Prepared for submission to University of Cambridge

# C1: Research Computing - Coursework Assignment

**Adnan Siddiquei**

University of Cambridge

E-mail: as3438@cam.ac.uk

## Contents



# 1 Introduction

There are a variety of algorithms that can be used to solve sudoku puzzles, some very complex algorithms and some much simpler ones. Backtracking is an example of a much simpler, brute force algorithm. Simply put, backtracking works by visitng each cell in a sudoku grid one by one, filling numbers into the cells, and backtracking to previous cells if no valid number can be filled into the current cell. This is the algorithm that we chose to implement in this project, due to it's simplicity, as it would allow this report to focus on the software development process rather than the algorithm itself.

In this report, we will discuss a variety of topics related to the software development process, starting from the solution design where we will discuss the prototyping involved. We will then discuss the development of the code which involves the tools, workflows and processes used to develop and optimise the code. We will then discuss unit testing and continuous integration, and finally we will discuss how we utilised documentation best practises and packaging tools.

# 2 Solution Design

## 2.1 Prototyping

Prior to writing any code, we prototyped the solution. Prototyping prior to coding allowed us to

- identify possible bugs and complexity earlier on in the development process, such that we could consider them prior to coding, rather than discover them after the fact;
- identify non-trivial and edge cases that might need to be considered while writing code and unit tests;
- identify API interfaces of the package, allowing us to write the unit tests beforehand (more on this when we talk about test driven development in Section(3.3)).
- explore different implementations of the backtracking algorithm;
- identify python packages and resources that we may need to use in the implementation, and make decisions on which might be the most appropriate;

The implementation of the sudoku solver consisted of two main components: parsing the user input (and handling associated errors), and the backtracking algorithm itself. The prototyping flowcharts of these two components is shown in Fig.(1) and Fig.(2). Additionally, we wrote some *pseudo-python-code* as shown in Fig.(3), allowing us to define the interfaces of the functions and classes that we would write - which is required if we desire to to write any unit tests prior to coding. The tests derived from this solution design are discussed more in Section(4). Further iterations on these initial prototypes are discussed more in Section(3) where we discuss how experimentation and profiling changed the end implementation in efforts to optimise the code.

### 2.2 API Interface

An important decision to make early on is how the code will be structured into modules and what the external API interface will look like, as shown in Fig.(3). We decided to model the `sudokusolver` package after well known `scikit-learn` [2] package because `scikit-learn`'s structure is extremely well thought out and simple to use. Each solver would be segmented into a class which can be instantiated with the hyperparameters of the solver (in this case, is there `multiple_solutions`), and then a `solve` method which takes in the data (the `unsolved_board`) and returns the class instance with the solved board as an attribute. This seemed like a fitting way to model the package as our solvers are akin to `scikit-learn`'s estimators, and the `solve` method is akin to `scikit-learn`'s `fit` method. To extend the package we simply need to add more solvers classes, and to extend a solver we simply need to add more hyperparameters to the class and modify the `solve` method to take these hyperparameters into account.

### 2.3 Key Conclusions

We decided to only implement the `BacktrackingSolver` with no `multiple_solutions` functionality in the initial `v1.0.0` implementation of the package, The implementation of `multiple_solutions` was not thought through in the prototyping stage but given that we now understood how to syntactically and structurally extend the package and the solvers within, it would be trivial to add this functionality in a future version of the package, which demonstrates the importance of blueprinting out the API interface.

Another key decision which was implicit and not thoroughly discussed was to utilise `numpy` arrays to represent the sudoku board, this is because `numpy` is well known to be the most performant array manipulation package in python. Alternatives include using `pandas` or python's in-built `list` object, however, `numpy` is generally known to be more performant

than both of these and `numpy` also includes numerous built-in functions that will be useful in the implementation of the backtracking algorithm.

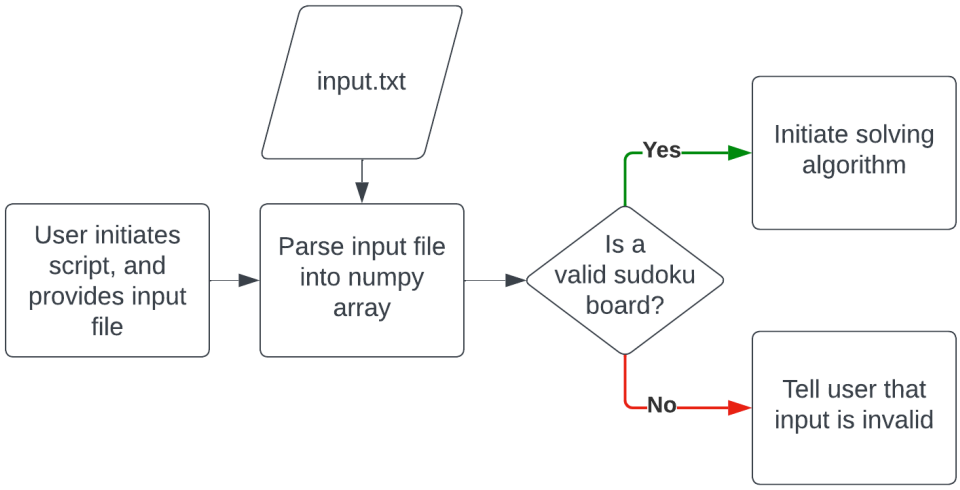


**Figure 1**: A flowchart for part 1 of solving a sudoku puzzle: parsing the user input.

## 3 Development, Experimentation and Profiling

Here we discuss several components of the development process, and reason why we chose to do things in certain ways.

### 3.1 Linting and Formatting - `ruff`

Linting and formatting is useful, especially in shared projects, as it allows for a consistent style across the codebase. There are a plethora of python linting and formatting tools available and for this project, we chose to use `ruff`. There were two primary reasons for this choice: speed and simplicity. `ruff` is faster than most other linting tools including `flake8`. In a test done by the developers of `ruff`, it managed to lint the CPython codebase 42x faster than `flake8` [1].

Additionally, `ruff` provides formatting functionality and as such it can also replace tools such as `black`. This makes the implementation of linting and formatting simpler, as we only need to use one tool. `ruff`'s configuration capabilities allow it to lint and format to any standard we want to, and therefore, it was configured to mimic `black` and `flake8`'s default config in accordance with PEP8.

### 3.2 Git and Gitlab Workflows

It is generally good practise to write detailed commit messages and merge requests, and traditionally in larger software projects, project management tools such as JIRA are used to track issues and tasks, with merge requests being linked to these issues. In the world of open source, and small projects like this, GitLab's issue tracker works as a perfect tool for this. Therefore, we utilised the issue tracker to create issues for tasks that needed to be

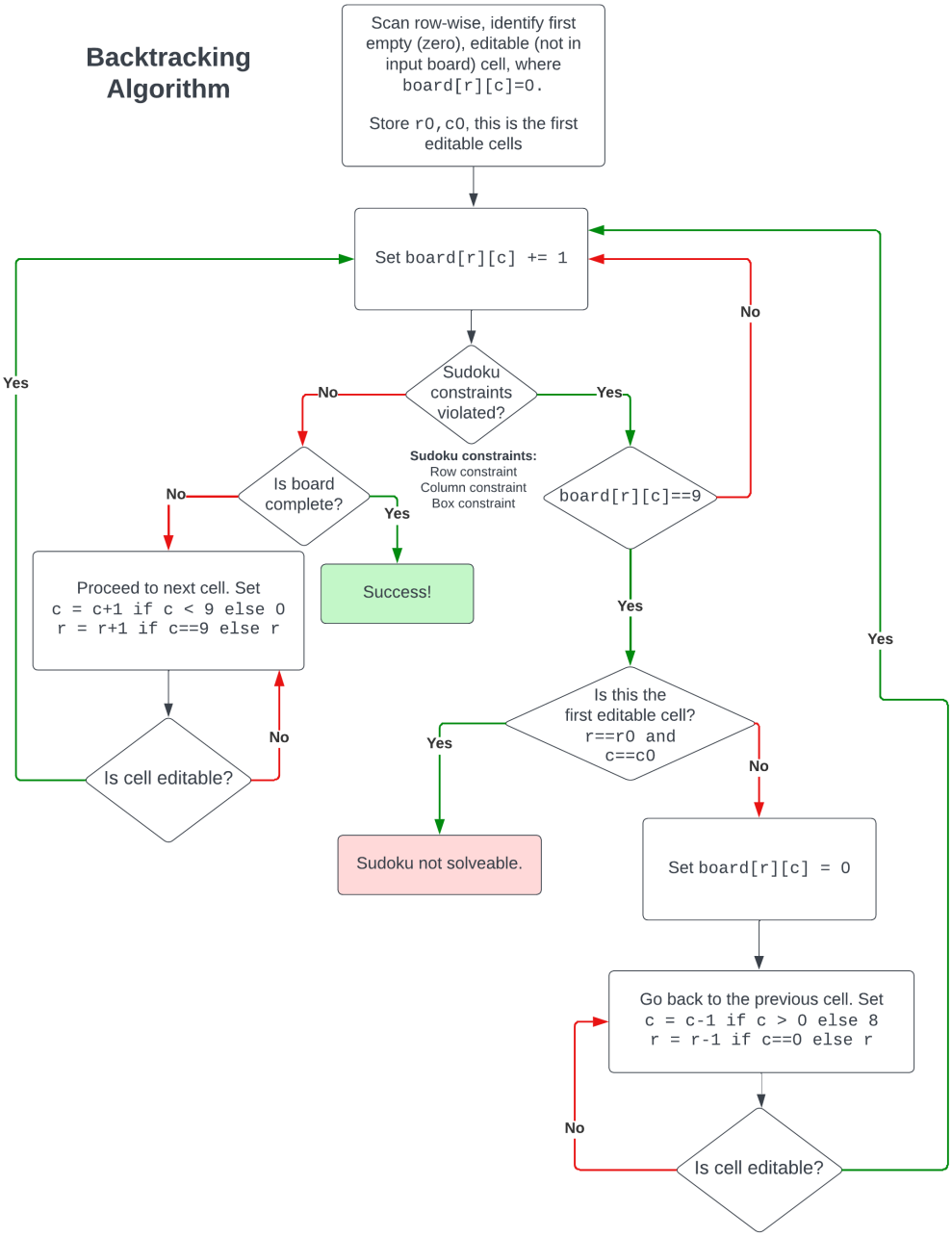


**Figure 2**: A flowchart for part 2 of solving a sudoku puzzle: the backtracking algorithm.

done, and then linked merge requests to these issues. In this way, we could maintain detailed documentation in the issue tracker of what was done, and why it was done, in a slightly more structured way than just using commit messages. Therefore, a new branch was created for every issue, and once the branch implemented or resolved the features in the issue, a merge request was created, linked to the issue, merged into main, and the issue was closed. As such, anyone with access to the repo can see the git commit history through the GitLab UI and

```
class BacktrackingSolver
    def __init__(self, multiple_solutions: bool = False):
        self.board  # will store solved board
        self.is_solved  # will store whether board is solved
        self.is_solvable  # will store whether board is solvable
        self.is_valid  # will store whether board is valid
    def solve(self, unsolved_board: numpy.NDArray):

def parse_input_file(input_file_path: str) -> numpy.NDArray

def save_board_to_txt(
    board: numpy.NDArray,
    output_file_path: str
) -> None

# entry point for script, argv = sys.argv
def handler(argv) -> None:
```

**Figure 3**: *pseudo-python-code* representation of the interfaces of the functions and classes within our `sudokusolver` package.

any references to issue numbers in commits can be clicked to link the user to the underlying issue which describes what was implemented and why.

Furthermore, a nice feature of many project management tools and IDE's is the ability to group branches into folders based on the branch name. Git allows forward slashes '/' in branch names such that branches can be structured into folders by the IDE, and so a naming convention was adopted for branches. Branches were named like the following: `feat/issue-1` or `tests/issue-4` where the 3 main 'folders' used were `feat` for feature, `tests` and `bug`, followed by the issue number it implemented. In practise, the `bug` folder was not used as no bugs were discovered.

Overall, these Git and GitLab workflows allowed for a structured development process, and a detailed history of what was done and why it was done, which is useful for future reference.

### 3.3 Test Driven Development

Where possible, we strived to do test driven development by writing tests first. This was made possible due to the careful prototyping and API design discussed in Section(2). The reason we decided to write tests first came from three primary reasons:

- it streamlined code writing, as we were already aware of what the code needed to do, and what the edge cases were.
- it defined clearly what the end product would be;
- it reduced debugging time. We could more quickly figure out what was going wrong if a bug arose;

The final tests that were written are discussed more in Section(4).

### 3.4 Profiling and Optimisation

```
222     31941    7794000.0    244.0      0.2          while not self.is_solved and self.is_solvable:
223     31940   52662000.0   1648.8      1.1              self.board[r, c] += 1
224
225     63880 4361697000.0  68279.5     94.9              if self._sudoku_constraint_violated(
226     31940    2931000.0     91.8      0.1                  r, c
227                                                       ):  # If any constraints have been violated
228     57434   61020000.0   1062.4      1.3                  if self._range_violated(
229     28717    2670000.0     93.0      0.1                      r, c
230                                                           ):  # and one of them is the range constraint, self.board[r, c] == 10
```

(a) An excerpt of the line profile of the `BacktrackingSolver.solve` function. Line 220 indicates the that the `BacktrackingSolver._sudoku_constraint_violated` function consumes the vast majority of the computation time.

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   118                                           def _sudoku_constraint_violated(self, r: int, c: int) -> bool:
   119                                               return (
   120         3     164000.0  54666.7     38.6              self._row_constraint_violated(r, c)
   121         1     106000.0 106000.0     24.9              or self._col_constraint_violated(r, c)
   122         1     151000.0 151000.0     35.5              or self._box_constraint_violated(r, c)
   123         1       4000.0   4000.0      0.9              or self._range_violated(r, c)
   124                                               )
```

(b) The line profile of the `BacktrackingSolver._sudoku_constraint_violated` function. This indicates that no single line of code is responsible for the majority of the computation time, it is equally shared.

```
215     31941    7069000.0    221.3      1.2          while not self.is_solved and self.is_solvable:
216     31940   56431000.0   1766.8      9.4              self.board[r, c] += 1
217
218     63880  427773000.0   6696.5     71.2              if self._sudoku_constraint_violated(
219     31940    3329000.0    104.2      0.6                  r, c
220                                                       ):  # If any constraints have been violated
221     57434   57080000.0    993.8      9.5                  if self._range_violated(
222     28717    2881000.0    100.3      0.5                      r, c
223                                                           ):  # and one of them is the range constraint, self.board[r, c] == 10
```

(c) The same line profile as in (a), but after optimisation. The key line here is line 218.

**Figure 4**: Line profiles of the `BacktrackingSolver.solve` function and the `BacktrackingSolver._sudoku_constraint_violated` functions. The headers of each column are shown in (b). (a), and (b) show the profiles before optimisation, and (c) shows a profile after optimisation. The key line in (a) and (c) is line 225 and 218 respectively, which show a 10.2x speedup after optimisation.

Given that this was a very low requirement project, there was not much testing of third-party packages required. However, the performance of the code was tested using a line profiler. Fig.(4a) shows the line profile of the `BacktrackingSolver.solve` function before optimisation, which allowed us to identify the bottleneck of the code, where most of the time was being spent. This was useful as it allowed us to identify where to focus our optimisation efforts, and not waste time optimising code that was not responsible for the majority of the computation time.

Given that `BacktrackingSolver._sudoku_constraint_violated` was the function that we identified as the bottleneck, the next steps was to optimise this function. Some options for this were re-write this function in Cython or C, or look into different algorithms or packages that could be used to implement this function. We decided to implement this function in C, by implementing a function `has_non_zero_duplicates` and made it executable in python using `ctypes`, a python standard library module. The functions called in lines 120 to 122 in Fig.(4b), were then amended to use this new C function. This resulted in a 10.2x speedup of the `BacktrackingSolver.solve` function, as shown in Fig.(4c). The raw speed-up from the implementation of `has_non_zero_duplicates` was closer to 20x

but addition of error handling in the python wrapper reduced this to 10.2x.

### 3.5 Error handling

General python best practises were followed with regard to error handling. Exceptions were raised when appropriate, and try except blocks were used to catch exceptions. Occasionally, a try except block was used to catch and re-raise exceptions (with no modification) for clarity. Likewise, try except blocks were used to catch potential errors, or invalid arguments to functions, and the in-built python exceptions were used to raise these errors to the user. The three most common exceptions that were raised were `ValueError`, `TypeError` and `FileNotFoundError`.

## 4 Validation, Unit Tests and CI set up

### 4.1 Unit Tests

Unit tests were implemented using `pytest`. These were planned and written prior to writing any code, as discussed in Section(3.3). All the functions described in the pseudocode in Fig.(3) were tested, as well as the python wrapper for the `has_non_zero_duplicates` C function, and generally speaking, the tests covered the following cases:

- correct outputs when the inputs were correct;
- expected behaviour and handling of exceptions when inputs were incorrect. A function could take parameters that were incorrect in multiple ways (value, shape, length, type, etc.), and as such, we tested for all of these cases, as well as different combinations of these cases.
- correct handling of edge cases, where inputs were at their extremes (such as empty lists, input sudoku already solved, etc.)
- correct handling of slight variations on inputs that were allowed to be correct. For example, in the case of the input file, we allowed slightly incorrectly formatted versions of the input file to be parsed correctly.

Designing the tests using this framework, we were confident that the code was working as expected and that the testing suite was comprehensive.

### 4.2 Continuous Integration - `pre-commit`

`pre-commit` was used to set up actions to run before every commit. In our case, we used `pre-commit` to run the `ruff` linter and formatter before every commit. This ensured that no incorrectly formatted code made it's way into main. It is good to also run tests periodically, some projects run tests before every commit, and some run tests before every PR into main by using a CI tool such as GitHub Actions. However, in our case, we decided not to add the running of unit tests before every commit to `pre-commit-config.yaml` as we were doing test driven development, and as such, on many commits and PRs into main, the tests would, by design, not pass. However, we ensured that the tests were run regularly and as the relevant code was being written.

## 5 Documentation, Packaging and Usability

### 5.1 Documentation - `sphinx` and NumPy Style Docstrings

Documentation is an important part of any software project, without it, the user would have no idea how to utilise third-party packages. Documentation comes in two forms: inline documentation (docstrings, comments and type definitions) and usage documentation (user guides and API references). The former is written whilst developing the code, and the latter is generated from the former with documentation tools.

The discussion of which documentation tool to use is quite an opinionated one, there is no strict "one is better than the other" in most cases. The two most popular ones in python are `sphinx` and `doxygen`. Whilst using either of these tools would have been sufficient, `sphinx` was chosen for this project primarily because of the vast universe of themes and extensions available for it. The reason for documentation is to help a user understand how to use the package, and as such, above all else the documentation output should be easy to read and navigate. `sphinx` has a number of themes available, and the one chosen for this project was `furo` which is used by `pip` [3] `black` [4] and the python developer's guide [5]. Whilst the setup for `sphinx` was a bit more hands on, it resulted in much more usable and aesthetic docs.

Another motivator for using `sphinx` was the ability to use NumPy style docstrings. More so than the desire to use NumPy style docstrings, was the desire to not use Epytext style or reStructuredText style docstrings, which is what Doxygen requires. We found these styles to be much less readable than NumPy style docstrings, which was another motivator for using `sphinx`. Another popular alternative is Google style doctrings, NumPy style was chosen only by personal preference, `sphinx` can handle both.

### 5.2 Packaging - conda, Docker and make

Packaging the code effectively is important for usability. Docker allows us to containerise the code and run it on any machine and as such, we created a `Dockerfile` for our project which would allow the user to run the code in a Docker container.

We also utilised `make` by creating a `Makefile` which would allow the user to easily build and run the docker image. The `Makefile` contained a `make run` target which would build the docker image, then run the code in the container with the relevant directories mounted, and finally place the user in a shell in the container so that they could interact with the code. This way, the user does not need to worry about understanding which directories need to be mounted or any other config with relation to getting the Docker container started. If the user did care to know, they could simply look at the `Makefile` and see what the `run` target does.

Conda was also utilised to create and manage environments for the code. The base image for the Docker container was `continuumio/miniconda3`, which allowed us to easily create and install all the dependencies for the Docker image using an `environment.yaml` file and conda.

Generally, the purposes of these tools were to make the code as easy as possible to run by any user, and this goal was achieved through the use of these tools.

## 6 Summary

In summary, we have implemented a python sudoku solver package, and have discussed the software development process involved in doing so. We have discussed the design of the

package, the development and experimentation process, the validation and testing process, the documentation, and finally, the packaging of the code.

## References


[1] Astral, *`ruff` GitHub Repository*. Available at: https://github.com/astral-sh/ruff [Accessed: 7-Dec-2023].

[2] scikit-learn, *`schikit-learn` GitHub Repository*. Available at: https://github.com/scikit-learn/scikit-learn [Accessed: 7-Dec-2023].

[3] The pip developers, *pip docs*. Available at: https://pip.pypa.io/en/stable/ [Accessed: 8-Dec-2023].

[4] Lukasz Langa and contributors to Black, *Black docs*. Available at: https://black.readthedocs.io/en/stable/ [Accessed: 8-Dec-2023].

[5] Python Software Foundation, *Python Developer's Guide*. Available at: https://devguide.python.org/ [Accessed: 8-Dec-2023].