Exploring the Sudoku Game via Depth-First Search on the Stack Data Structure

Abstract:

Sudoku is a ubiquitous game that highlights the beauty of simplicity, where endless combinations can be crafted from the 9 basic digits to fill in a grid where no digit is duplicated in any row, column, and mini-grid. Sudoku has not only attracted human players, but also challenged computer scientists in devising new algorithms for solving it. This project approaches different approaches for solving Sudoku puzzles and examines their efficiency. The solving process is fuelled by depth-first search, a recursive algorithm, on the Stack data structure, which is implemented on a Linked List. Depth-first search and stack are suitable for the Sudoku's game, because it accommodates efficient backtracking and attempting different values for every cell, one by one, instead of reforming the whole grid every time. My results show that the solving process of Sudoku will be optimized when the program starts out with filling empty cells in areas with the most pre-filled cells, as there is more information and fewer options, rather than attempting every cell in a chronicle order. This approach also simulates a human player's strategy for Sudoku solving – trying with the easy areas to reduce the number of viable options for the cells with less information.

Results:

In this project, experiments were run using 2 approaches: one is by filling the Sudoku board one by one, checking if the board is still valid after that filling, and refilling/backtracking if necessary, and the other is to locate empty cells with the fewest viable options left, start with filling those cells to reduce the number of options for the other cells, and move to cells with fewer options in a descending order. Both approaches were examined on solving Sudoku boards with a random number of initial (locked) values and contrasted in terms of speed for solving the board or determining if the board is unsolvable.

Firstly, the experiment was run on the basic method – solving the board by moving to every cell in a chronological order until all cells contain valid values – to test the functionality of the classes and the solving methods. This part of the experiment is run to solve a completely empty board of size 9x9, which contains 81 zero values. The puzzle is solved using a nested loop to move iteratively horizontally and vertically from cell of position [0,0] to position [80, 80] on the grid. The below image is a solved Sudoku board which had 0 locked values initially (all cells are 0):

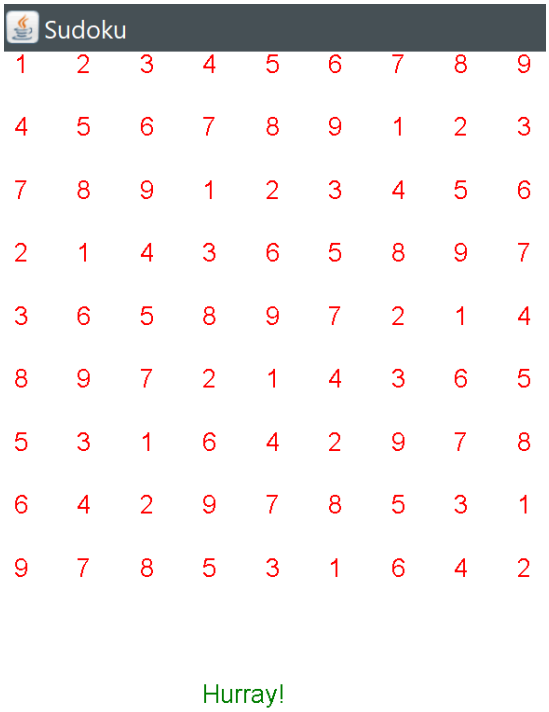

Figure 1. A solved Sudoku board which was initially empty (filled with all 0 values)

Applying the core rule sets of the Sudoku game, this solved board was manually verified to be a valid solution.

Next, I moved on to examining the effects of the board's initial settings on the solving speed of the Sudoku game. Specifically, I ran the simulation using the basic approach mentioned above on 3 different pre-written Sudoku grid boards, including the empty board above. All the boards were of size 9x9, but each of them had a different number of locked values – the number of values prescribed on the Sudoku board before solving. The difference of locked values between these boards were significant enough to provide insights about how this parameter impacted the time taken to solve a Sudoku game using the primitive approach.

The table below reports the simulation's results, including the number of locked values on each board and the time needed to solve them. All boards were of size 9x9, and the time was determined by the number of iterations run in order to solve the Sudoku game. Lastly, all the 3 boards were verified to be solvable (has at least one viable solution given the initial settings).

	Board 1	Board 2	Board 3
Number of locked values	10	20	0
Time/Number of steps/iterations	168	10960	391

Caption: The number of iterations needed to solve 3 Sudoku boards with starting numbers of locked values of 10, 20, and 0. Board 1 with 10 initially locked values yielded the fastest performance, while Board 2 took significantly much longer to be solved.

Board 1, which initially had 10 pre-filled (locked) values, yielded the best time efficiency for solving the Sudoku board – only 168 iterations. Meanwhile, Board 2, which had twice the number of prefilled values – 20, took significantly longer to be finished. The number of iterations for this board mounted up to 10960 iterations, nearly 70 times longer than for Board 1. Meanwhile, Board 3, with exactly 0 prefilled values, only took 391 iterations to get done, approximately twice the time for Board 1 and remarkably faster than Board 2.

This unexpectedly wide margin can be explained by the mechanism employed to solve the Sudoku puzzle. While a larger number of prefilled values does provide more information for solving the problem, the way the program solves it in a chronological order makes this prescribed information become a disadvantage. Specifically, when the board is filled from the start to the end, when it comes to area where there had been several pre-filled cells, there is a very high chance that the program will have to backtrack, because the options for filling cells in such areas are few and might have been used by some previous cells. For example, if there are 2 cells on the same row, one that appears sooner is not surrounded by any cells and can be filled with any value from 1 to 9, another that appears later and has already been surrounded in a way that 1 is its final option. Using the primitive approach, the former cell will take 1 and leave nothing for the latter cell, making the program backtrack multiple times until 1 is assigned to the latter cell.

The last part of this section provides a comparative analysis of 2 different approaches for solving the Sudoku problem: a primitive approach (whose results have been specified until now) and a human-inspired approach, which starts solving where there is the most information available. In this part only, the boards are generated randomly with a random number of locked values, ranging from 0 to 40. The two approaches will be compared and contrasted in terms of time efficiency - the time needed to solve the Sudoku or to determine that the game is not solvable. Each simulation has a threshold – it will “timeout” if the game is not solved/determined to be unsolvable after 100 iterations.

The table below reports the time efficiency of two 2 approaches in solving the 7 Sudoku problems. The numbers of locked values are integers ranging from 0 to 40, all the boards are of size 9x9, and the program will be interrupted if there is no answer after 100 iterations.

	3	17	19	31	14	7	39
Human-inspired	83	Timeout	88	Unsolvable (after 3 steps)	Timeout	74	Unsolvable (after 1 step)
Primitive	Timeout	90	Timeout	Unsolvable (after 67 steps)	Timeout	Timeout	Unsolvable (after 33 steps)

Caption: The number of iterations needed to solve or determine the solvability of 7 Sudoku puzzles with numbers of locked values ranging from 0 to 40. A simulation is considered “timeout” if it cannot solve the Sudoku or determine the solvability of the Sudoku after 100 iterations. It is evident from the table that the human-inspired approach reduces the rate of timeout and the time of solving the puzzles in most cases.

The human-inspired approach outdid the primitive approach in most cases. The number of timeout instances for each approach is 2 and 4 for each approach, respectively. Even in cases where both approaches could yield a result, the finish time of the human-inspired approach is also more impressive than the primitive approach. In the 2 cases where both approaches determine the board is unsolvable, the human-inspired only took very few steps (1 and 3), while the primitive took much

more (33 and 67). This is mainly due to the fact that the human-inspired approach explores the board from the densest areas, so when the number of pre-filled cells is large enough, this approach can quickly determine if the board is unsolvable.

From this table, we can also infer the relationship between the number of locked values and the time efficiency of solving the Sudoku problem. Specifically, when the number of locked cells is less than 10, the human-inspired approach is highly likely able to solve the Sudoku within 100 steps. However, when the number locked values was raised to between 10 and 20, both methods experienced timeout. However, when the number of locked values is large enough, the Sudoku is more likely unsolvable, and this can be determined within 100 steps using both methods.

Extensions:

Extension 2: The purpose of this extension is to create tester files for the classes associated with the Sudoku game.

CellTests:

CellTests is a class including the test cases for the Cell class. The CellTests is important because the whole game is built upon cells, so it is necessary to ensure it functions correctly. The CellTests class include these following test cases:

- Test the constructors and the associated getters: Create two cells with assigned positions (rows and columns) and values. Print the fields of each Cell and assert that the values attained by the getters equal their assigned values. Failure of this assertion indicates errors in the constructor of the Cell class or its associated getters.
- Test the setters: Create a Cell with prescribed position and value. Use the setters to play around with the fields of the Cell, including setting a new “locked” status and assigning a new value. Print the value of those fields before and after the setter is executed to track their changes. Lastly, assert the values attained by the getters are the newly set values; if this fails, there are either problems with the getters or the setters.

To run the CellTests, one needs to compile the CellTests class and enable assertions (-ea) for the class. In other words, the execution requires these 2 commands inputted to the Terminal:

- javac CellTests.java
- java -ea CellTests

Upon execution, the tester class will output values at designed positions, including the row, column, value, and the locked status of the cell initially and after the setters are applied, and error if any method has an unexpected behavior.

BoardTests:

BoardTests is a class including the test cases for the Board class. The BoardTests class include these following test cases:

- Test the constructors: The Board class has 3 constructors – one for an empty board, one for a specific number of locked values, and one for assigning a pre-written board. This block of code of the tester tests all these three constructors. For each board, the program calls the constructors and uses the getters to test if the fields of the boards are similar to the assigned

values. The tested fields are the number of rows, number of columns, and the number of locked cells.

- Test the getters: Apart from the basic getters for the number of rows, columns, and locked values, the Board class also has other getters that access specifically certain cells on the board, including their position, value, and locked status. This part of the tester file tests these methods by asserting the obtained values using these getters to the real value of cells on a prewritten board(board 1) and an empty board.
- Test the setters: the Board class has three different setters for its cells. This block of code of the BoardTests file examines all these 3 methods. Using the getter to access specific cells, printing out their initial conditions (value and status of locked), executing the setters, printing out new conditions, and implementing the assertions, the setter methods are tested. Notably, when testing the setter that changes the locked status of a cell, the program also keeps track of the total number of locked values of the whole Board.
- Test the validValue() method: Test the validValue() on an empty Board (“board3.txt”) and a solved Board (“solved.txt”). For the empty Board, the program runs a nested loop to iterate through every Cell and repeatedly asserts that it is not validValue(). If the method is inaccurate, the assertion will break the loop and print an error. Contrariwise, on the solved Board, the program constantly asserts that every Cell’s value is validValue().
- Test the validSolution() method: Test the validSolution() method. First, assert the method on the solved Board. Then, randomly set the value of some Cells on the solved Board to zero and add another assertion that the Board is not validSolution(). If the program errors in any of these 2 assertions, there are problems with the validSolution() method.

To run the BoardTests, one needs to compile the BoardTests class and enable assertions (-ea) for the class. In other words, the execution requires these 2 commands inputted to the Terminal:

- javac BoardTests.java
- java -ea BoardTests

Extension 3:

The purpose of this extension is to generalize the Sudoku game into a game that can be played on boards with multiple sizes. The only requirement for the game to be valid (playable) is that the size, or the length of each side of the grid, must be a square number.

Below is an example of the Sudoku game played on a 16x16 board. The board initially contains 45 locked values, ranging from 1 to 16.

```
15 0 0 0 0 0 0 0 14 0 0 0 0 0 0 0
0 0 0 11 0 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0 15 0 0 0 0 0 3 0 0 6 0
0 0 0 0 14 0 0 0 0 0 13 0 0 0 0 0
11 0 0 0 5 0 0 0 0 0 7 0 9 0 0 0
0 0 0 0 6 2 0 0 9 0 0 0 0 7 0 0
0 0 0 0 0 0 0 0 0 2 0 0 0 6 0 11
2 0 0 0 0 0 15 0 0 0 0 10 0 0 0 0
0 0 0 0 0 0 0 0 0 0 3 0 0 9 0 0
7 0 0 16 0 0 0 0 0 0 0 0 0 11 0 0
0 0 0 0 0 0 13 0 0 0 0 0 0 8 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 15 0
0 0 0 13 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0 0 15 0 0 0 4 0 8
0 0 0 0 8 0 14 12 4 0 5 0 0 2 0 0
8 0 0 0 9 0 0 0 0 0 0 0 0 0 0 13
```

```
15 3 6 1 2 10 16 9 14 7 11 12 8 5 13 4
10 8 13 11 3 4 12 6 1 5 9 2 14 15 16 7
9 14 16 2 7 15 5 13 8 10 4 3 11 12 6 1
4 7 12 5 14 1 8 11 15 6 13 16 2 10 3 9
11 6 4 15 5 14 1 8 3 12 7 13 9 16 10 2
13 16 5 8 6 2 3 10 9 14 1 11 4 7 12 15
12 1 3 10 13 9 4 7 16 2 8 15 5 6 14 11
2 9 7 14 11 12 15 16 5 4 6 10 1 13 8 3
14 12 11 4 15 8 10 2 13 16 3 5 7 9 1 6
7 13 8 16 12 3 9 4 6 1 15 14 10 11 2 5
5 15 1 6 16 11 13 14 2 9 10 7 3 8 4 12
3 2 10 9 1 7 6 5 11 8 12 4 13 14 15 16
6 10 2 13 4 5 11 15 7 3 16 8 12 1 9 14
16 5 9 7 10 13 2 3 12 15 14 1 6 4 11 8
1 11 15 3 8 6 14 12 4 13 5 9 16 2 7 10
8 4 14 12 9 16 7 1 10 11 2 6 15 3 5 13
```

Figure 2. a) A 16x16 Sudoku board with 45 locked values, b) The solved 16x16 Sudoku board

In order to simulate a Sudoku game of any size, one needs to compile and run the Exploration class (human-inspired approach) or the Sudoku class (primitive approach). No matter which class is chosen, the input argument to the Terminal must be of length 2, in which:

- The first argument is the size of the Sudoku board.
- The second argument is the number of locked values, ranging from 0 to the squared value of the size.

Specific syntaxes for this purpose can be found in the README.txt file of the “Extension 5 Project 3” directory. Also, in the main function of the Sudoku class and the Exploration class, there are also equivalent guidelines.

Acknowledgements:

I completed this project on my own.