Research Computing

Coursework Submission

A REPORT PRESENTED

BY

STEVEN DILLMANN

Departments
Department of Applied Mathematics and Theoretical Physics
Department of Physics (Cavendish Laboratory)
Institute of Astronomy

Degree
MPhil in Data Intensive Science

Module
C1 Research Computing

Supervision
Dr James Fergusson

ST JOHN'S COLLEGE
UNIVERSITY OF CAMBRIDGE
17TH DECEMBER 2023

List of Figures

List of Tables

List of Listings

Contents

LaTeX Word Count: 3039 (excluding Listings, Bibliography and Appendix)

1 Introduction

History of Sudoku

The history of Sudoku dates back to Leonard Euler's Latin Squares in 1782 [1], later featured in French newspapers from 1892 until WWI [2]. They are $n \times n$ grids with n unique symbols in each row and column, without a subgrid constraint [3]. Today's Sudoku was designed by the American architect Howard Garns in 1979 [4]. Popularised in Japan in 1984 [5], they were first introduced as "Suuji wa dokushin ni kagiru" (the digits must be unique) and later shortened to "Sudoku" [6].

Sudoku Solver

In this report, we present the development and functionality of a Python-based Sudoku solver [7]. Our Sudoku solver accepts a 9×9 Sudoku puzzle in a text file format, where zeros represent unknown values and cells are separated by '|', '+', '-', i.e.:

```
$ cat input.txt
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
```

It outputs the solved Sudoku including the solving time to the terminal with the option to save the sudoku in a file. It is run-able from the command line with:

```
$ python src/solve_sudoku.py input.txt [solver] [save_file]
```

where `input.txt` is the input sudoku file, `[solver]` is the optional solver argument (`"bt"`: backtracking algorithm, `"cs"`: constraint satisfaction algorithm, `"lp"`: linear programming algorithm) and `[save_file]` is the optional argument to save the solved sudoku to a file (`True`, `False`). The default values are `"lp"` and `False`. For further details, refer to the `README.md` file in the project repository under C1-Coursework-Submission-(sd2022).

2 Prototyping, Development, Experimentation

2.1 Prototyping

The first important step in designing our Sudoku solver it to build a prototype. We pose several key questions during prototyping, see Table 1, and answer these during an iterative process that involves idea conception and small model experiments.

Table 1: Sudoku solver prototyping considerations.

Question	Answer
1. Problem Size?	9×9 Sudoku: no severe memory/parallelisation/architecture concerns
2. Input Data?	Short `.txt` file with not yet solved sudoku (text representation)
3. Output Data?	Short `.txt` file with solved sudoku or print to Terminal
4. Operations?	String operations, list (not `numpy` array) operations
5. Temporary Data?	Array (list of lists) with sudoku (grid representation)
6. Modules: I/O?	`<algorithm>_solver.py`: not yet solved/solved Sudoku grid
	`checkers.py`: True/False
	`converters.py`: text/grid representation and vice versa

The prototyping process is visualised in the multi-step flow chart in Figure 1 (see next page).

Input, Solver and Output

Starting from the not-yet-solved Sudoku input (Question 2) of the format described in Section 1, we need to output a solved Sudoku (Question 3) of the same format (from now on called "text representation"). The central block stands for the solving algorithm.

Converters

As the algorithms are much more straightforward to implement with array or list operations rather than string operations, we convert the text representation to an array, i.e. either a `numpy` array or a list of lists (from now on called "grid representation"), and vice versa (Question 5). These are placed after the input and before the output.

Algorithms: `numpy` array or list operations

We prototype simple backtracking (BT) algorithms [8] with both `numpy` array and list operations for comparison. Eventually, we opt for list operations due to the following reasons:

(i) The small size of a 9×9 Sudoku problem (Question 1) and the simplicity of required algorithm operations (Question 4)

(ii) The advantages of `numpy` would only become apparent for larger and more complex problems and may even be disadvantageous in terms of speed here (later confirmed during experimentation/profiling)

(iii) The readability of the algorithm is barely affected by the choice of list vs. `numpy` operations for this problem. We consider using lists to be slightly more readable (subjective matter).

Checkers

In the next step, we introduce checkers for the Sudoku grid representations. The one before the solver checks if the Sudoku is valid according to Sudoku rules and the one after checks if the Sudoku is solved (valid and no empty cells).

We decide to modularise (Question 6) our code in a way that the solver could easily be exchanged during development without altering any other parts of the code, i.e. the checkers and converters.

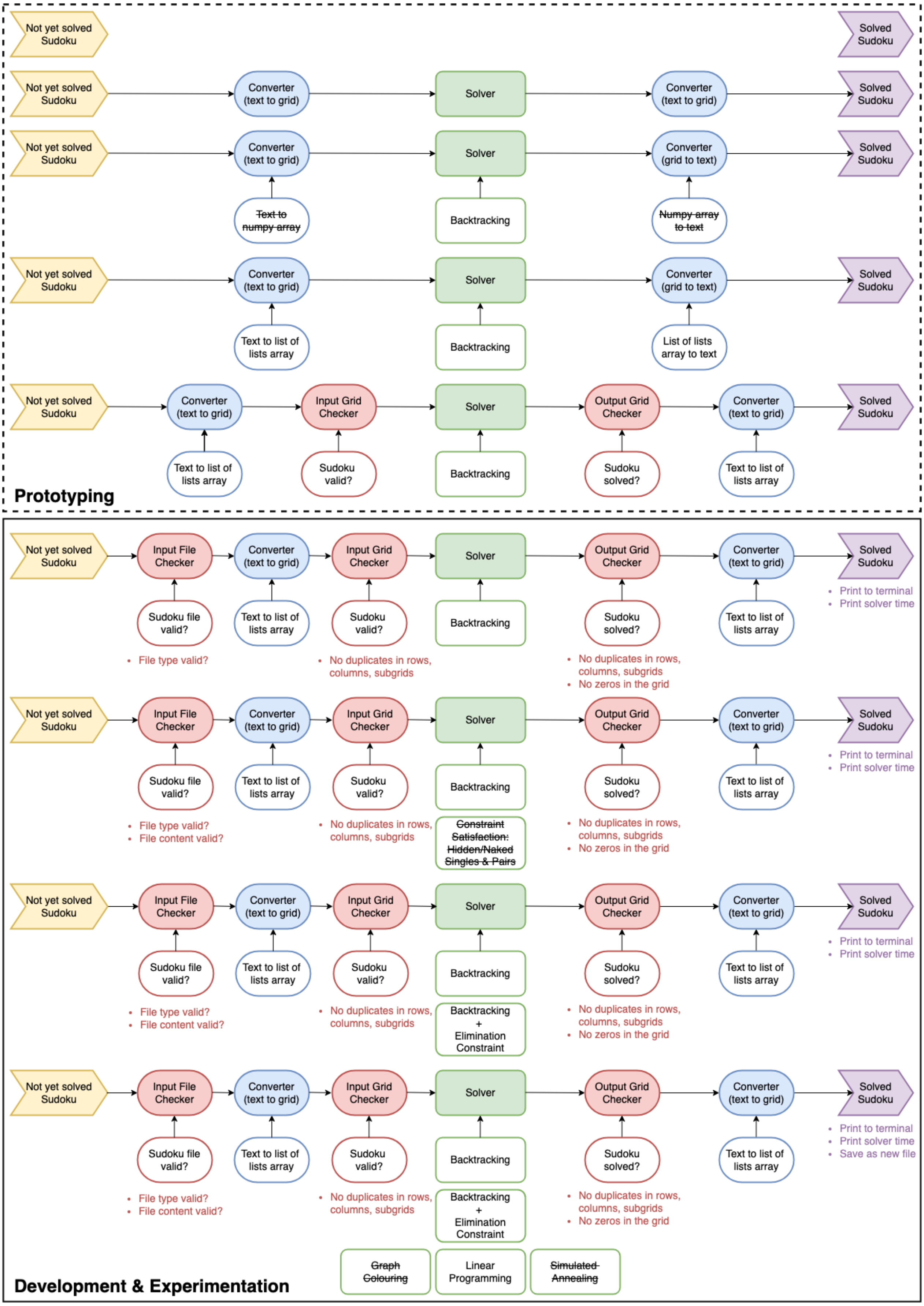

Figure 1: Sudoku solver prototyping, development and experimentation process including solver, converter and checker modules.

2.2 Development and Experimentation

Based on the initial Sudoku solver prototype, we start the development and experimentation process shown in Figure 1.

Object-oriented Programming (OOP) vs. Procedural Programming (PP)

Given the implementation of the Sudoku solver is relatively straightforward, we decide early on in the development process against the use of Classes and OOP. We consider the use of PP with a sequence of procedures executed step by step (as in the flowcharts in Figure 1) more straightforward and readable for this project.

Robustness and Output Feature Development

As a first development step, we make the code more robust by introducing an additional checker before the first converter to check for bad input files. This one checks if the file type and contents are valid. We also develop a feature to print the solving time and solved Sudoku solution to the terminal. This is later extended by adding a feature to save the solution to a file.

Algorithm Selection and Development

We experiment with a second solver using constraint satisfaction [9] with naked/hidden singles/-pairs [10, 11, 12, 13]. However, this requires redefining empty cells from zeros into lists containing digits 1 to 9. This would require making changes to checkers and converters, which makes the code less readable. Therefore, we abandon this approach and instead build an algorithm, which extends the current backtracking algorithm with an elimination constraint [14]. We manage to improve the performance of our solver with this algorithm on a number of experimental Sudoku puzzle inputs without negatively affecting the simplicity and readability of our code. Therefore, we introduce this constraint satisfaction (CS) approach as our second algorithm.

Having two fully functioning backtracking-based solvers, we run experiments on multiple Sudoku puzzles and find that for some very hard Sudokus with less than 20 clues, they can take several minutes to solve the Sudoku. This holds especially for Sudokus specifically designed to maximise the amount of backtracking required during the solving process, so called "anti-backtracking" puzzles, see Appendix A. This makes the exhaustive search in backtracking algorithms very expensive. Therefore, we decide to experiment with a number of different other algorithms including graph colouring (GC) [15], linear programming (LP) [16, 17] and simulated annealing (SA) [18]. Eventually, we decide to introduce LP as a third approach due to the following reasons:

(i) The LP approach is straightforward to implement with the `pulp` [19] library by setting up an LP problem, adding constraints and selecting an optimisation solver. The SA approach requires careful parameter tuning and the GC implementation is less concise.

(ii) The LP approach is able to solve the "anti-backtracking" puzzles within fractions of a second.

(iii) The LP approach shows very consistent performance across various experimental Sudokus, while e.g. the SA approach shows occasional convergence issues.

This initial algorithm selection is further refined in Section 3.

Final Product

The repository structure for the final product of the project is shown on the next page. Our final product has the following special abilities and features:

1. **Sudoku File and Puzzle Validation:** The programme informs the user if the file type or content is invalid, if the input puzzle is invalid according to Sudoku rules or is unsolveable.
2. **Sudoku File Fixing:** The programme detects bad input files, attempts to fix them and if successful saves the fixed sudoku to a file in the same directory as the input file with the following naming convention: `input_fixed.txt`.

3. **Sudoku Algorithm Selection:** The programme offers a range of solving algorithms: back-tracking, constraint satisfaction, and linear programming, empowering users to choose the most suitable algorithm for different difficulty levels.

In what follows, we provide information on the packages, modules and main function in the `src` directory. These are developed using best software development practices including branching for testing and feature development in the context of version control with `git`. For more details, such as helper functions, please refer to the code in the project repository.

```
/ sd2022
├── docs
│   └── Doxyfile
├── report
│   └── c1_sd2022_report.pdf
├── src
│   ├── processors
│   │   ├── __init__.py
│   │   ├── checkers.py
│   │   └── converters.py
│   ├── solvers
│   │   ├── __init__.py
│   │   ├── back_tracking_solver.py
│   │   ├── constraint_satisfaction_solver.py
│   │   └── linear_programming_solver.py
│   ├── profiling.py
│   └── solve_sudoku.py
├── tests
│   ├── test_processors
│   │   ├── __init__.py
│   │   ├── test_checkers.py
│   │   └── test_converters.py
│   ├── test_solvers
│   │   ├── __init__.py
│   │   ├── test_back_tracking_solver.py
│   │   ├── test_constraint_satisfaction_solver.py
│   │   └── test_linear_programming_solver.py
│   └── __init__.py
├── tests_resources
│   ├── easy_1.txt
│   ├── easy_1_solved.txt
│   ├── ...
│   ├── sudoku_valid_solved_rules_invalid.txt
│   └── sudoku_valid_unsolveable.txt
├── utils
│   ├── sudoku_demonstration.gif
│   └── sudoku_parser.py
├── .gitignore
├── .pre-commit-config.yaml
├── Dockerfile
├── environment.yml
├── Instructions.md
├── LICENSE
└── README.md
```

2.2.1 Sudoku Solver Script: `solve_sudoku.py`

The `solve_sudoku.py` script includes the overall Sudoku solver function, see Listing 1.

```python
def solve_sudoku(sudoku_file, solver="lp", save_file=False):
    """!@brief This is the main function to solve a sudoku.
    # ...
    """
    # Record the start time
    start_time = time.time()
    # Check if the input sudoku file is valid
    if not checkers.is_sudoku_file_valid(sudoku_file):
        return None
    # Convert the sudoku file to an array
    sudoku = converters.convert_sudoku_txt_to_arr(sudoku_file)
    # Check if the sudoku is valid
    if not checkers.is_sudoku_valid(sudoku):
        return None
    # Solve the sudoku with specified solver or default solver
    if solver == "bt":
        print("Use backtracking solver.")
        sudoku_solved = back_tracking_solver.solve_sudoku_bt(sudoku)
    elif solver == "cs":
        print("Use constraint satisfaction solver.")
        sudoku_solved = constraint_satisfaction_solver.solve_sudoku_cs(sudoku)
    elif solver == "lp":
        print("Use linear programming solver.")
        sudoku_solved = linear_programming_solver.solve_sudoku_lp(sudoku)
    else:
        print("Invalid solver specified.")
        print("Use default solver (constraint satisfaction solver).")
        sudoku_solved = constraint_satisfaction_solver.solve_sudoku_cs(sudoku)
    # Check if the sudoku was unsolveable
    if sudoku_solved is None:
        return None
    else:
        # Check if the sudoku is valid and solved
        if checkers.is_sudoku_solved(sudoku_solved):
            # Convert the solved Sudoku array back to text
            solution = converters.convert_sudoku_arr_to_txt(sudoku_solved)
            end_time = time.time()  # record the end time
            duration = end_time - start_time  # calculate the duration
            print(f"Solved in {duration:.5f} seconds with {solver} solver.\n")
            print("Sudoku solution:\n")
            print(solution, "\n")
            # Save the solved Sudoku string to a file if save_file is True
            if save_file:
                solved_file = os.path.splitext(sudoku_file)[0] + "_solved.txt"
                with open(solved_file, "w") as file:
                    file.write(solution)
                print(f"Solved sudoku saved to this file: {solved_file}")
        else:
            print("Sudoku solution is invalid or not solved. Returned 'None'.")
            return None
        return solution,

def main():
    """!@brief Parse command line arguments and call solve_sudoku function.
    # ...
    """
    if len(sys.argv) < 2 or len(sys.argv) > 4:
        print("Usage: python solve_sudoku.py input.txt [solver] [save_file]")
        return

    sudoku_file = sys.argv[1]
    solver = "lp"  # Default solver
    save_file = False  # Default save_file

    if len(sys.argv) >= 3:
        solver = sys.argv[2]

    if len(sys.argv) == 4:
        save_file = True if sys.argv[3].lower() == "true" else False

    solve_sudoku(sudoku_file, solver, save_file)

if __name__ == "__main__":
    main()
```

Listing 1: Main Sudoku solver script. It takes a Sudoku file as an input and returns a solved Sudoku with the option to save it to a file. The optional solver argument allows users to pick different algorithms.

2.2.2 Package: `processors`

The `processors` package includes (i) the `checkers.py` and (ii) the `converters.py` modules.

Module: `checkers.py`

This module includes tools to check if the `input.txt` file type and content is valid and if the Sudoku puzzle is valid according to Sudoku rules, see Listings 2, 3 and 4.

```python
def is_sudoku_file_valid(sudoku_file):
    """!@brief Check if the sudoku file is valid.
    # ...
    @see is_file_type_valid Function to check if the file type is valid
    @see is_file_content_valid Function to check if the file content is valid
    """
    # Check if the file type is valid
    if not is_file_type_valid(sudoku_file):
        print("WARNING SUMMARY: INVALID FILE TYPE!\n")
        return False
    # Check if the file content is valid
    if not is_file_content_valid(sudoku_file):
        print("WARNING SUMMARY: INVALID FILE CONTENT!\n")
        return False
    return True
```

Listing 2: Main function in `checkers.py`: `is_sudoku_file_valid`. This function checks if the Sudoku file is valid by checking if the file type and content are valid.

```python
def is_sudoku_valid(sudoku_arr):
    """!@brief Check if the sudoku puzzle is valid.
    # ...
    """
    # Check if the sudoku is a list of lists
    if not isinstance(sudoku_arr, list) or not all(
        isinstance(row, list) for row in sudoku_arr
    ):
        raise TypeError("Input Sudoku should be a list of lists.")
    # Store error messages in a list
    error_list = []
    # Check if a list of numbers has duplicates (excluding zeros)
    def check_duplicates(list_of_numbers):
        non_zeros_list = [num for num in list_of_numbers if num != 0]
        return len(non_zeros_list) != len(set(non_zeros_list))
    # Check rows for duplicates
    for row_idx, row in enumerate(sudoku_arr):
        if check_duplicates(row):
            error_list.append(f"Duplicate numbers in row {row_idx + 1}.\n")
    # Check columns for duplicates
    for col in range(9):
        column = [sudoku_arr[row][col] for row in range(9)]
        if check_duplicates(column):
            error_list.append(f"Duplicate numbers in column {col + 1}.\n")
    # Check subgrids for duplicates
    for i in range(0, 9, 3):
        for j in range(0, 9, 3):
            subgrid = [
                sudoku_arr[a][b]
                for a in range(i, i + 3)
                for b in range(j, j + 3)
            ]
            if check_duplicates(subgrid):
                error_list.append(
                    f"Duplicate numbers in subgrid starting"
                    f"at cell ({i + 1}, {j + 1}).\n"
                )
    # Print error messages if there are any and return False
    if error_list:
        for error_message in error_list:
            print("Error: ", error_message)
        print("WARNING SUMMARY: INVALID SUDOKU!\n")
        return False
    return True
```

.

Listing 3: Main function in `checkers.py`: `is_sudoku_valid`. This function checks if the Sudoku puzzle is valid by examining for duplicates in the rows, columns, or subgrids.

```python
def is_sudoku_solved(sudoku_arr):
    """!@brief Check if the sudoku puzzle is solved.
    # ....
    """
    # Check if the sudoku is a list of lists
    if not isinstance(sudoku_arr, list) or not all(
        isinstance(row, list) for row in sudoku_arr
    ):
        raise TypeError("Input Sudoku should be a list of lists.")
    # Store error messages in a list
    error_list = []
    # Check rows and columns for unique numbers and zeros
    for i in range(9):
        row_numbers = set(sudoku_arr[i])
        col_numbers = set(sudoku_arr[j][i] for j in range(9))

        if len(row_numbers) != 9:
            error_list.append(f"Duplicate/missing numbers in row {i+1}.\n")

        if len(col_numbers) != 9:
            error_list.append(f"Duplicate/missing numbers in column {i+1}.\n")
    # Check subgrids for unique numbers and zeros
    for a in range(0, 9, 3):
        for b in range(0, 9, 3):
            subgrid_numbers = set()
            for i in range(3):
                for j in range(3):
                    subgrid_numbers.add(sudoku_arr[a + i][b + j])
            if len(subgrid_numbers) != 9:
                error_list.append(
                    f"Duplicate/missing numbers in subgrid"
                    f"starting at cell ({a+1}, {b+1}).\n"
                )
    if error_list:
        for error_message in error_list:
            print("Error: ", error_message)
        print("WARNING SUMMARY: SUDOKU NOT SOLVED!\n")
        return False
    return True
```

Listing 4: Main function in `checkers.py`: `is_sudoku_solved`. This function checks if the Sudoku puzzle is solved by examining for duplicates and zeros in the rows, columns, or subgrids.

Module: `converters.py`

This module includes tools to convert Sudokus between the text and grid representations, see Listings 5 and 6.

```python
def convert_sudoku_txt_to_arr(sudoku_txt: str) -> list:
    """!@brief Converts a sudoku text file to a sudoku array (list of lists).
    # ...
    """
    # Check if the sudoku text file exists
    try:
        # Open the file in read mode
        with open(sudoku_txt, "r") as file:
            # Read sudoku text and split into lines
            sudoku_txt = file.read()
            sudoku_lines = sudoku_txt.split("\n")
            # Remove separator rows ('---+---+---')
            sudoku_lines = [line for line in sudoku_lines if "+" not in line]
            # Create sudoku array and return by iterating over each line
            sudoku_arr = []
            for line in sudoku_lines:
                line = line.replace("|", "")  # remove '|' separators
                row = [int(char) for char in line]
                sudoku_arr.append(row)
        return sudoku_arr
    except FileNotFoundError:
        raise FileNotFoundError("Sudoku text file does not exist.\n")
```

Listing 5: Main function in `converters.py`: `convert_sudoku_txt_to_arr`. This function converts a Sudoku in text representation into its grid representation.

```python
def convert_sudoku_arr_to_txt(sudoku_arr: list) -> str:
    """!@brief Converts a sudoku array (list of lists) to a sudoku text file.
    # ...
    """
    # Check if the sudoku array is empty
    if not sudoku_arr:
        raise ValueError("Sudoku array is empty.\n")
    # Check if the sudoku array is 9x9
    if len(sudoku_arr) != 9 or len(sudoku_arr[0]) != 9:
        raise ValueError("Sudoku array is not 9x9.\n")
    # Initialise the sudoku text
    sudoku_txt = ""
    # Create sudoku text by iterating over each row
    for row in sudoku_arr:
        line = "".join(str(num) for num in row)
        line = "|".join(textwrap.wrap(line, width=3))  # insert '|' separators
        sudoku_txt += line + "\n"
    # Insert separator rows ('---+---+---')
    sudoku_lines = sudoku_txt.split("\n")
    sudoku_lines.insert(3, "---+---+---")
    sudoku_lines.insert(7, "---+---+---")
    sudoku_txt = "\n".join(sudoku_lines[:-1])  # exclude the last empty line
    return sudoku_txt
```

Listing 6: Main function in `converters.py`: `convert_sudoku_arr_to_txt`. This function converts a Sudoku in grid representation into its text representation.

2.2.3 Package: `solvers`

The `solvers` package includes the algorithm solver modules: (i) `back_tracking_solver.py`, (ii) `constraint_satisfaction_solver.py` and (iii) `linear_programming_solver.py`.

Module: `back_tracking_solver.py`

This module includes tools to solve Sudoku using the BT algorithm, see Listing 7.

```python
def solve_sudoku_bt(sudoku):
    """!@brief This is the main function to solve a sudoku using the
    backtracking algorithm.
    # ...
    @see is_number_valid Function to check if a number is valid in sudoku
    """
    # Create copy of initial sudoku
    sudoku_solved = [row[:] for row in sudoku]
    # Run backtracking algorithm
    def solve():
        for row in range(9):
            for col in range(9):
                # Find empty cell
                if sudoku_solved[row][col] == 0:
                    # Try all numbers for this cell
                    for num in range(1, 10):
                        # Check if number is valid for this cell based on rules
                        if is_number_valid(sudoku_solved, row, col, num):
                            sudoku_solved[row][col] = num
                            # Solve the updated sudoku with recursion
                            if solve():
                                return True
                            # Backtrack if the number doesn't lead to solution
                            sudoku_solved[row][col] = 0
                    return False
        return True
    # Return the solved sudoku if the sudoku is valid
    if solve():
        return sudoku_solved
    else:
        # Print a warning if sudoku is invalid/unsolveable
        print("Unsolveable Sudoku. Returned 'None'.")
        return None
```

Listing 7: Main function in `back_tracking_solver.py`: `solve_sudoku_bt`. It takes a Sudoku array (list of lists) as an input and returns a solved Sudoku array (list of lists). The algorithm works by iterating through each cell in the Sudoku, trying all numbers and backtracking when necessary to ensure a valid solution.

Module: `constraint_satisfaction_solver.py`

This module includes tools to solve Sudoku using the CS algorithm, see Listing 8.

```
def solve_sudoku_cs(sudoku):
    """!@brief This is the main function to solve a sudoku using the
    backtracking algorithm with an elimination constraint.
    # ...
    @see get_valid_numbers Function to get all valid numbers for a cell
    """
    # Create copy of initial sudoku
    sudoku_solved = [row[:] for row in sudoku]
    # Run backtracking algorithm including elimination constraint
    def solve():
        for row in range(9):
            for col in range(9):
                # Find an empty cell
                if sudoku_solved[row][col] == 0:
                    # Get all valid numbers for this cell based on rules
                    valid_numbers = get_valid_numbers(sudoku_solved, row, col)
                    # Trigger backtracking if no valid numbers
                    if len(valid_numbers) == 0:
                        return False
                    # Try the valid numbers for this cell
                    for num in valid_numbers:
                        sudoku_solved[row][col] = num
                        # Solve the updated sudoku with recursion
                        if solve():
                            return True
                        # Backtrack if the number doesn't lead to solution
                        sudoku_solved[row][col] = 0
                    return False
        return True
    # Return the solved sudoku if the sudoku is valid
    if solve():
        return sudoku_solved
    else:
        # Print a warning if sudoku is invalid/unsolveable
        print("Unsolveable Sudoku. Returned 'None'.")
        return None
```

Listing 8: Main function in `constraint_satisfaction_solver.py`: `solve_sudoku_cs`. It takes a sudoku array (list of lists) as an input and returns a solved sudoku array (list of lists). The algorithm works by iterating through each cell in the sudoku, trying all valid numbers and backtracking when necessary to ensure a valid solution.

Module: `linear_programming_solver.py`

This module includes tools to solve Sudoku using the LP algorithm, see Listing 9.

```
def solve_sudoku_lp(sudoku):
    """!@brief This is the main function to solve a sudoku using the linear
    programming algorithm.
    # ...
    @see define_constraints Function to define the constraints for the
    linear programming problem
    @see extract_sudoku Function to extract the solved sudoku numbers
    @see is_sudoku_solved Function to check if the sudoku is solved
    """
    # Create a linear programming problem
    sudoku_lp = LpProblem("Sudoku_LP_Problem", LpMinimize)
    # Create all combinations of rows, columns and possible sudoku numbers
    cells = [
        (row, col, num)
        for row in range(9)
        for col in range(9)
        for num in range(1, 10)
    ]
    # Create a decision variable: number k exists in cell (i,j) or not (0 or 1)
    decision = LpVariable.dicts("Cell", cells, 0, 1, LpInteger)
    # Fix initial sudoku numbers
    for row in range(9):
        for col in range(9):
            if sudoku[row][col] != 0:
                sudoku_lp += decision[(row, col, sudoku[row][col])] == 1
    # Add cell, row, column and subgrid constraints
    define_constraints(sudoku_lp, decision)
    # Solve the linear programming problem
    sudoku_lp.solve(solver=GLPK(msg=0))
    # Extract the solved sudoku numbers from the decision variables
    sudoku_solved = extract_sudoku(decision)
    # Return the solved sudoku if the sudoku is valid
    if is_sudoku_solved(sudoku_solved):
        return sudoku_solved
    else:
        # Print a warning if sudoku is invalid/unsolveable
        print("Unsolveable Sudoku. Returned 'None'.")
        return None
```

Listing 9: Main function in `linear_programming_solver.py`: `solve_sudoku_lp`. It takes a Sudoku array (list of lists) as an input and returns a solved Sudoku array (list of lists). The algorithm works by creating a linear programming problem with a decision variable for each possible Sudoku number in each cell. The algorithm then adds constraints to the problem to ensure that each cell contains exactly one Sudoku number, each row contains unique Sudoku numbers, each column contains unique Sudoku numbers and each subgrid contains unique Sudoku numbers. The algorithm then solves the linear programming problem.

3 Algorithm Selection and Profiling

3.1 Standard Profiling

We use the `cProfile` [20] and `memory_profiler` tools [21] for time and memory profiling, see `profiling.py`. We profile all algorithms using an easy (45 clues), medium (35 clues), hard (25 clues) and an extreme (17 clues) Sudoku, see Appendix A. The results are summarised in Table 2.

Table 2: Time and memory profiling results for different algorithms and difficulties.

Difficulty	Backtracking		Constraint Satisfaction		Linear Programming	
	Time [ms]	Memory [kiB]	Time [ms]	Memory [kiB]	Time [ms]	Memory [kiB]
Easy	0.97	68	0.88	72	36.40	184
Medium	10.73	76	7.14	80	39.78	220
Hard	545.63	108	344.07	120	37.91	312
Extreme	$328.06 \cdot 10^3$	120	$193.45 \cdot 10^3$	136	38.54	272

The LP solver is generally more memory-intensive and slower for easy and medium puzzles. However, it's performance exceeds significantly for the hard and extreme puzzle with similar solving times across all puzzles.

3.2 Algorithm Selection

During our initial algorithm selection, we have excluded algorithms such as GA and SA. We now run a more detailed performance-based algorithm selection to choose our default algorithm. We download 10,000 Sudoku puzzles each for difficulties easy, medium and hard from [22]. The solving time evolution across the first 200 puzzles in each category is shown in Figure 2.

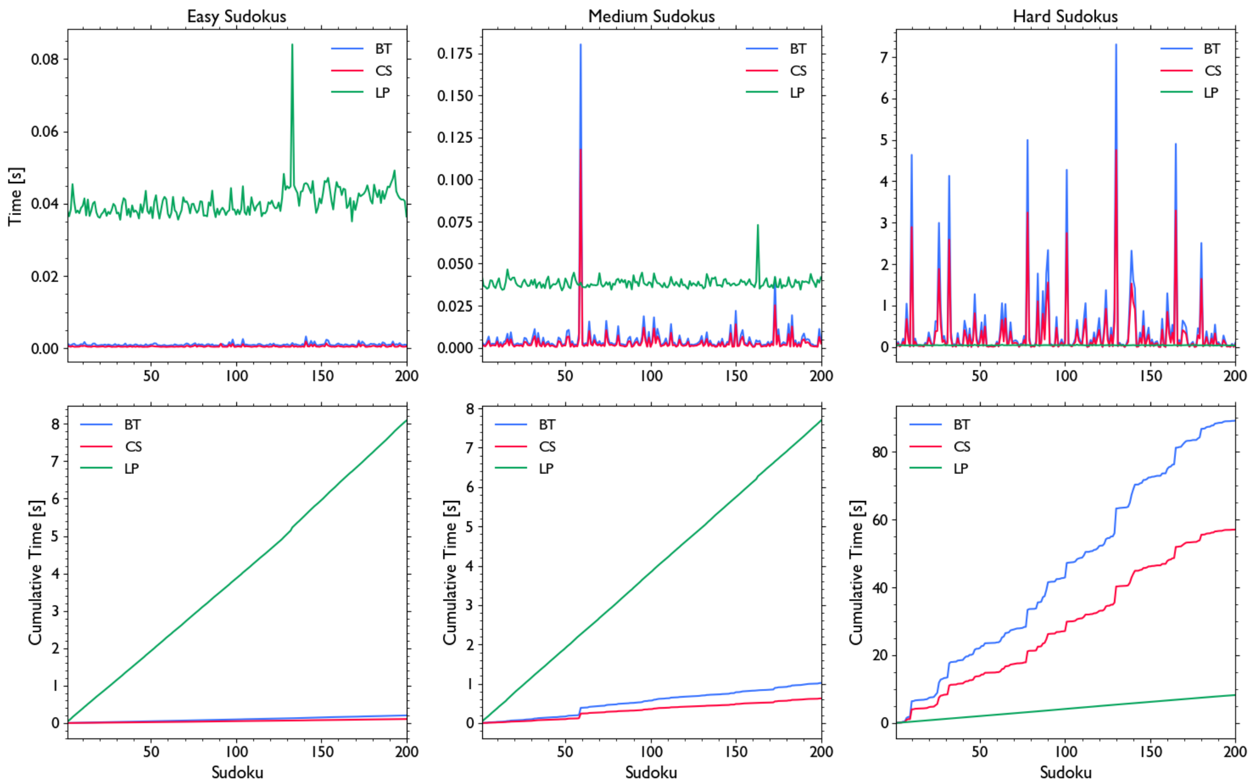

Figure 2: Sudoku solving times of different algorithms across 200 easy, medium and hard puzzles.

The CS algorithm beats the BT algorithm across all difficulty levels, while the LP algorithm beats CS and BT solvers for hard puzzles only. The performance across all 10,000 puzzles in each category is visualised in Figure 3 in form of box-and-whisker plots.

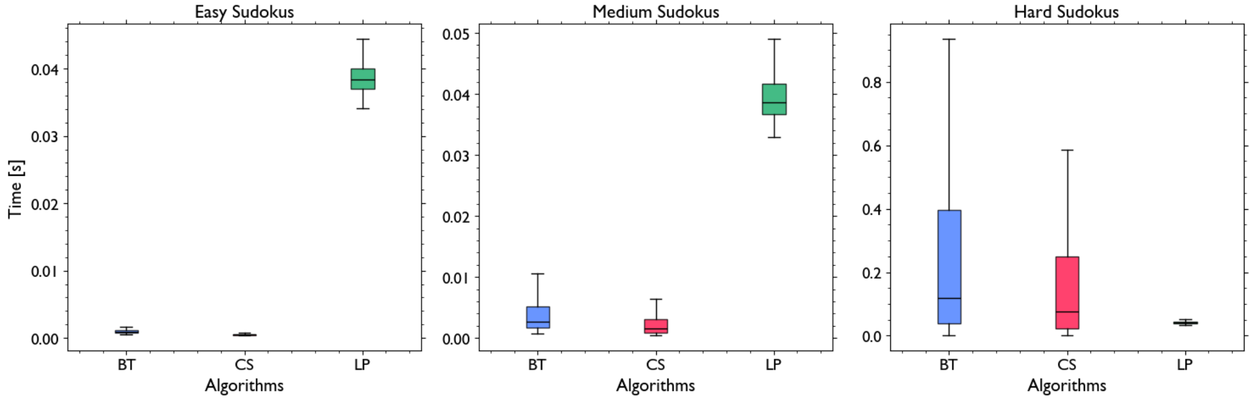

Figure 3: Box-and whisker plots for the solving times of different algorithms across 10,000 easy, medium and hard puzzles (outliers not displayed).

We report the average solving times across all puzzles for each category and algorithm in Table 3.

Table 3: Average solving times of different algorithms across 10,000 easy, medium and hard puzzles.

Algorithm	**Easy** [ms]	**Medium** [ms]	**Hard** [ms]	**Combined** [ms]
Backtracking	1.01	4.92	513.21	173.05
Constraint Satisfaction	0.53	2.99	327.06	110.20
Linear Programming	38.94	39.57	42.01	40.17

Based on our performance study, we present the recommended Sudoku solver for different puzzle difficulties in Table 4.

Table 4: Recommended Sudoku solvers based on the difficulty level.

Difficulty	**Number of Clues**	**Recommended Solver**
Easy	∼45	Constraint Satisfaction
Medium	∼35	Constraint Satisfaction
Hard	∼25	Linear Programming
Extreme	<20	Linear Programming
Default Solver: Linear Programming		

We select the LP solver as our default algorithm due to the following reasons:

(i) Even though the LP solver performs worse than the BT and CS solvers for easy and medium Sudokus, the solving time is still very short (on the order of ms).

(ii) The LP algorithm solves hard Sudokus significantly faster than the BT and CS solvers.

(iii) The LP solver's performance is highly consistent (∼ 40 ms on average) for all difficulties.

The data used in this study is stored in Google-Drive-C1-Submission-Data.

4 Validation, Unit Testing and CI Set-up

4.1 Validation

For the code validation, we implement type checking, error trapping and debugging as validation tools. We do not include Input/Output (I/O) tools such as parameter files with the iniparser package [23], because the Sudoku solver is a relatively simple script with only two optional arguments [solver] and [save_file]. We consider passing arguments via the command line as more user-friendly, which outweighs potential benefits from a parameter/configuration file.

4.1.1 Type Checking

We implement type checking for the converter.py module functions to make sure they receive the correct input type and convert it to the correct output type. For this purpose, we use the typing package, which gives hints to the linting tool flake8 [24], which is configured in the .pre-commit-config.yaml file. It is set up by adding ':' and '->' in this way:

```
def convert_sudoku_arr_to_txt(sudoku_arr: list) -> str:
    # Code to convert a sudoku array (list of lists) to a sudoku
    return sudoku_txt
```

4.1.2 Error Trapping

For consistency and readability, most error trapping is done with if and print statements. In a few cases, errors are better handled with try-except blocks or raise statements.

Validation Checkers

In the checker.py module, we use a number of ways to check if the Sudoku file and its grid are valid. For example, one of the file content checks is to check if the file has the correct number of lines. If not, the user is presented with the expected input format:

```
def is_file_content_valid(sudoku_file):
    #...
    # Check the number of lines
    if len(fixed_sudoku_lines) != 11:
        print("Error: Incorrect number of lines in the file.\n")
        print("Make sure the input file matches the following format:")
        print(
    """
        xxx|xxx|xxx
        xxx|xxx|xxx
        xxx|xxx|xxx
        ---+---+---
        xxx|xxx|xxx
        xxx|xxx|xxx
        xxx|xxx|xxx
        ---+---+---
        xxx|xxx|xxx
        xxx|xxx|xxx
        xxx|xxx|xxx
        """
        )
        return False
    #...
    return True
```

Exception Handling

We introduce `try-except` blocks in the `checker.py` module, e.g. when we try to convert an unexpected file format (`.docx`, `.md`, `.pdf` etc.) into a fixed Sudoku file with `.txt` extension:

```
def is_file_type_valid(sudoku_file):
    # ...
    # Check if the extension is .txt
    if file_extension == ".txt":
        return True
    else:
        # Try to create a new file with fixed .txt extension
        print("Attempting to create a new file with the '.txt' extension...\n")
        fixed_sudoku_file = file_name + "_fixed.txt"
        try:
            shutil.copyfile(sudoku_file, fixed_sudoku_file)
            print(
                "Fixed: Created a new file with the '.txt' extension:"
                f"'{fixed_sudoku_file}'.\n"
            )
            print("Please use this new file as input.\n")
        # Print an error message if creating the new file fails
        except Exception as e:
            print(f"Error: Failed due to the following error: {e}.\n")
            print(
                "Make sure the input has the correct file extension: '.txt'\n"
            )
        return False
```

Raising Exceptions

Exceptions are raised with `raise` statements, e.g. to check if the input to the `is_sudoku_solved` function is a list of lists rather than a `numpy` array or any other type:

```
def is_sudoku_solved(sudoku_arr):
    # ...
    # Check if the sudoku is a list of lists
    if not isinstance(sudoku_arr, list) or not all(
        isinstance(row, list) for row in sudoku_arr
    ):
        raise TypeError("Input Sudoku should be a list of lists.")
    # ...
    return True
```

4.1.3 Debugging

Using the Python Debugger (PDB) has proven to be a very helpful code validation tool throughout the development process. We run the debugger from the command line with:

```
$ python -m pdb src/solve_sudoku.py
```

Debugging commands like `n` (next line), `s` (step into function), `c` (continue execution), `b` (set breakpoint) and `l` (list code) help step through the code and diagnose issues. Using PDB, we resolve a number of issues, e.g. it identifies a `SyntaxError` in one of our `converters` module functions after adding type checking tools:

```
  File "/Users/steven/Desktop/C1/sd2022/src/processors/converters.py", line 18
    def convert_sudoku_txt_to_arr(sudoku_txt: str): -> list:
                                                  ^

SyntaxError: invalid syntax
```

4.2 Unit Testing

While using PDB is helpful, most of our debugging is actually done via unit testing with `pytest`.

4.2.1 Testing Setup

The testing directory `tests` mirrors our `src` directory and is made a package by adding an `__init.py__` file. The `tests_resources` include a number of test Sudoku files.

```
/ sd2022
├── ...
├── src
│   ├── processors
│   │   ├── __init__.py
│   │   ├── checkers.py
│   │   └── converters.py
│   ├── solvers
│   │   ├── __init__.py
│   │   ├── back_tracking_solver.py
│   │   ├── constraint_satisfaction_solver.py
│   │   └── linear_programming_solver.py
│   ├── profiling.py
│   └── solve_sudoku.py
├── tests
│   ├── test_processors
│   │   ├── __init__.py
│   │   ├── test_checkers.py
│   │   └── test_converters.py
│   ├── test_solvers
│   │   ├── __init__.py
│   │   ├── test_back_tracking_solver.py
│   │   ├── test_constraint_satisfaction_solver.py
│   │   └── test_linear_programming_solver.py
│   └── __init__.py
├── tests_resources
└── ...
```

4.2.2 Testing Led Development

We create tests during development for the main functions in the `processors` and `solvers` packages, often before we create the functions itself to check whether they work as expected. We automate the tests with a pre-commit hook in the `.pre-commit-config.yaml` file, which requires all tests passing before a commit. For example, we test if the `is_sudoku_file_valid` function recognises a bad character '`#`' in the input file and thus returns `False`:

```
$ cat tests_resources/sudoku_invalid_bad_numbered_lines.txt
000|007|000
000|009|504
000|050|169
---+---+---
080|000|305
075|0#0|290
406|000|080
---+---+---
762|080|000
103|900|000
000|600|000
```

```
sudoku_files_valid = "tests_resources/sudoku_invalid_bad_numbered_lines.txt"
expected_valid = False

def test_is_sudoku_file_valid(sudoku_files_valid, expected_valid):
    # ...
    assert checkers.is_sudoku_file_valid(sudoku_files_valid) == expected_valid
```

We identify numerous code issues by running 63 `pytest` tests in total as summarised in Table 5.

Table 5: Summary of unit test results.

Test Module	Test Function	Passed	Cumulative
tests/test_processors/			
test_checkers.py	test_is_sudoku_file_valid	27/27	27/63 (43%)
	test_is_sudoku_valid	3/3	30/63 (48%)
	test_is_sudoku_solved	9/9	39/63 (61%)
test_converters.py	test_convert_sudoku_txt_to_arr	3/3	42/63 (67%)
	test_convert_sudoku_arr_to_txt	3/3	45/63 (71%)
tests/test_solvers/			
test_back_tracking_solver.py	test_solve_sudoku_bt	6/6	51/63 (81%)
test_constraint_satisfaction_solver.py	test_solve_sudoku_cs	6/6	57/63 (90%)
test_linear_programming_solver.py	test_solve_sudoku_lp	6/6	63/63 (100%)

4.3 Continuous Integration

Ideally, we should use continuous integration (CI) functionalities within GitLab. For demonstration purposes, we instead present the set-up of the `.pre-commit-config.yaml` file. This file configures the pre-commit hooks, which run automatically before a commit in `git`. To enforce PEP8 style guidelines, we use `flake8` [24] and `black` [25] within our hooks.

'pre-commit-hooks' Repository Hooks

- **trailing-whitespace:** Removes trailing whitespace in files.
- **end-of-file-fixer:** Ensures consistent line endings.
- **mixed-line-ending:** Checks for mixed line endings.
- **check-yaml:** Checks YAML files for syntax errors.
- **debug-statements:** Warns about potential debug statements in the code.

'psf/black' Repository Hook

- **black:** Runs the `black` code formatter with an added argument setting the line length to 79 characters to enforce PEP8 style.

'PyCQA/flake8' Repository Hook

- **flake8:** Lints Python files using `flake8` with an added dependency for type checking.

Local Testing Hook

- **testing:** Runs `pytest` on the test files in the `tests/` directory before every commit.

5 Packaging, Usability and Documentation

5.1 Packaging

In what follows, we describe the installation and deployment of the Sudoku solver.

5.1.1 Requirements

The basic requirements to install and deploy the Sudoku solver are as follows:

- Python 3.9 or higher installed on your system.
- Conda installed (for managing the Python environment).
- Docker (for containerised deployment).

The rest of the requirements are specified in the `environment.yml` file.

5.1.2 Installation

To clone the repository to your local machine, use the following command:

```
$ git clone https://gitlab.developers.cam.ac.uk/phy/data-intensive-science-
    mphil/c1_assessment/sd2022
```

Alternatively, simply download the repository from C1-Coursework-Submission-(sd2022). Next, navigate to the project directory on your local machine with:

```
$ cd /full/path/to/sd2022
```

and replace `/full/path/to/` with the full path to the repository on your local machine.

5.1.3 Deployment

We can now perform a traditional or containerised deployment.

Traditional Deployment (Local Setup)

We set up the required environment on our local machine with:

```
$ conda env create -f environment.yml
$ conda activate sd2022_c1_env
```

Containerised Deployment (Docker)

Make sure to run Docker, which can be installed from Docker Download. You can build a Docker image with the following command:

```
$ docker build -t [image] .
```

and replace `[image]` with the name of the image you want to build.

5.2 Usability

We can use the Sudoku solver via the command line or using Docker.

5.2.1 Traditional Usage (Command Line)

We can now run the Sudoku solver from the command line with the following command:

```
$ python src/solve_sudoku.py input.txt [solver] [save_file]
```

where `input.txt` is the path to the input Sudoku file, `[solver]` is the optional solver argument and `[save_file]` is the optional argument to save the solved Sudoku to a file.

The `[solver]` argument specifies which Sudoku solver algorithm should be used:

1. `"bt"`: backtracking algorithm
2. `"cs"`: constraint satisfaction algorithm
3. `"lp"`: linear programming algorithm

If no or an invalid `[solver]` argument is specified, the script will use the linear programming algorithm by default.

The `[save_file]` specifies whether the solved Sudoku should be saved to a file in the same directory as the input file with the naming convention `input_solved.txt`:

1. `True`: save the solved Sudoku to a file
2. `False`: do not save the solved Sudoku to a file

If no or an invalid `[save_file]` argument is specified, the script will not save the solved Sudoku to a file by default.

The default output is the solved Sudoku and the solving time printed to the terminal. For further details and a usage demonstration, refer to the `README.md` file of the project repository.

5.2.2 Containerised Usage (Docker)

If you want to run `solve_sudoku.py` on an input file within the Docker container `input.txt`, first create and run a container with the following command:

```
$ docker run -it --name=[container] [image]
```

Then simply run the script with the following command:

```
$ python src/solve_sudoku.py tests_resources/input.txt [solver] [save_file]
```

If you want to run the script on an input file outside the Docker container `input_2.txt`, you need to link the file on your local machine with the container using the following command:

```
$  docker run -it --name=[container] -v /path/to/input_2.txt:/sd2022/
   tests_resources/input_2.txt [image]
```

Next, simply run the script with the following command again:

```
$ python src/solve_sudoku.py tests_resources/input_2.txt [solver] [save_file]
```

In all commands, replace `[container]` with the container name and `[image]` with the image name you created.

5.3 Documentation

Code documentation is generated with Doxygen [26] and the `Doxyfile` in the documentation folder `docs`. Make sure you have Doxygen installed on your system.

Navigate to the `docs` folder and build the documentation with:

```
$ doxygen
```

Navigate to the generated `latex` folder and build a PDF document with:

```
$ make
```

You can now view the generated documentation in PDF format under `refman.pdf`.

6 Summary

Conclusion

In this project, we design a Python-based Sudoku solver according to best software development practices. Starting from a simple prototype, we develop the final solver to include validity checks, type converters and a feature to choose from three different solver algorithms. We validate our code with error trapping and unit testing. Finally, we explain the solver's packaging and usability.

Future Work

There are a number of potential enhancement areas for the Sudoku solver. Below, we present three improvement ideas:

1. **Optimised Cell Selection in Backtracking:** Currently, both BT and CS algorithms start on the top left of the board, continue with the next cells and backtrack if necessary. Special "anti-backtracking" Sudoku puzzles can exploit this and make them slow. To mitigate this, the solver could start with the cells having the least amount of possible values first and continue accordingly.
2. **Parallelisation**: Currently, both BT and CS algorithms solve Sudokus as one problem. Parallelisation could help break down the problem into smaller simultaneous tasks to be solved leading to potential performance improvements.
3. **Dynamic Solver Selection based on Sudoku Complexity:** Currently, the default solver is the LP algorithm. However, the CS algorithm performs better for easy and medium Sudokus. The programme could be improved by checking the number of clues in the Sudoku first to then select the appropriate algorithm as a default.

Bibliography

[1] Leonhard Euler. Recherches sur un nouvelle espéce de quarrés magiques. Verhandelingen uitgegeven door het zeeuwsch Genootschap der Wetenschappen te Vlissingen, pages 85–239, 1782.

[2] Christian Boyer. Sudoku's french ancestors. The Mathematical Intelligencer, 29(1):37–44, 2007.

[3] Jyotirmoy Sarkar and Bikas K Sinha. Sudoku squares as experimental designs. Resonance, 20:788–802, 2015.

[4] Jaysonne A. Pacurib, Glaiza Mae M. Seno, and John Paul T. Yusiong. Solving sudoku puzzles using improved artificial bee colony algorithm. In 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC), pages 885–888, 2009.

[5] J. Jilg and J. Carter. Sudoku evolution. In 2009 International IEEE Consumer Electronics Society's Games Innovations Conference, pages 173–185, 2009.

[6] Sunanda Jana, Arnab Kumar Maji, and Rajat Kumar Pal. A novel sudoku solving technique using column based permutation. In 2015 International Symposium on Advanced Computing and Communication (ISACC), pages 71–77, 2015.

[7] Guido Van Rossum and Fred L. Drake. Python 3 Reference Manual. CreateSpace, Scotts Valley, CA, 2009.

[8] Dhanya Job and Varghese Paul. Recursive backtracking for solving 9* 9 sudoku puzzle. Bonfring International Journal of Data Mining, 6(1):7–9, 2016.

[9] Helmut Simonis. Sudoku as a constraint problem. In CP Workshop on modeling and reformulating Constraint Satisfaction Problems, volume 12, pages 13–27. Citeseer, 2005.

[10] Obvious singles - sudoku rules. https://sudoku.com/sudoku-rules/obvious-singles/. Accessed: December 11, 2023.

[11] Hidden singles - sudoku rules. https://sudoku.com/sudoku-rules/hidden-singles/. Accessed: December 11, 2023.

[12] Obvious pairs - sudoku rules. https://sudoku.com/sudoku-rules/obvious-pairs/. Accessed: December 11, 2023.

[13] Hidden pairs - sudoku rules. https://sudoku.com/sudoku-rules/hidden-pairs/. Accessed: December 11, 2023.

[14] Notes in sudoku - sudoku rules. https://sudoku.com/sudoku-rules/notes-in-sudoku/. Accessed: December 11, 2023.

[15] Tommy R Jensen and Bjarne Toft. Graph coloring problems. John Wiley & Sons, 2011.

[16] Robert J Vanderbei et al. Linear programming. Springer, 2020.

[17] Lakshmi Ajay. Solve sudoku using linear programming. https://towardsdatascience.com/solve-sudoku-using-linear-programming-python-pulp-b41b29f479f3, 2023. Accessed: December 10, 2023.

[18] Peter JM Van Laarhoven, Emile HL Aarts, Peter JM van Laarhoven, and Emile HL Aarts. Simulated annealing. Springer, 1987.

[19] Stuart Mitchell et al. PULP: Python Linear Programming. https://pypi.org/project/PuLP/, 2009. Accessed: December 12, 2023.

[20] Python Software Foundation. cProfile. https://docs.python.org/3/library/profile.html, 2023. Accessed: December 13, 2023.

[21] Python Software Foundation. Fabian pedregosa and philippe gervais. https://pypi.org/project/memory-profiler/, 2022. Accessed: December 13, 2023.

[22] Free printable 9x9 sudoku puzzles. https://www.phptutorial.info/scripts/sudoku/, 2023. Accessed: November 27, 2023.

[23] Nicolas Devillard. iniparser. https://github.com/ndevilla/iniparser, 2022. Accessed: December 10, 2023.

[24] Ian Stapleton Cordasco. Flake8: Your tool for style guide enforcement. https://flake8.pycqa.org/, 2016. Accessed: December 11, 2023.

[25] Łukasz Langa. Black: The uncompromising code formatter. https://github.com/psf/black, 2023. Accessed: December 11, 2023.

[26] Dimitri van Heesch. Doxygen. https://www.doxygen.nl/, 2023. Accessed: December 17, 2023.

A Sudoku Puzzle Examples

The below Sudoku is an example of an easy puzzle:

```
$ tests_resources/extreme_1.txt
001|700|509
573|024|106
800|501|002
---+---+---
700|295|018
009|400|305
652|800|007
---+---+---
465|080|071
000|159|004
908|007|053
```

The below Sudoku is an example of a medium puzzle:

```
$ tests_resources/extreme_1.txt
290|500|007
700|000|400
004|738|012
---+---+---
902|003|064
800|050|070
500|067|200
---+---+---
309|004|005
000|080|700
087|005|109
```

The below Sudoku is an example of a hard puzzle:

```
$ tests_resources/extreme_1.txt
000|075|400
000|000|008
080|190|000
---+---+---
300|001|060
000|000|034
000|068|170
---+---+---
204|000|603
900|000|020
530|200|000
```

The below Sudoku is an example of an extreme “anti-backtracking” puzzle:

```
$ tests_resources/extreme_1.txt
000|000|010
400|000|000
020|000|000
---+---+---
000|050|407
008|000|300
001|090|000
---+---+---
300|400|200
050|100|000
000|806|000
```