Prepared for submission to University of Cambridge

An implementation of a python sudoku solver package and a complete review of the software development process involved.

Adnan Siddiquei

University of Cambridge

E-mail: as3438@cam.ac.uk

Contents

1 Introduction

2 Solution Design

2.1 Selection of Solution Algorithm

Why did I choose backtracking?

2.2 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 to write any unit tests before 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.

2.3 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 own module, containing 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 in their own modules, 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.

2.4 Key Conclusions

We decided to only implement the `BacktrackingSolver` with no `multiple_solutions` functionality in the initial `v0.0.1` 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 likely be useful in the implementation of the backtracking algorithm.

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

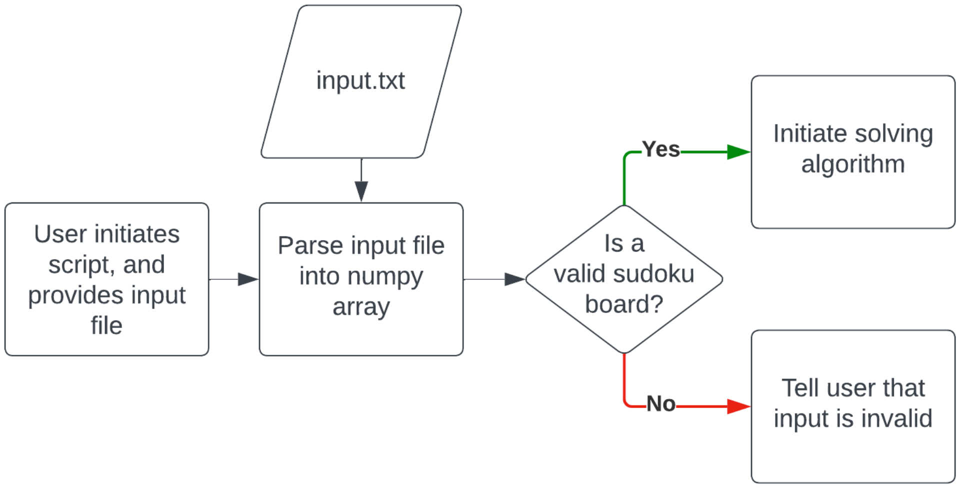

Figure 1. A flowchart for part 1 of solving a sudoku puzzle: parsing the user input.

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]. Whilst speed of linting is not a primary concern for this project, it doesn't hurt to pick the faster option.

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 as such, 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, and merge requests are 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 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

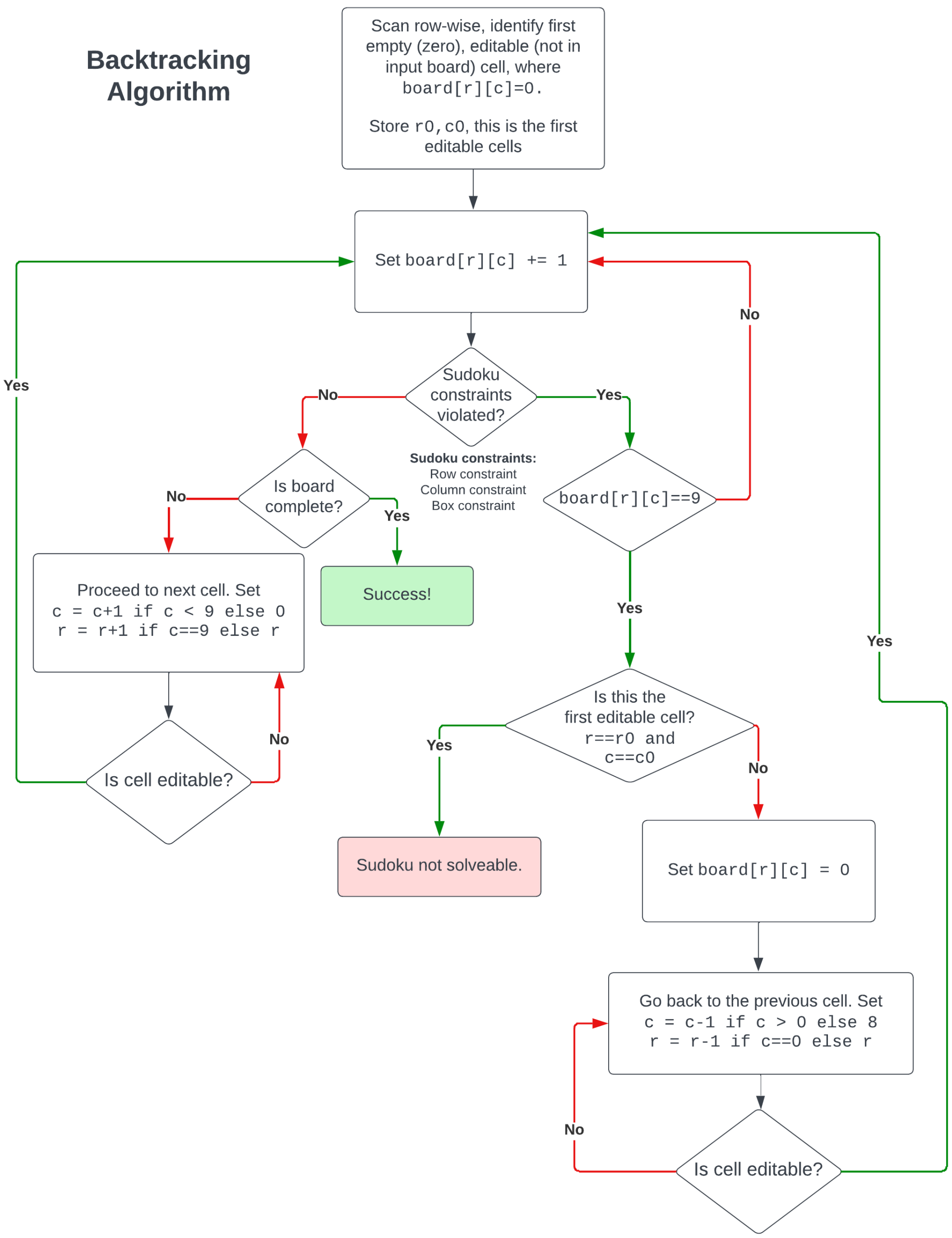

Figure 2. A flowchart for part 2 of solving a sudoku puzzle: the backtracking algorithm.

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, as such a naming convention was adopted for branches. Branches were named like the following:

```
class BacktrackingSolver
    def __init__(self, multiple_solutions: bool = False):
        self.board  # will store solved board
        self.is_solvable  # will store whether board is solvable
        self.is_solved  # will store whether board is solved
    def solve(self, unsolved_board: numpy.NDArray) -> None:

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

def save_board(
    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.

`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 bugs as no bugs were discovered. Occasionally a `report` branch was created when only the report needed to be updated, these were not linked to any issues.

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

Where did i test different packages for speed? Why did I choose numpy? Show how I profiled my code to identify where the bottleneck was. How did I work around this? Cython?

3.5 Coding Best Practises

Modularisation. typing. Exceptions. Error handling, try except. Never catch all exceptions.

4 Validation, Unit Tests and CI set up

Talk through why my unit tests are sufficient. How did I put this into the CI? With pre-commit.yaml

5 Documentation, Packaging and Usability

5.1 Documentation - `sphinx`

Why did I use sphinx over Doxygen? Sphinx is more popular, more support, more documentation, more features.

5.2 Packaging - Conda and Docker

What benefits does conda have over pip? Why did I choose conda? Why Docker?

6 Summary

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].