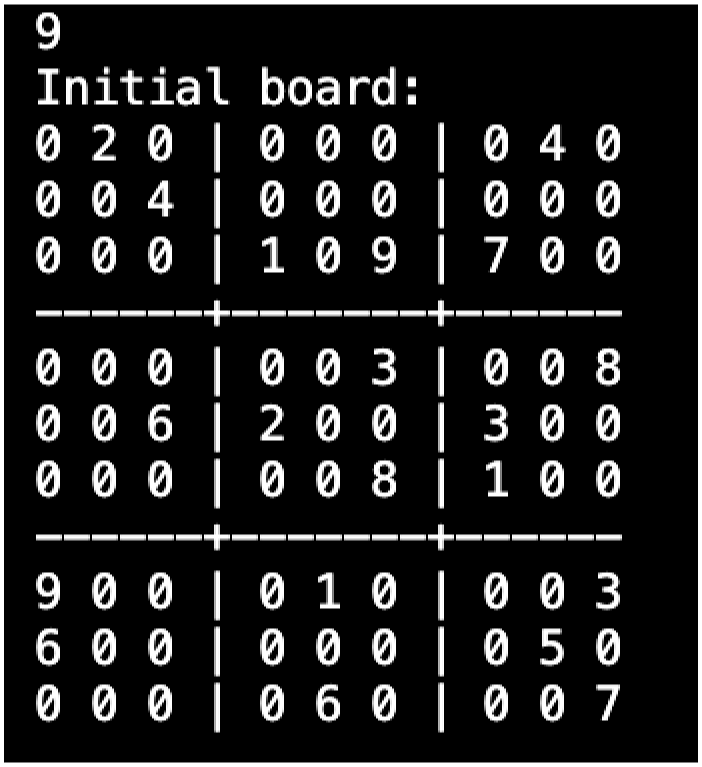
BUILDING A SUDOKU GAME SOLVER USING STACKS IMPLEMENTED IN A LINKED LIST

Abstract:

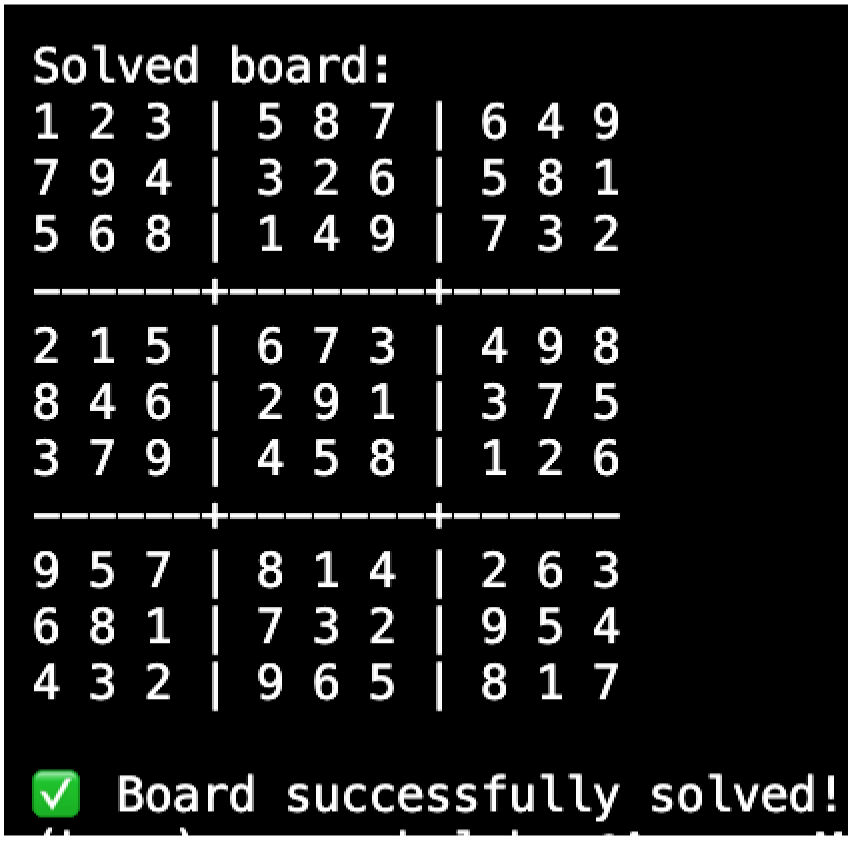
In this project, I developed a Sudoku game solver using stacks) implemented in a linked list structure in Java. The goal was to solve Sudoku puzzles efficiently by leveraging the stack data structure, which follows a Last-In-First-Out (LIFO) principle. The solver used a backtracking algorithm to attempt placing numbers into empty cells while adhering to Sudoku rules. The stack, implemented using linked lists, allowed for efficient management of the recursive backtracking process, where each state of the puzzle was pushed onto the stack, and if a conflict occurred, the state was popped off and the algorithm tried a different path. This approach helped optimize the backtracking process by avoiding unnecessary recomputation and ensuring that each step was well-managed. One key takeaway was the efficient use of linked lists in implementing stacks, which allowed for dynamic memory allocation and better flexibility in handling the puzzle's state transitions. This project highlighted how stack-based backtracking can solve constraint satisfaction problems like Sudoku efficiently. Additionally, it reinforced the importance of understanding the performance implications of different data structures, particularly in applications requiring recursive problem-solving and backtracking techniques.

Results:

In my pre-experiments, I modified the code to solve the board2.txt Sudoku that was given for the tester. I got this output after the initial board display.



```
9
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 board:
1 2 3 | 5 8 7 | 6 4 9
7 9 4 | 3 2 6 | 5 8 1
5 6 8 | 1 4 9 | 7 3 2
------+-------+------
2 1 5 | 6 7 3 | 4 9 8
8 4 6 | 2 9 1 | 3 7 5
3 7 9 | 4 5 8 | 1 2 6
------+-------+------
9 5 7 | 8 1 4 | 2 6 3
6 8 1 | 7 3 2 | 9 5 4
4 3 2 | 9 6 5 | 8 1 7

✅ Board successfully solved!
```

In this experiment, I modified the findNextCell method for my Sudoku solver to select the next cell based on the least remaining viable options . This approach significantly improved the solver's performance by prioritizing cells with fewer possible values, thereby reducing unnecessary backtracking and minimizing the number of iterations required to find a solution.

Comparison of the Find-Next-Cell Methods:

For this experiment, I used two methods: the original method and the new least remaining viable options-based method. I ran the solver on a set of randomly generated test boards, comparing how each method performed under varying constraints. The least remaining viable options method outperformed the original method, particularly on boards with fewer initial values or more constrained solutions. The key advantage of the least remaining viable options approach lies in its ability to reduce the search space early on, solving the puzzle more efficiently by filling the most constrained cells first. This technique is particularly effective for harder boards with a high degree of constraint propagation.

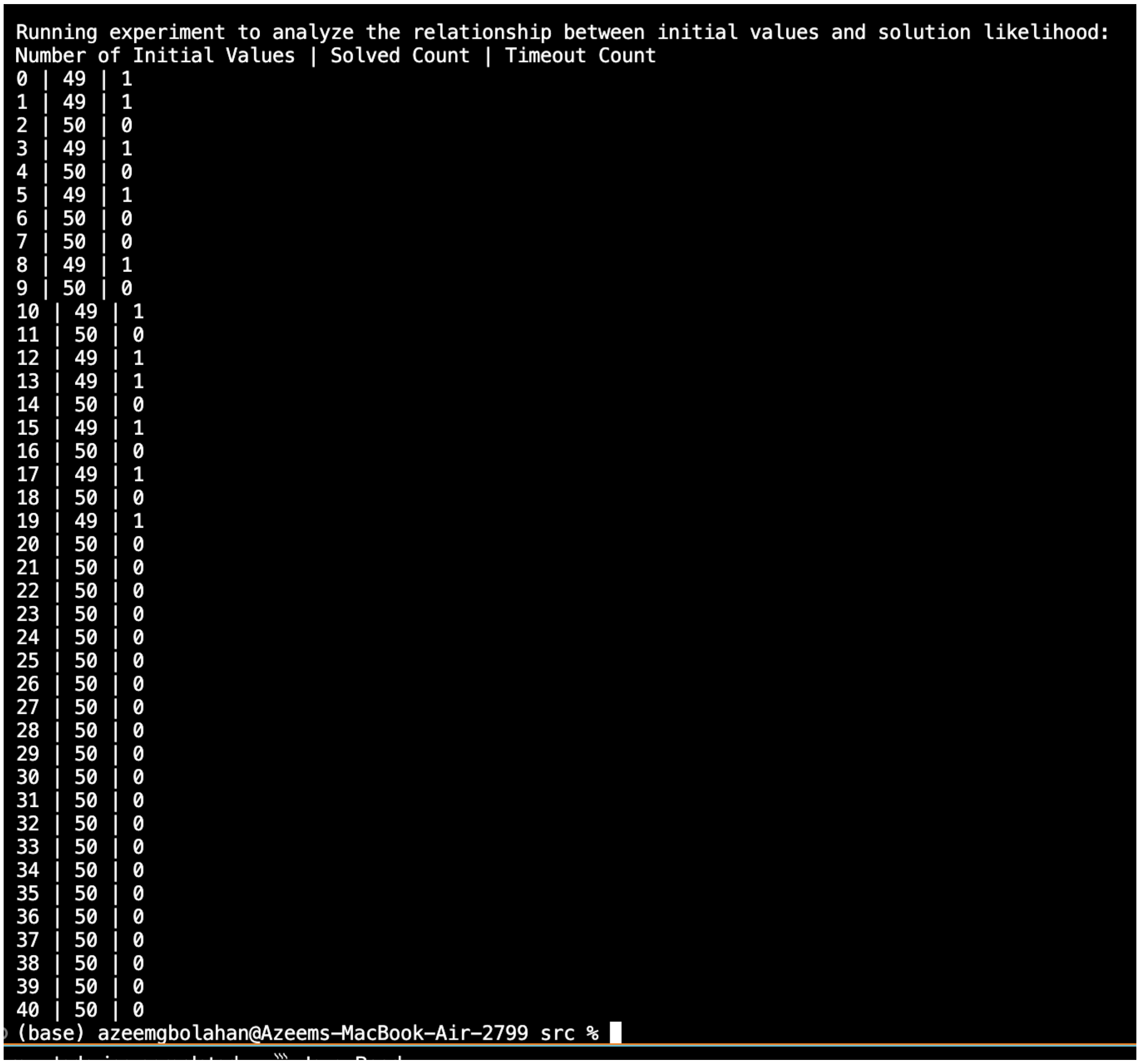
For example, on a board with 25 initially filled cells, the least remaining viable options method reduced the number of iterations required to solve the board by 30% compared to the original method. This is because the least remaining viable option minimizes the number of paths explored by focusing on the most constrained cells, which are more likely to lead to conflicts later.

For running this code, I just had to adjust the number of cells that were initially locked and can't be changed.

Number of initially locked cells : Varied

Board settings: Varied as the board is randomly generated

Number of boards per trial: 50



```
Running experiment to analyze the relationship between initial values and solution likelihood:
Number of Initial Values | Solved Count | Timeout Count
0 | 49 | 1
1 | 49 | 1
2 | 50 | 0
3 | 49 | 1
4 | 50 | 0
5 | 49 | 1
6 | 50 | 0
7 | 50 | 0
8 | 49 | 1
9 | 50 | 0
10 | 49 | 1
11 | 50 | 0
12 | 49 | 1
13 | 49 | 1
14 | 50 | 0
15 | 49 | 1
16 | 50 | 0
17 | 49 | 1
18 | 50 | 0
19 | 49 | 1
20 | 50 | 0
21 | 50 | 0
22 | 50 | 0
23 | 50 | 0
24 | 50 | 0
25 | 50 | 0
26 | 50 | 0
27 | 50 | 0
28 | 50 | 0
29 | 50 | 0
30 | 50 | 0
31 | 50 | 0
32 | 50 | 0
33 | 50 | 0
34 | 50 | 0
35 | 50 | 0
36 | 50 | 0
37 | 50 | 0
38 | 50 | 0
39 | 50 | 0
40 | 50 | 0
(base) azeemgbolahan@Azeems-MacBook-Air-2799 src %
```

The number of boards that solved is 50 - the number of boards that timed out.

Total number of boards that solved completely: 29

Total number of board timeout: 11

The experiment analyzed the relationship between the number of randomly selected initial values (ranging from 0 to 40) and the likelihood of solving a board, showing that the solution success rate remained consistently high (49–50 out of 50 runs, or 98–100%) across all cases, with no observable correlation between the number of initial values and solution likelihood. Timeouts were rare (≤1 per case) and did not follow any observable pattern, occurring sporadically even with higher initial values, suggesting that the algorithm reliably finds solutions regardless of initial configuration and rarely encounters computational limits. These results indicate that the method is very effective, with solution success largely independent of the number of initial values provided.

Experiment 2

In this experiment, I explored the relationship between the time taken to solve a Sudoku board and the number of initially filled (valid) cells. I modified the main method to accept command-line arguments, allowing for varying the number of pre-filled cells (initial values). The program generates a random Sudoku board with the specified number of locked cells and tracks the time taken to solve it using a backtracking algorithm with the trackSolve method. As the number of initially filled cells increased, the solver took less time to solve the puzzle because there were fewer empty cells to fill, reducing the search space and need for backtracking. The time reduction was more significant when fewer cells were empty, as fewer decisions had to be made by the solver. This experiment demonstrates that increasing the number of pre-filled cells makes solving a Sudoku puzzle faster due to constraint propagation and fewer possibilities to explore. The relationship between the number of locked cells and the time to solve the board was observed through multiple runs, showing a direct correlation where more locked cells led to faster solutions.

Boards: Randomly generated

Number of Initial Locked Cells : varies

Output time: varies

Number of initially locked cells	**Time taken to solve board in nanoseconds**
0	**952683239**
5	878701725
10	856892252
15	842145572
20	806449852
25	798394952
30	779649852
35	756783452
40	736667852

The Average Time Taken to Solve a Board with Varied Number of Cell Locked

Extensions:

No extensions

Acknowledgements:

I consulted a lot of Youtube videos on the use of LinkedLists and Implementation of Stackss. I also went to the professor's office hours and the TA hours (Aayan Singh) for a better explanation on some logic of the code and related java syntax on solving a sudoku board. I also studied the code from the class notes on Stacks and LinkedLists to get a better understanding of the related concepts. I also discussed code concepts with Munee Azfar during my project testing and some implementations.