C1 Coursework

Cailley Factor

December 4, 2023

1 Introduction

This project aimed to develop a Sudoku solver algorithm with an input.txt file containing an example puzzle of the following format to be run from the command line with the 0 used to specify empty cells:
000—007—000
000—009—504
000—050—169
—+—+—
080—000—305
075—000—290
406—000—080
—+—+—
762—080—000
103—900—000
000—600—000

To use research computing best practice, we conducted prototyping for both the unit tests and algorithm before we developed the code. Before coding, we set up a .gitignore file. Additionally, we set up a .pre-commit-config.yaml file to run pre-commit checks before allowing the code to be committed to the remote repository.

For each function added to the main code, we developed robust unit tests. Before pushing any code to the remote git repository, we ran the unit tests. We implemented version control using git, branching before adding new functions to the code and before any significant development changes, and merging to git once the functions passed the unit tests and pre-commit checks. Once we developed the initial code, we added the unit tests to the pre-commit checks. We profiled the code using CProfiler and LineProfiler to ensure its efficiency. Additionally, we produced a doxyfile to utilise Doxygen for documentation, and finished by containerising the code with a dockerfile with which to use Docker.

2 Selection of Solution Algorithm and Prototyping

To select a solution algorithm, we considered several solution algorithms but decided on starting with the backtracking algorithm, in part, due to its simplicity. The advantages of the backtracking algorithm are that a solution is guaranteed as long as the puzzle is a valid Sudoku puzzle. Some of its disadvantages are that it can take variable time to solve the Sudoku puzzle depending on the input puzzle and can be relatively slow compared to other algorithms. Additionally, the backtracking algorithm finds one solution to a puzzle, even if there are several valid solutions.

For prototyping, we developed the following framework, graphically displayed in Figure 1. Firstly, a function removed delimiters and converted the input.txt file to a NumPy array in order to easily use vector NumPy operations to speed up the code. The prototyping diagram included error traps to ensure a correct input including making sure the digits were all between 0 and 9, that 81 digits were provided, and that the puzzle contained no duplicates in rows, columns, or blocks. Following this, a function made a list of all

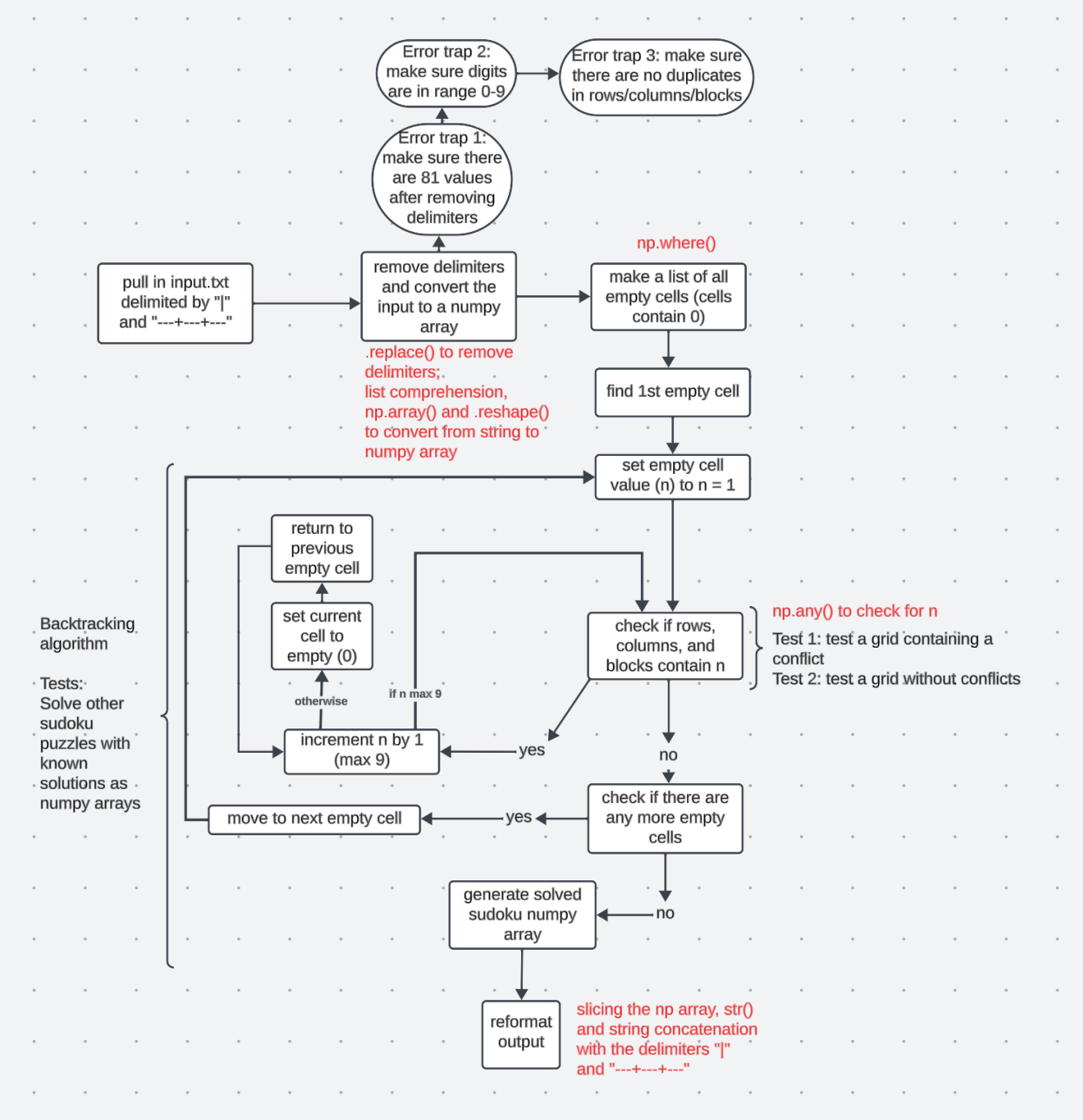

Figure 1: Prototyping diagram

empty cells, i.e., cells containing zero. A function found the first empty cell from the list and set the empty cell value (n) to n=1. A function then checked if rows, columns, and blocks contained n.

If the function returned that there was a conflict, the empty cell value (n) was incremented by 1 with a maximum of n=9. The function checked again for a conflict. If the function could not be incremented, as n was already 9, then the function backtracked. The function set the current empty cell value to zero and returned to the previous empty cell. The function incremented its previous value by 1 and the process continued.

If the function returned that there was not a conflict, then the function checked that there were more empty cells. If there were more empty cells, the function returned to the next empty cell, set its value to n=1 and checked for conflicts, repeating the rest of the process. If there were no more empty cells, the function generated the solved Sudoku puzzle as a NumPy array and converted it to the appropriate final output.

In the prototyping stage, unit testing ideas were developed, including testing that the conflict function correctly resolved conflicts for both a puzzle containing conflicts and one that did not contain conflicts. An idea for a unit test for the Sudoku solver was testing that an example puzzle could be converted into the correct solution.

3 Development, Experimentation and Profiling

3.1 Initial Development

The code was designed to be run from the terminal utilising code in the main.py file which called functions from the src module using an input file called input.txt. The solved puzzle was output to the terminal, as were any error trapping messages, described in more detail in section 4.

The research computing principles of version control and modularity were implemented. The assessment git repository was cloned to a personal machine. Branching was used when adding new functions or other large changes to the code to prevent any accidental damage to the existing working functions. Functions were only committed and pushed to the remote git repository after the functions were completed in full and passed pre-commit checks. Functions were merged to the main branch, only after appropriate unit testing, as described in section 4, and the inclusion of appropriate documentation code. In hindsight, code could have been pushed more frequently to the branches of the remote repository due to the individual nature of the project. Additionally, following a fixed convention for branch names, attempted later in the project, could have improved the version control methodology.

Following prototyping and during development, the file structure evolved to make the code more modular. The functions involved in the Sudoku solver were separated into different files in an src module and the module was imported into the main.py file from the root directory. The main.py file contained relative imports for the input.txt file, containing the input puzzle of the form outlined in the assessment guidelines. Unit tests were also included in a separate test folder, discussed in more detail in section 4.

The code which made a list of all the empty cells of the input puzzle was also moved inside of a wrapper function for the Sudoku solver. This was not anticipated during prototyping and facilitated unit testing of the Sudoku solver, as the wrapper function could be unit tested with only an input puzzle and did not require a list of empty cells, enabling puzzle and solution pairs to be utilised easily as unit tests. Additionally, the input conversion code was moved to the input_conversion file of the src module and converted to a function called input_converter. This enabled appropriate unit testing of the input_conversion process. Similarly, the output_conversion code was moved to the output_conversion file of the src module and converted to a function called output_converter to also enable greater modularity, and improved unit testing.

3.2 Sudoku Solver Approach

To make the run-time of the backtracking algorithm as fast as possible, we minimised loops and favoured the use of vector operations, as indicated in the prototyping stage. Following import to the main.py file using relative imports, the input puzzle was converted into a NumPy array by the input_converter function of input_conversion.py, eliminating the delimiters using the .replace function. For efficiency, a list comprehension was used instead of a loop to convert the input text into digits for a NumPy array. This conversion to a NumPy array enabled the efficient use of NumPy vector operations in the functions used in the Sudoku solver

algorithm. The inputs were validated as described in section 4 and main.py called the solve_sudoku_wrapper function from the src module.

As discussed, the solve_sudoku_wrapper function created a list of the indices of empty cells, i.e., cells containing the value zero. The wrapper function then called the solve_sudoku function also in the src module. The solve_sudoku function executed the backtracking algorithm, beginning with the first empty cell, obtained from the existing list of empty cells, defined by the wrapper function.

The solve_sudoku function utilised a base case and a recursive case to implement backtracking, as outlined in the prototyping diagram of section 2. The base case checked if the index of the list of empty cells was equal to its length. This ensured that the solve_sudoku function returned true when all the empty cells were filled, so the wrapper function could produce a solution to the puzzle.

If the empty cells were not all filled, the recursive case looped through the values one through nine to fill the empty cell with a value which did not create conflicts with existing values in the same row, block, or column of the puzzle. The exists_conflict function of the exists_conflict.py file of the src module checked that the value set for the empty cell was not already found in the same row, block, or column. The exists_conflict function defined the row, column, and block of the empty cell. Rather than a slow loop, the vector function np.any was used to check for conflicts.

If there were no conflicts, the solve_sudoku function called itself recursively to solve the next empty cell. If there was a conflict, which was not resolved by looping the current empty cell value from one through nine, then the function backtracked. It reset the current empty cell to zero and tried the next number in the loop for the previous empty cell. If the solve_sudoku function returned False, an error trap was added, as described in section 4, in order to raise an error when puzzles were invalid, despite the input puzzle having no row, block, or column value duplicates.

Lastly, the output_converter function was run on the output from the solve_sudoku_wrapper function by main.py to convert the NumPy array back to the appropriate delimited string. This function looped through each row to make the appropriate text changes.

3.3 Profiling

Due to the efficiency best practice of section 3.2, the Sudoku solver approach utilising the backtracking algorithm only took 0.314 seconds to execute using the example input puzzle defined in the instructions.md file. To get a function-level understanding of profiling, CProfile was run from the terminal and the outputs saved to a file called output.prof in the profile file. To easily access the information from the file, the pstats module was used to extract this data. It was sorted by the total time spent on each function and the top ten functions extracted and printed. Only the exists_conflict and solve_sudoku functions appeared in the list, having total times of respectively 0.078 and 0.010 seconds. Line profiling was then used to understand what part of the exists_conflict function was slowest, given that CProfiler only provided information on a per function level. Decorators were added with @Profile to the function definitions and the line profiling output viewed in the terminal. From the exists_conflict function, the vastly most time-intensive part of the function (71.4% of the time) was the line using np.any to check for a value n occurring in the puzzle row, puzzle block, or puzzle column, defined based on the empty cell. Given that this line was already using vector NumPy operations, instead of loops, no change was made to the code based on this analysis.

3.4 Further improvements

Although the input puzzle was efficiently executed by the backtracking algorithm. The following puzzle designed to be extremely difficult for backtracking proved very slow[1]:

900—800—000
000—000—500
000—000—000
—+—+—
020—010—003

[1]Armando Matos. "The most difficult puzzles are quickly solved by a straightforward depth-first search algorithm". (2016). Available at: `https://www.dcc.fc.up.pt/~acm/sudoku.pdf` (Accessed: 16 December 2023)

010—000—060
000—400—070
—+—+—
708—600—000
000—030—100
400—000—200

With the initial backtracking approach, a solution was not generated even after running the code for half an hour. Given the significant difference in the run time of this puzzle versus the example provided in the assessment guidelines, algorithmic improvements were sought rather than function changes. To speed up the backtracking algorithm for otherwise slow puzzles, two new functions were added. The rearrange function defined three blocks of three rows of the puzzle and rearranged them by the number of filled squares in the row block. In other words, from top to bottom, the row blocks were ordered with the most to the least puzzle clues. This function design was because the np.where function of the solve_sudoku_wrapper function found the list of empty cells row by row from left to right. Thus, having more clues earlier in the puzzle prevented as many mistakes in defining earlier cell values and thus reduced the amount of backtracking necessary to derive a puzzle solution. The rearrange function also returned a list of keys stating the order of the blocks. The arrange_back function was defined to convert the row blocks of the solved puzzle back to their order in the initial puzzle by arranging the row blocks in ascending orders of their corresponding keys. Upon profiling, the previously long-running puzzle took only 0.542 seconds to find a solution and the cProfiler output was saved to output_hard_puzzle.prof in the profiling folder. The puzzle defined in the assessment instructions was then profiled again using cProfiler and executed efficiently in 1.033 seconds, though this was longer than the time taken without the added code functions.

4 Validation, Unit Tests and CI set up

For input validation, a try-except block was used to discover if the input.txt file was imported correctly using relative imports, and if not raised an error that the file was not found. During the reformatting of the input.txt file in the input_converter function, the function raised an error if extra spaces were eliminated during the reformatting. Error traps for the input included making sure that the input contains only digits and exactly 81 digits after the delimiters were removed. Additionally, the function produced a message if there were certainly multiple solutions to the puzzle based on having less than 17 clues, i.e., filled in cells[2]. Lastly, errors were raised if there were duplicates in a row, column, or block. Np.unique was used to count the number of times numbers other than zero were present and an error was raised if this count was more than 1 for each column, row, or block. Output validation was implemented by ensuring that an error message was raised in the solve_sudoku_wrapper function if the solve_sudoku function returned false, i.e., the puzzle could not be solved, but the input puzzle did not contain any row, column, or block duplicates.

Unit tests were used to ensure that the functions of the src module were correctly implemented. Before committing any changes to git and pushing them to the remote repository, the unit tests were run, initially from the terminal itself before the entire code was complete, and later the unit tests were implemented in the continuous integration file. One unit test for the exists_conflict function included checking that the exists_conflict function did not return true for several numbers placed just outside of the block, row, and column containing the empty cell that the function was analysing. Another unit test for the exists_conflict function involved ensuring that the exists_conflict function returned true for each of three conflicts, one in the same row, one in the same column, and one in the same block as the empty cell. A unit test for the solve_sudoku_wrapper function involved solving an example Sudoku puzzle to ensure the function returned the correct answer. The example puzzle and solution for the test were generated with the help of ChatGPT, as detailed in the appendix section. Additional unit tests were added upon creating the input_converter and output_converter functions. These unit tests utilised an example delimited text and its respective NumPy array to ensure that the input_converter function converted the delimited text to a NumPy array, and the output_converter function could do the reverse conversion. Similarly, upon adding the rearrange and

[2]Gary McGuire, Bastian Tugemann, Gilles Civario. "There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem". arXiv preprint arXiv:1201.0749 (2013). Available at: `https://arxiv.org/abs/1201.0749`

arrange_back functions, unit tests were added. The rearrange unit test utilised an example grid to test that the function could rearrange the row blocks in order of most to least clues. The arrange_back unit test utilised an example grid with a corresponding list of keys to test that the arrange_back function could arrange the row blocks of the grid in ascending order of the keys.

In terms of continuous integration, a git ignore file was set up. The .gitignore file was set up based on the reference on the github website [3]. A .pre-commit-config.yaml file was set up with hooks containing Flake8 and Black for code formatting. The testing suite was later added to the .pre-commit-config.yaml file to prevent any code edits disrupting the code functionality, once the key components of the Sudoku solver were functional.

5 Packaging and Usability

For packaging and usability, we included a Conda environment file RC_environment.yml, specifying standard Conda channels and the inclusion of the NumPy package, as a dependency. Additionally, a Dockerfile was included in the base repository in order to generate an image to run the code. The Dockerfile utilised a minimal Docker image which had Miniconda3 pre-installed. The working directory inside the container was set to /cf593_doxy and the contents of the current directory, the cloned git root directory, were copied into this directory in the container. The RC_environment.yml file was then used to create a new environment using Conda. Inside this Conda environment, the main.py file was then run with the argument input.txt in order to execute the Sudoku solver and return the output to the terminal.

6 Summary

In summary, the Sudoku solver utilised a backtracking algorithm with input Sudoku puzzles in a specific delimited text format. From prototyping, the code was designed to be efficient by favouring the use of NumPy vector operations. As the development evolved, the code became more modular with functionality removed from the main.py file and packaged into functions of the src module. This enabled extensive unit testing before committing to the main branch of the repository. Error trapping was also used to ensure solvable input puzzles. The backtracking algorithm efficiency was improved by adding functionality to rearrange the row blocks of the puzzle to provide the most clues at the beginning of the puzzle. Another function was added to return the solved puzzle with the original row block order. This enabled puzzles very slow for traditional backtracking to be solved very efficiently. The final code was packaged using docker for maintainability with Doxygen commands provided for documentation generation.

7 Appendix

As also specified in the README.md file, ChatGPT version 4.0 was used for the following aspects of the code development:

- Prototyping the import of the input.txt file into the main.py module. The following prompt was used: “methods of importing files efficiently from another directory in python”. Several options were provided and in the end, we decided on simple relative imports.

- Prototyping the efficient conversion of a string of integers into a NumPy array. The following prompt was used: “how to most efficiently convert a string of integers into a numpy array”. This provided the insight to use list comprehension to achieve this and an example list comprehension code. This was modified to be used in the Sudoku solver context.

- Prototyping maximising the use of numpy vector operations over loops. The following prompt was used:“vectorisation operations in numpy”. This gave some insights into useful vector operations such as incorporating np.any() and np.unique()

[3]GitHub Repository ”gitignore”: `https://github.com/github/gitignore` (Accessed: 26 November 2023)

- Changing the format of the output of "np.where(puzzle == 0)" in the solve_sudoku_wrapper function of the solve_sudoku.py module. The following prompt was used "how to index np.where() outputs". The code was used to convert a tupule of arrays into a list of tupules containing the indices.

- Coding a python script to read from a CProfile file. The following prompt was used: "standard practice to save a CProfile file and read from it". The code was used in the profile.py file to read from the profile_output.prof file and extract key information

- Fixing the base environment in the Dockerfile. The prompt was the error message received, when utilising a different base image. ChatGPT suggested the use of miniconda as a good base image, which we implemented.

- The example puzzle and solution for the unit testing for the solve_sudoku_wrapper function was created using ChatGPT with prompt: "example sudoku puzzle and answer in python as NumPy array". The NumPy array puzzles were then used in the unit test.

- Storing and accessing values from a dictionary storing tuples. The following prompt was used "how to sort and access values from a dictionary storing tuples". The code was used to understand how to sort and access the dictionary values within the rearrange function of the rearrange.py module.