaps empty - but this might take too long to run), and a single backtracking algorithm. The logic rules must be chosen among `Obvious Singles`, `Hidden Singles`, `Hidden Pointers`, and `Obvious Pairs`. More details on them can be found on the Sudoku.com website.

The backtracking algorithms available are a simple `Naive Backtracker`, as well as a ”smarter” `Selective Backtracker`, which makes progress on the least defined cell. The latter is recommended as it can save a substantial amount of backtracking steps and is not significantly more computationally expensive.

A.5.2 Visualization

There are 2 types of visualization - text-based or animation. A text-based visualization will present the user with a step-by-step progress report. If a cell is decided, the full board will be displayed, while if only the cell possibilities have changed a single line describing the change will be presented.

The animation will present the full state of the board in the form of a Matplotlib animation. It will only be saved if the ”Output” section of the configuration file has been specified. An example of the solver in action can be seen below:

Solution Example

The animation can be viewed by linking Visual Studio Code to the container, or by copying it from the container to the local machine using:

```
docker cp <container name>:/ivp24/<source> <destination>
```

A.5.3 Documentation

All documentation for the repository can be generated through Doxygen by running the `doxygen` command inside the `/docs` folder. This file will be used as the front page.

A.5.4 Output

An output path may be specified in the configuration so that multiple scripts can be run at the same time, with the results being stored in the specified files.

A.6 Frameworks

The entire project was built on **Python** and uses the following packages: - For computation and parsing: - NumPy - configparser - For plotting: - matplotlib - For maintainability/documentation: - doxygen - pytest - pre-commit

A.7 Build status

Currently, the build is complete and the program can be used to its full capacity.

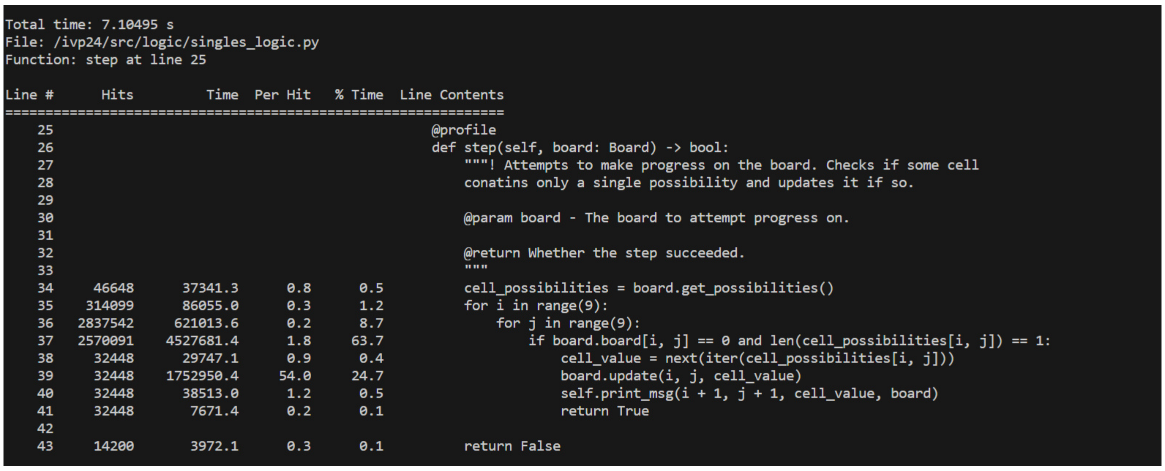
A.8 Credits

The `.pre-commit-config.yaml` configuration file content has been adapted from the Research Computing lecture notes. Ideas for the logic rules were taken from the Sudoku.com website. The example sudoku boards were found in Dimitri Fontaine's Git repository and Sudopedia.

Directory tree

```
/
├── docs
│   └── Doxyfile - contains the Doxygen configuration
├── src
│   ├── __init__.py
│   ├── animation.py - small animation module
│   ├── exceptions.py - contains custom exceptions for error trapping
│   ├── logic - module containing the logic operations for the solver
│   │   ├── __init__.py
│   │   ├── backtracking.py - The backtracking algorithms (naive and selective)
│   │   ├── base_logic.py - The basic interfaces for logic and backtracking
│   │   ├── complex_logic.py - Advanced logic rules(Obvious pairs, Hidden
│   │   │   pointers)
│   │   └── singles_logic.py - Basic logic rules on single cells (Obvious and
│   │       hidden)
│   ├── parsing - file reading module
│   │   ├── __init__.py
│   │   ├── config_parsing.py - For configuration file reading
│   │   ├── preprocessing.py - For cleaning and processing file input for the
│   │   │   board
│   │   ├── sudoku_parser.py - core methods for opening file and converting to
│   │   │   data
│   │   └── validation.py - for detecting errors in the input
│   └── solver - module for aggregating the separate parts of the solver
│       ├── __init__.py
│       ├── board.py - the data structure for the board representation
│       └── solver.py - wrapper, combining all parts of a solver
├── test
│   ├── __init__.py
│   ├── test_backtracking.py - unit tests for backtracking
│   ├── test_board.py - unit tests for correct board representation
│   ├── test_logic.py - unit tests for logic rules
│   └── test_parsing.py - unit tests for correct file processing
├── run_all_samples.py - runs a set of all samples using default parameters
├── run_solver.py - runs the solver using a set board or configuration
├── .gitignore
├── .pre-commit-config.yml
├── Dockerfile
├── environment.yml
├── Instructions.md
├── LICENSE.md
└── README.md
```

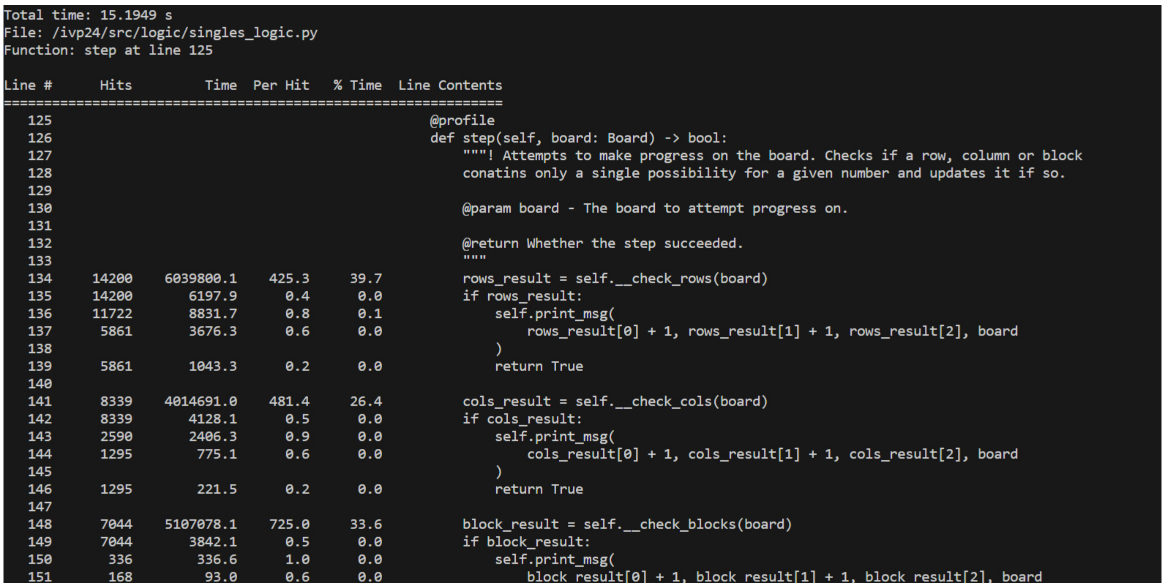
Profiling



```
Total time: 7.10495 s
File: /ivp24/src/logic/singles_logic.py
Function: step at line 25

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    25                                               @profile
    26                                               def step(self, board: Board) -> bool:
    27                                                   """! Attempts to make progress on the board. Checks if some cell
    28                                                   conatins only a single possibility and updates it if so.
    29
    30                                                   @param board - The board to attempt progress on.
    31
    32                                                   @return Whether the step succeeded.
    33                                                   """
    34     46648      37341.3      0.8      0.5          cell_possibilities = board.get_possibilities()
    35    314099      86055.0      0.3      1.2          for i in range(9):
    36   2837542     621013.6      0.2      8.7              for j in range(9):
    37   2570091    4527681.4      1.8     63.7                  if board.board[i, j] == 0 and len(cell_possibilities[i, j]) == 1:
    38     32448      29747.1      0.9      0.4                      cell_value = next(iter(cell_possibilities[i, j]))
    39     32448    1752950.4     54.0     24.7                      board.update(i, j, cell_value)
    40     32448      38513.0      1.2      0.5                      self.print_msg(i + 1, j + 1, cell_value, board)
    41     32448       7671.4      0.2      0.1                      return True
    42
    43     14200       3972.1      0.3      0.1          return False
```

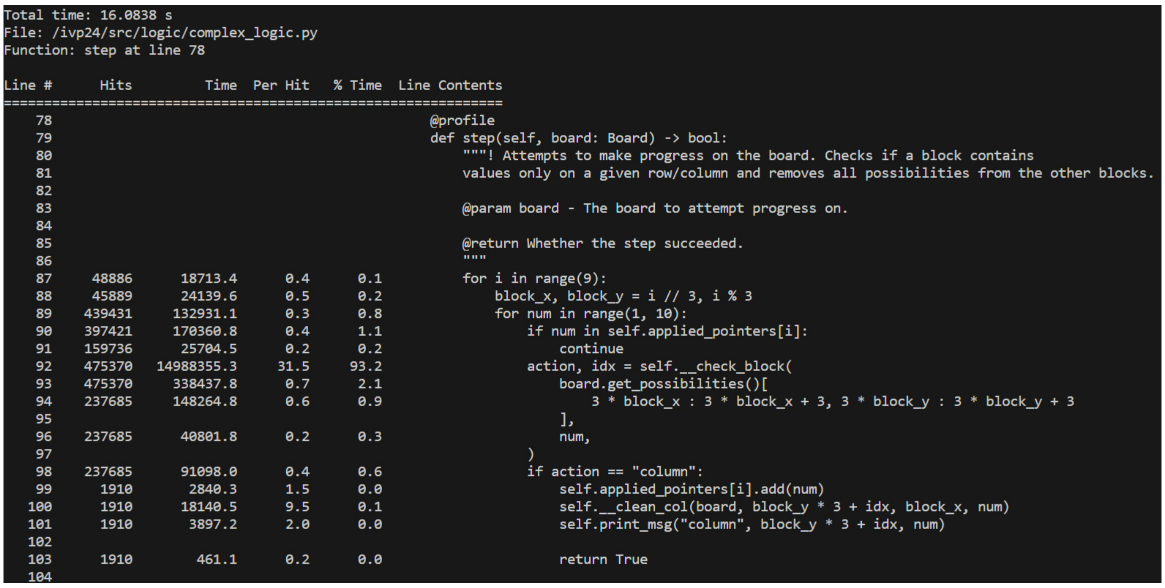
Figure A.1: *Obvious Singles step function profiling.*



```
Total time: 15.1949 s
File: /ivp24/src/logic/singles_logic.py
Function: step at line 125

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   125                                               @profile
   126                                               def step(self, board: Board) -> bool:
   127                                                   """! Attempts to make progress on the board. Checks if a row, column or block
   128                                                   conatins only a single possibility for a given number and updates it if so.
   129
   130                                                   @param board - The board to attempt progress on.
   131
   132                                                   @return Whether the step succeeded.
   133                                                   """
   134     14200    6039800.1    425.3     39.7          rows_result = self.__check_rows(board)
   135     14200       6197.9      0.4      0.0          if rows_result:
   136     11722       8831.7      0.8      0.1              self.print_msg(
   137      5861       3676.3      0.6      0.0                  rows_result[0] + 1, rows_result[1] + 1, rows_result[2], board
   138                                                       )
   139      5861       1043.3      0.2      0.0              return True
   140
   141      8339    4014691.0    481.4     26.4          cols_result = self.__check_cols(board)
   142      8339       4128.1      0.5      0.0          if cols_result:
   143      2590       2406.3      0.9      0.0              self.print_msg(
   144      1295        775.1      0.6      0.0                  cols_result[0] + 1, cols_result[1] + 1, cols_result[2], board
   145                                                       )
   146      1295        221.5      0.2      0.0              return True
   147
   148      7044    5107078.1    725.0     33.6          block_result = self.__check_blocks(board)
   149      7044       3842.1      0.5      0.0          if block_result:
   150       336        336.6      1.0      0.0              self.print_msg(
   151       168         93.0      0.6      0.0                  block_result[0] + 1, block_result[1] + 1, block_result[2], board
```

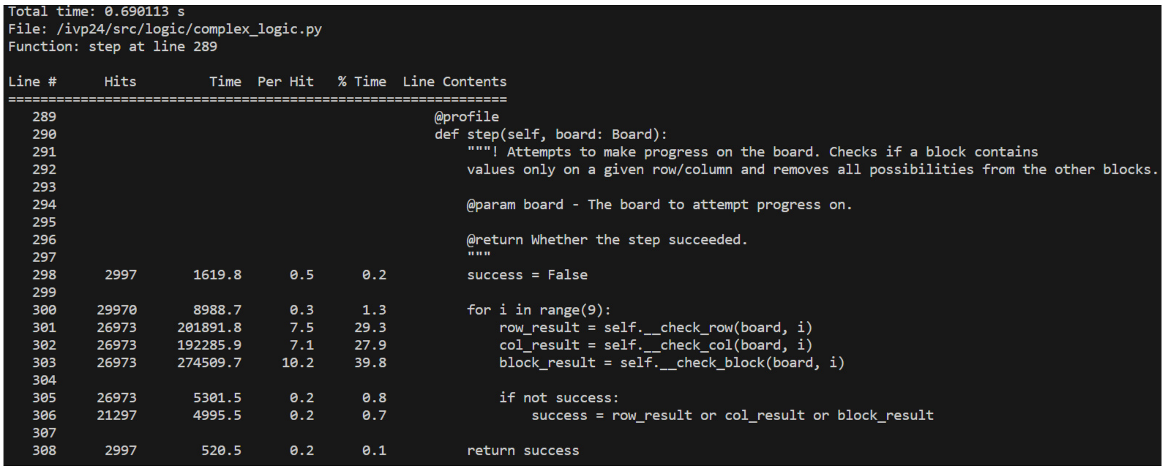
Figure A.2: *Hidden Singles step function profiling.*



```
Total time: 16.0838 s
File: /ivp24/src/logic/complex_logic.py
Function: step at line 78

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    78                                               @profile
    79                                               def step(self, board: Board) -> bool:
    80                                                   """! Attempts to make progress on the board. Checks if a block contains
    81                                                   values only on a given row/column and removes all possibilities from the other blocks.
    82
    83                                                   @param board - The board to attempt progress on.
    84
    85                                                   @return Whether the step succeeded.
    86                                                   """
    87     48886      18713.4      0.4      0.1          for i in range(9):
    88     45889      24139.6      0.5      0.2              block_x, block_y = i // 3, i % 3
    89    439431     132931.1      0.3      0.8              for num in range(1, 10):
    90    397421     170360.8      0.4      1.1                  if num in self.applied_pointers[i]:
    91    159736      25704.5      0.2      0.2                      continue
    92    475370   14988355.3     31.5     93.2                  action, idx = self.__check_block(
    93    475370     338437.8      0.7      2.1                      board.get_possibilities()[
    94    237685     148264.8      0.6      0.9                          3 * block_x : 3 * block_x + 3, 3 * block_y : 3 * block_y + 3
    95                                                               ],
    96    237685      40801.8      0.2      0.3                      num,
    97                                                           )
    98    237685      91098.0      0.4      0.6                  if action == "column":
    99      1910       2840.3      1.5      0.0                      self.applied_pointers[i].add(num)
   100      1910      18140.5      9.5      0.1                      self.__clean_col(board, block_y * 3 + idx, block_x, num)
   101      1910       3897.2      2.0      0.0                      self.print_msg("column", block_y * 3 + idx, num)
   102
   103      1910        461.1      0.2      0.0                      return True
   104
```

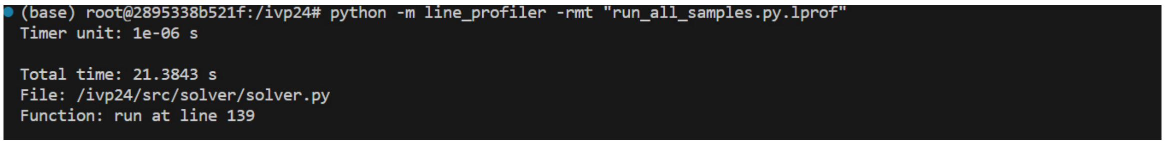
Figure A.3: *Hidden Pointers step function profiling.*



```
Total time: 0.690113 s
File: /ivp24/src/logic/complex_logic.py
Function: step at line 289

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   289                                           @profile
   290                                           def step(self, board: Board):
   291                                               """! Attempts to make progress on the board. Checks if a block contains
   292                                               values only on a given row/column and removes all possibilities from the other blocks.
   293
   294                                               @param board - The board to attempt progress on.
   295
   296                                               @return Whether the step succeeded.
   297                                               """
   298      2997       1619.8      0.5      0.2          success = False
   299
   300     29970       8988.7      0.3      1.3          for i in range(9):
   301     26973     201891.8      7.5     29.3              row_result = self.__check_row(board, i)
   302     26973     192285.9      7.1     27.9              col_result = self.__check_col(board, i)
   303     26973     274509.7     10.2     39.8              block_result = self.__check_block(board, i)
   304
   305     26973       5301.5      0.2      0.8              if not success:
   306     21297       4995.5      0.2      0.7                  success = row_result or col_result or block_result
   307
   308      2997        520.5      0.2      0.1          return success
```

Figure A.4: *Obvious Pairs step function profiling.*



```
(base) root@2895338b521f:/ivp24# python -m line_profiler -rmt "run_all_samples.py.lprof"
Timer unit: 1e-06 s

Total time: 21.3843 s
File: /ivp24/src/solver/solver.py
Function: run at line 139
```

Figure A.5: *Post-optimization results.*