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.

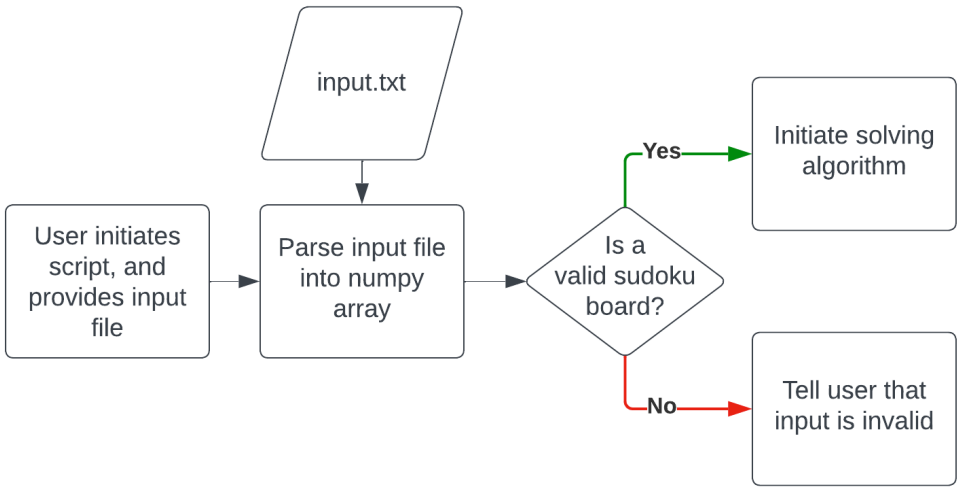

Figure 1. A flowchart for part 1 of solving a sudoku puzzle: parsing the user input.

2.3 Conclusions

An important decision made here was to utilise `numpy` arrays to represent the sudoku board, `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. 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.

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

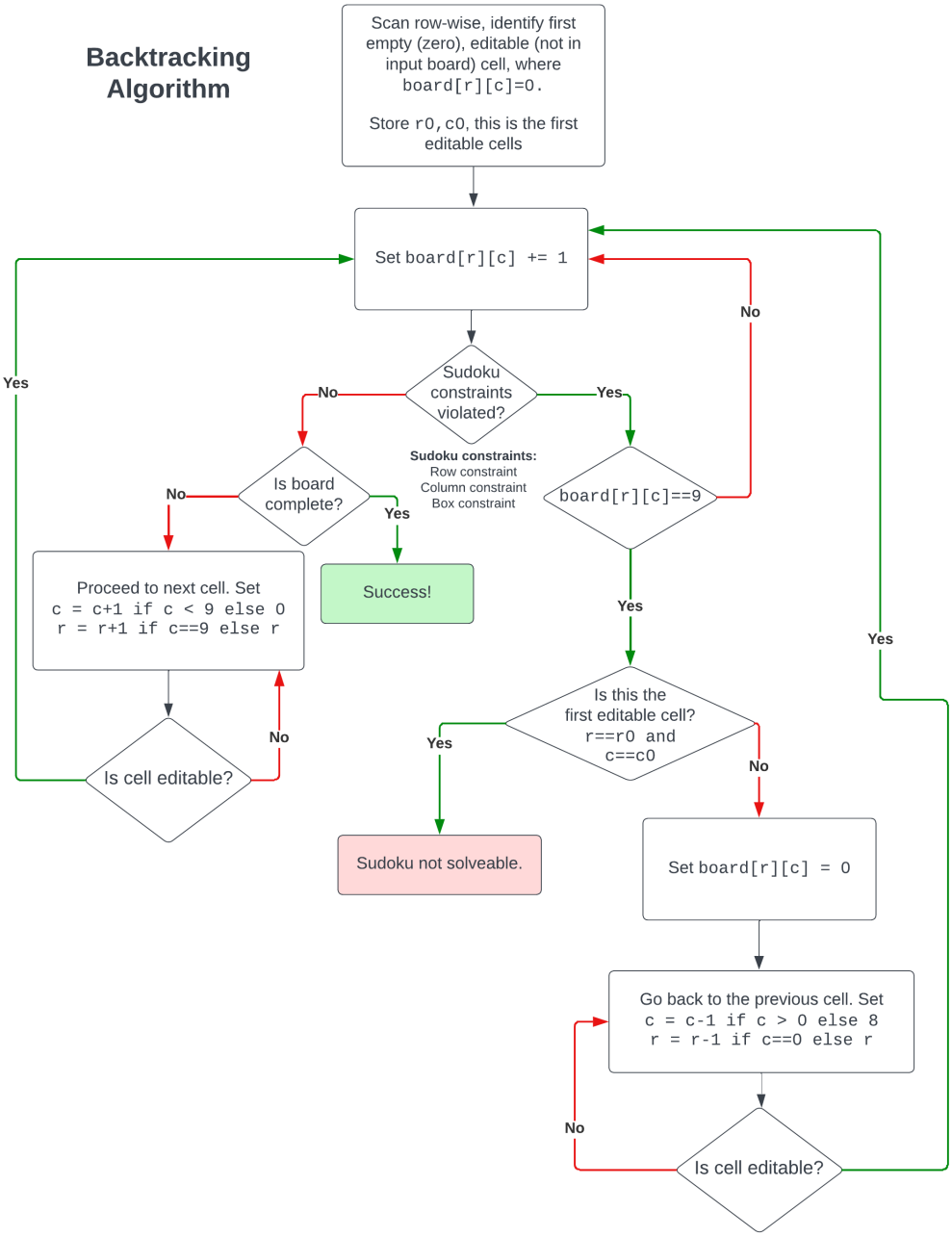

Figure 2. A flowchart for part 2 of solving a sudoku puzzle: the backtracking algorithm.

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

```
class BacktrackingSolver
    def __init__(self, unsolved_board: numpy.NDArray)
        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) -> 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.

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 pipeline

Why branches? Branch naming convention. Creating issues for tasks. Why? Project management. JIRA. New branch and merge per issue. Labels for issues because easy to and can folder the branches with those labels (show PyCharm example).

3.3 Test Driven Development

Explain why I wrote the tests first.

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].