Coursework Report: Sudoku Solver Implementation Using Simulated Annealing

C. Grivot

December 2023

1 Introduction

This coursework undertaking centres on creating a Sudoku solver through the application of simulated annealing. The choice to utilise simulated annealing as the algorithmic solution stems from previous exposure to the method during the undergraduate thesis. The following sections delve into the decision-making process, prototyping, development, experimentation, profiling, and packaging facets of the project. Simulated annealing, a probabilistic optimisation algorithm, involves mimicking the annealing process in metallurgy to find near-optimal solutions by navigating a solution space and gradually reducing the probability of accepting sub-optimal solutions over time.

2 Selection of Solution Algorithm and Prototyping

2.1 Solution Algorithm

The choice of simulated annealing as the solution algorithm was influenced by an extensive review of various algorithms documented on Wikipedia. Given familiarity with simulated annealing from the bachelor's thesis, its selection was deemed advantageous.

2.2 Prototyping

The prototyping phase commenced with the establishment of the project structure. A meticulous organisation was achieved, featuring clear separation of concerns, main scripts in the `src/` directory, modular design, a dedicated test directory (`test/`), and comprehensive documentation in the `docs/` directory.

The modular design includes components such as `solve_sudoku.py`, `sudoku_simulated_annealing.py`, and `sudoku_utils.py`. Version control using `git`, pre-commit hooks, Docker containerisation (`Dockerfile`), and Conda environment management (`environment.yml`) were integral to the development process.

The goal was to prioritise clarity, modularity, and maintainability, enhancing the overall development experience. A flowchart was employed to outline the functions necessary for the simulated annealing approach. The main challenge of the prototyping was figuring out the implementation of the constraints for the binary quadratic model which is our objective function for the simulated annealing solver. The constraints themselves are quite straightforward, no duplicated digits in a row, column or box. However, the pseudocode and the algorithm were more challenging to figure out. This will be discussed in the following section.

3 Development, Experimentation, and Profiling

The initial development phase involved coding in a monolithic manner to comprehend the constraints of the binary quadratic matrix. The first challenge was to build the objective function. In the development phase, it was first thought of using a loop that would assert/ check the inequality for each cell of the matrix using the indexes i and j. But then the difficulty was then how do we put this in a binary quadratic form so that it works with the simulated annealing sampler. The Binary quadratic models (BQMs) are problems of the form: $E(\mathbf{v}) = \sum_{\mathbf{i}} \mathbf{a_i v_i} + \sum_{\mathbf{i<j}} \mathbf{b_{i,j} v_i v_j} + \mathbf{c} \qquad \mathbf{v_i} \in \{\mathbf{0}, \mathbf{1}\}$

Doing more research, one could find the dimod.generators.combinations function which generates a BQM that is minimised when k of n variables are selected, where the BQM is minimised for each of the k-combinations of its variables. The 2 parameters of this function are

- n – Variable labels. If `n` is an integer, variables are labelled [0, n).
- k – The number of selected variables (variables assigned value 1) that minimises the generated BQM, resulting in an energy of 0.

The energy for the BQM is then given by $(\sum_i x_i - k)^2$. An example of it's usage is: bqm = dimod.generators.combinations(['a', 'b', 'c'], 2) bqm.energy('a': 1, 'b': 0, 'c': 1) 0.0 bqm.energy('a': 1, 'b': 1, 'c': 1) 1.0 The plan is then to generate the constraints using this function each time and then add them together. In our Sudoku solver, the variable labels are the cell in the Sudoku grid and the number of selected variables is 1. For example, we have to make sure that each row has unique digits. The constraint creates combinations of variables representing cells in each row and adds them to the BQM. The constraints must collectively enforce the rules of Sudoku:

- Each cell must contain exactly one digit.
- Each row must contain unique digits.
- Each column must contain unique digits.
- Each 3x3 box must contain unique digits.

- Known values provided in the input matrix are fixed and must be included in the solution.

Subsequently, the simulated annealing sampler from DWave was utilised, with comparisons against the simulated annealing sampler from Neal and Dimod. Attempts were made to package `sudoku_utils.py` and `sudoku_simulated_annealing.py` into a package, encountering challenges with unit test accessibility, hence the idea was discarded.

One more step that was done during the development is too add 2 specific decorators. Here's an overview of why it was decided to implement these decorators:

- Execution Time Profiling: The 'timing decorator' has been introduced to measure the execution time of critical functions such as 'get bqm' and 'solve simulated annealing'. This profiling capability is invaluable for performance analysis and optimisation efforts. By applying this decorator, we gained a clear understanding of the time each function takes to execute.

- Function Call Logging: The 'logging decorator' facilitates detailed logging of function calls, capturing information about the function name, arguments, and return values. This feature aided in debugging and provides a comprehensive record of the program's execution flow. The inclusion of this decorator was undertaken with the specific purpose of examining the cause behind the ineffective parallelisation implementation in the simulated annealing solver. The decorator's capability to trace function calls can be pivotal in the identification and resolution of issues.

- Code Profiling and Insight: Combining timing and logging decorators offers a holistic view of the code's performance and function call sequence. This information is valuable for profiling and understanding the behaviour of the Sudoku solver. It allows us to pinpoint performance bottlenecks and unexpected behaviours during program execution.

- Enhanced Modularity and Maintainability: Decorators contribute to the separation of concerns by isolating additional functionality, such as timing and logging, from the core logic of the functions. This modular approach enhances code maintainability, as each decorator focuses on a specific aspect of behaviour without introducing complexity to the primary implementation.

The decision to incorporate these specific decorators reflects a commitment to improving the overall quality of the Sudoku solver module. These enhancements not only provide valuable insights into the run-time behaviour of the code but also contribute to a more modular and maintainable codebase.

3.1 Profiling

First the algorithmic complexity of the code was examined:

- Reading Input (Linear Complexity): Reading the input file and parsing it into a matrix is a linear operation, as each cell of the input file is processed once.
- Building the Binary Quadratic Model (Polynomial Complexity): The 'get bqm' function has a nested loop structure that iterates over rows, columns, and digits. This results in a polynomial time complexity, it is quadratic in terms of the size of the Sudoku grid (n^2), where n is the number of rows or columns in the Sudoku.
- Simulated Annealing Solver (Depends on the Solver, Typically Polynomial or Log-Linear): The 'solve simulated annealing' function uses the simulated annealing solver from D-Wave. The time complexity of simulated annealing depends on the specific implementation and parameters used, which we do not have access to. It can be polynomial or log-linear, depending on factors like the number of iterations, temperature schedules, etc.

Profiling using CProfile on November 25th highlighted a total of 313,763 function calls, with a cumulative time of 124.130 seconds. The main focus for optimization was the `solve_simulated_annealing` function because it is the most time-consuming function, which took 123.779 seconds. Attempts were made to parallelise the code with multiple start points, but it resulted in non-conclusive outcomes, possibly due to issues with parallel execution. The thought behind is that simulated annealing is a stochastic algorithm, and its performance can depend on the initial solution. We can then run multiple instances of simulated annealing with different initial solutions in parallel. At first an erroneous code for this new function was implemented, and realisation came that there are a couple of issues:

1. Passing Function References: When submitting tasks to the executor, we should have passed the function reference along with its arguments. What we did was calling the function `solve_simulated_annealing(bqm, matrix)` immediately, and passing its result to `executor.submit`. It was then corrected and instead we used `functools.partial` to create a callable function with the specified arguments.
2. Avoiding Repeated Computation: Since `solve_simulated_annealing` is a stochastic algorithm, We might want to ensure that each run starts with a different initial state to explore a broader solution space. The mistake was that all runs are using the same initial state because we are calling the function outside of the `submit` method.

After fixing these issues, the code with the decorators and the "solve simulated annealing parallel" took 248.5 seconds. Which is still longer than using the initial function without parallelisation. A hypothesis is that maybe for a more complex Sudoku with more rows and columns, this method might be faster.

4 Validation, Unit Tests, and CI Setup

4.1 Validation

The concept of validation was implemented through the `is_correct` function in `sudoku_utils.py`, ensuring the correctness of Sudoku matrices. The function checks the validity of rows, columns, and boxes, providing detailed error messages in case of discrepancies. This function checks whether the rows, columns, and 3x3 boxes of the Sudoku matrix contain all the digits from 1 to 9 without repetition. If any of these conditions are not met, the function prints an error message and returns False, indicating that the Sudoku solution is not correct. If all conditions are satisfied, it prints a success message and returns True.

4.2 Unit Tests

Robust unit tests were developed for various functions in both `sudoku_utils.py` and `sudoku_simulated_annealing.py`. These tests include `test_get_matrix()`, `test_is_correct()`, `test_matrix_to_format()`, and `test_get_bqm()`. The unit tests cover parsing, correctness checking, formatting, and Binary Quadratic Model creation, ensuring the reliability of implemented functions. The first test creates a temporary file with a specified Sudoku puzzle, calls `get_matrix()`, to parse the puzzle from the file, and compares the result with the expected matrix. Then we have a test that provides two matrices, one with a duplicate in the first row, which gives us an incorrect Sudoku, and one representing a correctly solved Sudoku puzzle. The test asserts that the `is_correct()` function correctly identifies the correctness of each matrix. The third test function checks the correctness of the `matrix_to_format()`function. It provides an input matrix and compares the result of converting it to the expected formatted string. The last test checks the correctness of the `get_bqm()`function. It provides a Sudoku puzzle matrix and checks whether the resulting Binary Quadratic Model has the expected number of variables.

4.3 CI Setup

Before starting implementing the functions for each module, the pre-commit file was setup. This configuration defines a set of pre-commit hooks from various repositories, each serving a specific purpose like formatting, linting, and testing. In our case, we used Black for code formatting, Flake8 for linting, and custom testing hooks using pytest. Continuous Integration (CI) was set up on the same branch used for development and testing, providing immediate feedback on changes and facilitating early issue identification. It must be understood that Automated testing becomes crucial in this scenario to catch issues early. Furthermore, it was believed that robust automated testing practices can mitigate the risks associated with testing on the same branch as development. We have a main branch where we would update once we are sure that what's on the test branch is functioning, well formatted and reliable.

5 Packaging and Usability

5.1 Packaging

Conda environment specification (`environment.yml`) facilitates consistent dependency management, specifying Python version and channel selection. A Dockerfile is provided for containerisation, ensuring the code's runnable state in a container without additional efforts beyond image generation.

5.2 Usability

The project prioritises usability through a streamlined environment creation process using the specified Conda environment. A clear documentation (README.md) in the project repository was added, explaining how to set up and use the solver. It includes information about how to activate the Conda environment and to run the solver. The code includes docstrings for functions and modules, providing information about their purpose, parameters, and return values. A Doxygen-style documentation block is present at the beginning of the main script, introducing the project and its structure. Also, there are added enhance error messages and user feedback in the solver script, this way clear messages will help users understand and troubleshoot issues. The solver does the formatting of solved Sudoku output for better user readability.

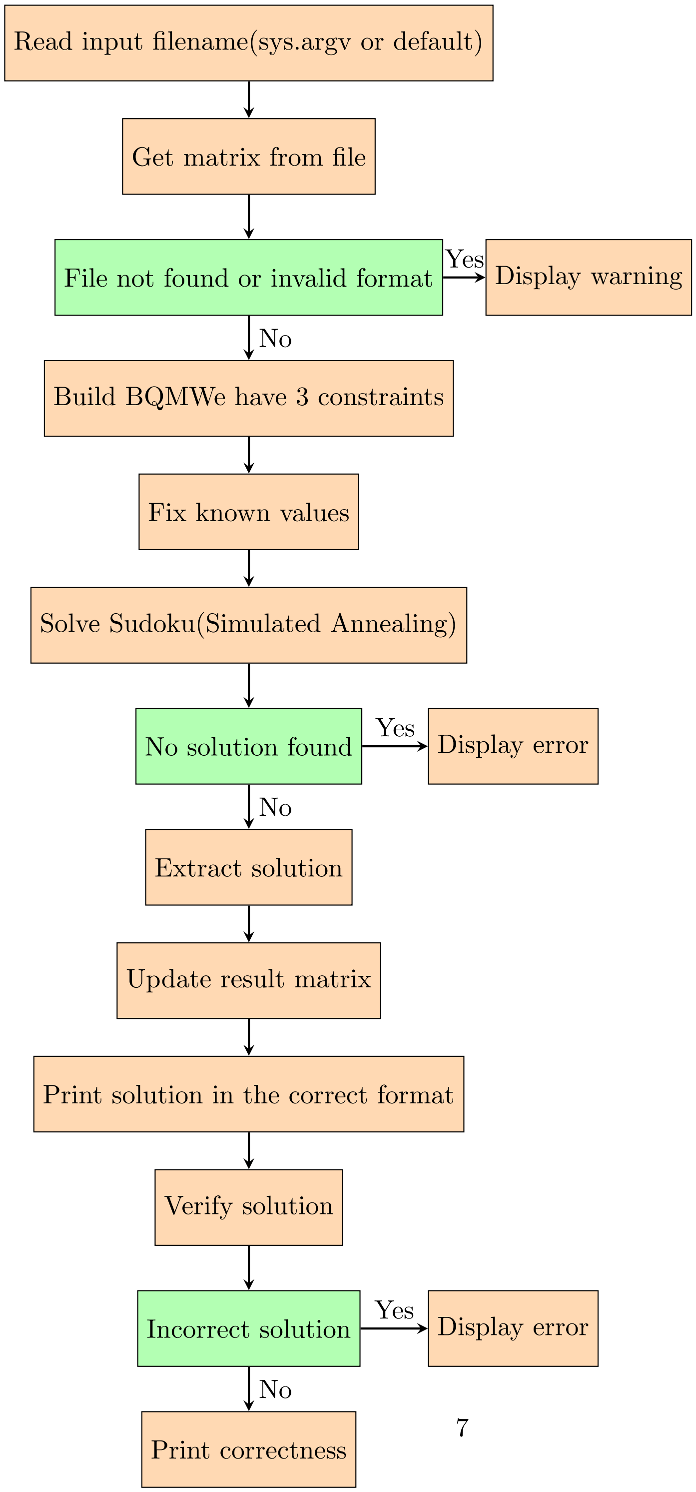
What could be interesting for future uses is that the inclusion of dwave-ocean-sdk means that the Sudoku solver project has the option to run on a D-Wave quantum computer using other samplers than the Simulated annealing sampler.

6 Summary

The Sudoku Solver project delivers a Python-based solution emphasising clarity, modularity, and reproducibility. Following best practices in software development, the project ensures effective version control, comprehensive documentation, proper structuring, and portability through containerisation. The report outlines the project's evolution from algorithm selection to validation, unit testing, and profiling, culminating in a user-friendly and well-packaged Sudoku solver implementation.

7 Annexe

7.1 Flowchart



7.2 Usage of ChatGPT

Prompt 2: How can I consider optimizing or parallelizing a simulated_annealing function?

Answer:

```
import concurrent.futures

def solve_simulated_annealing_parallel(problem, num_start_points=5):
  with concurrent.futures.ProcessPoolExecutor() as executor:
      futures = [executor.submit(solve_simulated_annealing, problem) for
      ↪  _ in range(num_start_points)]

  # Get results from parallel runs
  results = [future.result() for future in
  ↪  concurrent.futures.as_completed(futures)]

  # Return the best result
  best_result = min(results, key=lambda x: x[1])
  return best_result

# Usage
best_solution, best_cost =
↪  solve_simulated_annealing_parallel(your_problem_instance)
print(f"Best Solution: {best_solution}, Best Cost: {best_cost}")
```

Modification I have made:

```
def solve_simulated_annealing_parallel(bqm, matrix, num_start_points=5):
   with concurrent.futures.ProcessPoolExecutor() as executor:
       futures = [
           executor.submit(solve_simulated_annealing(bqm, matrix))
           for _ in range(num_start_points)
       ]

   # Get results from parallel runs
   results = [f.result() for f in
   ↪  concurrent.futures.as_completed(futures)]

   # Return the best result
   best_result = min(results, key=lambda x: x[1])
   return best_result
```

Prompt 1: Give me a full sudoku in the matrix format that is correct.

Answer:

```
[
    [5, 3, 4, 6, 7, 8, 9, 1, 2],
    [6, 7, 2, 1, 9, 5, 3, 4, 8],
    [1, 9, 8, 3, 4, 2, 5, 6, 7],
    [8, 5, 9, 7, 6, 1, 4, 2, 3],
    [4, 2, 6, 8, 5, 3, 7, 9, 1],
    [7, 1, 3, 9, 2, 4, 8, 5, 6],
    [9, 6, 1, 5, 3, 7, 2, 8, 4],
    [2, 8, 7, 4, 1, 9, 6, 3, 5],
    [3, 4, 5, 2, 8, 6, 1, 7, 9]
]
```

Usage:

In `def test_is_correct():` as a good matrix input.