Solve Sudoku with Depth-First Search and Minimum Remaining Values

Abstract

I modeled Sudoku as a constraint satisfaction problem and built a solver that uses an explicit stack depth-first search (DFS) with the Minimum Remaining Values (MRV) heuristic, along with fast row/column/subgrid validity checks and optional visualization. Using core CS ideas like stacks for recursion-by-iteration, backtracking search, and heuristic ordering, I ran 50 randomized trials for each clue count from 0 to 40 and recorded solve rate, timeouts, and solve time among successes. The solver achieved a 100% success rate up to 11 clues, then success declined steeply (≈68% at 20 clues and near 0% by the low 30s); timeouts were rare and concentrated around 12–19 clues, while MRV kept solve times sub-millisecond in the easy regime with occasional spikes near the solvable/unsolvable boundary. These results show that random valid-at-placement givens rapidly become globally inconsistent beyond the mid-teens, explaining why practical Sudoku generation starts from a full solution, and they highlight MRV's strong payoff in pruning search on typical 9×9 boards.

Results

- **Experiment**

I ran a sweep over the number of initially locked clues numFixed from 0 to 40 in steps of 1. For each setting, I generated 50 random boards (valid-at-placement) and solved them with my DFS+stack backtracker using MRV (minimum remaining values) to pick the next cell. I recorded successes, failures (proved unsatisfiable before the cap), timeouts (hit an iteration cap of 200,000 iterations), and the average wall-clock time among successes (avgSolveMs(success)). These metrics tell us, respectively, the prevalence of solvability, the prevalence of contradiction, search blow-ups, and conditional speed on the subset that is solvable.

- **Hypothesis**

With very few clues, the grid is flexible, and many completions are possible, so I expect the success rate to be high and the solve times to be low. As clues increase to a moderate range, added constraints should further prune the search, potentially lowering the time. Past a threshold, however, random locally valid clues increasingly conflict globally, so the success rate should drop and many trials should fail quickly (MRV detects zero-option cells early). Near the solvable/unsolvable boundary, we should also see occasional timeouts from deep backtracks.

- **Table of results**

Table 1. Solve Outcomes vs. Number of Clues (Columns: numFixed, trials, successes, failures, timeouts, avgSolveMs(success), success_rate)

numFixed	trialCount	successes	failures	timeouts	avgSolveMs(success)	success_rate
0	50	50	0	0	0.830	1.0

1	50	50	0	0	0.631	1.0
2	50	50	0	0	0.455	1.0
3	50	50	0	0	0.359	1.0
4	50	50	0	0	0.355	1.0
5	50	50	0	0	0.345	1.0
6	50	50	0	0	0.326	1.0
7	50	50	0	0	0.320	1.0
8	50	50	0	0	0.301	1.0
9	50	50	0	0	0.295	1.0
10	50	50	0	0	0.305	1.0
11	50	50	0	0	0.261	1.0
12	50	49	0	1	0.264	0.98
13	50	48	1	1	0.254	0.96
14	50	49	1	0	0.274	0.98
15	50	49	1	0	0.344	0.98
16	50	47	2	1	0.811	0.94
17	50	43	6	1	1.425	0.86
18	50	48	1	1	0.524	0.96
19	50	42	6	2	3.171	0.84
20	50	34	16	1	0.317	0.68
21	50	30	20	0	0.432	0.60
22	50	29	21	1	0.206	0.58
23	50	26	24	0	0.176	0.52
24	50	21	29	0	0.167	0.42
25	50	15	35	0	0.206	0.30
26	50	13	37	0	0.661	0.26
27	50	5	45	0	0.149	0.10
28	50	6	44	0	0.154	0.12
29	50	2	48	0	0.138	0.04
30	50	1	49	0	0.088	0.02
31	50	0	50	0	0.0	0.0
32	50	1	49	0	0.089	0.02
33	50	0	50	0	0.0	0.0

34	50	0	50	0	0.0	0.0
35	50	0	50	0	0.0	0.0
36	50	0	50	0	0.0	0.0
37	50	0	50	0	0.0	0.0
38	50	0	50	0	0.0	0.0
39	50	0	50	0	0.0	0.0
40	50	0	50	0	0.0	0.0

The success rate holds at 1.00 for numFixed = 0…11, then drops to 0.98 at 12–15, 0.94 at 16, 0.86 at 17, 0.84 at 19, 0.68 at 20, and falls to ≤0.04 by 29–30, reaching 0.00 for most ≥31—i.e., the curve stays flat at 1.00 through 11 and then descends steadily toward zero by the low 30s. Randomly sprinkling valid-at-placement clues quickly creates globally inconsistent boards once you pass the mid-teens; by ~20 clues, most instances are unsatisfiable. This explains why practical Sudoku generation starts from a full solution and removes clues while preserving global consistency and uniqueness.

Conditional solve times are sub-millisecond in the easy region and dip around 10–15 clues (e.g., ~0.26–0.36 ms), then spike near 17–19 clues (e.g., 1.425 ms at 17; 3.171 ms at 19), marking a hardness ridge where constraints force deeper backtracking without immediate contradiction. Timeouts appear sparsely between 12–19 and vanish thereafter, not because later boards are easier, but because most are unsatisfiable and are refuted quickly by MRV.

- **Synthesis**

Empirically, random-placement boards are reliably solvable by DFS+MRV up to roughly 15 clues; between 16–20 clues, the solvability rate drops, and occasional hard instances emerge; past ~20, most random instances fail. For generations, this argues strongly against naive random givens; for solver design, it highlights the payoff of MRV, which front-loads forced moves, reduces branching, and prevents many fruitless dives that a row-major policy would trigger.

Acknowledgements

I would like to thank Professor Harper and Professor Al Madi for providing the project template and guidance site, as well as the TAs for guiding me through the debugging process. Thanks to Oracle and W3Schools for the detailed Java documentation.

Extensions:

What

I implemented four next-cell selection strategies inside Sudoku.java to study how variable ordering affects a DFS backtracker on randomly generated 9×9 boards:

1) Row-major (first empty cell), and the rule is to scan the board left-to-right, top-to-bottom; pick the first empty, unlocked cell. If it has at least one legal value (checked with validValue), assign the first legal value and continue; if it has no legal values, report a dead end to trigger backtracking. This

strategy ignores constraint structure and often branches on high-degree cells (lots of peers) with large domains (many legal values), thereby increasing the branching factor and backtracking depth on harder boards. I use this strategy because it's simple, has near-zero overhead, and gives me a control line to compare other strategies.

2) MRV (minimum remaining values), which is from the original project: Among all empty, unlocked cells, compute each cell's domain size (how many digits 1–9 are currently legal by row/col/box). Choose the cell with the smallest domain; if any cell's domain is zero, immediately return a dead end. Assign the next valid value to that cell (using findNextValue, which tries values > current).

3) MRV + Degree (MRV with the most-constraining tie-breaker by picking the variable that constrains the most others), and the rule is to start with MRV. If multiple cells tie on the smallest domain size, choose the one with the largest degree; i.e., the cell that intersects the most empty peers across its row, column, and 3×3 box. Then assign a legal value. This strategy design was inspired by the slides https://www.cs.cornell.edu/courses/cs4700/2011fa/lectures/05_CSP.pdf.

4) Random empty (pick a random empty cell that has a valid value), and the rule is to collect all empty, unlocked cells; shuffle; pick the first that has a legal move, and assign the next valid value. If none have a legal move, report a dead end. I made this strategy because it frequently branches into bad parts of the tree, causing many timeouts or failures on boards that MRV solves quickly. Thus, it is a stochastic control that shows how much informed ordering matters.

The goal was to quantify their impact on solvability and timeouts by varying the number of locked clues.

Outcome

I ran 50 trials per setting for numFixed = 0…40, recording successes, failures, timeouts, and avgSolveMs(success). MRV-family strategies maintain high solve rates longer than the baselines as clue count increases; row-major performs well on easy instances but degrades sooner; random is essentially non-viable (near-universal timeouts or failures). In the hardness ridge (≈15–22 clues), MRV+Degree often yields higher success rates than row-major and plain MRV on the same boards, though its per-success time can be spikier due to extra computation and deeper, but more targeted search.

Table 1. Strategy Comparison across Clue Counts (50 trials each). S/T: Successes/Timeouts

numFixed	Row-major S / T	MRV S / T	MRV+Degree S / T	Random S / T
0	50 / 0	50 / 0	50 / 0	0 / 50
5	48 / 2	50 / 0	50 / 0	0 / 50
10	46 / 4	50 / 0	50 / 0	0 / 50
12	48 / 2	50 / 0	50 / 0	0 / 50
13	46 / 3	50 / 0	48 / 2	0 / 50

numFixed	Row-major S / T	MRV S / T	MRV+Degree S / T	Random S / T
14	45 / 4	50 / 0	48 / 0	0 / 50
15	42 / 8	48 / 2	46 / 2	0 / 50
16	41 / 8	45 / 0	43 / 3	0 / 50
17	38 / 11	43 / 1	40 / 2	0 / 50
18	36 / 12	40 / 2	34 / 1	0 / 50
19	39 / 7	41 / 1	41 / 0	0 / 50
20	34 / 13	37 / 0	38 / 0	0 / 50
21	26 / 17	31 / 0	35 / 0	0 / 50
22	25 / 21	34 / 0	29 / 0	0 / 50
23	25 / 13	23 / 0	27 / 0	0 / 50
24	17 / 6	24 / 0	15 / 0	0 / 50
25	12 / 14	9 / 0	19 / 0	0 / 50
27	6 / 7	8 / 0	7 / 0	0 / 50
28	2 / 2	1 / 0	5 / 0	0 / 50
30	1 / 0	0 / 0	3 / 0	0 / 50
31	0 / 0	4 / 0	0 / 0	0 / 50
33	0 / 0	0 / 0	1 / 0	0 / 50
40	0 / 0	0 / 0	0 / 0	0 / 18

Success remains perfect (50/50) for MRV variants up to ~12–14 clues and then decays; row-major starts losing ground earlier (e.g., 10–15 show more timeouts); random virtually never solves and mostly times out. We observe that informed variable orderings (MRV, MRV+Degree) substantially increase solvability on the same boards, especially in the mid–low-20s, where naive ordering collapses. Random selection is not competitive for systematic solving. The original data set is in experiment_algorithms_results.txt under the ext folder.

How to run

Run and compile ExperimentAlgorithm.java. You can edit the code in the ExperimenAlgorithm class and choose a strategy via the constructor:

new Sudoku(numLocked, 0, 0) → row-major
new Sudoku(numLocked, 0, 1) → MRV (default)
new Sudoku(numLocked, 0, 2) → MRV+Degree
new Sudoku(numLocked, 0, 3) → random
Disable visualization for speed: pass delay = 0.