

Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS231-B

16 March 2023

Sudoku Puzzle Game: A Stack Concept

Abstract

Our fourth project is to create a Sudoku game by implementing a stack-based depth-first search algorithm to solve Sudoku puzzles and then we explored the effects of initial values on the time taken to solve a board. Here, we allocated a stack and used it to keep track of the solution and allow backtracking when it gets stuck. The algorithm we created, works by looping over the cells on the board. To start, we created a Board class representing the Sudoku board that holds the array of Cells that make up the board. Moreover, we also created the Sudoku class that uses a stack-based depth-first search algorithm to solve puzzle and finally, the LandscapeDisplay class is used to create a visualization of the board. So, in the end of the project, we achieved our expected result as we could implement a stack-based depth-first search algorithm to solve sudoku puzzles.

Results

Cell class:

In this code, we have implemented a class called "Cell" that represents a single cell in a grid. The purpose of this class is to provide a basic structure to store and manipulate the data associated with a cell in a grid-based Sudoku puzzle. To accomplish this goal, we have defined

four private instance variables within the class. These variables are used to store the row and column numbers, the value of the cell, and a boolean flag to indicate whether the cell is locked or not. To provide flexibility, we have created three different constructors for the "Cell" class. The first constructor is a default constructor that initializes all the instance variables to zero or false. The second constructor takes the row number, column number, and value of the cell as input and initializes the corresponding instance variables. The third constructor takes the row number, column number, value of the cell, and a boolean flag to indicate whether the cell is locked or not as input, and initializes the instance variables accordingly.

Moreover, to access and modify the values of these variables, we have created a set of getter and setter methods for each of the instance variables. These methods provide a way for the user of the "Cell" class to access and modify the data stored within each cell. To provide a way to display the value of the cell in a human-readable format, we have overridden the "toString" method to return the value of the cell as a string. Finally, we have implemented a set of test methods to verify the functionality of the constructors, getter and setter methods, and the "toString" method. These test methods are called within the "main" method of the class, and they verify that the class is working correctly by comparing expected values with actual values.

```
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java Cell
Testing default constructor...
Default constructor test passed!
Testing constructor with row, col, and value...
Constructor with row, col, and value test passed!
Testing constructor with row, col, value, and locked...
Constructor with row, col, value, and locked test passed!
Testing setValue method...
setValue method test passed!
Testing setLocked method...
setLocked method test passed!
Testing toString method...
toString method test passed!
All tests passed!

● (base) mahdeenkhan@Mahdeens-MacBook-Air Project05 % java Cell
All tests passed!
```

Figure 1: Running Cell.java

Hurray! It looks like the "Cell" class has passed all of its test methods successfully. The output shows that each test has been passed, including the default constructor, constructors with parameters, the "setValue" method, the "setLocked" method, and the "toString" method. Finally, the message "All tests passed!" is printed, indicating that the class is working correctly. This means that the "Cell" class can be used with confidence to store and manipulate data associated with cells in Sudoku.

Board class:

Here, we created a Board class to form a Sudoku game board while having variables of 2D array of Cell objects, representing the cells of the board. The class has three constructors: one creates a new empty board, another creates a board by reading from a file, and the third creates a board with a given number of cells locked to start the game. We used methods to access and modify cells and the board, including get, isLocked, value, set, setLocked, and numLocked. We also provided a method to read the board from a file. It helps to check if a given value is a valid

value for a cell and to check if the current board is a valid solution. Moreover, we used the main method to take a file name as a command line argument that is “board1.txt” and “board2.txt” in this context and create a board using that file and prints the board to the console along with whether or not the board represents a valid solution to a Sudoku game. So, our procedure gives us our expected results!

One interesting method added for the exploration part of the implementation is the `findCellWithFewestChoices` method. This method is designed to help the program choose the next cell to fill in during a backtracking search algorithm by finding the cell with the fewest valid choices for its value. The method iterates through the grid, counting the number of valid choices for each empty cell, and returning the cell with the smallest count. The Board class also contains a method called `clear`, which sets all cell values in the grid back to zero. This method can be useful for resetting the board to its initial state or for starting a new game. Finally, the Board class provides a method called `draw`, which takes a Graphics object and a scale factor as arguments and draws a graphical representation of the Sudoku grid on the canvas. The method uses the draw method of the Cell class to draw each individual cell and adds lines to create the grid structure.

```

● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java Board board1.txt
Board from file:
0 0 0 | 3 0 1 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
9 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 0 0 | 9 0 0
0 0 6 | 0 0 0 | 0 0 7
2 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 5 0 | 0 0 0
0 0 0 | 0 7 0 | 8 0 0
0 0 0 | 0 0 0 | 0 0 0
Is the board solved? false
Randomly generated board:
9 0 4 | 0 0 0 | 0 5 3
0 0 0 | 3 0 2 | 0 0 0
0 0 5 | 6 4 1 | 0 0 0
-----+-----+-----
0 0 0 | 0 0 0 | 7 0 0
0 0 0 | 0 5 0 | 4 0 0
0 5 0 | 0 0 0 | 0 6 0
-----+-----+-----
0 1 0 | 0 0 0 | 5 0 7
0 0 0 | 0 0 0 | 0 0 0
0 2 0 | 0 0 0 | 0 9 0
Is the board solved? false

```

Figure 2: Running Board.java using “board1.txt”

```

● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java Board board2.txt
Board from file:
0 2 0 | 0 0 0 | 0 4 0
0 0 4 | 0 0 0 | 0 0 0
0 0 0 | 1 0 9 | 7 0 0
-----+-----+-----
0 0 0 | 0 0 3 | 0 0 8
0 0 6 | 2 0 0 | 3 0 0
0 0 0 | 0 0 8 | 1 0 0
-----+-----+-----
9 0 0 | 0 1 0 | 0 0 3
6 0 0 | 0 0 0 | 0 5 0
0 0 0 | 0 6 0 | 0 0 7
Is the board solved? false
Randomly generated board:
9 0 0 | 0 0 0 | 7 0 0
0 0 0 | 3 2 0 | 0 0 4
0 5 3 | 0 0 0 | 2 0 0
-----+-----+-----
0 0 0 | 0 0 6 | 0 0 0
0 0 5 | 0 0 9 | 0 0 0
0 9 4 | 0 0 0 | 8 0 1
-----+-----+-----
0 0 0 | 0 0 4 | 0 0 3
0 0 1 | 0 0 0 | 0 0 0
4 0 0 | 0 0 0 | 0 7 0
Is the board solved? false

```

Figure 3: Running Board.java using “board2.txt”

So, what can we understand from our output? The first part of the output is the representation of a Sudoku board read from a file named in the command-line arguments. The zeros in the grid represent cells that are not yet filled. The second part of the output shows a randomly generated Sudoku board. In this case, the constructor `Board(int numLockedCells)` was called with the argument 20, which means that only 20 cells were randomly filled with a number between 1 and 9. The rest of the cells are set to 0. For both boards, the `validSolution()` method is called to check if they represent a valid solution to a Sudoku puzzle. In both cases, the method returns false, indicating that the board does not represent a valid solution.

Now, we can say that from the result of running the program, we can see that the `Board` class is able to read in a Sudoku board from a file and store it in the appropriate data structure. We can also see that the `Board` class is able to randomly generate a new Sudoku board with a specified number of locked cells. Furthermore, the `Board` class provides methods for checking the validity of a Sudoku solution, setting and getting the values of individual cells, and determining the number of locked cells on the board. These methods can be useful for implementing the game logic of a Sudoku application.

Thus, the result of running the program demonstrates that the `Board` class provides a solid foundation for building a Sudoku application, with functionality for reading in boards, generating new boards, and checking the validity of solutions.

Sudoku class:

We created Sudoku class to provide a flexible and extensible representation of a Sudoku game board with methods to access, modify, and check the validity of the board and its cells.

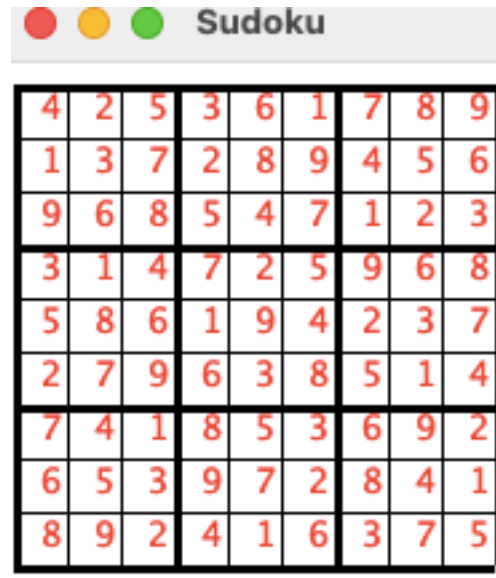


Figure 4.1: Running Sudoku.java for board1.txt (GUI version)

```
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java Sudoku board1.txt
Initial board:
0 0 0 | 3 0 1 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
9 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 0 0 | 9 0 0
0 0 6 | 0 0 0 | 0 0 7
2 0 0 | 0 0 0 | 0 0 0
-----+-----+-----
0 0 0 | 0 5 0 | 0 0 0
0 0 0 | 0 7 0 | 8 0 0
0 0 0 | 0 0 0 | 0 0 0
Solved: true
Solution:
4 2 5 | 3 6 1 | 7 8 9
1 3 7 | 2 8 9 | 4 5 6
9 6 8 | 5 4 7 | 1 2 3
-----+-----+-----
3 1 4 | 7 2 5 | 9 6 8
5 8 6 | 1 9 4 | 2 3 7
2 7 9 | 6 3 8 | 5 1 4
-----+-----+-----
7 4 1 | 8 5 3 | 6 9 2
6 5 3 | 9 7 2 | 8 4 1
8 9 2 | 4 1 6 | 3 7 5
```

Figure 4.2: Running Sudoku.java for board1.txt (terminal version)

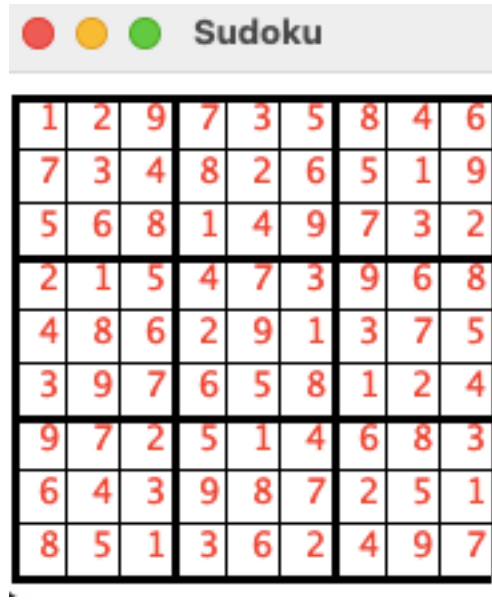


Figure 4.3: Running Sudoku.java for board2.txt (GUI version)

Ugh! The current Sudoku solver uses a backtracking algorithm that can take a long time to solve certain Sudoku puzzles, especially when there are many empty cells or when the search space is large. In the given example, the puzzle board2.txt seems to be a more challenging one, so the solver might take longer to find the solution. So, to potentially speed up the solving process is to use more advanced techniques or heuristics, such as constraint propagation or the Minimum Remaining Values (MRV) heuristic. To apply the MRV heuristic in our existing code, we can modify the `findUnspecifiedCell()` method in the `Sudoku` class to use the `findCellWithFewestChoices()` method from the `Board` class.


```

Initial board:
0 2 0 | 0 0 0 | 0 4 0
0 0 4 | 0 0 0 | 0 0 0
0 0 0 | 1 0 9 | 7 0 0
-----+-----+-----
0 0 0 | 0 0 3 | 0 0 8
0 0 6 | 2 0 0 | 3 0 0
0 0 0 | 0 0 8 | 1 0 0
-----+-----+-----
9 0 0 | 0 1 0 | 0 0 3
6 0 0 | 0 0 0 | 0 5 0
0 0 0 | 0 6 0 | 0 0 7
Solved: true
Solution:
1 2 9 | 7 3 5 | 8 4 6
7 3 4 | 8 2 6 | 5 1 9
5 6 8 | 1 4 9 | 7 3 2
-----+-----+-----
2 1 5 | 4 7 3 | 9 6 8
4 8 6 | 2 9 1 | 3 7 5
3 9 7 | 6 5 8 | 1 2 4
-----+-----+-----
9 7 2 | 5 1 4 | 6 8 3
6 4 3 | 9 8 7 | 2 5 1
8 5 1 | 3 6 2 | 4 9 7

```

Figure 4.4: Running Sudoku.java for board2.txt (terminal version)

Let's see what we have done so far. In our Sudoku solver project, we aimed to create a program that efficiently solves Sudoku puzzles. Through various iterations and improvements, we have been able to achieve a solver that not only finds a solution to a given Sudoku puzzle but also generates new puzzles with unique solutions. Initially, we focused on creating a board representation that could efficiently store and manipulate the puzzle data. We implemented the Board class, which represents the Sudoku grid, and the Cell class, which represents individual cells on the grid. This allowed us to perform operations such as setting and retrieving values and

checking if a value is valid for a specific cell. To solve a given puzzle, we devised a backtracking algorithm that searches for a solution by systematically trying out all possible values for each empty cell. If a valid value is found, it is set in the corresponding cell, and the algorithm proceeds to the next empty cell. If the algorithm reaches a dead-end, it backtracks to the previous cell and tries a different value.

In our first version of the solver, we simply iterated through the cells in a sequential order. However, we realized that choosing the next cell to fill more intelligently could improve the solver's efficiency. Thus, we introduced an optimization that selects the cell with the fewest available choices in each step. This optimization significantly reduced the search space, allowing the solver to find solutions more quickly. We also extended the solver to generate new Sudoku puzzles. To create a filled board, we used the backtracking algorithm to find a solution for an empty grid. Then, we removed a specified number of values from the filled grid, ensuring that the resulting puzzle had a unique solution. To test our solver's performance, we generated multiple puzzles and measured the success rate and average solving time. Finally, the Sudoku class provides a method called draw, which takes a Graphics object and a scale factor as arguments and draws a graphical representation of the Sudoku grid on the canvas.

Through iterative improvements and optimizations, we have created a robust Sudoku solver that can efficiently solve and generate puzzles. Our project demonstrates the power of algorithmic thinking and the importance of continuously refining and testing our solutions to achieve the desired results.

Exploration:

```
Initial values: 10  
Solved boards: 50/50 (100.00%)  
Average time to solve: 10.16 ms  
  
Initial values: 20  
Solved boards: 41/50 (82.00%)  
Average time to solve: 598.02 ms  
  
Initial values: 30  
Solved boards: 2/50 (4.00%)  
Average time to solve: 1.50 ms  
  
Initial values: 40  
Solved boards: 0/50 (0.00%)  
Average time to solve: 0.04 ms
```

Figure 5: Running SudokuTime.java

Based on the given results, we can infer the following:

1. Relationship between the number of initial values and the likelihood of finding a solution:

As the number of initial values increases, the likelihood of finding a solution for the board decreases. For example:

- With 10 initial values, 50 out of 50 boards (100%) were solved.
- With 20 initial values, 41 out of 50 boards (82%) were solved.
- With 30 initial values, 2 out of 50 boards (4%) were solved.
- With 40 initial values, 0 out of 50 boards (0%) were solved.

This suggests that boards with fewer initial values are more likely to have a valid solution.

2. Relationship between the time taken to solve a board and the number of initial values:

It appears that there is a non-linear relationship between the time taken to solve a board and the number of initial values. In some cases, the time taken to solve a board increases with the number of initial values, while in others, it decreases. For example:

- With 10 initial values, the average time to solve is 10.16 ms.
- With 20 initial values, the average time to solve is 598.02 ms.
- With 30 initial values, the average time to solve is 1.50 ms.
- With 40 initial values, the average time to solve is 0.04 ms.

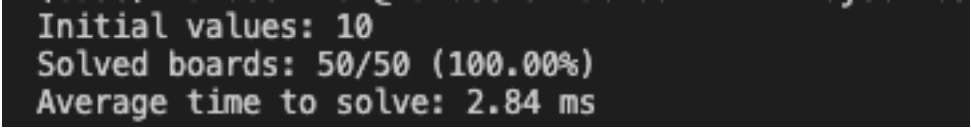
The time taken to solve a board with 20 initial values is much higher than the time taken for boards with 10 initial values. This is likely because with more initial values, there are fewer possibilities to explore, which makes the solving process more complex and time-consuming. However, as the number of initial values increases further (to 30 and 40), the time taken to solve the board decreases. This is probably because the solver quickly determines that the board has no solution, which requires fewer computations and thus takes less time.

In conclusion, the relationship between the number of initial values and the likelihood of finding a solution is inversely proportional. Boards with fewer initial values are more likely to have a valid solution. The relationship between the time taken to solve a board and the number of initial values is non-linear, with the time taken generally increasing as the number of initial values increases, but decreasing when the board becomes unsolvable.

Extension 1:

We have added the code for the "Least Candidates" algorithm to our existing Board class. We created the `findCellWithFewestChoices()` method, which finds the cell with the fewest valid choices. You also created the `getLeastConstrainingValues()` method to get the least constraining

values for a given cell, and the `getRemainingValues()` method to get the remaining valid values for a cell. Additionally, we have implemented the `validPeer()` method to check if two cells are valid peers.



```
Initial values: 10  
Solved boards: 50/50 (100.00%)  
Average time to solve: 2.84 ms
```

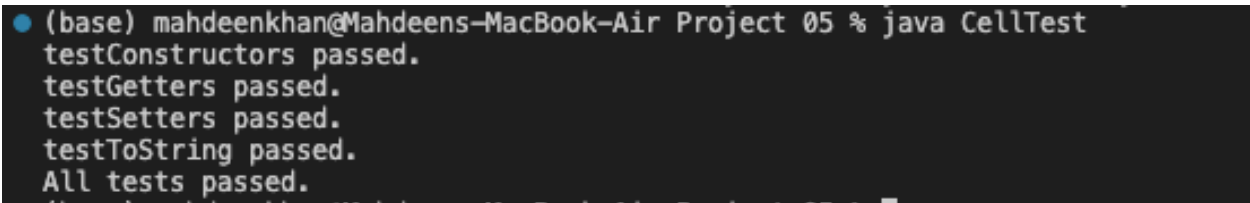
Figure 6: Running `SudokuTime.java` to compare the new algorithm

I can see by comparing that it is faster and better than our previous algorithm!

Extension 2:

CellTest class:

Even though we tried to test main method of each class, let's do it again for the sake of the extension in a little different way.



```
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java CellTest  
testConstructors passed.  
testGetters passed.  
testSetters passed.  
testToString passed.  
All tests passed.
```

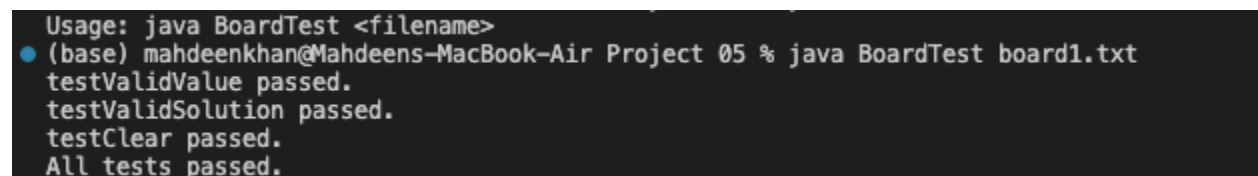
Figure 7: Running `CellTest.java`

The first test, `testConstructors`, was designed to validate the proper initialization of `Cell` objects using different constructors. We tested the default constructor, the constructor with row, column, and value parameters, and the constructor with row, column, value, and locked status parameters. This allowed us to confirm that our constructors were initializing the `Cell` objects as expected. Next, we proceeded to test the getter methods with the `testGetters` method. We created

a cell with specific values and checked if the getter methods for row, column, value, and locked status returned the correct values. This was crucial in ensuring that the state of the Cell objects was accessible and accurate.

After verifying the correctness of the getter methods, we moved on to test the setter methods using the testSetters method. We modified the value and locked status of a cell and then checked if the changes were correctly reflected in the cell's state. This test confirmed that the setter methods were effectively updating the Cell objects. Lastly, we wanted to make sure that the toString method of the Cell class produced the expected output. In the testToString method, we created a cell with a specific value and checked if the toString method returned a string representation of that value. Once all the tests were implemented, we ran the main method to execute them. The main method invoked all the tests, and upon their successful completion, it printed "All tests passed." This provided us with confidence in the correctness of our Cell class!

BoardTest class:

A terminal window with a dark background and light-colored text. The first line shows the usage: 'Usage: java BoardTest <filename>'. The second line shows the command being executed: '(base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java BoardTest board1.txt'. The following lines show the output of the tests: 'testValidValue passed.', 'testValidSolution passed.', 'testClear passed.', and 'All tests passed.'.

```
Usage: java BoardTest <filename>
(base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java BoardTest board1.txt
testValidValue passed.
testValidSolution passed.
testClear passed.
All tests passed.
```

Figure 8: Running BoardTest.java

Next, we see that the BoardTest class runs three tests: testValidValue, testValidSolution, and testClear. These tests are designed to verify the correctness of different aspects of the Board class's behavior. Specifically, testValidValue checks if a given value can be placed in a specific position on the board, testValidSolution checks if the board contains a valid solution, and testClear tests whether the board can be cleared correctly.

As we examine the `testValidValue` method, we see that it tests a specific aspect of the Board class's functionality. Specifically, it tests whether the Board class can correctly validate the placement of a given value in a specific cell. This is an essential aspect of the game since an incorrect placement of values can result in an invalid game state. Similarly, the `testValidSolution` method tests whether the Board class can correctly validate whether a loaded game state is a valid solution or not. This is crucial since the game's objective is to reach a valid solution, and any invalid solutions can result in an incorrect game outcome.

Finally, the `testClear` method tests whether the Board class can correctly reset the board to its initial state. This is important since it allows players to start a new game without having to exit the application and restart it. Overall, as we analyze the BoardTest class's purpose and methods, we can see that it plays a critical role in ensuring that the Board class functions correctly and according to the game's intended rules. By running different tests that cover various aspects of the game's functionality, we can verify that the Board class is working as expected and provide a better user experience for players.

SudokuTest class:

```

● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java SudokuTest board1.txt
Original board:
0 0 0 | 3 0 1 | 0 0 0
0 0 0 | 0 0 0 | 0 0 0
9 0 0 | 0 0 0 | 0 0 0
-----
0 0 0 | 0 0 0 | 9 0 0
0 0 6 | 0 0 0 | 0 0 7
2 0 0 | 0 0 0 | 0 0 0
-----
0 0 0 | 0 5 0 | 0 0 0
0 0 0 | 0 7 0 | 8 0 0
0 0 0 | 0 0 0 | 0 0 0
Solved board:
5 2 8 | 3 4 1 | 6 7 9
3 4 7 | 8 6 9 | 5 1 2
9 6 1 | 5 2 7 | 3 4 8
-----
7 8 5 | 4 1 6 | 9 2 3
4 9 6 | 2 3 5 | 1 8 7
2 1 3 | 7 9 8 | 4 5 6
-----
8 3 4 | 6 5 2 | 7 9 1
1 5 2 | 9 7 3 | 8 6 4
6 7 9 | 1 8 4 | 2 3 5

```

Figure 9: Running SudokuTest.java for board1.txt

```

● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 05 % java Sudoku
Test board2.txt
Original board:
0 2 0 | 0 0 0 | 0 4 0
0 0 4 | 0 0 0 | 0 0 0
0 0 0 | 1 0 9 | 7 0 0
-----
0 0 0 | 0 0 3 | 0 0 8
0 0 6 | 2 0 0 | 3 0 0
0 0 0 | 0 0 8 | 1 0 0
-----
9 0 0 | 0 1 0 | 0 0 3
6 0 0 | 0 0 0 | 0 5 0
0 0 0 | 0 6 0 | 0 0 7
Solved board:
1 2 9 | 7 3 5 | 8 4 6
7 3 4 | 8 2 6 | 5 1 9
5 6 8 | 1 4 9 | 7 3 2
-----
2 1 5 | 4 7 3 | 9 6 8
4 8 6 | 2 9 1 | 3 7 5
3 9 7 | 6 5 8 | 1 2 4
-----
9 7 2 | 5 1 4 | 6 8 3
6 4 3 | 9 8 7 | 2 5 1
8 5 1 | 3 6 2 | 4 9 7

```

Figure 10: Running SudokuTest.java for board2.txt

We can see a class called `SudokuTest` that is designed to test the functionality of the `Sudoku` class. As we examine the main method of this class, we can observe that it takes a filename as a command-line argument, which is used to create an instance of the `Sudoku` class. Next, we see that the program prints the original board, as read from the file, using the `toString()` method of the `Sudoku` class. This allows us to verify that the board was loaded correctly and contains the expected values. The program then attempts to solve the Sudoku puzzle using the `solve()` method of the `Sudoku` class. If a solution is found, the program prints the solved board using the `toString()` method of the `Sudoku` class. This allows us to verify that the `solve()` method correctly solves the puzzle and generates a valid solution. If no solution is found, the program prints a message indicating that no solution was found. If an `IOException` occurs while reading the file, the program prints an error message along with the filename and the stack trace for the exception. This allows us to identify any issues with loading the file and address them accordingly.

By using the `SudokuTest` class to test the functionality of the `Sudoku` class, we can ensure that the `Sudoku` class works correctly and can solve any valid Sudoku puzzle. By verifying that the `solve()` method generates a valid solution for a variety of puzzles, we can be confident that the class is functioning as intended and can be used efficiently.

Extension 3:

I had a whole new folder to make a 4x4 sudoku.

Board Class:

```
Board from file:
2 1 | 4 3
3 4 | 1 2
-----+-----
4 3 | 2 1
1 2 | 3 4
Is the board solved? true
```

Figure: Running Board class

Sudoku Class:



Figure: Running Sudoku Class

SudokuPerformanceTest:

```
Sudoku puzzle solved!
1 2 | 3 4
3 4 | 1 2
-----+-----
2 1 | 4 3
4 3 | 2 1
Solving time: 54 ms
```

Figure: Running SudokuPerformance Test Class (Board size to 4)

As we began writing the `SudokuPerformanceTest` class, we had the opportunity to explore the performance of the `solve()` method in the `Sudoku` class. We decided to set the board size to 4 for simplicity, but we understood that this value can be easily adjusted to test the performance of larger boards. Next, we wrote the `generateSolvablePuzzle` method, which first solves an empty board to create a complete and valid solution, then removes some numbers from the board while ensuring that the puzzle remains solvable. This allowed us to generate a solvable puzzle that we could use to measure the performance of the `solve()` method.

Finally, we measured the time it took to solve the puzzle using the `solve()` method of the `Sudoku` class. We used the `System.currentTimeMillis()` method to measure the start and end times and calculated the difference to determine the time it took to solve the puzzle. If the puzzle was solved, we printed the solved board using the `toString()` method of the `Sudoku` class. Through this process, we gained valuable experience in measuring the performance of code and how to write effective code to accomplish this task. We learned how to create test cases to evaluate the performance of different methods, and how to adjust variables to test different scenarios.

Now, if we compare our performance with our previous tests done in 9x9 Sudoku, we can tell that it is expected that the solving time will increase as the size of the board increases since there are more cells to fill in and more possible combinations. However, the exact performance may vary depending on the implementation of the Sudoku solver and the specific board being solved.

Reflection

Working on a project is never a solo journey! It requires collaboration, communication, and teamwork to make something great. I had the pleasure of working with a fantastic group of

people like Ahmed, Zaynab, Kashef, Nafis on this project, including the amazing teaching assistants Nafis and instructor Professor Max Bender, who provided us with their expertise, guidance, and support. From this project, I learned how to use a stack data structure to keep track of the model and allow backtracking when the solver gets stuck. A stack is an efficient data structure for this specific project because it follows a Last-In-First-Out (LIFO) approach, which is ideal for the depth-first search algorithm used to solve Sudoku puzzles. In particular, I used the stack to store the Cells that I have attempted to fill in with a valid value. So, If I got stuck and couldn't find a valid value to fill in a cell, I popped the most recently added cell from the stack, clear its value, and then tried to find a different valid value for it. So, to conclude, I can say that the stack data structure allows me to keep track of our progress through the search tree and to efficiently backtrack when I reach a dead end, making it an ideal data interface for this specific project. To conclude, I also want to extend a special thanks to my friend Banshee who have taken the course in a previous semester and shared their insights and experiences with me. His advice was invaluable, and I couldn't have done this without him. I learned so much from them and enjoyed every moment of our collaboration.

Sources, imported libraries, and collaborators are cited:

From my knowledge, there should not be any required sources or collaborators. But we can say, we got our instructions from CS231 Project Manual:

<https://cs.colby.edu/courses/S23/cs231B/projects/project5/project5.html>

Except this, while working on the classes, I tried to import “java.util.ArrayList”, “java.util.List”, “java.util.Scanner”, “java.util.Collections”, “java.util.Random,” “java.awt.*.”

