AMS 325 Final Project (Sudoku Tester): Rohan Basavaraju, Patrick McLoughlin

1. Introduction

Sudoku took the puzzle world by storm in the early part of the 21st century. It's almost inconceivable to wait for a flight at an airport or to read a newspaper puzzle section today without coming across an opportunity to play the game. Its exercise of elementary combinatorial skills and logical thinking has been suggested (Grabbe, 2017) to positively affect the working memory of individuals who played, particularly in aging adults.

Despite the explosive popularity Sudoku has enjoyed just within the past two decades, the diversion has centuries-old ancestors. Swiss mathematician Leonhard Euler developed "Latin Squares" in 1783, of which present-day Sudoku can be considered a particular version (Smith, 2005). In the late 1800s, various French periodicals published a *carré magique diabolique* or "diabolical magic square", which could be recognized today as a version of the modern game (Boyer, 2006). As authors, we can confirm learning the game without mobile or digital technology can invoke pencil-and-eraser frustration worthy of such a devilish moniker!

New York's own *Dell Puzzle Magazines* can claim credit for introducing the modern version of the numerical riddle with "Number Place" in 1979 (Smith, 2005). The accessible yet confounding brain teaser spread to an apt audience of Japanese problem solvers in 1984 under publisher *Nikoli*, who christened it under the name we know today: Su Doku is short for *Sūji wa dokushin ni kagiru*, or "the numbers which are single".

Thirteen years later, New Zealander Wayne Gould was browsing a Hong Kong bookstore when he was so taken by a collection of Sudoku puzzles as to develop his own computer program to generate games. Through family and commercial contacts, this led to publication in the *Times* of London. By 2005, Sudoku was ubiquitous in western media, from European television to campus newspapers at universities and colleges across America.

Modern sudoku's rules involve a 9×9 grid, which is partitioned into nine 3×3 sub-grids. Successful completion of the puzzle is placement of the numbers 1 through 9 exactly once each in each column, row, and 3×3 sub-grid of the 9×9 grid (Easybrain LTD, 2018). The grid would

contain integer values ranging from one to nine, inclusive. However, other versions of the game may use letters (e.g. ‘A’-‘I’, inclusive). The Japanese Sudoku version even uses different dimensions (e.g. 4x4 grid of 4 2x2 sun-grids with a range of integer values between one and four, inclusive) (Moler, 2011).

In this project, the primary objective was to create an application that would assist in the gameplay of Sudoku. In particular, there were multiple key tasks that needed to be completed to achieve this objective: creating a viable Sudoku board (across different styles of the game), checking what values a position within the grid (can) have, testing if a filled Sudoku board is valid, and analyzing the results of the Sudoku board using matrix visualizations.

2. Methodology

This computational Sudoku project required the use of a variety of tools and technologies. To implement the core functions for the proposed Sudoku module, the Python3 programming language was used. To enhance the mathematical computing power of the module implementation, the NumPy package (from Python) was used. The Matplotlib package (from Python) was used to create visual plots based on the fundamental functions in the Sudoku project. This included both the PyPlot and Colors packages within Matplotlib as they were used to chart the data points and visually distinguish between data points, respectively. Additionally, the Atom, Visual Studio, and Google Colab integrated development environments (IDEs) were used to develop and test the Python programs. As the code was iteratively and collaboratively developed, GitHub was used as the version control system. This was especially useful as the creators of this project could explore how code changes are synced together in real-time using open source software. As evident, the computing tools utilized for this project are widely used in industry, which was a key motivation of the creators of this project as they wished to learn more about real-world software engineering practices and widen their technical toolkit.

While there are multiple versions of Sudoku that can be played like Japanese and letter-based Sudoku, the core rules of the game are similar across each version. Namely, a grid

with the same number of columns and rows is used to play the game. Each position in the grid can have one value from a range of possibilities. The total number of possibilities is at most the same as the number of columns and rows in the grid. However, depending on the values in the other positions within the grid, certain values may not be possible in a given position under those conditions. Likewise, the Sudoku board will typically be preset with some positions already filled in with valid values. As a result, these predetermined values are unable to be changed. As for the specifics of the Sudoku game, these parameters will depend upon the version of Sudoku being played. In the case of a traditional Sudoku game, there will be nine rows and columns with possible values falling within the inclusive integer range of one through nine. Japanese Sudoku is very similar, except its board uses four columns and rows and has a range of possible values inclusively falling between one and four. Lastly, the letter Sudoku game will also have a board of nine columns and rows, but its range of possible values includes the following nine characters: A, B, C, D, E, F, G, H, and I.

In terms of the code written for this Sudoku tester project, a total of eight different functions were used. For each of these functions, docstrings were included to describe each function's inputs and outputs while comments were used throughout to denote any significant pieces of the function's code. Test cases were also created for each function to evaluate the function's success across the three core flavors of Sudoku (normal, letter, Japanese). The first function was createEmptyBoard(), which took the Sudoku version as its only input (with "normal" set as the default option). This generated a 2-dimensional Python array that would be initially filled with zeroes as its values. This is important to note as zeroes do not fall under the integer range of one to nine (inclusive) or the character range of 'A' to 'I' (inclusive). As a result, they are a suitable value for indicating that a space on the Sudoku board has not been filled as of yet. The normal or letter Sudoku board would output a nine by nine grid while the Japanese Sudoku board would output a four by four grid. The next function was checkElement(), which used a user-provided Sudoku board along with a given position and Sudoku value to determine (i.e.

return a Boolean value) if the value was valid for that particular position. This coincided with the possibleElements() function, which returned a list of valid Sudoku values for a given position on a Sudoku board. The addElement() and removeElement() functions were implemented to replace a value at a given position with either a new Sudoku value or an empty Sudoku space (e.g. the integer value of zero). The final three functions focused on evaluating the success of a given Sudoku board. First, testBoard() checked if every value in a Sudoku board met the criteria for being a valid solution. As outlined previously, the board must only have valid Sudoku values (for the given Sudoku version) with each value appearing only once in each column, row, and sub-board. Next, the accuracyBoard() printed the total number of positions in the Sudoku board, the number of correct values, and the percent of spaces on the board that were correct. Lastly, the viewAccuracyPlot() used the NumPy and Matplotlib packages to create a colorful plot of the distribution of correct and incorrect Sudoku values (for the given Sudoku version). In particular, a grid with the appropriate dimensions would have either a green or red filled-in box at every position to indicate if the value at the position was correct or incorrect, respectively.

3. Results and Discussion

For the first function, createEmptyBoard(), three different test cases were used as the version of the Sudoku was set to normal, letter, or Japanese for the test cases. For the first two Sudoku versions, a 9x9 2D array was successfully created with each value in the grid initialized at zero. For the Japanese Sudoku test case, similar success was found as the expected output of a zero-initialized 4x4 2D array was returned as the function's output. For the next four functions (checkElement(), possibleElements(), addElement(), removeElement()), three attempted Sudoku boards were used to test each function's success across each version of Sudoku.

For normal Sudoku, the test case used the board 'b1': [[8,1,6,5,7,9,2,4,3], [3,5,7,6,2,4,9,8,1], [4,9,2,1,3,8,7,6,5], [2,4,3,0,1,6,5,7,9], [9,8,1,3,5,7,6,2,4], [7,6,5,4,9,2,1,3,8], [5,7,9,2,4,3,8,1,6], [6,2,4,9,8,1,3,5,7], [1,3,8,7,6,5,4,9,2]]. Every element in this Sudoku board

was valid except for the element at the 2D array index (3,3), which has a value of zero and represents an unfilled position within the board. When the checkElement() function was run for a value of 8 at (3,3), the function returned True (as expected). Likewise, the function returned False (as expected) for a value of 2 as any other value would not be valid when playing normal Sudoku for this board at position (3,3). This resembles what was found for the possibleElements() function as an array with the element 8 was retrieved as output when the input used the position (3,3). For the final two of these functions, the value 8 was successfully added and then removed (and replaced with the value 0) at position (3,3), respectively.

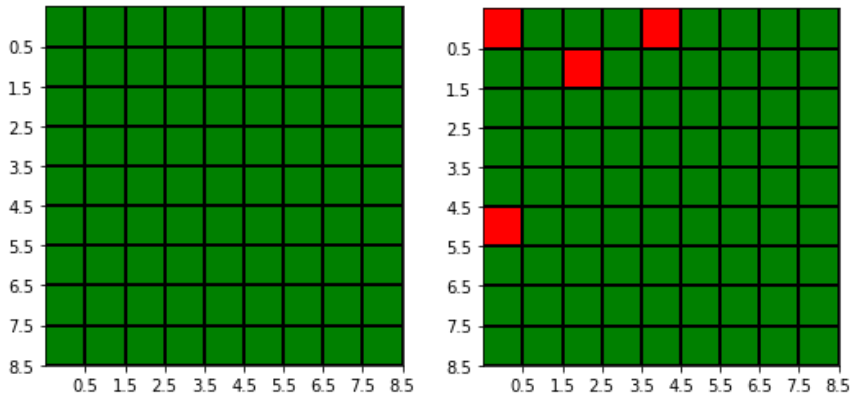
For letter Sudoku, the test case used the board ‘b3’: [['H', 'A', 'F', 'E', 'G', 'I', 'B', 'D', 'C'], ['C', 'E', 'G', 'F', 'B', 'D', 'I', 'H', 'A'], ['D', 'I', 'B', 'A', 'C', 'H', 'G', 'F', 'E'], ['B', 'D', 'C', 0, 'A', 'F', 'E', 'G', 'I'], ['I', 'H', 'A', 'C', 'E', 'G', 'F', 'B', 'D'], ['G', 'F', 'E', 'D', 'I', 'B', 'A', 'C', 'H'], ['E', 'G', 'I', 'B', 'D', 'C', 'H', 'A', 'F'], ['F', 'B', 'D', 'I', 'H', 'A', 'C', 'E', 'G'], ['A', 'C', 'H', 'G', 'F', 'E', 'D', 'I', 'B']]. This is the same as ‘b1’, except the integer values were replaced with their corresponding English capital alphabet letters (e.g. 1 -> ‘A’, 9 -> ‘I’, 0 -> 0). The same four functions were run at position (3,3) with the correct and incorrect values of ‘H’ and ‘G’, respectively. The functions yielded similarly successful results, which indicated that the functions are scalable across integer and string data types alike. This included receiving an array with the character ‘H’ for possibleElements() and updating ‘b3’ with added and removed elements.

For Japanese Sudoku, the test case used the board ‘b5’: [[1,4,2,3], [3,2,4,1], [4,1,3,2], [2,3,1,0]]. When testing the same four functions as described above, the functions provided successful results like showing value 3 was valid while value 4 was invalid at position (3,3). This led to seeing only the value 3 was valid at this position and being able to add and remove an element from the specified position.

For the final three functions (testBoard(), accuracyBoard(), viewAccuracyPlot()) in this Sudoku tester project, three pairs of Sudoku boards were used where each pair included a correctly and incorrectly solved Sudoku board for each Sudoku version (normal, letter,

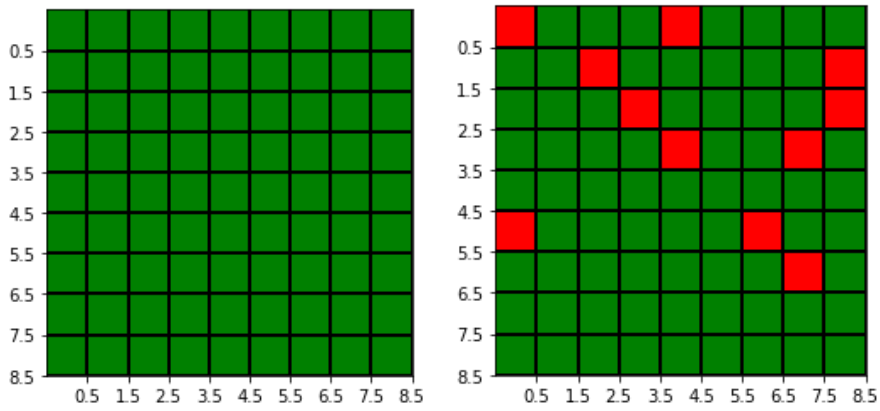
Japanese). The correctly solved Sudoku boards for each Sudoku version are the same as the ones previously tested (i.e. ‘b1’, ‘b3’, ‘b5’) for normal, letter, and Japanese, respectively, but their value at (3,3) has been replaced with the correct values to allow for a valid solution.

For normal Sudoku, the incorrect board was represented by board ‘b2’: [[7,1,6,5,7,9,2,4,3], [3,5,7,6,2,4,9,8,1], [4,9,2,1,3,8,7,6,5], [2,4,3,8,1,6,5,7,9], [9,8,1,3,5,7,6,2,4], [7,6,5,4,9,2,1,3,8], [5,7,9,2,4,3,8,1,6], [6,2,4,9,8,1,3,5,7], [1,3,8,7,6,5,4,9,2]]. When testBoard() was run for normal Sudoku, ‘b1’ produced a positive result while ‘b2’ produced a negative result as the value in ‘b2’ at position (0,0) was changed from 8 to 7. To analyze this solution, the accuracyBoard() function printed that 100% of the values in ‘b1’ were correct but only 77 of the 81 spaces were accurate, leading to a 95% accuracy rate. This is interesting to note as changing only one value in the Sudoku board led to four values in the board being inaccurate: the changed value itself and the repeated values in the same row, column, and sub-grid as the changed value. As shown by the figures below, this was reflected when running viewAccuracyPlot() as ‘b1’ led to a completely green grid while ‘b2’ resulted in four points on the grid being red instead of green. These pinpoint the exact positions in the board at which 7 repeats twice in the same row, column, and sub-grid.

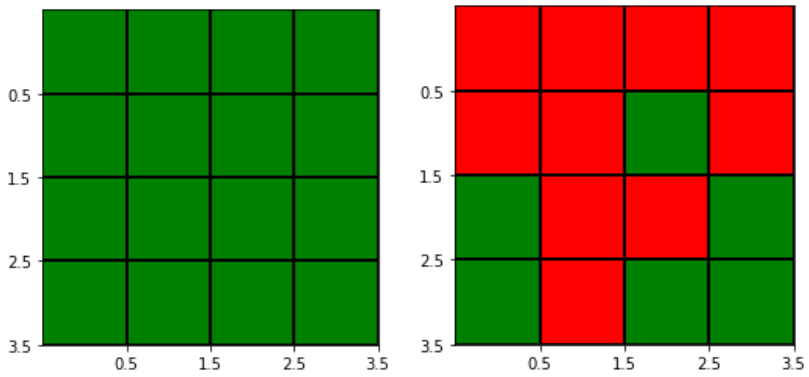


For letter Sudoku, the incorrect board was expressed as board ‘b4’: [['H', 'A', 'F', 'E', 'G', 'I', 'B', 'D', 'C'], ['C', 'E', 'G', 'F', 'B', 'D', 'I', 'H', 'A'], ['D', 'I', 'B', 'A', 'C', 'H', 'G', 'F', 'A'], ['B', 'D', 'C', ‘H’, 'A', 'F', 'E', 'A', 'I'], ['I', 'H', 'A', 'C', 'E', 'G', 'F', 'B', 'D'], ['G', 'F', 'E', 'D', 'I', 'B', 'A', 'C', 'H'], ['E', 'G',

'I', 'B', 'D', 'C', 'H', 'A', 'F'], ['F', 'B', 'D', 'I', 'H', 'A', 'C', 'E', 'G'], ['A', 'C', 'H', 'G', 'F', 'E', 'D', 'I', 'B']]. When running the testBoard() function, the result correctly showed ‘b3’ was a valid Sudoku solution while ‘b4’ was invalid. When further exploring these results, the accuracyBoard() function indicated that 100% of the 81 values in ‘b3’ were valid while only 70 of the 81 values (i.e. 86%) were valid in ‘b4’. Looking at the figures below, these results were charted using viewAccuracyPlot() and provided similar visual results as the 11 positions at which a repeating letter Sudoku value was used (e.g. positions (1,8) and (2,8) both had ‘A’). This demonstrated how changing only a couple of values in the letter Sudoku board can lead to a significant decrease in the board’s solution accuracy.



For Japanese sudoku, the incorrect 4x4 grid was expressed as ‘b6’: [[1,3,3,3], [3,1,4,1], [4,3,3,2], [2,3,1,4]]. By executing testBoard(), the expected success and failure of ‘b5’ and ‘b6’, respectively, was confirmed. Through more introspective analysis, accuracyBoard() printed that ‘b5’ had 100% accurate results while only 6 of the 16 values in ‘b6’ were accurate, leading to a 37.5% accuracy rate. As shown in the figures below, these results were corroborated by using viewAccuracyPlot() to highlight the grid positions with incorrect values in red. Unlike the other charts, most of the values in this plot were red instead of green. This was intentional as a variety of accuracy levels were intended to be successfully tested.



After evaluating the aforementioned project results, it can be ascertained that the testing, analysis, and visualization of the accuracy of correct and incorrect Sudoku boards for each Sudoku version may help Sudoku enthusiasts to better diagnose and debug their errors when attempting to solve their own Sudoku puzzles, leading to a more efficient process for tackling their Sudoku problems without being automatically given the answer to the Sudoku puzzle.

4. Conclusion

In this report, the writers documented their work with creating a Sudoku tester for the normal, letter, and Japanese versions of Sudoku and then implementing their code in a Python file. As evident, the code successfully met the project objectives of generating an empty Sudoku board, checking if certain values could be used in a box, updating the Sudoku board, testing if the filled Sudoku board was a valid solution, and analyzing the results using a percent accuracy and colorful visualization. From this project, both students learned a great deal about the domain of Sudoku, best practices in software engineering, and the importance of collaboration and communication. In particular, they learned how to combine their technical aptitude and interpersonal skills while completing their presentation slides and written report as well as assigning roles for completing project tasks. They also learned the importance of succinctly documenting their code using docstrings and comments. From a technical perspective, the students gained more proficiency with the Python3 programming language, the Google Colab

platform, and the creative process of designing and testing edge cases for each of their coded features.

While this project successfully executed the testing and analysis of a given Sudoku board, future investigations of this project should tackle expanding the Python code's functionality to generating novel unsolved Sudoku boards and then computationally solving the Sudoku boards (e.g. using a backtracking algorithmic technique). Another area to investigate is the scalability of the Sudoku tester to work with a Sudoku board of a variable number of dimensions (i.e. n x n) or a unique set of values (i.e. any range of integers, character, etc.). In turn, this Sudoku project could comprehensively provide an end-to-end opportunity for Sudoku enthusiasts and avid mathematical and computational scientists alike.

References

Boyer, C. (2006, May 24). 'Les ancêtres français du sudoku' *Pour La Science*

Easybrain LTD (2018, August) 'Sudoku Rules' <https://sudoku.com/sudoku-rules/>

Grabbe, J. (2017, January 2). 'Sudoku and Changes in Working Memory Performance for Older Adults and Younger Adults' *Activities, Adaptation, and Aging*

Moler, C. (2011). Experiments with MATLAB [E-book]. MathWorks. Retrieved July 5, 2022, from https://www.mathworks.com/moler/exm.html

Smith, D. (2005, May 14). 'So you thought sudoku came from the Land of the Rising Sun' *The Guardian*