**Project Summary**

In this week’s project, we implemented array-based stacks to explore the game of Sudoku. This method is called depth-first research algorithm.

**Task Solutions**

**Board Class:** building on the board class that I created during lab, I first created a field for it that is 9\*9 2D array of Cells, each of which holds the value of 0 when they are initialized. Then I wrote the method toString () to print the board that is divided into 3\*3 blocks into the terminal. I then added more utility methods that helps me access the information of Cells on board given its location.

* I then updated the read method and asks it to keep reading lines until it reaches the end of the document and set the numbers in each line to each row on the board.
* For the validValue() method, I first checked if the value is in the bound of [1, 9]. Then I used two for loops to check if the value is valid in its column and row by looping through every element in the row/column but skipping the position itself.

Then I checked if the value is unique in its 3\*3 block. I first locate the 3\*3 block given the location of a cell using integer division (as shown below).

* Finally, I wrote the validSolution() method that returns true if the board is solved, i.e. every number on the board is valid in its position and there’s no 0’s. Otherwise the method will return false.

**Sudoku Class**: the Sudoku class implements the board and solves the puzzle. The class first initiates a board of 0’s.

* + The second constructor takes in a parameter that let the user set how many initial values are there on the board. When I was testing this second instructor in the main method, I found that when the number of initial values is more than a certain value (in my trials, 70), the board will take much longer time to initialize, due to the increasing time in finding valid values for each cell.
  + Solve() method took at the initial values on the board and loop through each cell to try out possible values for them. The algorithm is fairly time-consuming, because for every location, it tries values starting from 1 to 9.

The method includes a big while loop that keeps the program proceed to the next cell. For an unlocked cell, it starts looping from 1 to 9 and breaks the loop if one of them works. If there’s no solution for the cell at this location, it backs up to the previous cell. The method will return false if there’s no cell in the stack to pop and return true if it eventually finds a solution for the board.

* + I tested the solve() method by testing it on a board of 0’s, and found that the steps taken is always 701 with the same outcome, as shown below.

When I tested it on board 10, it took 197 steps and the outcome is shown below.

**LandscapeDisplay Class**: I first created the draw method in both Cell class and Board class that prints out the values as strings. To make the presentation of the Sudoku better, I divided the board into 3\*3 blocks by drawing lines. I also set each number a different color. Below is a picture of my first solved board.

**Explorations:**

1. What is the relationship between the number of randomly selected (but valid) initial values and the likelihood of finding a solution for the board?

I run 5 boards with initial values of 0, 10, 20, 30, 40 and 50, and here is the result that shows the number of boards solved.

I found out that when the number of initial values is more than 25, the possibility of solving it decrease drastically.

1. Is there a relationship between the time taken to solve a board and the number of initial values?

Below is the snippet of the time taken and the number of initial values in 5 trials.

I found that the time taken for a board of 0’s is always 701, and but a board of 5 values actually takes less time to solve on average. When the number keeps increasing, the time taken increases drastically.

However, 5 simulation not enough to draw conclusions of the relationship between the number of initial values and the time taken. In extension, I tried to simulate a large number of boards to calculate the percentage of solved boards.

**Extensions**

1. I found that giving each number a different color doesn’t help the user to locate locked values, so I set the color of locked values into red, and the rest being black, as shown below.
2. In Sudoku class, when I tried to run 5 boards to test the relationship between the number of initial values and possibility of having a solution, instead of creating 5 Sudoku objects one by one, I created an ArrayList with the length of 5 and pushed 5 Sudoku references. Then I looped through each Sudoku object and try to solve them to get the number of solvable games.
3. I updated the read method in board class, enabling it to handle the board 20, which has space lines and spaces at the beginning of the lines. In the while loop of reading lines, I added an if statement to test if the line is blank ( line.equalsIgnoreCase(“”) ), as shown below.

I also called the trim method of line to trim off the spaces at the beginning of lines.

1. I created a Sudoku game called UserSudoku, which is a child class of Sudoku and allows the user to enter value for cells on the board. It follows the same algorithm. It will skip the locked values and pop up a JoptionPane that asks for user input of value for the cell at a position. Then the program will check if the input is valid, and will ask the user to try again if it is not by returning to the start of the while loop. The cell will be set to the valid input value and pushed into a stack. The value on the Sudoku will also be updated. Below is the snippet of the codes and a screenshot of the program.

The program also allows the user to type “/”, which will pop the stack of cells back up to the previous value that they chose to enter previously. It basically follows the same algorithm as the solve() method when the program needs to back up to previous cell. An error message will pop up if there’s no more cell to pop in the stack, as shown below.

However, I find although this program allows more user interaction, but it complicates the process because it doesn’t allow users to choose which cell to put value but force them to go through each cell in order. So it will take a very long time before the Sudoku is solve.

1. Then I created a Simulation class which is also a child class of the Sudoku class. The purpose of this class is to create a lot of boards simultaneously (i.e. 100). To stop the program from displaying 100 Sudoku windows as they do in the parent class, I created a boolean field in the parent class constructor decides if the LandscapeDisplay object gets created and if the program needs to print out results.

The program asks for user input of how many initial values they wish to create in the simulation and the number of boards to be created, and finally calculate the percentage of solvable boards. Here is the result I get by running 50 boards with 10 initial values.

I also tried to run 100 boards with 10 initial values, but the program took more than 20 minutes to run so I terminated it. I think that when the number of initial values is more than 0, the possibility that the board is solvable is rather random, because when I run 50 boards with 10 initial values, the second trial took much longer than the first time, but the result is always 100%.

**Conclusion**

This project helps me better understand the concept of stack and the logic behind a depth-first search, which appears to be effective but time-consuming in the context of solving a Sudoku. Understanding the algorithm behind the solve() method has been a challenge, but very helpful for me to understand while loops and stacks.

Credits:

Professor Maxell and his lecture notes

Roujia

Enzo