C1: Research Computing - Coursework

William Knottenbelt, wdk24

December 16, 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 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. 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 (indexing, slicing, arithmetic operations, etc.). 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 rest 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 maintain

a `development` branch for integration of different features, as well as for minor changes which do not require their own branch. This allows us to validate features and changes before merging into the `main` branch, which is reserved for stable code only 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. We used a '`.gitignore`' file to specify which files and directories not to commit.

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 of first implementing backtracking to assess its suitability for our purposes before exploring other techniques, so as to not waste time implementing additional functionality if backtracking is sufficient.

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. 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. 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 solutions. 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. See the prototype before and after this update in appendix A.

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 from Project Euler's problem 96 [5] and from Peter Norvig's Sudoku solving project [6]. We also manually copied the two hardest Sudoku puzzles in the world, devised by mathematician Dr Arto Inkala in 2010 [7] and 2012 [8]. We then 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 first implemented a function, '`singles_filler`', that can repeatedly fills in naked singles (on branch '`feature-singles-filling`'). Our plan for the generation algorithm was to apply backtracking to an empty board to create fully-solved puzzles, then removing numbers while ensuring that the puzzle remains solvable by our '`singles_filler`' function. A challenged we encountered when prototyping this algorithm 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 '`puzzles/hard/hard_01.txt`' before we manually interrupted it. However, other hard puzzles, such as '`hard_03.txt`' and '`hard_04.txt`', were solved within a couple minutes, and Dr Inkala's 'world's hardest' puzzles were solved in only a few seconds each. Nevertheless, it was clear that more advanced solving techniques were needed. Additionally, it was revealed that the speed at which many puzzles were solved could be dramatically increased by filling in naked singles prior to applying backtracking, and for puzzles containing only naked singles. This demonstrated the benefit of adding additional functionality to assist with backtracking. To see the project repository at the point when this assessment was carried out, use '`git checkout assessment`'. After this assessment, the need for additional solving techniques was clear. We chose to use 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') [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 began prototyping how to implement candidate elimination most efficiently. 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 B), but chose NumPy arrays instead for their versatile manipulation and intuitive similarity to our '`puzzle`' object. 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 solving functionality. 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` vs `585 ms`, which can be explained by the efficient interplay between techniques (changes from one technique is immediately considered by others). The implementation of these two options and the code used to time them can be found in appendix C.

After this '`all_elimination`' function had been constructed, we integrated it into our backtracking algorithm to reduce the search space, and performed a fast-forward merge into '`development`'. 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 D), 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. Names of all testing files and functions began with '`test_`', such that the full suite can be run with a single '`pytest`' command. 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, making it easy to address.

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 a logical structure of 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 related to solving. The second package, '`toolkit`', contains all utilities which are not direct parts of the solving process. 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, and '`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. For readability, all components of the project contained detailed comments and docstrings, which were made compatible with Doxygen using special commands. A '`Doxyfile`' was written to configure the settings for Doxygen, allowing the users to generate documentation for the project. We added the MIT license to the repository to convey that it is freely available for anyone to use or modify, and created a README containing feature descriptions, installation instructions, usage examples, framework information and credits.

To guarantee the project is universally usable (and reproducible), we saved a Conda environment necessary for running the project in an '`environment.yml`' file, and used Docker for containerisation. In '`environment.yml`', we included the just the bare essentials for running the project, since the exact environment used during development contains packages and versions that are only available for Mac OS (eg. '`libgfortran`'), and we want users with different operating systems (who don't use Docker) to be able to run the project. 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. The Dockerfile also includes commands to automatically download and set up the puzzle datasets used during development for assessment and evaluation of the solving functionality. The custom script '`convert_data.py`' is used to convert these datasets into the conventional puzzle format used throughout the project. Instructions to do this without the use of Docker were included in the README. Since not all puzzles were necessary for the testing suite, they were not committed, but they are important for reproduction of certain results discussed in the report.

Summary

In summary, this project successfully developed an efficient Sudoku solver utilising the backtracking algorithm enhanced with advanced candidate elimination techniques, emphasizing proper planning/prototyping, robust validation, readability, and usability. The program possesses the ability to find multiple solutions to puzzles, can deal with unsolvable or invalid puzzles, and handles input and output through text files. The final solver was optimized using profiling tools, revealing the use of `copy.deepcopy` on the candidates grid as a significant bottleneck which was subsequently addressed. The development process involved extensive use of Git for version control, branching for new features, and maintaining a development branch for feature integration. Validation was enacted using unit tests, error trapping and pre-commit hooks, ensuring reliability of all components. For user-friendliness, the project was equipped with logical modular structure, proper documentation and a descriptive README file. Finally, a Dockerfile and Conda environment was included in the repository, ensuring effortless usability.

References

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

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

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

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

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

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

[7] Fiona Macre. *It took three months to create, so how long will it take to crack... the world's hardest Sudoku?* Retrieved from 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?* Retrieved from https://www.telegraph.co.uk/news/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html

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

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

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

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

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

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

Appendix

A Backtracking prototyping

Below is the initial backtracking prototype, which did not allow for finding multiple solutions.

```
def backtracker(puzzle):

    def solve(puzzle):
        for i in range(9):  # iterate over rows
            for j in range(9):  # iterate over columns
                if puzzle[i][j] == 0:  # find an empty cell
                    for num in range(1, 10):  # try all numbers from 1 to 9
                        if possibility(puzzle, num, i, j):
                            puzzle[i][j] = num

                            if solve(puzzle):
                                return True

                            puzzle[i][j] = 0  # backtrack if puzzle was not solved

                    return False # trigger backtracking if no number is possible

        return True  # puzzle solved

    # run backtracking
    solved = solve(puzzle)

    if not solved:
        return "UNSOLVABLE"

    return puzzle
```

We corrected the prototype to allow for finding multiple solutions, by eliminating certain return statements to prevent the cascading 'break out' effect once a solution is found.

```
def backtracker(puzzle, num_solutions=1):
    solutions = []

    def solve(puzzle, solutions):
        # break recursion when we have found enough solutions
        if len(solutions) >= num_solutions:
            return

        for i in range(9):  # iterate over rows
            for j in range(9):  # iterate over columns
                if puzzle[i][j] == 0:  # find an empty cell
                    numbers = np.arange(1, 10)
                    np.random.shuffle(numbers)  # introduce randomness

                    for n in numbers:
                        if possibility(puzzle, n, i, j):
                            puzzle[i][j] = n  # fill square if number is a possibility
                            solve(puzzle, solutions)  # recursively fill puzzle
                            puzzle[i][j] = 0  # backtrack
                    return  # trigger backtracking if there are no possibilities

        # we only get here if puzzle is solved
        solutions.append(puzzle.copy())

    solve(puzzle, solutions)

    # if there are no solutions, the puzzle is unsolvable
    if not solutions:
        return "UNSOLVABLE"

    # if we want 1 solution, don't return as a list
```

```
    if num_solutions == 1:
        assert len(solutions) == 1
        return solutions[0]

    # assert all solutions are unique
    flattened_tuple_solutions = map(lambda x: tuple(x.flatten()), solutions)
    unique_solutions = set(flattened_tuple_solutions)
    assert len(solutions) == len(unique_solutions), "Solutions found are not unique"

    return solutions
```

B Candidates dictionary vs array

During the prototyping stage for the candidate elimination techniques, we timed whether a dictionary or NumPy array was faster for the following operations: getting candidates, setting candidates, discarding candidates.

```
# 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)

# getting candidates
%timeit candidates_dict[idx]
%timeit candidates_arr[idx]

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

# discarding candidates
%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)
```

C Combining elimination techniques

Find below the two options considered for the `all_elimination` function, which combines all four candidate elimination techniques. The first option (implement each technique individually it's own loop to eliminate all candidates possible, before moving onto the next technique) was:

```
def all_elimination(candidates):

    # type-checking and taking copy to avoid mutating the original
    assert isinstance(candidates, np.ndarray) and candidates.dtype == object
    assert candidates.shape == (9, 9)
    candidates = copy.deepcopy(candidates)

    # repeatedly apply each candidate elimination technique individually
    # until no changes are made, before moving onto the next technique
    # repeat the full sequence until there are no changes to the candidate grid
    old_candidates_main = None
    while not np.array_equal(candidates, old_candidates_main):
        old_candidates_main = copy.deepcopy(candidates)

        old_candidates = None
        while not np.array_equal(candidates, old_candidates):
            old_candidates = copy.deepcopy(candidates)
            candidates = naked_singles_elimination(candidates)

        old_candidates = None
        while not np.array_equal(candidates, old_candidates):
            old_candidates = copy.deepcopy(candidates)
            candidates = hidden_singles_elimination(candidates)

        old_candidates = None
        while not np.array_equal(candidates, old_candidates):
            old_candidates = copy.deepcopy(candidates)
            candidates = obvious_pairs_elimination(candidates)

        old_candidates = None
        while not np.array_equal(candidates, old_candidates):
            old_candidates = copy.deepcopy(candidates)
            candidates = pointing_elimination(candidates)

    return candidates
```

The second option (implement each technique sequentially in a single loop) was:

```
def all_elimination(candidates):

    # type-checking and taking copy to avoid mutating the original
    assert isinstance(candidates, np.ndarray) and candidates.dtype == object
    assert candidates.shape == (9, 9)
    candidates = copy.deepcopy(candidates)

    # Apply all elimination techniques until candidates grid stops changing
    old_candidates = None
    while not np.array_equal(candidates, old_candidates):
        old_candidates = copy.deepcopy(candidates)
        candidates = naked_singles_elimination(candidates)
        candidates = hidden_singles_elimination(candidates)
        candidates = obvious_pairs_elimination(candidates)
        candidates = pointing_elimination(candidates)
```

```
    return candidates
```

To time each technique, we used the following code, where '`easy_filepaths`' were the filepaths of all 50 easy puzzles sourced from Project Euler's Problem 96. The range [7:47] includes only puzzles which can be solved using the candidate elimination alone, hence they were used to test the speeds of the two options.

```
%%timeit

for filepath in easy_filepaths[7:47]:
    puzzle = load_puzzle(filepath)
    candidates = init_candidates(puzzle)
    candidates = all_elimination(candidates)
    solution = filler(puzzle, candidates)
    assert validate_solution(puzzle, solution) == "Valid"
```

For the first option, we obtained the output:

```
585 ms ± 4.75 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

And for the second option:

```
286 ms ± 2.98 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Hence we implemented the second option.

D Example Profiling Results

Below are the `line_profiler` results from profiling the `naked_singles_elimination` function. We see that the calling `copy.deepcopy` took 37.4% of the total time of the function. Similar results were obtained for all elimination functions.

```
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 singles technique.
    11
    12                                               @details The naked singles technique works by finding squares that
    13                                               only have a single candidate, then that candidate can be eliminated
    14                                               from all other squares in the same row, column and block
    15
    16                                               Reference: https://sudoku.com/sudoku-rules/obvious-singles/
    17                                               """
    18       489        104.0      0.2      0.0      assert isinstance(candidates, np.ndarray) and candidates.dtype == object
```

```
19       489         59.0      0.1      0.0      assert candidates.shape == (9, 9)
20
21                                                # take copy of candidates grid to avoid modifying the original
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 squares in the same row
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(value)
34
35                                                        # eliminate this candidate from all squares in the same column
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(value)
39
40                                                        # eliminate this candidate from all squares in the same block
41     33898       3792.0      0.1      0.6                  block_i = 3 * (i // 3)  # row index of top left square in block
42     33898       3672.0      0.1      0.6                  block_j = 3 * (j // 3)  # col index of top left square in block
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 + 3):
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(value)
47
48       489         55.0      0.1      0.0      return candidates
```