Research Computing Coursework Report

James Hughes

December 12, 2023

Introduction

Explain some basics of Sudoku (citation!), and why this problem relates to/necessitates software development. Some salient points such as uniqueness of solution, 17 clues minimum [1] and so on. Basic definitions as in the documentation. Also "row condition" and so on.

Algorithm Selection and Prototyping

I decided to base my solution on the "templates" method combined with "backtracking" as described on the Wikipedia page about Sudoku solving algorithms [2]. This is a brute-force method, which relies on the fact that the rules of Sudoku inherently limit the number of possible arrangements of any given digit into nine cells on the grid. We can see this using a combinatorial argument filling in the 3x3 boxes from left-to-right then top-to-bottom: in the first box there are 9 choices, then 6 choices (one row already occupied), then 4 choices (two rows), then 6 choices (one column) and so on.

$$\text{Number of possible templates} = 9 \cdot 6 \cdot 3 \cdot 6 \cdot 4 \cdot 2 \cdot 4 \cdot 2 \cdot 1 = 46656$$

The method involves generating all such templates, and then finding the subset of templates that are valid in the context of each digit, with respect to the given clues. Then, based on the remaining possibilities, we search through all possible combinations of entries of the empty cells. I opted for a brute-force algorithm since this would be robust; since all possibilities are considered, a solution will be always found so long as it exists. Moreover, due to the simplicity of the Sudoku-solving problem, even in the worst case (as I discuss later) brute-force does not take too long provided the code is optimised. In particular, this method has a far smaller search space than backtracking without using templates, and so should be faster, so long as finding the subsets of templates is implemented to be fast.

In my original plan for the software, the first consideration was the memory required to store all the template arrays, which is the most memory demanding data structure in my solution. Assuming the default NumPy behaviour of storing integer arrays as 64-bit signed integers, this would require $46656 \cdot 81 \cdot 8 \approx 3 \cdot 10^7$ bytes, or 30MB, which poses no issues. I decided to store the abstract representation of the grid as a `np.ndarray` type of shape

(9, 9) since this greatly improves the speed of the high number of mathematical operations performed on the grid to find the solution. However I also anticipated that this could have been subject to change as I started to develop the code. For this reason, I decided to code the solving part of the code first, and then the next major parts, namely handling the input and output of data, could be coded according to the final choice of representing the grid internally. It appeared that the two main parts of the code were the solving and then the input and output of data, so I decided that a reasonable modular approach would be to create two separate modules, `solve.py` and `data.py` under the package `sudokutools`. Finally, I also noted that significant error catching was required somewhere between the input and solve stages, primarily because beyond 'obvious' violations from inputs that do not resemble a sudoku grid, there is the more subtle violation of a grid that does not have a solution, and could therefore cause the solver code to continue indefinitely if not caught.

INSERT DIAGRAM

Development, Experimentation and Profiling

I implemented a testing-led development strategy. This meant that some of the earliest steps that I took in developing any code was orchestrating unit tests to ensure that the next anticipated phase of development was working properly. The main advantage of this was that it heavily directed my development path, keeping me on the right track. For instance, in one of my earliest commits (3132ebb) I created the following unit test:

```
def test_templates():
count_array = np.zeros((9, 9))
for template_array in generate_templates():
    count_array += template_array
count_array_expected = 5184 * np.ones((9, 9))
assert count_array == count_array_expected
```

Besides its purpose for validating my future code, this also ensured that in the next phase of development, I specifically had a function called `generate_templates()`, and that its output was of the form of a 9x9 NumPy array with binary entries. This approach during development in general, I already had an idea of what functions I intended to create, and what their inputs and corresponding outputs should look like.

I also made full use of Git to version control my code. I used three branches: `main`, `dev`, and `test`. Broadly speaking, I developed the code primarily in the `dev` branch, and then merged to main once I had code that passed the current testing suite. The `test` branch was for experimental features, but I only used this at one point which I mention later; this was when I had an idea that conflicted with my original plan. I also had a strictly self-enforced commit message style for almost all commits, broadly following the "Convential Commits" guidance [3]. In particular, while my commit messages weren't as detailed as suggested there, I aimed for a regular structure of a full sentence starting with an imperative term ("Create", "Change", "Remove", "Profile", etc.), and with one of the given "type" words suggested by [3], such as "Fix", "Test", "Docs", "Build", and "CI".

According to my plan the first development of functional code handled implementing the templates part of the solving method. This comprised two functions, `generate_templates` and later `find_valid_templates`. The former implemented a generator which iterated through all possible templates. I found that an efficient way of doing this was to:

- enumerate each arrangement of a fixed digit (which I labelled 1) on the grid by a list of length 9, with the i^{th} digit of the array giving the column to which the 1 in the i^{th} row belonged - such an arrangement inherently satisfies the row condition;
- ensure the column condition is satisfied by only iterating through permutations of (1,...,9);
- ensure the box condition is satisfied by finding the remainder of each entry of the array when divided by 3, corresponding to the three boxes on that row, and checking the first, second and third group of 3 entries of the array are individually permutations of (0, 1, 2).

Developing the first versions of the template routines that passed the tests was followed by refactoring of some of the functionality. For instance in one commit, the way that `find_valid_templates` checked whether any of the 8 remaining digits' clues conflicted with a template was massively improved. Originally, this was done using a double-nested for loop, checking each cell, and making a list of the other digits on each iteration. This was then changed to initialise the list of `OTHER_DIGITS` as a global variable in the module, so that it was only constructed once, saving computational time. The nested for loops were also substituted for the use of the NumPy method `isin()` which later became the use of NumPy's `any()` method. This refactoring not only improved the computational speed of the routine, but also achieved the same functionality in fewer lines, improving readability of the code.

The first prototype of the solving routine experimentally used a simple for loop to iterate through the Cartesian product of the subsets of templates for each digit, using the `product` method from `itertools`. This was tried before the backtracking algorithm since it only required a very short (11 lines), readable routine, and was therefore worth investigating to see if it was viable. While this passed the testing suite, even for the most difficult starting grids, it was immediately found to be very slow for any non-trivial starting grid.

At this point, considering how one would solve some of the hardest Sudoku grids by hand illuminated the fact that a further 'smart' filtering of the subsets of templates was possible. In particular, in some cases the initial clues for a given digit allow certain cells to be filled by using the fact that they are a kind of 'fixed point' in the subset of templates, that is, in all valid templates, a certain cell is always occupied by that digit, but this isn't given as an explicit clue. The simplest case of this is when, say, the top left box does not contain a clue for the digit 7, but the other boxes occupying the top row and the left column, do. In turn, in many cases this allows us to fill in some cells on the grid with certainty, so we can re-use the `find_valid_templates()` routine on the updated grid, repeat this for all digits in a loop. This experimental feature was developed in the `refine_valid_templates()` method in the `test` branch, with a loop halting when the grid is no longer updated after iterating through all digits. The advantage of the feature is that it also uses a brute-force style approach, but in each instance the search space is very small (assuming the initial clues included at least one of each digit 1-9, the set of templates for a given digit to search through is no more than 5184). In some of the starting grids with more clues, this routine was found to be sufficient to solve the grid entirely.

After this, it was clear that the backtracking part of the solving algorithm was necessary to solve—in a reasonable time-frame—the grids that were still fairly empty after the new template step. In the first prototype of this routine, a nested list of generators was instantiated at the start of the routine, with each generator representing the possible digits that could occupy the corresponding cell. A NumPy array `search_positions` was used to store the coordinates of the cells that were empty, and thus formed part of the backtracking procedure. The variable `search_idx` was initialised at zero, and used to keep track of the current location of the backtracking procedure on the grid. At each stage, this variable was incremented by 1 to move to next cell in the search, when a valid entry was found for the current cell, and decreased by 1 to move to the previous cell, when all options for the current cell were exhausted. Once this variable reached the maximum (the number of coordinate pairs in `search_positions`) and the entries of the search cells constituted a valid grid, the backtracking was halted.

Next, the data handling part of the code package was developed. This module has two functions, `read_grid` and `write_grid`. Their development began with simple functionality that effectively handled the valid input cases where the arguments they received were in the expected format. For instance, the first prototype of `read_grid` simply read the specified file line-by-line, building a nested list of characters in each line that was a digit character, and returning this as a NumPy array. This worked for text files formatted in the prescribed way, but lacked robustness, raising errors in the face of the most minor violations to this format.

I then implemented extensive error catching starting with the data handling routines in `data.py`. This was appropriate since this module contains front-end functions where the input is more likely to created by a user who could make a mistake such as:

- Formatting the grid incorrectly
- Specifying the wrong filepath, or similarly not using a .txt file.
- Specifying a non-existing file.

The error catching was developed to handle such transgressions. The approach taken was to take a lenient approach in the case of a badly formatted grid; so long as the text file contained exactly 9 lines that contained any number of digit characters, and each of these contained exactly 9 digit characters, the input file was accepted. This meant that input files unambiguously specifying a Sudoku grid were accepted, despite any bad formatting with the characters ”—”, ”+” and ”-”, and also reduced the complexity of the code as these decorative characters were not checked. In the case of any of the other violations above, an informative string was printed explaining the error, and an empty Sudoku grid was returned.

At this point the `sudokutools` had been fully constructed with all necessary functionality, and steps such as testing and documentation had been implemented. The next step was to profile the code using `line_profiler`, whose outputs were exported as .txt files in a separate `prof` directory. This was done in the `dev` branch of the repository. The profiling was focused on the solving functions, which were the main bottleneck in the end-to-end solution of a grid specified by an input .txt file, to be outputted to the terminal or a .txt file. In particular, the function `solve_backtrack` was profiled with input grids of a variety of difficulties, since this function also called all of the other backend solving functions. Regardless of the starting grid, it was found that the largest percentage (between 75.5% and 99.5%) of time was spent on the lines calling the `find_valid_templates()` and `refine_valid_templates()` functions. Further profiling of the latter revealed that most of its own computational time was spent calling the former function, which was therefore the main bottleneck in producing a solution in a short amount of time.

In order to optimise the `find_valid_templates()` function, the main change was to represent the full space of templates as a 3D NumPy array of shape `(46656, 9, 9)` rather than as a list of arrays. Moreover, where before the subsets of valid templates for each digit were stored in a nested list, this was also changed to be a NumPy array of shape `(9, 46656)`, with each row representing the indices of valid templates for that digit. This enabled the use of NumPy's `any()` method; rather than looping through the list of all templates, to find the list of valid templates. This was compatible with the representation of the grid as a 2D NumPy array itself; the check involved creating two new `(9, 9)` shaped arrays, representing the locations of the digit of interest, and the 'other digits' in the grid, and then checking that both were compatible with the any given template array by using the any() method across the 2nd and 3rd axes of the `ALL_TEMPLATES` array.

Validation, Unit Tests and Continuous Integration

Validation ?

Testing: reason for multiple files

pre-commit: testing excluded, changes to the args

Packaging and Usability

Structure of packages, modularity, refactored code routines

Usability: docker, documentation, error catching?

Summary

Reasons for software development practices

Lessons learnt?

References

[1] wikipedia.org. Mathematics of sudoku, 2023. Date accessed: 7/12/2023.

[2] wikipedia.org. Sudoku solving algorithms, 2023. Date accessed: 7/12/2023.

[3] Conventional commits, 2023. Date accessed: 11/12/2023.