
CS231: Project 3

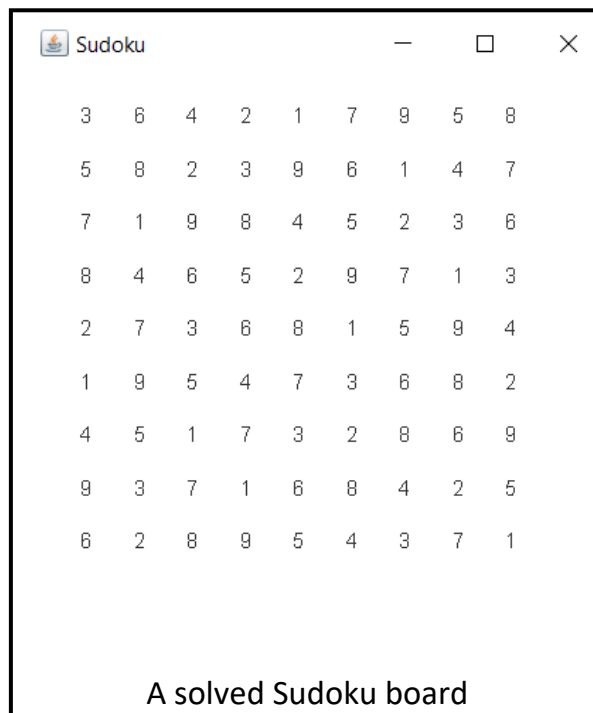
Sudoku

Parth Parth | CS231L-A | Dr. Alan Harper

Abstract

In the project, a Sudoku solver has been coded. The current project reads from the *test.txt* file and prints the solution out to the terminal, the solved board. In the meantime, the GUI window shows how the solver progresses step by step. Then, I have explored how long the board will take to solve when it starts with different numbers of locked values.

The data structure that has been used to implement this solver is a stack (same a vertical stack of books). This data structure works on the principle of LIFO which stands for Last In First Out. The stack saves all the cells that the program has attempted to solve with their values and is helpful in easily manipulating (adding and removing at the top) the Cells and their values to go through them until a solution is found.



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

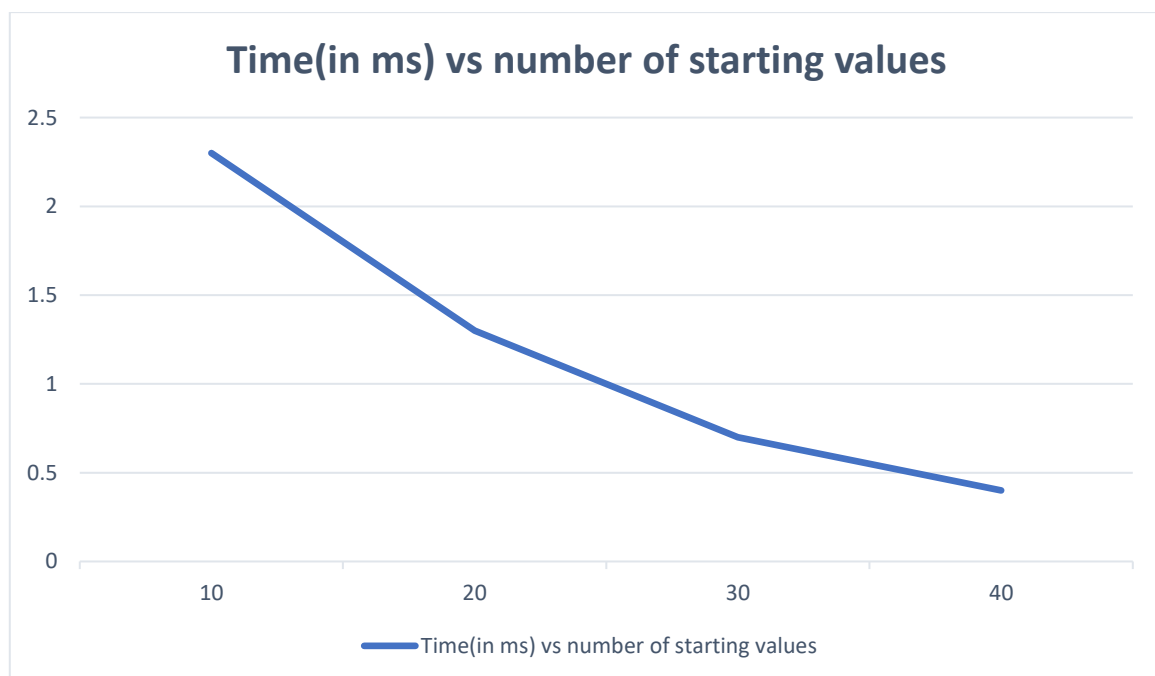
A solved Sudoku board

Exploration

1. Values for different numbers of starting values are tabulated below:

	10 values	20 values	30 values	40 values
1.	5	3	1	2
2.	9	2	0	0
3.	0	0	2	1
4.	1	1	1	0
5.	3	0	0	0
6.	1	2	1	0
7.	1	4	2	0
8.	0	0	0	0
9.	1	0	0	0
10.	2	1	0	1
Average	2.3	1.3	0.7	0.4

2. Plotting the data, we get:



As such, it is evident that as the number of starting values goes up, the time taken to solve the board goes down.

Extension 0

My zeroth extension is incorporated into the main project itself. In this extension, I have changed the *nextBestCell()* method to look for a Cell by searching the grid for the cell with the least number of possible values and returns the first cell with the least possible values. This happens by design as the following code shows:

```
private Cell nextBestCell() {
    Cell best=null;
    int numOfSolutions=9;

    for(int row=0; row<this.game.getRows(); row++) {
        for(int col=0; col<this.game.getCols(); col++) {
            // retrieve Cell at row,col
            Cell temp=this.game.get(row, col);

            // if Cell is locked or if it is already filled, skip Cell
            if(temp.isLocked() || temp.getValue()!=0) {
                continue;
            }

            // count number of solutions
            int tempNumOfSolutions=0;
            for(int value=1; value<10; value++) {
                if(this.game.isValidValue(row, col, value)) {
                    tempNumOfSolutions++;
                }
            }

            // if Cell at row,col has less solutions than previous Cell
            // make it new best Cell
            if(tempNumOfSolutions<numOfSolutions) {
                best=temp;
                numOfSolutions=tempNumOfSolutions;
            }
        }
    }

    // by this point either best has a best cell in it
    // or it is still null which would happen only when
    // no unlocked empty cells or ones valid values could be found
    return best;
}
```

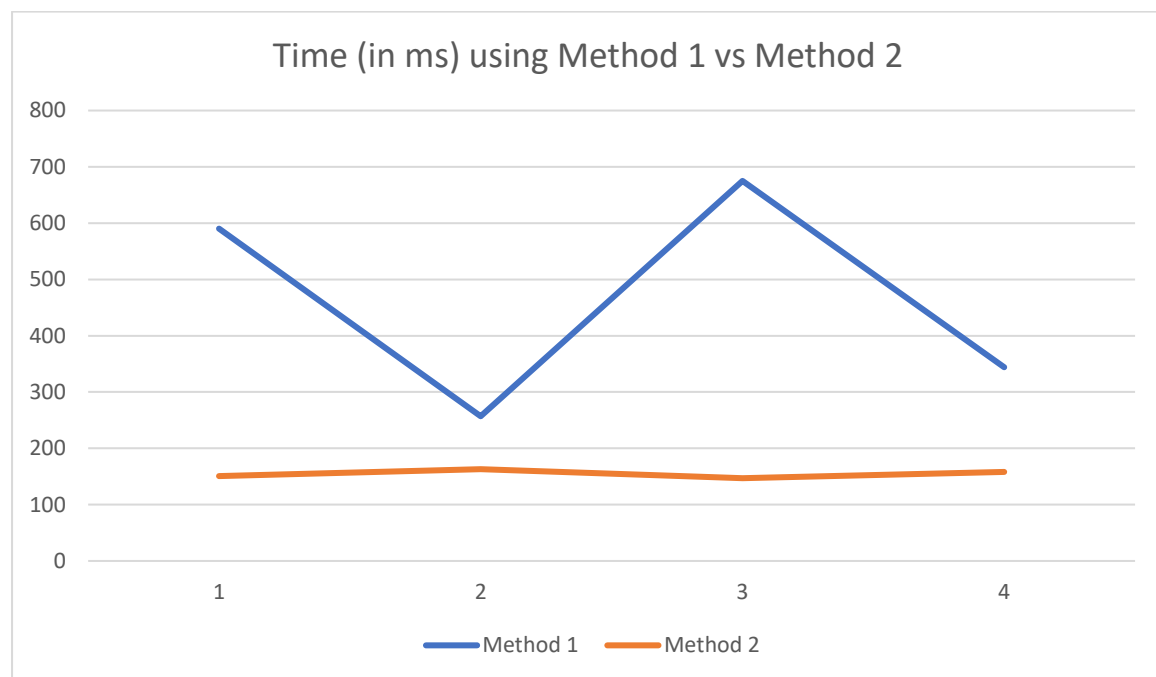
Extension 1

My first extension explores the difference between the two following versions of the *nextBestCell()* method:

1. Return the first cell (row-wise) cell that has a value of 0.
2. Return the first cell with the least number of possible values.

To do this, I have run 100 simulations of both methods for 10 initial values four times. My results are as follows:

	Method 1	Method 2
1.	590ms	151ms
2.	257ms	163ms
3.	675ms	147ms
4.	344ms	158ms
Average	466.5ms	154.75ms



As such, we can see that using Method 2 gives us almost 300% the speed that we get from using Method 1.

Based on the empirical evidence, it can be concluded that Method 2 is better for solving when starting from 10 initial values. However, these results cannot be generalized to all number of starting values without actually testing them.

Extension 2

My second extension accounts for the extra spaces in files. It is a suggested extension from the project instructions. With this extension, a file that is valid (has 9 rows of nine separated numbers) can be read by the *read()* method of the *Board* class. The following are screenshots of the example file being read and solved along with the GUI output screen:



This was implemented by picking numbers off each line and making them into a new String, while skipping the whole line if it was empty after removing all whitespace from the ends of the line which was done using the *trim()* method of the *String* class.

References/Acknowledgements

I consulted both, Prof. Harper and Prof. Al Madi to discuss why my solve method might not be working. Prof. Harper gave me an idea which led to me doing extension 1. I also worked with Quan Phan to go over my code and help me find the bug that was causing my display to be rotated by 90 degrees. The *LandscapeDisplay* class was retrieved from <https://cs.colby.edu/aharper/courses/cs231/f21/labs/lab03/LandscapeDisplay.java>. The algorithm from the project page at <https://cs.colby.edu/aharper/courses/cs231/f21/labs/lab03/assignment.php> was used for the *solve()* method.