C1: Research Computing - Coursework

William Knottenbelt, wdk24

December 10, 2023

Introduction

In this project we created a program that can be used to solve any Sudoku puzzle using the combination of four candidate elimination techniques (namely; naked singles, hidden singles, obvious pairs, pointing pairs/triples) and brute force search. The project was built in stages, starting with the creation of functionality to load, validate and save puzzles, then moving onto basic backtracking prototyping and implementation. This was followed by collection and generation of test puzzles, to lay the foundation for a robust testing suite of the final solver. Then, the four candidate elimination techniques were prototyped, implemented and integrated into the backtracking algorithm, and the final solver was constructed. Finally, profiling was used to optimize performance, and the documentation for the project was finalised. Unit testing and error trapping was performed throughout, and frequently tests were created in advance of the components they test. Pre-commit hooks were used from the beginning for automated formatting, linting and testing. Git was used for version control, with branching being utilised for new features and major changes. We maintained a development branch for the integration of different features before merging into main, as well as for minor changes which do not require their own branch. Docker was used for containerisation, to ensure universal usability.

Setting up

Before beginning development, we made broad decisions about the project. Inputs to the program would be text files containing Sudoku puzzles, and outputs would be solutions (saved in text files and printed to the console), hence the need for functions to load, print and save puzzles was apparent. For error catching purposes, we also needed a module for validating puzzles. To ensure quick resolution of errors, we decided all validation functions would return a message (either the string "Valid" or the relevant error message) rather than just a boolean. We also needed some way to represent Sudoku puzzles during the solving process. Since we would need to perform operations like filling empty squares and extracting whole rows/columns/squares, we decided to use NumPy arrays as they offer fast and flexible manipulation (eg. indexing, slicing, arithmetic operations). We also decided on the some variable naming conventions: '`puzzle`' for a Sudoku board as a NumPy array, '`row, col, block`' to refer to full rows, columns and blocks, and letters `i` and `j` for indices of rows and columns, respectively.

Prior to the development of any solving tools, we created the `toolkit` package to load, save, print and validate Sudoku puzzles, since it assists with development, experimentation and testing of the other components of the project. Initially, this package was called '`puzzleloading`' and implemented on branch '`loadingpuzzles`', but upon completion we renamed the package to the more suitable '`toolkit`', and reformulated the way we utilised Git. We decided all new features and large changes should have their own branches to prevent them interfering with other aspects of the project, and we would use a `development` branch for integration of different features (and for minor changes). This allows us to ensure stability of the features before merging into the `main` branch, which is reserved for stable code and would never be committed to directly. Additionally, we added tags to mark milestones in the project, making it easy to revert to specific versions if needed.

Important Note: If you checkout earlier commits, ensure that you have a `puzzles/` directory in the root of the project before running the testing suite, since the test `test_save_puzzle` originally depended on this. (The `Dockerfile` will create this directory for storing datasets anyway so it is unlikely to be an issue - but I mention it here just in case)

Selection of Solution Algorithms and Prototyping

Our primary aim for the project was to construct a solver which could solve any Sudoku puzzle on a reasonable time-scale of a few seconds or less, with the ability to find multiple solutions to a puzzle if desired and available. After light research into Sudoku solving algorithms [1] [2], we selected the brute force approach 'backtracking' [3] [4], since it is guaranteed to find a solution to any puzzle given enough time, and it is easily optimised by integrating more advanced solving techniques to reduce the search space. We decided on the development route to first implement backtracking and assess its suitability for our purposes, before exploring other techniques.

The backtracking algorithm works by placing numbers in empty squares one by one, and upon arriving at a square with no possible solutions, the algorithm backtracks by removing the last number added and trying the next one. This process continues until either the puzzle is solved or the algorithm has backtracked all the way back to the first square and there are no numbers left to try, indicating that the puzzle is unsolvable. We decided that a good way to implement this is by using a recursive `solve` function to iteratively fill in squares and return True when the puzzle is solved, triggering a cascade effect to return True at every preceding level of recursion. We implemented this simple backtracking prototype on branch `feature-backtracking`, and created solution-validating tools to ensure it functioned correctly. This original prototype can be found at Git tag '`backtracking-prototype`'. However, one challenged we faced with this initial backtracking prototype is that it did not allow for finding multiple solutions since this cascade effect terminates the search when the first solution is found. To address this, we modified the `solve` function to trigger further backtracking when the puzzle is filled, thus continuing the search for other solution. The revised function accepts a list `solutions`, which new solutions are added to and which triggers a condition to terminate recursion when the length of the list reaches a predetermined `num_solutions`. This was implemented on branch `feature-multiple-solutions` to ensure stability before altering the backtracking implementation on the `development` branch.

During this process, on a separate branch, we collected and generated a comprehensive set of test puzzles to be used in the testing suite, and to assess the need for additional solving tools. These were collected in part from Project Euler's problem 96 [5], from Peter Norvig's Sudoku solving project [6], and we manually copied the two hardest Sudoku puzzles in the world, devised by mathematician Dr Arto Inkala in 2010 [7] and 2012 [8]. We also created unsolvable puzzles by filling squares incorrectly in some of the puzzles we collected. We wanted the easiest bracket of our testing suite to be puzzles that are solvable by filling in 'Naked Singles' [9] (squares which have just one possibility), but were unable to find a public dataset providing this, so we generated them ourselves. To do this, we implemented a function that can repeatedly fills in 'Naked Singles' on branch `feature-singles-filling`, to be used to construct the generator function on branch `feature-generation`. For the generation algorithm, we planned to create filled puzzles by applying backtracking to an empty puzzle, then we could empty squares to generate puzzles. A challenged we encountered when prototyping this function was that our initial prototype for backtracking was deterministic, hence it would create the same filled puzzle every time starting from an empty board. To address this, we added randomness by shuffling the order in which we attempt to place numbers on a square.

This set of example puzzles was then used to test the backtracking algorithm to assess the need for additional solving techniques. We discovered that indeed, pure backtracking was too slow for certain hard puzzles, spending 18 minutes searching for a solution to `tests/test_puzzles/hard_01.txt` before we manually interrupted it. This was not the case for all hard puzzles though, as it found solutions to `hard_03.txt` and `hard_04.txt` in 80s and solved the two 'world's hardest' in only a few seconds each. Additionally, it was revealed that for puzzles containing only naked singles, the singles_filler function was 12-13x faster than backtracking alone, and for other puzzles, the speed at which they are solved is dramatically increased by filling in naked singles prior to applying backtracking. To see the project repository at (roughly) the point when this assessment was carried out, use `git checkout assessment`. Please note, since our `backtracker` implementation includes randomness, the time it takes to find solutions to the same puzzles may vary. After this assessment, the need for additional solving techniques was clear. We chose to implement candidate elimination techniques for their ability to reduce search space, which is critical for addressing the slow performance of the backtracking method. The methods we selected were 'Naked Singles' (AKA 'Obvious Singles' - discussed earlier) [10], 'Hidden Singles' [11], 'Obvious Pairs' [12], 'Pointing Pairs/Triples' [13] [14], which were each chosen for their simplicity and effectiveness.

After selecting our approach, we evaluated the most efficient method for candidate elimination in puzzles. Initially we considered calculating the candidates for all squares each time we reached a new square during the brute force search, but quickly realised that this would lead to unnecessary recalculations. Instead, we opted for a 'candidates grid', a structure like a dictionary or NumPy array, to store and update candidates. We tested both options for speed and found dictionaries slightly faster (see appendix A), but chose NumPy arrays for their versatile manipulation (with slicing) and intuitive similarity to our puzzle structure. We chose the naming convention of `candidates` for the candidates grid, since it is short and clear.

Development, Experimentation and Profiling

After selecting and prototyping the solution algorithms, we moved to develop the final solver. For each of the candidate elimination techniques, we wrote the tests in advance of implementation for early bug detection and to ensure that development would be closely aligned with the pre-defined requirements. We implemented them on branch `feature-elimination`, so as to not interfere with the rest of the code base. Once all techniques had been developed and thoroughly tested, we began experimenting with how best to combine the techniques in a single function: `all_elimination`. We considered two distinct approaches: either repeatedly apply each technique until no further changes in candidates occur before proceeding to the next technique, or implement all techniques sequentially in a single loop and continue this combined cycle until there are no more changes in candidates. To compare the two approaches, we collected a subset of 40 puzzles from our datasets, which were solvable purely using these elimination techniques. We timed how long it took each approach to find all 40 solutions, and discovered that the latter approach was significantly faster: `286 ms ± 2.98 ms` vs `585 ms ± 4.75 ms`, which can be explained by the efficient interplay between techniques (changes from one technique is immediately considered by others).

After this `all_elimination` function had been constructed, we performed a fast-forward merge into `development`, and integrated this into our backtracking algorithm to reduce the search space. We then built the main solving script `solve_sudoku.py`, which initially fills squares using `all_elimination`, and if the solution is not found, backtracking is employed. We tested the solver on all 50 easy, 95 hard and 11 hardest puzzles, and it completed all in 120s. Additionally, it solved the 2010 'worlds hardest' puzzle in 2.1s and the 2012 puzzle in 13s. To see the state of the full solver prior to profiling, use `git checkout before-profiling`.

We only began profiling once the full solver had been implemented and tested, since pre-mature profiling is likely to cause us to waste time optimizing the wrong things. We used `line_profiler` for line-by-line profiling of the main script and all functions we suspected would be time-consuming. This revealed that 99% of the total time was due backtracking and 95% of backtracking was due to `all_elimination`. Within `all_elimination`, the four elimination techniques took (in the order they were called) 33%, 20%, 9% and 32% of the total time. Upon examining the profiling results of each individual elimination function, we identified that taking a deep copy of the candidates grid at the start of each function was a major bottleneck, taking roughly 37% of the total time for three of the techniques (see example of this in appendix B), and 89% of the last technique. This line is unnecessary and was removed. We then tested the full solver again on all 50 + 95 + 11 = 156 puzzles, and it took 81s, which is an improvement of 40s compared to the solver before profiling. We repeated profiling again to identify additional bottlenecks and tried several changes, but were unable to make any significant optimizations. For instance, we saw that in `naked_singles_elimination`, the `discard(value)` method was a major bottleneck, so we tried replacing it with `candidates[i, r] -= value`, but this only made it slower.

Validation, Unit tests and CI set up

Within the `toolkit` package, we created a module `validation.py` to verify that puzzles coming into the project adhere to Sudoku rules, and to guarantee that the solutions found are valid. We had all validation functions return error messages when the validation fails, explaining the reason for the failure. This ensures that when errors arise, they are quick and easy to resolve. We included additional validation features elsewhere in the code base, such as in the function `toolkit.input.parse_sudoku_string`, which verifies that the input string has a valid Sudoku format before attempting to parse it into a NumPy array. Additionally, we included functionality in the backtracking algorithm to return the string "UNSOLVABLE" when no solution is found, so that it is clear to the user that the puzzle they provided cannot be solved.

Throughout development, we wrote unit tests to verify each component's functionality, which drastically increased the reliability of the program, and made it far less risky to make alterations to components, since it makes it immediately clear when the change has introduced bugs. Frequently, we wrote unit tests in advance of implementation of the components they test, to ensure early error catching and adherence to the predefined specifications. For additional robustness, we employed error trapping and type checking in all functions and scripts throughout the project, primarily through the use of `assert` statements. We added print statements to these `assert`s so that when they do raise errors, it is clear what the problem is, which ensures rapid error identification and fixing.

From the beginning, we made use of pre-commit hooks, which were configured using the `pre-commit` package and a `pre-commit-config.yaml` file. We included `black` for automated formatting, `flake8` for linting and `pytest` to run the testing suite before each commit. This guaranteed that all code we committed had the correct PEP8 style, and passed all tests, which significantly reduced the likelihood of committing code containing bugs. On certain occasions, it was necessary to remove `pytest` hook before committing, for instance during testing-led development when committing tests before writing the components.

Packaging and Usability

For user convenience, the components of the project were separated into distinct packages and modules. Our first package, `engine`, contains all solving functionality and includes three modules; `backtracking.py` for implementation of the backtracking algorithm, `elimination.py` containing all the candidate elimination techniques and `basics.py` for the basic essential functions assisting solving. The second package, `toolkit`, contains all utilities which are not directly related to the solving pipeline. It is made up of four modules; `generation.py` for generation of test puzzles, `input.py` for handling inputs to the program, `output.py` for managing visualisation and outputs from the program, `validation.py` for validating puzzles and solutions. This organisational structure makes clear the range of features available, and allows users of the project to cherry-pick components they need. Furthermore, it allows for isolated testing and debugging, which significantly improves the speed of identifying and resolving issues.

To guarantee the project is universally usable (and reproducible), we used Docker for containerisation. We saved a conda environment necessary for running/reproducing the project in a `environment.yml` file. In the environment we included only the bare essentials for running the project, not the exact environment used during development, since this contains packages and versions that are only available for Mac OS and we want the project to be universally usable on all operating systems. We wrote a Dockerfile for automated generation of the docker image, which uses the `continuumio/miniconda3` base image, installs `git` and `curl`, copies the root directory of the project to the image working directory and recreates the conda environment. We also added commands to the Dockerfile that obtain the full datasets of example puzzles used for evaluation/assessment of the solving algorithms during development. These were not all used in the testing suite, so are not necessary to be commit, but are important for reproduction of the assessment results discussed in the report.

Summary

In summary, this project successfully developed a Sudoku solver utilising advanced candidate elimination techniques integrated into a backtracking algorithm, emphasizing proper planning/prototyping, robust testing, code quality, and usability. The program handles input and output through text files, with puzzles and candidate grids represented as NumPy arrays for efficient manipulation. The development process involved extensive use of Git for version control, branching for new features, and maintaining a development branch for feature integration. The final solver was tested and optimized using profiling tools, revealing the use of `copy.deepcopy` on the candidates grid as a significant bottleneck which was subsequently addressed. The project also focused on validation and unit testing to ensure reliability of components and solutions, along with setting up pre-commit hooks for automated code quality checks. Modularity was employed effectively for user-friendliness, and containerisation with docker was used to ensure universal usability.

References

[1] Wikipedia contributors. *Sudoku solving algorithms.* `https://en.wikipedia.org/wiki/Sudoku_solving_algorithms`

[2] sudoku.com, *Sudoku rules.* https://sudoku.com/sudoku-rules

[3] geeksforgeeks. *Backtracking Algorithms.* https://www.geeksforgeeks.org/backtracking-algorithms/?ref=lbp

[4] computerphile. *Python Sudoku Solver.* https://www.youtube.com/watch?v=G_UYXzGuqvM&t=543s

[5] Project Euler. *Problem 96: Su Doku.* https://projecteuler.net/index.php?section=problems&id=96

[6] Peter Norvig. *Solving Every Sudoku Puzzle.* https://norvig.com/sudoku.html

[7] Fiona Macre for the Daily Mail. *It took three months to create, so how long will it take to crack... the world's hardest Sudoku?* https://www.dailymail.co.uk/news/article-1304222/It-took-months-create-long-crack--worlds-hardest-Sudoku.html

[8] Nick Collins. *World's hardest sudoku: can you crack it?* https://www.telegraph.co.uk/news/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html

[9] Sudoku9x9.com. *Naked Single* https://www.sudoku9x9.com/techniques/nakedsingle/

[10] sudoku.com. *"Obvious Singles" technique* https://sudoku.com/sudoku-rules/obvious-singles/

[11] sudoku.com. *"Hidden Singles" technique* https://sudoku.com/sudoku-rules/hidden-singles/

[12] sudoku.com. *"Obvious Pairs" technique* https://sudoku.com/sudoku-rules/obvious-pairs/

[13] sudoku.com. *"Pointing Pairs" technique* https://sudoku.com/sudoku-rules/pointing-pairs/

[14] sudoku.com. *"Pointing Triples" technique* https://sudoku.com/sudoku-rules/pointing-triples/

Appendix

A Candidates dictionary vs array

```
# candidates dictionary
candidates_dict = {}
for i in range(9):
    for j in range(9):
        candidates_dict[(i,j)] = set(range(1,10))

# candidates array
candidates_arr = np.full((9,9), set(range(1,10)))

# Timing how long it takes to retrieve, set and eliminate
idx = (3,2)

%timeit candidates_dict[idx]
```

```
%timeit candidates_arr[idx]

%timeit candidates_dict[idx] = {2,3,4,5,6}
%timeit candidates_arr[idx] = {2,3,4,5,6}

%timeit candidates_dict[idx].discard(3)
%timeit candidates_arr[idx].discard(3)
```

Output:

```
25.3 ns ± 0.185 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
36.1 ns ± 0.0392 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
82.4 ns ± 0.12 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
96.2 ns ± 0.12 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
36.5 ns ± 0.5 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
47.5 ns ± 0.0321 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

B Example Profiling Results

```
Total time: 0.587134 s
File: /Users/willknott/Desktop/DIS/coursework/rc/wdk24/src/engine/elimination.py
Function: naked_singles_elimination at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           @profile
     8                                           def naked_singles_elimination(candidates):
     9                                               """
    10                                               @brief Eliminate candidates using the naked sing
    11
    12                                               @details The naked singles technique works by fi
    13                                               only have a single candidate, then that candidat
    14                                               from all other squares in the same row, column a
    15
    16                                               Reference: https://sudoku.com/sudoku-rules/obvic
    17                                               """
    18       489        104.0      0.2      0.0      assert isinstance(candidates, np.ndarray) and ca
    19       489         59.0      0.1      0.0      assert candidates.shape == (9, 9)
    20
    21                                               # take copy of candidates grid to avoid modifyin
    22       489     219467.0    448.8     37.4      candidates = copy.deepcopy(candidates)
    23
    24      4890        556.0      0.1      0.1      for i in range(9):
    25     44010       4327.0      0.1      0.7          for j in range(9):
    26     39609       7418.0      0.2      1.3              if len(candidates[i, j]) == 1:
    27                                                           # extract the single candidate
    28     33898       8220.0      0.2      1.4                  value = list(candidates[i, j])[0]
    29
    30                                                           # eliminate this candidate from all
```

```
31    338980      30153.0      0.1      5.1                  for c in range(9):
32    305082      23532.0      0.1      4.0                      if c != j:
33    271184      49818.0      0.2      8.5                          candidates[i, c].discard(val
34
35                                                           # eliminate this candidate from all
36    338980      29498.0      0.1      5.0                  for r in range(9):
37    305082      22641.0      0.1      3.9                      if r != i:
38    271184      48109.0      0.2      8.2                          candidates[r, j].discard(val
39
40                                                           # eliminate this candidate from all
41     33898       3792.0      0.1      0.6                  block_i = 3 * (i // 3)  # row index
42     33898       3672.0      0.1      0.6                  block_j = 3 * (j // 3)  # col index
43    135592      14883.0      0.1      2.5                  for r in range(block_i, block_i + 3)
44    406776      43760.0      0.1      7.5                      for c in range(block_j, block_j
45    305082      26555.0      0.1      4.5                          if r != i or c != j:
46    271184      50515.0      0.2      8.6                              candidates[r, c].discard
47
48       489         55.0      0.1      0.0      return candidates
```