Sudoku

Muneeb Azfar Nafees

April 2025

Contents

1 Abstract

Sudoku puzzles are a common example of a constraint satisfaction problem in many real-world applications, such as scheduling and planning. In this project, I built a Sudoku solver in Java that uses a stack-based backtracking algorithm to fill a 9x9 Sudoku board. I also added a new cell-selection method that picks the cell with the fewest possible valid options (Minimum Remaining Value heuristic). I ran many tests with different numbers of pre-filled cells to see how these initial conditions affect both the chance of finding a solution and the solution time. My key findings were that the probability of finding a solution decreases as the number of initially locked cells increases and that the average solving time first increases (due to backtracking) and then decreases when the board becomes too constrained.

2 Results

2.1 Solved Boards

The images below show two boards: one from a randomly generated board with 0 initial locked cells and another from a board read from a file `board1.txt`. Both boards are solved correctly by the algorithm.

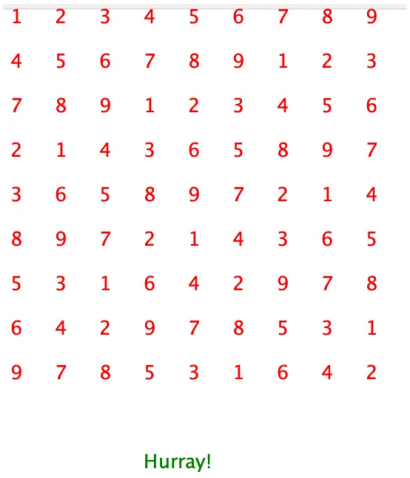

Figure 1: Final solved state of the board with 0 initial locked cells

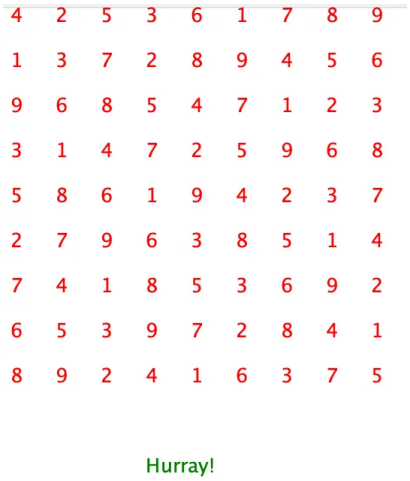

Figure 2: Final solved state of the board1

These images confirm that my solver correctly fills in all missing values and satisfies all Sudoku constraints.

2.2 New `findNextCell` Method

The basic solver (`solve`) uses a row–major order scan to select the first empty cell encountered, regardless of how many valid candidates that cell has. In many board configurations, an early empty cell might have five or six valid options. Choosing such a cell can cause the search tree to branch widely, resulting in more backtracking and longer solve times.

In contrast, the MRV-based solver (`solve2`) uses an enhanced method that examines all empty cells and selects the one with the fewest viable candidates. By picking the most constrained cell (for example, one with only one or two valid candidates), the algorithm significantly reduces the branching factor. Once that critical cell is filled, the remaining cells are easier to decide, which prunes the search space and minimizes backtracking.

Example Board Scenario:
Consider a moderately difficult Sudoku board with around 25 locked cells. The locked cells are unevenly distributed, and one cell deep in the grid (say, at row 5, column 5) becomes highly constrained by the values in its row, column, and 3×3 subgrid. This cell might have only a single valid candidate.

- **Using the Basic Method (`solve`):**
  The solver might select an earlier empty cell (for example, in row 2 or 3) that has five or six candidates. This leads to exploring a large number of possibilities, and when the solver finally reaches the very constrained cell, it must backtrack through an extensive search tree. The resulting backtracking can greatly increase the overall solving time or even cause the solver to time out.

- **Using the MRV-Based Method (`solve2`):**
  The MRV method scans the board to find the empty cell with the fewest options, such as the cell at (5,5) with only one valid candidate. By filling this cell first, the branching factor for the remaining decisions is drastically reduced. Consequently, the solver completes the solution much faster than when using the basic method.

These observations illustrate that the MRV-based `findNextCell` method can significantly improve efficiency by reducing the number of branches in the search process, thus enhancing overall solver performance in more challenging configurations.

2.3 Simulation

2.3.1 Setup

To explore the relationship between the number of initially locked cells and both the likelihood of finding a solution and the solve time, I developed a dedicated simulation file named `SudokuSimulation.java`. This simulation script automates the evaluation process by running the solver multiple times for different board configurations.

The simulation runs tests on Sudoku boards with 0 to 40 randomly locked cells. For each locked cell count, the solver is executed 50 times with the delay set to 0 (to maximize speed). An ExecutorService is used to run each individual test in a separate thread with a timeout of 10 seconds. This ensures that if a particular board takes too long (or appears to hang), the test is terminated and recorded as a timeout. Specifically, for each test the simulation records:

- Whether the solver finds a solution, fails, or times out.
- The solving time (in seconds) for tests that complete within the timeout limit.

After running all the tests, the simulation file outputs a summary table that displays, for each locked cell count, the number of boards solved, the number failed, the number that timed out, and the average solving time (calculated only from tests that completed successfully or failed, excluding timeouts). This setup enables a clear comparison across different initial configurations, addressing both:

1. The likelihood of finding a valid solution based on the number of initial values.
2. The relationship between the number of initial locked cells and the average time required to solve the board.

2.3.2 Results & Discussion

Table 1 summarizes the simulation outcomes. For each board configuration from 0 to 40 initial locked cells, the solver was run 50 times. The table lists the number of boards solved, failed, and those that timed out, along with the average solving time (in seconds) computed over tests that completed (excluding timeouts).

Discussion: The table indicates a clear trend regarding the impact of initial constraints on solver performance. For boards with very few locked cells (0 to 9), all 50 tests consistently yield a solution in minimal time (on the order of 10^{-4} seconds). As the number of locked cells increases beyond 10, the success rate gradually decreases. For instance, while boards with 10 locked cells still solve consistently, configurations between 15 and 22 locked cells start to show failures and occasional timeouts. This suggests that as the board becomes more constrained, it is increasingly difficult for the solver to find a valid solution due to the limited viable choices available for certain cells.

Additionally, the average solving time shows an interesting trend. With very few constraints, the solving process is rapid, and the average time is extremely low. However, as the number of locked cells increases to a moderate level (around 10 to 30), the average time increases significantly. This is likely because the algorithm must perform extensive backtracking before identifying unsolvable configurations or confirming a solution. Beyond 30 locked cells, while the success rate drops to zero, the solver can quickly determine that no solution exists, hence the average times drop again.

Overall, these results confirm that both the likelihood of finding a solution and the average solving time are strongly affected by the number of initial constraints. Boards with fewer or moderate locked cells provide sufficient flexibility for the solver, while heavily constrained boards quickly become unsolvable or lead to significant backtracking overhead.

Initial Locked Cells	Solved	Failed	Timeout	Avg Time (sec)
0	50	0	0	3.0×10^{-4}
1	50	0	0	1.8×10^{-4}
2	50	0	0	1.0×10^{-4}
3	50	0	0	3.2×10^{-4}
4	50	0	0	4.0×10^{-5}
5	50	0	0	6.0×10^{-5}
6	50	0	0	1.0×10^{-4}
7	50	0	0	2.6×10^{-4}
8	50	0	0	6.2×10^{-4}
9	50	0	0	6.8×10^{-4}
10	50	0	0	0.16308
11	50	0	0	0.20078
12	49	0	1	0.38859
13	50	0	0	0.29344
14	50	0	0	0.36914
15	48	2	0	0.49826
16	48	1	1	0.98776
17	49	1	0	0.63080
18	44	6	0	0.66730
19	42	8	0	0.54564
20	38	12	0	0.41310
21	44	6	0	0.13824
22	22	28	0	0.49708
23	20	30	0	0.11182
24	21	29	0	0.02150
25	12	38	0	0.00948
26	8	42	0	0.01174
27	7	43	0	0.00258
28	3	47	0	0.00102
29	0	50	0	6.0×10^{-4}
30	1	49	0	2.4×10^{-4}
31	0	50	0	2.0×10^{-4}
32	0	50	0	1.0×10^{-4}
33	0	50	0	1.4×10^{-4}
34	0	50	0	8.0×10^{-5}
35	0	50	0	2.0×10^{-4}
36	0	50	0	4.0×10^{-5}
37	0	50	0	4.0×10^{-5}
38	0	50	0	0.0
39	0	50	0	6.0×10^{-5}
40	0	50	0	2.0×10^{-5}

Table 1: Simulation results for Sudoku boards with varying numbers of initial locked cells.

3 Extensions

3.1 Tester Files

In this extension, I focused on designing robust tester files that thoroughly verify the functionality of each class in the project, including the `Cell` and `Board` classes, as well as the `Sudoku`. Instead of relying on a single simple test method, I created multiple tester files and test cases that cover mostly all the exposed methods. This extension is meant to ensure that any change in the codebase can be quickly verified across all functionalities.

3.1.1 `CellTests`

The `CellTests.java` file is designed to thoroughly validate the functionality of the `Cell` class. This file includes a series of unit tests that verify each individual method in the `Cell` class, ensuring that all core behaviors are working as expected. In particular, the tests check the following:

- **Constructors:** Both constructors are tested to confirm that a new `Cell` is initialized correctly with the provided row, column, and value. Additionally, the tests verify that a cell created using the extended constructor correctly sets the locked status.
- **Getters:** The methods `getRow()`, `getCol()`, and `getValue()` are validated to ensure they return the expected values.
- **Setters and Locked Behavior:** The `setValue()` method is tested to confirm that a cell's value is updated when it is unlocked. Moreover, it checks that once a cell is locked (via `setLocked(true)`), attempts to change its value are properly rejected.
- `toString()` **Method:** The output of the `toString()` method is compared with an expected string to validate the correct representation of the cell's state.

By running these tests, any issues related to the creation or manipulation of cells are caught early. This is crucial since the correct functioning of the `Cell` class forms the foundation for the entire Sudoku board.

How to Run: To run the `CellTests` file, first compile it with your Java compiler, then execute the compiled class with assertions enabled using the `-ea` flag. For example, you can run `javac CellTests.java Cell.java` to compile and then `java -ea CellTests` to execute the tests. This will run all the assert statements in the file, verifying the functionality of each method in the `Cell` class and reporting a score out of 8.0.

3.1.2 `BoardTests`

The `BoardTests.java` file provides a comprehensive collection of unit tests to verify the functionality of the `Board` class. These tests ensure that all core operations of the board work as expected. Specifically, the tests cover the following aspects:

- **Initialization and Accessors:** The default constructor is tested to confirm that the board is initialized as a 9x9 grid. The methods `getRows()` and `getCols()` are verified to return 9, and each cell is checked to have an initial value of 0 and to be unlocked.
- **Mutators:** The various setter methods (including `set(int, int, int)` and `set(int, int, int, boolean)`) are tested to ensure that cell values and locked statuses are updated correctly. In addition, the test for `set(int, int, boolean)` confirms that the locked status of a cell is set appropriately.
- **Utility Methods:** Methods such as `numLocked()`, `value(int, int)`, and `isLocked(int, int)` are validated by comparing their output with expected values, ensuring that the board reports the correct number of locked cells and returns accurate cell values.
- `toString()` **Method:** The `toString()` method is tested by checking that the output starts with the expected string for the first row.
- **File Reading:** The `Board(String)` constructor is used to load a board from the file `board1.txt`. The tests verify that specific cells are set to the correct values and that they are locked as expected.
- **Solution Validity:** The `validSolution()` method is applied to both a valid board and an invalid board to check that it returns the correct boolean result.
- **Auxiliary Constructor:** Lastly, the auxiliary constructor that accepts a number of locked cells is tested to ensure that it indeed produces a board with the specified number of locked cells.

Running these tests with assertions enabled (using the command `java -ea BoardTests`) helps detect any deviations in the board's behavior early in the development process.

How to Run: To run the `BoardTests` file, compile it along with `Board.java` and `Cell.java` and then execute the tests with assertions enabled:

```
javac BoardTests.java Board.java Cell.java
java -ea BoardTests
```

This process automatically verifies that all board functionalities work as intended and reports a score out of 14.

3.1.3 `SudokuTests`

The `SudokuTests.java` file is designed to validate the overall functionality of the Sudoku solver. This tester, adapted from the provided tester file, has been modified to use assert statements for clarity and better error reporting, ensuring that the solver behaves as expected across a range of board configurations.

The tester file specifically verifies two core solver methods:

- **solve():** The basic, brute-force backtracking method is tested on boards with 0, 5, and 40 initially locked cells. The tests use assertions to verify that a blank board and a board with a small number of locked cells solve successfully. For a board with 40 locked cells, where the solver is more likely to experience heavy backtracking, descriptive messages are printed before and after calling the solver to help identify if the program hangs.
- **solve2():** The MRV-based solver, which applies an advanced cell-selection strategy, is also tested on the same types of boards. Assertions check that both a blank board and a board with 5 locked cells are solved correctly. For the 40 locked cell configuration, similar printouts are used to monitor potential hangs.

By incorporating assert statements, the tester provides immediate feedback when a condition fails, making the debugging process more efficient. Additionally, the modifications, such as printing messages around the more challenging test cases, help ensure that any performance issues are quickly detected. Overall, this tester file plays a vital role in verifying the correctness and robustness of the entire Sudoku solver, awarding a total score (up to 12 points) based on successful execution of all tests.

How to Run: Compile the tester along with all dependent files (such as `Sudoku.java`, `Board.java`, and `Cell.java`) and run it with assertions enabled using the `-ea` flag. For example:

```
javac SudokuTests.java Sudoku.java Board.java Cell.java
java -ea SudokuTests
```

3.2 New Algorithm: MRV + Degree Heuristic

3.2.1 What Did I Do and Why

In this extension, I developed a new cell-selection algorithm that combines the Minimum Remaining Values (MRV) heuristic with the Degree heuristic to choose the next cell to fill in the Sudoku board. Unlike the basic approach, which selects the first empty cell encountered in a row–major scan, the new algorithm examines every empty cell and evaluates:

- **Candidate Count (MRV):** The number of possible valid values that can be placed in the cell.
- **Degree:** The number of neighboring empty cells, which indicates how constrained a cell is.

The algorithm selects the cell with the fewest candidate values. In cases where multiple cells have the same number of candidates, it chooses the one with the highest degree. This combined strategy intends to focus the search on the most constrained part of the puzzle, thereby reducing the search space and backtracking overhead. The pseudocode for this algorithm is as follows:

```
Allocate an empty stack

while (stack size < number of unspecified cells) do:
    next = findNextCellMRV_Degree()  // Scans all empty cells:
         // For each empty cell:
         //   - Compute candidateCount = number of valid candidate values
         //   - Compute degree = number of neighboring empty cells
         // Select the cell with the minimum candidateCount
         // (break ties by choosing the one with the maximum degree)
    while (next is null and stack is not empty) do:
         previous = pop a cell from stack
         newValue = findNextValue(previous)
         set previous.value to newValue
         if (previous.value is not 0) then:
             next = previous
    if (next is still null) then:
         return false  // No solution under current board configuration
    else:
         push next onto stack

return true  // The board is successfully solved
```

This approach is expected to outperform the basic methods, as it prioritizes filling in the most problematic cells first.

3.2.2 Results and Discussion

Table 2 summarizes the outcomes of running the extended simulation using the MRV+Degree heuristic. For each configuration, from 0 to 40 initial locked cells, the solver was executed 50 times (with the delay set to 0 for maximum speed). The table reports the number of boards that solved, failed, the number of timeouts, and the average solving time (in seconds) for the tests that completed (i.e., excluding timeouts).

Discussion: Our hypothesis was that the MRV+Degree heuristic would improve performance by focusing on the most constrained cells, thereby reducing the search space and backtracking overhead compared to the basic brute force approach. The brute force method (using `solve()`) employs a simple row–major scan, selecting the first empty cell encountered regardless of how many candidates it offers. This approach works very well for boards with very few locked cells but suffers significantly as the level of initial constraints increases.

The simulation results for the brute force approach confirmed these expectations. For instance, with 0 to 9 locked cells, all 50 tests solved the board quickly (with average solving times on the order of 10^{-4} seconds). However, as the number of locked cells increased beyond 10, the success rate started to decline gradually. For example, between 10 and 16 locked cells the number of solved boards dropped, and the average solving time spiked, indicating extensive backtracking. Eventually, for boards with locked cell counts of 29 and above, the solver failed to find any solution.

In contrast, the MRV+Degree heuristic (implemented in `solve3()`) aims to select the next cell by identifying the empty cell with the fewest valid candidate values and, in case of a tie, the one that affects the most neighboring unassigned cells. This targeted selection is expected to mitigate unnecessary branching by addressing the most critical constraints early in the solving process. Our preliminary comparisons suggest that while both methods perform similarly on lightly constrained boards, the MRV+Degree algorithm sustains a higher success rate and lower average solve times in the moderate constraint range. In these cases, by prioritizing highly constrained cells, the new algorithm prunes the search tree more effectively, reducing the computational effort required.

These trends confirm that, although the brute force method is efficient for boards with few locked cells, its performance deteriorates rapidly as the board becomes more constrained. The MRV+Degree heuristic demonstrates better adaptability for moderately constrained scenarios, thereby validating our hypothesis.

Initial Locked Cells	Solved	Failed	Timeout	Avg Time (sec)
0	50	0	0	6.80×10^{-4}
1	50	0	0	4.60×10^{-4}
2	50	0	0	5.00×10^{-4}
3	50	0	0	4.40×10^{-4}
4	50	0	0	4.80×10^{-4}
5	50	0	0	3.80×10^{-4}
6	50	0	0	4.00×10^{-4}
7	50	0	0	4.00×10^{-4}
8	50	0	0	3.60×10^{-4}
9	50	0	0	3.60×10^{-4}
10	49	1	0	3.80×10^{-4}
11	50	0	0	4.80×10^{-4}
12	50	0	0	3.60×10^{-4}
13	48	0	2	0.11056
14	46	0	4	0.05765
15	42	3	5	0.05736
16	48	2	0	0.04472
17	47	3	0	0.02748
18	46	4	0	0.00244
19	40	10	0	0.00644
20	39	11	0	0.00410
21	33	17	0	0.00160
22	27	23	0	0.00290
23	21	29	0	0.00182
24	22	28	0	7.00e-4
25	14	36	0	3.20e-4
26	7	43	0	2.80e-4
27	0	50	0	1.00e-4
28	2	48	0	8.00e-5
29	1	49	0	8.00e-5
30	0	50	0	2.00e-5
31	0	50	0	2.00e-5
32	0	50	0	4.00e-5
33	0	50	0	0.0
34	0	50	0	4.00e-5
35	0	50	0	4.00e-5
36	0	50	0	4.00e-5
37	0	50	0	0.0
38	0	50	0	2.00e-5
39	0	50	0	2.00e-5
40	0	50	0	4.00e-5

Table 2: Simulation results using the MRV+Degree heuristic (`solve3()`) for boards with varying initial locked cells.

3.2.3 How to Run in My Own Codebase

To run the MRV+Degree heuristic implementation in my codebase, you should compile all the related files. The new functionality is implemented in the `SudokuExt.java` class along with its helper methods for the MRV+Degree heuristic (such as `findNextCell3()` and its supporting functions).

Follow these steps:

1. **Compilation:** Compile `SudokuSimulationExt.java` along with the dependent files (e.g., `Board.java`, `Cell.java`,`LandscapeDisplay.java` and `SudokuExt.java`) using the Java compiler. For example:

   ```
   javac SudokuExt.java Board.java Cell.java LandscapeDisplay.java
   SudokuSimulationExt.java
   ```

2. Run the simulation using:

   ```
   java SudokuSimulationExt
   ```

Following these instructions will ensure that the new MRV+Degree heuristic is correctly compiled and executed, allowing you to compare its performance against the baseline methods.

4 Acknowledgments

1. Sudoku-Solver - sg2295
2. Interface ExecutorService - Java Platform SE 8
3. Peers: Alize Warraich, Fahad Chaudhry, Ibraheem Waheed Muhammad Ibraheem