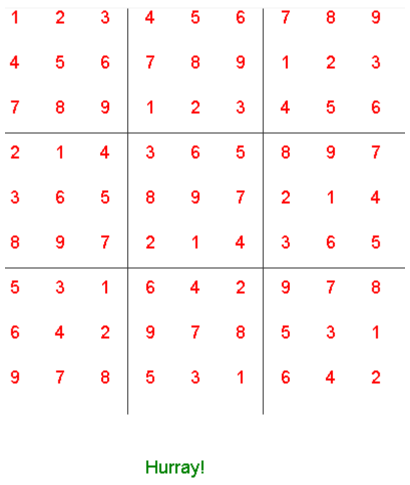
CS231 Project 4: Sudoku

October 3rd, 2023

Abstract

This project is meant to teach CS Students how to write a stack-based solver for the game that implements a depth-first search algorithm. The focus of this exploration lies in the context of Sudoku puzzles, a widely recognized game, where the solver navigates through different complexities based on the number of given starting values. By manipulating the initial values, students gain insights into how these alterations impact the search process, enhancing their understanding of stack-based solving techniques and their application in real-world problem-solving scenarios.

Results

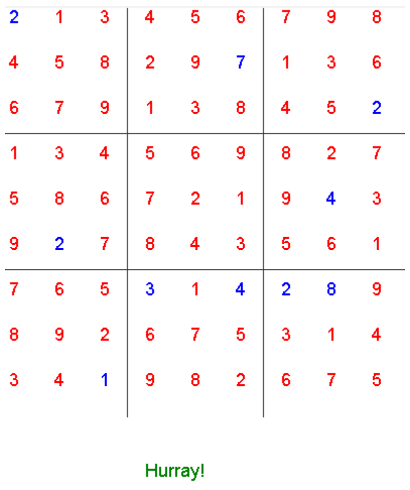


1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

Hurray!

Solved Board with No Initial Values

This is a solved Sudoku board with no predetermined values put into place. This approximately took 6 seconds to solve.



2	1	3	4	5	6	7	9	8
4	5	8	2	9	7	1	3	6
6	7	9	1	3	8	4	5	2
1	3	4	5	6	9	8	2	7
5	8	6	7	2	1	9	4	3
9	2	7	8	4	3	5	6	1
7	6	5	3	1	4	2	8	9
8	9	2	6	7	5	3	1	4
3	4	1	9	8	2	6	7	5

Hurray!

Solved Board with 10 Initial Values

This is a solved Sudoku board with 10 predetermined values put before it was solved. They are indicated by blue numbers on the board. This approximately took 2 seconds to solve.

Discussion

Overall, looking at the results of Sudoku and watching its process of solving what was given, they do make sense and explain how the game plays. Backtracking is the most common method used to solve Sudoku puzzles. It's a recursive depth-first search algorithm that tries out different numbers for each empty cell, backtracks when a contradiction is encountered, and explores other possibilities. It starts with an empty cell, tries placing a number in it and checks if the placement is valid. If yes, move to the next empty cell and repeat the process recursively. If a contradiction is found (same number in the row, column, or subgrid), it backtracks to the previous step and tries a different number. This process is repeated until the entire puzzle is filled.

Exploration

What is the relationship between the number of randomly selected (but valid) initial values and the likelihood of finding a solution for the board? For example, if you run 5 boards for each case of 10, 20, 30, and 40 initial values, how many of each case have a solution?

I ran 5 boards for each case of 10, 20, 30 and 40 initial values, and the results can be displayed here:

10	5/5
20	2/5
30	0/5
40	0/5

Judging from the data, it seems like the more predetermined values there are, the less solutions are possible. This might be because when you have more initial values in a Sudoku puzzle, it provides more constraints. These constraints, as compared to a puzzle with little to no initial values limit how much an algorithm can backtrack or search through.

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

Yes, as shown from the results above, Sudoku solving algorithms, like backtracking, rely on constraint propagation. The initial given values provide the initial constraints that the algorithm uses to determine possible values for empty cells. More initial values mean more constraints, which can significantly reduce the search space. The algorithm has to explore fewer possibilities, making it faster to find the solution. When there are more initial values, it is easier to determine the possible values for the empty cells. With each additional given value, the possible values for neighboring cells are immediately constrained. In contrast, with fewer initial values, there are more possibilities to consider for each empty cell. This increased uncertainty leads to a more complex search process, making it take longer to find the solution. When there exists initial values above 30, the board takes less time to solve because it quickly realizes that there is no solution.

Initial Values	10	20	30	40
Time (nanoseconds)	5696021417 (10 digits)	1045179125 (10 digits)	676501542 (9 digits)	226615750 (9 digits)

Reflection

Why did we implement a Stack in this project? Why was this an efficient data interface for this specific project?

The results from using a stack-based solver to implement a search algorithm does make sense, as when there exists more fixed values on a board, it is more difficult to find a solution since the search requires finding valid values that will suffice the starting values while also not repeating them. With little room to choose values, it causes the complexity of the search to increase, resulting in more "no solutions" than finding solutions. On the other hand, less fixed values gives the search more chances in finding valid values for a solution, and reaching a solution.

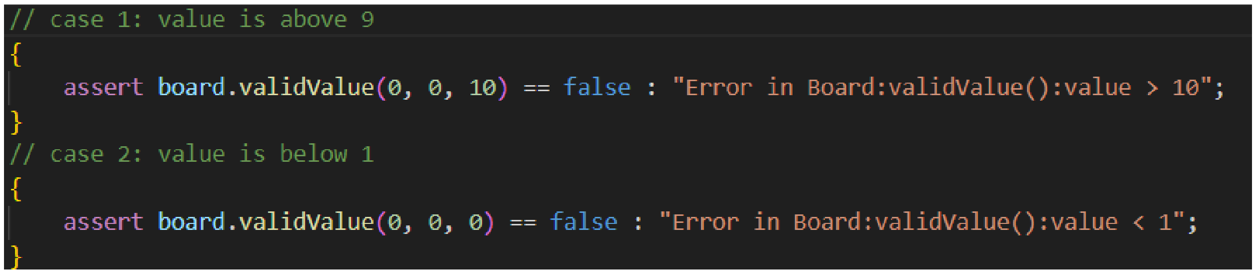
Extensions

Board Tester Files

The majority of the methods in Board were tested in the main method, but for ones such as read(), validValue(), and validSolution(), I made a separate class to test them.

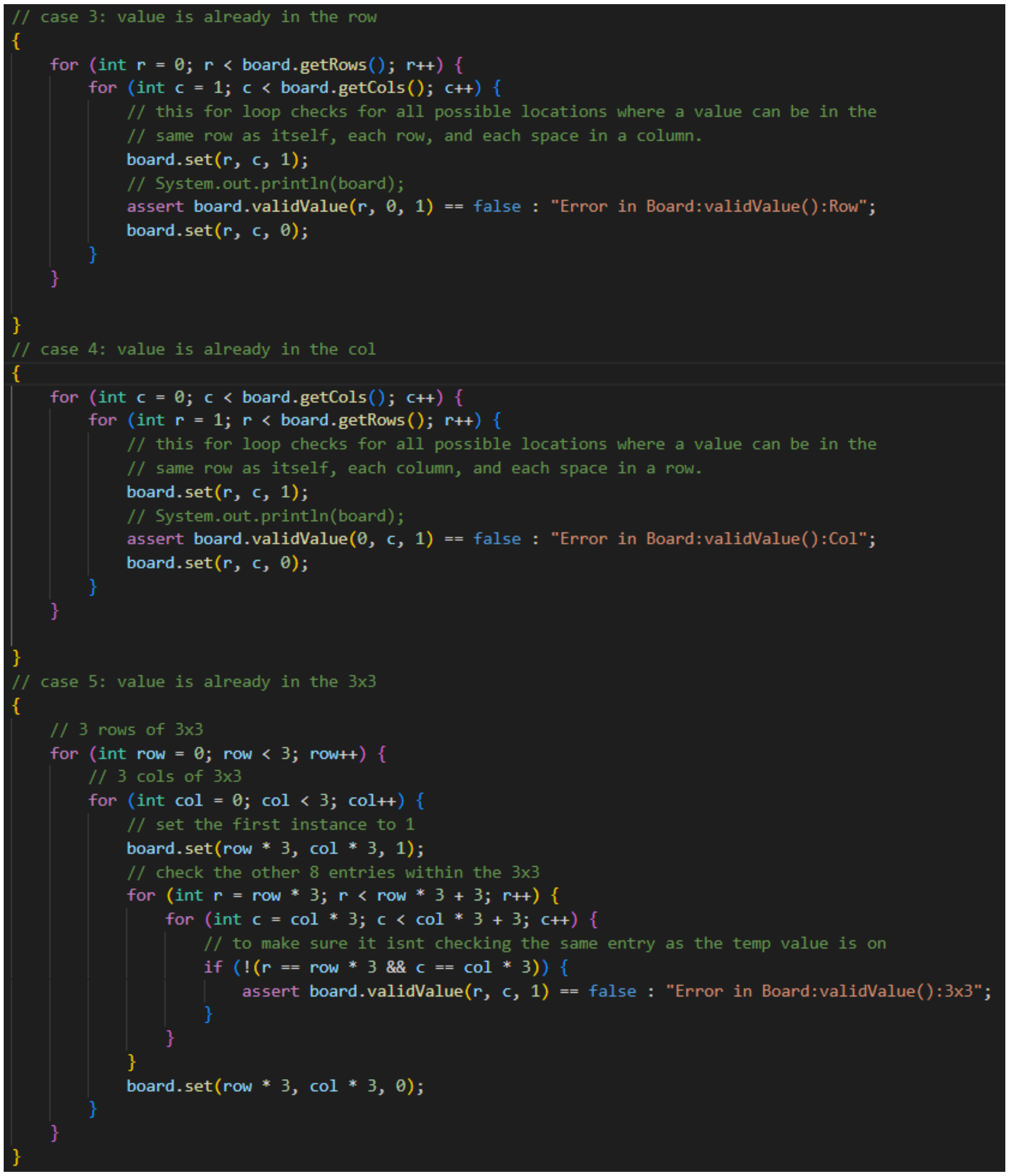
read() had its own print statement, so to know if it worked I just needed it to run correctly and print all of its statements within it to check.

To check if a Value was valid, the value has to be both between 1-9, and unique in its row, in its column, and in its local 3x3 square.



```java
// case 1: value is above 9
{
    assert board.validValue(0, 0, 10) == false : "Error in Board:validValue():value > 10";
}
// case 2: value is below 1
{
    assert board.validValue(0, 0, 0) == false : "Error in Board:validValue():value < 1";
}
```

Although testing if a value was out of bounds was fairly straightforward, I wanted to take extra care towards making sure checking if it was unique in its row, column and local 3x3 square.



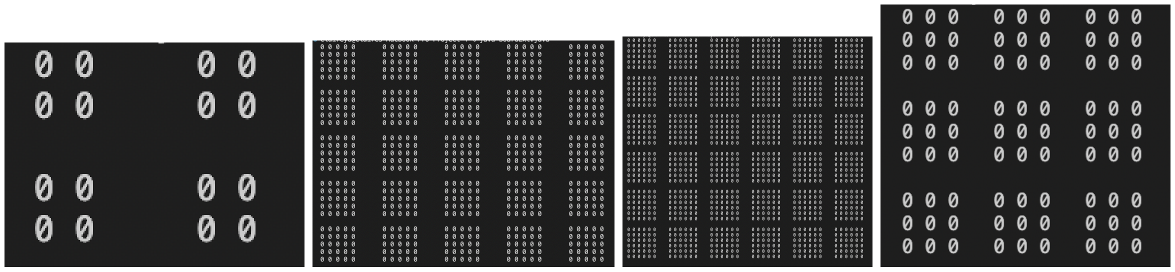
```java
// case 3: value is already in the row
{
    for (int r = 0; r < board.getRows(); r++) {
        for (int c = 1; c < board.getCols(); c++) {
            // this for loop checks for all possible locations where a value can be in the
            // same row as itself, each row, and each space in a column.
            board.set(r, c, 1);
            // System.out.println(board);
            assert board.validValue(r, 0, 1) == false : "Error in Board:validValue():Row";
            board.set(r, c, 0);
        }
    }

}
// case 4: value is already in the col
{
    for (int c = 0; c < board.getCols(); c++) {
        for (int r = 1; r < board.getRows(); r++) {
            // this for loop checks for all possible locations where a value can be in the
            // same row as itself, each column, and each space in a row.
            board.set(r, c, 1);
            // System.out.println(board);
            assert board.validValue(0, c, 1) == false : "Error in Board:validValue():Col";
            board.set(r, c, 0);
        }
    }

}
// case 5: value is already in the 3x3
{
    // 3 rows of 3x3
    for (int row = 0; row < 3; row++) {
        // 3 cols of 3x3
        for (int col = 0; col < 3; col++) {
            // set the first instance to 1
            board.set(row * 3, col * 3, 1);
            // check the other 8 entries within the 3x3
            for (int r = row * 3; r < row * 3 + 3; r++) {
                for (int c = col * 3; c < col * 3 + 3; c++) {
                    // to make sure it isnt checking the same entry as the temp value is on
                    if (!(r == row * 3 && c == col * 3)) {
                        assert board.validValue(r, c, 1) == false : "Error in Board:validValue():3x3";
                    }
                }
            }
            board.set(row * 3, col * 3, 0);
        }
    }
}
```

These 3 methods check the remaining 8 spots that a value could possibly repeat itself compared to its initial value.

Grid Sizes

Since the game Sudoku is generally played in a 9x9 grid, I wanted to extend the project by using size as a parameter to create different sizes of grids. Having size as a parameter, I could determine how performance depends on the parameter size, as well as find the minimum number of fixed cells for there to reach no solution.

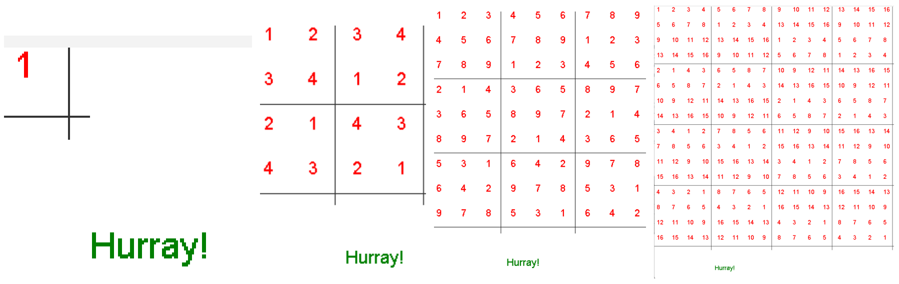
Since I had a lot of fixed numbers in my code to create the 9x9 grid, I decided to create new classes to construct other sizes of grids. (These are all labeled FLEX) Having created these classes, I then changed methods that had fixed numbers and rather replaced them with the SIZE parameter. I also updated the draw method of Cell so that it does not take in symbols/letters. After changing these methods, I could produce different sizes of boards.

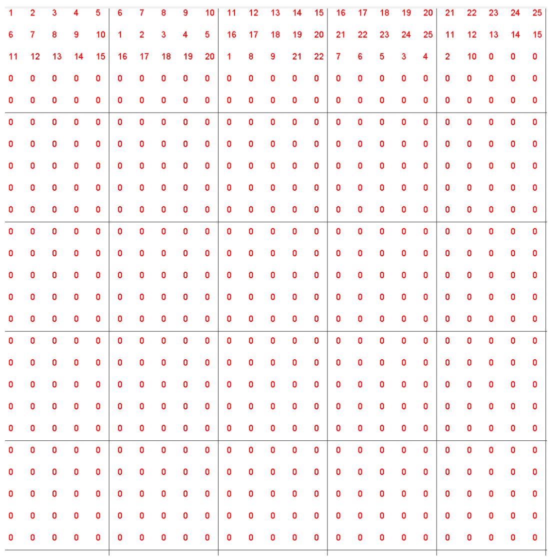


```
0 0    0 0
0 0    0 0

0 0    0 0
0 0    0 0
```

```
0 0 0   0 0 0   0 0 0
0 0 0   0 0 0   0 0 0
0 0 0   0 0 0   0 0 0

0 0 0   0 0 0   0 0 0
0 0 0   0 0 0   0 0 0
0 0 0   0 0 0   0 0 0

0 0 0   0 0 0   0 0 0
0 0 0   0 0 0   0 0 0
0 0 0   0 0 0   0 0 0
```

I also changed the LandscapeDisplay so that there would be lines separating the boxes appropriately.





1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
6	7	8	9	10	1	2	3	4	5	16	17	18	19	20	21	22	23	24	25	11	12	13	14	15
11	12	13	14	15	16	17	18	19	20	1	8	9	21	22	7	6	5	3	4	2	10	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(The 25x25 takes an extremely long time to run)

Performance depending on size

When there is an increase in the parameter size, the performance of the solve method greatly diminishes, as it takes longer and is less efficient in finding a solution. This could be explained by how when the board becomes increasingly bigger, it is able to accept a greater range of values, and therefore, having more tests to do to find the correct value at a certain location. With the increase in size, it becomes more difficult for the board to find a solution, needing to backtrack various times and causing a longer runtime (where I have yet to reach a solved board above 16x16). Unlike a smaller grid, it is easier to determine if there will be a solution or not since there are not that many values that need to be checked. To be specific, a 4x4 board has only 16 cells, while a 36x36 board has 1296 cells, which also shows how many possibilities of numbers there are in comparison. Furthermore, when creating these boards, it is easy to identify why most Sudoku games are played on a 9x9 grid. This grid allows room to be easily solved but also presents some difficulty.

References/Acknowledgements

I asked a TA for help troubleshooting since I made some mistakes while typing some of my logic out. Despite a bunch of painful head scratching undertaken to locate mistakes, we eventually solved the issues.
