C1: Research Computing

Coursework Assignment

University of Cambridge

Department of Physics

Prepared by:
Sabahattin Mert Daloglu

December 17, 2023

1 Introduction

Sudoku is a logic-based puzzle game played by inserting numbers from 1 to 9 into a 9x9 grid of 81 cells. The grid is divided into 3x3 boxes and some of the cells are pre-filled with numbers as *clues* for the player. The objective of Sudoku is to fill the remaining cells in such a way that no number is repeated within each 3x3 box, in each row, and each column. While it is possible for certain Sudokus to have more than one solution, this coursework focuses on developing an algorithm that provides a single solution for simplicity and computational efficiency.

2 Selection of Solution Algorithm and Prototyping

2.1 Sudoku Solving Algorithms

In the era of supercomputers and GPUs, brute-force algorithms are occasionally favored for easier implementations and certainty of the solutions with an expense in computational power. In the case of the 9x9 Sudoku solver algorithm, the computational expense is less of a worry even with brute-force algorithms, hence, the brute-force method provides a practical approach. This coursework is written to demonstrate a well-structured software development project on one of the most widely used Sudoku-solving algorithms, backtracking. The advantages of the backtracking algorithm are that a solution to a given valid Sudoku is guaranteed and the code implementation is relatively more straightforward. The most prominent setback of this method is that it tends to be slower compared to other Sudoku-solving methods. Backtracking is a brute-force algorithm that finds empty cells on the grid and explores every possible configuration related to a consequence of a single random assignment. This type of search is also referred to as *depth-first search* and can be visualized by a tree analogy. The first choice of empty cell is the base of the tree which stems into 9 branches in the 9x9 Sudoku case. The choice of which branch to proceed towards is constrained by the three rules of Sudoku mentioned above. The sub-branches related to this node of the tree are explored recursively under the same three rules, exhaustively. A full branch of solutions is explored in *depth* until no new assignments to an empty cell under Sudoku rules can be made before moving into another branch. This branching is illustrated in Figure 1 for a 4x4 Sudoku.

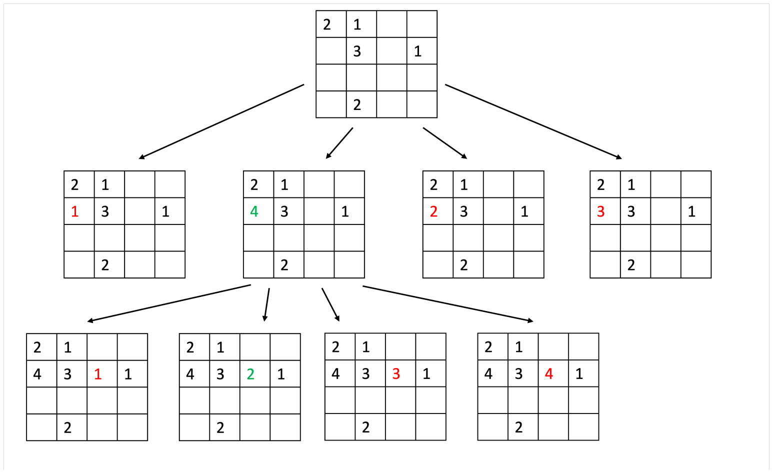


Figure 1: An example of branching in a backtracking algorithm used to solve a 4x4 Sudoku. For each empty cell, the algorithm explores every possible guess before moving on to the next empty cell.

2.2 Prototyping

Unlike the algorithm of choice, we shall avoid *depth-first* code implementation(start coding until a mistake is caught) in this coursework to reduce the number of times `git revert` is used during the development stage. Thus, a thorough list of prototyping in implementing the backtracking algorithm is given in the steps below. Furthermore, a flowchart illustrating the mind map of the code structure is presented in Figure 2.

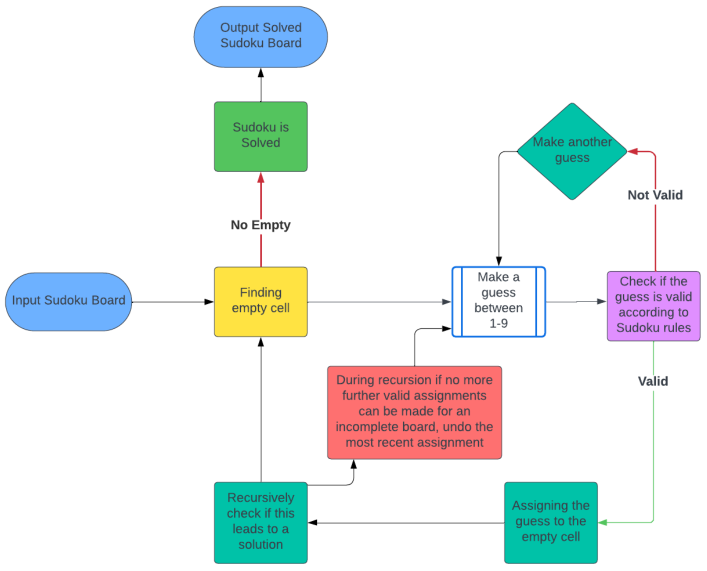

Output Solved Sudoku Board
Sudoku is Solved
No Empty
Make another guess
Not Valid
Input Sudoku Board
Finding empty cell
Make a guess between 1-9
Check if the guess is valid according to Sudoku rules
Valid
During recursion if no more further valid assignments can be made for an incomplete board, undo the most recent assignment
Recursively check if this leads to a solution
Assigning the guess to the empty cell

Figure 2: A flowchart created in Lucidchart[1] illustrating the prototyping stage of the project.

- The number of possible 9x9 Sudoku grids is in the order of $\sim 10^{21}$[2]. However, there is no computationally expensive calculation involved in exploring each branch of the solution. Therefore, parallelization is not expected to be required.
- The input data is a text file containing the Sudoku board. This file will be specified in the command line, which should be extracted using the `sys` package.
- Input file should be opened and read by a function which can be named `load_txt`.
- The read input file will output a string format, which should be converted into a 9x9 2-dimensional `NumPy` array containing only numbers inside the cells. Naming the function `board_to_array` would be sensible for this purpose.
- After the Sudoku board is solved, a function converting a `NumPy` array to a text file is required. Naming the function `array_to_board` would be sensible for this purpose.
- A module named `board` can be created to store `array_to_board`, `board_to_array`, and `load_txt` functions. This module should store any other functions related to data parsing and board manipulation.
- A function that searches for empty cells in the two dimensions of the Sudoku array will be required. Naming this function `find_empty` would be sensible for this purpose.
- Another function is required to validate the assignment of a number between 1 and 9 to an empty cell, ensuring it follows the three Sudoku rules. Naming this function `check_guess` seems appropriate for its purpose.
- Functions from `NumPy` such as `where` and `any` can be used inside the search algorithm to avoid double for-loops and decrease the running time for efficiency.
- Local variables will be useful in constructing modules or the main function. Coordinates of the 3x3 box where the initial guess belongs will be required when checking for the validity of the guess.
- A module named `backtracking` can be created to store `find_empty` and `check_guess` functions. These two functions will play a central role in implementing the backtracking algorithm.

3 Development, Experimentation and Profiling

3.1 Development

The development part of the coursework was started by focusing on the two main functions that will be used for the backtracking algorithm. Finding the empty cells of a Sudoku board is the first step in attempting a solution if we exclude data parsing steps which will be discussed in the Experimentation section. A pseudo-code explaining the `find_empty` function is given in Algorithm 1.

Algorithm 1 Find empty cell inside a Sudoku board

```
function FIND_EMPTY(sudoku: np.ndarray))→ tuple
    for $i$ in RANGE(len(sudoku)) do                          ▷ Itirate over rows
        for $j$ in RANGE(len(sudoku[$i$])) do                 ▷ Itirate over columns
            if sudoku[$i, j$] == 0 then
                empty_cell ← ($i, j$)
                Return empty_cell
            end if
        end for
    end for
end function
```

The algorithm provided above is fairly straightforward demonstrating a 2-dimensional search of the Sudoku board through nested for loops. Given that the input file denotes empty cells with 0's, the `find_empty` function returns the position of the first cell containing zero. Subsequently, a number is assigned to this empty cell using a simple for-loop within the range of 1-9. After this assignment, the next step is to check if this guess aligns with Sudoku rules. To achieve this, the `check_guess` function is designed as outlined in Algorithm 2.

Algorithm 2 Check Validity of Guess in Sudoku

```
function CHECK_GUESS(sudoku: np.ndarray, guess: int, empty_cell: tuple) → bool
    $x, y$ ← empty_cell[0], empty_cell[1].            ▷ Extract the position of the empty cell

    $x\_box$ ← $(x//3) \times 3$                      ▷ Coordinates of the 3x3 box where empty cell belongs
    $y\_box$ ← $(y//3) \times 3$
    $box\_y\_range$ ← $[y\_box, y\_box + 1, y\_box + 2]$
    $box\_x\_range$ ← $[x\_box, x\_box + 1, x\_box + 2]$

    for $i$ in $box\_x\_range$ do.        ▷ Checking the uniqueness of assigned guess inside the 3x3 box
        for $j$ in $box\_y\_range$ do
            if sudoku[$i, j$] == guess and $(i, j) \neq$ empty_cell then return False
            end if
        end for
    end for
    for $i$ in Range(len(sudoku[0])) do        ▷ Checking the uniqueness of assigned guess in a row
        if sudoku[$i, y$] == guess and $i \neq x$ then return False
        end if
    end for
    for $j$ in Range(len(sudoku)) do        ▷ Checking the uniqueness of assigned guess in a column
        if sudoku[$x, j$] == guess and $j \neq y$ then return False
        end if
    end for
     return True
end function
```

The output of the `find_empty` function (stored as a tuple in the variable `empty_cell`) serves as input for the `check_guess` function. Algorithm 2 introduces local variables to assign the coordinates of the 3x3 box to which the input empty cell belongs. When checking the uniqueness of the assigned guess inside the box, it is important to check it against the other 8 cells inside the box, excluding the chosen `empty_cell`. This is done by the second condition in line 11 of Algorithm 2, and similarly line 16 for rows and line 20 for columns. However, since the chosen empty cell will always contain the value zero, and the guess always falls within the range 1-9, checking the guess against the empty cell has no impact on the uniqueness checks. Consequently, removing the second conditions in lines 11, 16, and 20 would not change the output of the algorithm, but they are kept to provide a comprehensive demonstration of Sudoku rules.

3.2 Experimentation and Profiling

After developing the main functions required for the backtracking algorithm, the next step is to create data parsing functions in order to test and experiment with different Sudoku boards. These could also be covered in the Development section, however, parsing functions required more *trial-and-error* approach due to the variety of ways to handle different types of input board structures and catch possible errors. The natural starting point is the creation of a function named `load_text` designed to open and read the input text file. Within `load_txt`, Python's `open` function is employed with the `'r'` argument to exclusively read the input file. The use of `try` and `except` error-handling functions is important in catching an `IOError`. This error may indicate issues such as an invalid file name, an incorrect location, or insufficient permission— all of which are related to the input file. The `load_txt` function reads and returns the Sudoku board in a single string format. However, for subsequent processing and manipulation, it is essential to convert the elements' data type to integers and eliminate the `|, -, +` characters used as separators for the 3x3 boxes. The `board_to_array` function employs a straightforward approach to convert the input Sudoku board, initially represented as a string, into a 2-dimensional `NumPy` array. It begins by creating an array called `sudoku_number` in line 4 of Algorithm 3, containing string representations of numbers from 0 to 10. The function then scans each element of the input board, checking if it matches any of the values in `sudoku_number`. For each matching value, it appends the first nine elements to the `row_array` (line 7), which represents a row of the resulting 2-dimensional array. This process continues, appending each row to the outer array (line 9), effectively stripping non-number values from the input board. Finally, each element of the resulting array is converted to an integer type in line 14. When experimented with complete Sudoku boards, `board_to_array` function works fast, and accurately. However, when applying this function to a Sudoku board containing missing cells (cells with no assigned values), the resulting array deviates from the expected shape of 9x9. Although this does not cause a particular error for this function, an incorrect board size error will be carried out to the main backtracking functions. To avoid this happening, line 15 was implemented inside a try-except block to catch the `ValueError` from propagating further into the algorithm, ensuring the robustness and proper functionality of the overall backtracking process.

Algorithm 3 Convert Sudoku Board to 2D NumPy Array

```
 1: function BOARD_TO_ARRAY(board: str) → np.ndarray
 2:     SET data_array TO np.array([])                          ▷ Outer array of 2D NumPy array
 3:     SET row_array TO np.array([])                           ▷ Inner arrays representing rows
 4:     SET sudoku_numbers TO np.arange(0, 10, 1).astype(str)   ▷ Sudoku numbers
 5:     for value IN board: do
 6:         if value IN sudoku_numbers: then
 7:             SET row_array TO np.append(row_array, value)
 8:             if len(row_array) EQUALS 9: then                ▷ Row is complete, switch to the next row
 9:                 SET data_array TO np.append(data_array, row_array)
10:                 SET row_array TO np.array([])               ▷ Emptying the row_array for the next row
11:             end if
12:         end if
13:     end for
14:     SET data_array TO [eval(i) FOR i in data_array]         ▷ Converting elements from str to int
15:     SET sudoku_array TO np.reshape(data_array, (9, 9))
16:     return data_array
17: end function
```

Although complete, the input Sudoku board can also be invalid that is, it does not obey the Sudoku rules from the beginning. Thus, it is a good practice to check the validity of the input board before attempting a solution to ensure that the backtracking algorithm does not get stuck. The `check_board` function was created to check if the input Sudoku board is valid by assuring no element is repeated in each row, column, and 3x3 box. Conceptually, `check_board` function follows a structure analogous to Algorithm 2 but this time instead of checking a given guess against other cells, `np.unique` function was utilized. For example, while checking each column, the number of non-zero elements in a column was calculated and compared to the number of elements after the column was passed into the `np.unique` function. If these two lengths are the same then there is no repetition of a value. However, if two lengths are not equal, this indicates a repetition and an invalid Sudoku. In the event of such repetition, the function provides additional guidance by printing the corresponding column, row, or box number. This printed information serves as a helpful guide for users to rectify the input board and ensure its validity.

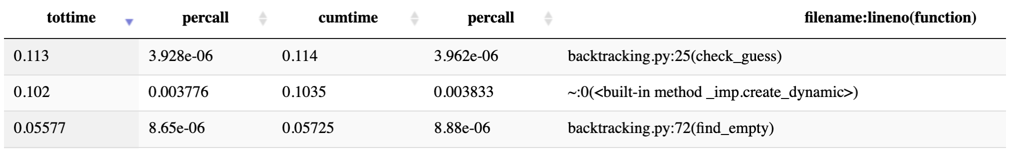
After developing the individual functions as outlined above, the creation of a main function becomes essential. This main function, residing in a separate `solve_sudoku.py` file, manages the sequence of predefined functions in a structured manner, as illustrated in Figure 2. The pseudo code providing an overview of the main function is presented in Algorithm 4. Initially, the function utilizes the `find_empty` function from the backtracking module to identify an empty cell. If no empty cell is found, indicating the completion of the Sudoku board, the function returns `True`. Conversely, if an empty cell is identified, the algorithm proceeds to try guesses within the range of 1-9. The validity of each guess in the empty cell is assessed using the `check_guess` function (as shown in line 8 of Algorithm 4). If the guess follows the Sudoku rules, it is assigned to the empty cell; otherwise, another guess is attempted. The process continues recursively (as seen in line 11) to fill empty cells until no further valid assignments can be made. In such cases, the function removes guesses starting from the last one (as depicted in line 14) until a branch leading to a solution is found. This search strategy, elaborated in Section 2 as *depth-first* search, involves exhaustively exploring each potential sequence of assignments, known as a branch, before progressing to the next one.

Algorithm 4 Main Sudoku Solving Function

```
 1: function SOLVE_SUDOKU(sudoku_array: np.ndarray) → bool
 2:     if find_empty(sudoku_array) = None then            ▷ This implies that the sudoku is solved
 3:         return True
 4:     else
 5:         SET empty_cell TO find_empty(sudoku_array)
 6:     end if
 7:     for guess IN range(1, 10): do
 8:         SET valid TO check_guess(sudoku_array, guess, empty_cell)
 9:         if valid: then
10:             SET sudoku_array[empty_cell[0], empty_cell[1]] TO guess              ▷ Recursion
11:             if solve_sudoku(sudoku_array): then
12:                 return True
13:             end if
14:             SET sudoku_array[empty_cell[0], empty_cell[1]] TO 0     ▷ Invalid assignment, undo
15:         end if
16:     end for
17:     return False
18: end function
```

Lastly, a function transforming the solved Sudoku into a string type so that it can be written in a text file is required. The `array_to_board` function was written by first flattening the input array board into 1x81 `NumPy` array. Then every element was converted into a string format. The separators `|, -, +`, and new line character `\n` are added to the string to preserve initial input text file format.

Profiling the code plays a crucial role in optimizing the run time of the previously designed algorithm. While the existing algorithm demonstrates relatively fast performance, there is always room for improvement, and recommendations for future versions of the algorithm are encouraged. For profiling, Python's built-in `cProfile` module was employed, offering more detailed profiling information compared to the `timeit` function. The main `solve_sudoku.py` was executed with a test board, and `cProfile` was passed as an argument to profile the entire algorithm. The profiling output was saved in a file with a `.prof` extension, which was then visually explored using the `SnakeViz` package. Below is a snippet of this visualization, showing three functions consuming the most amount of run time.



tottime	percall	cumtime	percall	filename:lineno(function)
0.113	3.928e-06	0.114	3.962e-06	backtracking.py:25(check_guess)
0.102	0.003776	0.1035	0.003833	~:0(<built-in method _imp.create_dynamic>)
0.05577	8.65e-06	0.05725	8.88e-06	backtracking.py:72(find_empty)

Figure 3: A screenshot of the first three functions of the `cProfile` output visualized by using the `SnakeViz` package.

The second line is related to loading modules dynamically during run time and is a consequence of profiling the main sudoku solver function which imports `backtracking` and `board` modules when executed. Since modular coding is preferred, this line is ignored. The first line indicates that the `check_guess` function in line 25 of the `backtracking` module takes 0.113 seconds in run time. This is expected due to the number of for loops used in the Algorithm 2 including nested double-for-loop. One improvement could be utilizing `numpy.any` function to make a search without the use of a for loop which is shown in the pseudo code in Algorithm 5. After this implementation, the algorithm is profiled one more time and the run time associated with `check_guess` was found to decrease to 0.09318 seconds which corresponds to around 82% decrease in run time. This change was integrated to the originally developed code for optimization. Another improvement can be suggested for the `find_empty` function which takes 0.05577 seconds currently looking at the last line of Figure 3. The same approach of using `numpy.any` was implemented to Algorithm 1, as shown in Algorithm 6. After profiling one more time, the run time for the `find_empty` function was reduced to 0.00921 seconds corresponding to around 17% decrease in run time. Although not a very significant improvement was achieved for this optimization, it was still integrated to the original code.

Algorithm 5 Optimization of `check_guess`

```
1: if np.any($sudoku[x\_box : x\_box + 3, y\_box : y\_box + 3]$ == guess) then    ▷ Check 3x3 box
2:     return False
3: end if
4: if np.any($sudoku[:, y]$ == guess) then    ▷ Check column
5:     return False
6: end if
7: if np.any($sudoku[x, :]$ == guess) then    ▷ Check row
8:     return False
9: end if
10: return True    ▷ If the guess is valid, then return True
```

Algorithm 6 Optimization of `find_empty`

```
1: indices $\leftarrow$ np.where(sudoku == 0)    ▷ Find indices of zero elements
2: if indices[0].size $>$ 0 then    ▷ Check if any zero element is found
3:     return indices[0][0], indices[1][0]
4: else
5:     return None
6: end if
```

4 Unit Tests, CI set up, and Validation

4.1 Unit Testing

Unit testing is an effective way to ensure that each function behaves as intended throughout the development process. Three test functions were developed for unit testing. The first one is for `check_board` function under the `board` module. The test file `test_check_board.py` is designed to verify that the `check_board` function effectively identifies invalid boards. It achieves this by testing the function using boards that contain repeating values in either a row, column, or 3x3 box. The second testing function focuses on the `check_guess` function within the `backtracking` module. Within the `test_check_guess.py` test file, the objective is to confirm that the `check_guess` function correctly recognizes invalid guesses. This is accomplished by assessing the validity of various guesses, both valid and invalid, in different positions of a test Sudoku board. The final test function `test_find_empty.py` was created to make sure `find_empty` function in the `backtracking` module correctly identifies empty cells of a Sudoku board. To achieve this, the function was tested using two 2-dimensional `NumPy` arrays as test boards—one with no zero elements and the other with a single zero element.

4.2 Continuous Integration

To automate the use of the test functions above with `pytest`, every time a version control commit is made locally, continuous integration (CI) was used. Additionally, `black` code formatter and `flake8` linter were used along with the 4.0.1 version of `git hooks` locally. The configuration file used for CI was named `.pre-commit-config.yaml` and placed in the root directory of the project.

4.3 Validation

The completed backtracking algorithm was tested on various Sudoku boards generated with the use of `ChatGPT 3.5` explained in Appendix A, located in the `test_boards` directory. While the algorithm successfully solved most of the test boards, it faced challenges with certain boards deliberately designed to be difficult for the backtracking approach. Although the backtracking algorithm, given enough time, guarantees a solution through a brute force method, for simplicity, a `count` variable was introduced into the main `solve_sudoku` function. This global variable increments with each recursive call of the function and is utilized in an if-statement to terminate the program when *count* reaches 10^6. A corresponding exit statement is then printed, recommending the use of an alternative solver for the provided input Sudoku board.

5 Packaging and Usability

The software project contains a Dockerfile which uses a `miniconda3` base image. This base image contains a minimal `Linux` operating system, `Conda`, and `Python` dependencies. A conda environment with specific dependencies defined in `environment.yml` file is built and activated when the container starts. The instructions on running the project inside this container is given in the `README.md` file. Finally, the documentation of the project is also available via the configuration `Doxyfile` inside the `docs` directory. To build documentation, `doxygen` should be run in the `docs` directory, after which `index.html` file will be available to view inside the `html` directory.

6 Summary

In conclusion, this coursework demonstrates a well-structured, modular software development of the backtracking algorithm for Sudoku solving. The prototyping and unit testing stages played a crucial role in identifying possible challenges during the development process such as data parsing and error trapping for invalid or incomplete inputs. The profiling and experimentation phases were important in refining code optimization and exploring various modular structures. Ultimately, the backtracking algorithm demonstrated success in the validation phase, exhibiting effective performance on the majority of the test boards and illustrating an efficient utilization of a brute-force approach for solving smaller-scale problems.

References

[1] Created in lucidchart. https://www.lucidchart.com.

[2] B. Felgenhauer and F. Jarvis. Enumerating possible sudoku grids. 07 2005.

A Appendix: Generative AI

Generative AI was used during the process of making Sudoku test boards for the validation section. The prompt submitted to ChatGPT 3.5 is given below:

- ”Can you generate 10 test sudoku boards as a text file in the following format?

  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

  Where 0’s represent empty cells.”

- ”Generate 5 more test sudoku boards in the same format but this time increase the difficulty of Sudokus, and make them harder especially for a solver using the backtracking algorithm.”

ChatGPT was also used for suggesting alternative wordings and grammar suggestions while writing the report. The following prompts were used during this process

- ”What is another word for *input word*?”
- ”Is this sentence grammatically correct? *input sentence*”
- ”Is this paragraph clear for a reader? *input paragraph*”
- ”How to rephrase this sentence to make it more clear? *input sentence*”

The outputs were partially adapted in a way that only alternative wordings were used and not the whole output while rephrasing the Introduction and Summary parts of the report.

Furthermore, the suggestions from the autocomplete feature of ‘GitHub Copilot‘ were utilized during the documentation of the unit testing functions, and code development of the project.