CSC 403 Data Structures II
Term Project Documentation
Trevor Siegler
November 10, 2024

SudokuSolver.java

Summary

The Sudoku Solver application takes a puzzle in the format of a two-dimensional array of integers and solves it if it is a valid puzzle. The application uses a brute-force solution approach by testing every possible answer for each cell and backtracks with a Stack until a solution or no solution is found. It stores the answers for each cell in HashSet constraints for rows, columns, and boxes. This allows for constant-time inserts, searches, and removals while solving and backtracking. Additionally, the application allows the user to enter custom puzzles or it can generate a random unique puzzle for the user (meaning it has only one solution).

User Documentation and Instructions for Developers

The SudokuSolver.java has to be inside of the java folder in algs4 to access the Stack class

solvePuzzle() Section

The application follows a simple flow of functions to solve the puzzle if there is a valid solution (the public function **solvePuzzle(int[][] board)** for example). The flow chart of the function:

Puzzle Input → Initialize Variables (HashSet) → Valid Starting Board → Solve Puzzle → Valid Solution

The **solvePuzzle** function takes a two-dimensional array of integers as an argument for the starting puzzle or board. The next step is to initialize the HashSet constraints to the size of the board. This is a critical step at the beginning because we use the HashSet to check and update the constraints of the puzzle. After we initialize the HashSet, the application continues to the next step, which is checking if the starting puzzle is valid.

The function **checkIfValidStartBoard(int[][] board)** receives the board as the argument and ensures the puzzle is valid. It will throw errors if there are any invalid

lengths, numbers, or any numbers that do not follow the constraints. If there are no errors thrown, the HashSet constraints are updated with the values of the starting board. This is also a critical step because this function updates the global HashSet constraints with the starting numbers that the next function uses.

The next function, **solveBoard(int[][] board)**, takes the valid board as input and works on the mutable array to solve it. It starts by declaring and initializing a local Stack, which will store two-dimensional positions of cells that were answered in order. This allows for quick and easy backtracking if we run into a cell that has no answer. The function also declares and initializes a couple of variables: an integer that keeps track of the starting number of a cell (int startingNumber) and an array of integers that keeps track of the board position (int[] rowCol). The function uses two helper functions to determine if a valid solution has been found or if it needs to backtrack due to an impasse.

The first helper function, **findNextZero(int[][] board, int startRow, int startCol)**, receives the board, row, and column as arguments and returns the position of the next empty cell as an array of integers. The goal of the helper function is to find the next empty cell (0) and return its position to the **solveBoard** function. Providing the row and column location as arguments allows the helper function to be more efficient by not checking previous cells since the application solves the puzzle in order. If there are no more empty cells, the function returns null, so the application has to check for null in the **solveBoard** function.

The second helper function, **findAnswer(int row, int col, int startNum)**, receives the row, column, and starting number as arguments and returns an integer. The helper function uses the location of the row and column to find a possible answer that complies with the HashSet constraints. It checks the numbers (startNum + 1) through 9 in a for loop and returns the answer if it's not within the set. Starting at (startNum + 1), rather than 1, allows the application to avoid getting stuck in an endless loop and try the next solution for the available cell. If no solution is found, the function returns 0, so the **solveBoard** function must check for 0.

Returning to the **solveBoard** function, it uses these two functions within a while loop with backtracking until a solution is found or no solution is found for the board. The while loop only breaks if **findNextZero** returns null (indicating the puzzle is solved with no empty cells remaining) or if the backtracking Stack is empty when attempting to pop, which means all possible solutions have been exhausted without finding a valid answer. After all of this, the last step is to check if the solution is valid or if the puzzle did not have any solutions.

The last function is **checkIfValidSolution(int[][] board)**, which receives the solved board as the argument and returns true if valid or false if not valid. This function is similar to **checkIfValidStartBoard**. It clears and empties the HashSet constraints and goes through the entire board, adding the answers to the HashSet while checking for

any 0s on the board (empty cells). After this, it goes through all of the HashSet constraints to verify that each contains the numbers 1 through 9. If it passes all the tests, it returns true. If it fails any of the tests, it returns false. It is critical that the user checks whether the function returns true or false. If the function returns true, the solution to the puzzle has been found, and the user has the solved puzzle on the mutable board array that they input at the beginning of the **solvePuzzle** function.

Time Complexity:

The **solvePuzzle** function uses a brute-force approach to solving, which means the time complexity for a worse case scenario: O(N^((N*N) - K)). Where N is the board's dimension and K is pre-filled cells with answers.

Space Complexity:

The space complexity of this function is primarily due to the Stack used for backtracking and the HashSet constraints for rows, columns, and boxes. Each HashSet stores up to N elements, where N is the board's dimension. And there are 3*N HashSets (N rows + N columns + N boxes), which is 3N * N = 3N^2 = O(N^2). As for the Stack, in the worst case scenario, the Stack will hold up to N^2 positions which is O(N^2).

generateRandomPuzzle() Section

The next section focuses on the public function, **generateRandomPuzzle()**, which returns a random unique starting board or puzzle as a two dimensional array of integers. This next part will not be going in as in-depth as the previous section, but will gloss over the important parts of the function. The flowchart of the function:

Initialize Board and Variables → Fill Random Cell with Random Possible Answer → Count Solutions for Board:

→ 2 Solutions = Loop back to Fill Random Cell

→ 1 Solution = Unique Solution / Finished

→ 0 Solutions = Remove recent answer and push position to the failsafe stack

The function starts by initializing a board to an empty two-dimensional array of integers. It also initializes an empty Stack of integers that will be used for a failsafe in case our randomization results in a no solution puzzle.

The next part is the randomization for the position of the board and answers. For this part we use a few helper functions: **shuffleBoardHelper()**, **findRandomCell(List<Integer> indices)**, and **findRandomAnswer(int row, int col)**.

shuffleBoardHelper creates a List of integers that will hold all of the indexes or positions of the board. It uses an ArrayList because it's going to need access to the shuffle function for randomization and removal function for removing random indices. The shuffled List is returned and is used by **findRandomCell**, which takes the List and grabs the value at the random index between 0 and the length. It then returns that value(board position) and removes the index from the List. Lastly, **findRandomAnswer** creates another List of Integers for all of the possible answers for that position and either returns 0 (if there are no possible answers) or a random possible answer.

Next, the application starts the while loop and updates the HashSets with **checkIfValidBoard**. If **findRandomAnswer** returned a possible answer then we add them to the HashSets and then send the updated board to be solved in the function **countUniqueSolutions(int[][] board)**. This function is very similar to **solveBoard**, but it has a few key differences that are necessary to understanding how the function works.

The first key difference is that the function returns an integer, which is the solution count for the current puzzle. Instead of breaking out of the while loop once it finds a solution, it increments the count and continues trying to solve the puzzle as if it did not find a solution. The solution count is capped at two, so if there are two solutions found, it stops and returns 2.

The second key difference is that, because the function has to try every possible solution, it needs a more efficient way of solving the puzzle. This function is called every time a new number is added, so it needs to be more efficient in recognizing a pathway that is not going to lead to the answer. So, instead of using **findNextZero**, it uses **findNextZeroMRV(int[][] board)**. This function takes only the board as an argument and returns an array of integers representing the position. Instead of returning the first empty cell, it returns the empty cell with the minimum remaining values for answers. If there is an empty cell with no possible answers, it returns that cell so the function can instantly backtrack. However, this means every time this function is called, the whole board is scanned and every empty cell has its possible answers counted. This may seem inefficient, but it saves a lot more time because it catches the backtrack earlier.

Returning to the main function, when the solution count is equal to 2, we loop again and another random answer to a random cell. If the solution count is equal to 0, remove the previous answer and push the position of the cell to the failsafe Stack. This Stack is used if the random index List becomes empty because there are instances of the random cells chosen that are a bad sequence and there are no unique solutions. So, if the List becomes empty, it pops all of the failed positions back into the List and continues the loop. Once the solution count is equal to 1, the random board is finished; it exits the loop and returns the board.