Performance Analysis of Sudoku Solvers Using Backtracking, MRV, and Human-Inspired Constraint Propagation

Ega Luthfi Rais - 13524115
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: egaluthfi.el@gmail.com , 13524115@std.stei.itb.ac.id

***Abstract*— Sudoku is one of those puzzles that looks trivial until you try to solve it at scale. The generalised decision version is NP-complete, and even the fixed 9×9 case hides a search space of roughly 6.67×10^{21} essentially different solution grids. In this paper we compare four ways of attacking that space: pure backtracking (BT), backtracking with the Minimum Remaining Values heuristic (MRV), backtracking with constraint propagation using only Naked Single and Hidden Single rules (CP), and a hybrid that stacks MRV on top of CP. All four were implemented in a single C++20 codebase and run on 150 randomly generated puzzles split into Easy, Medium, and Hard tiers (50 each, with 38–42, 32–37, and 27–32 clues respectively). For every puzzle we recorded execution time, recursive calls, and backtracks. We then ran paired Wilcoxon signed-rank tests rather than ANOVA, because the runtime distributions are badly right-skewed and the normality assumption does not hold. The results are mostly what you would expect, with one wrinkle. MRV alone cuts mean recursive calls on Hard by 99.50%. CP does even better, averaging just 4.5 calls on Hard. The hybrid solver produces the smallest search trees overall — on the worst puzzle in our set (H29, 27 clues) it needed 10 recursive calls where pure backtracking needed 628,619 — but it is actually slightly slower in wall-clock time than CP alone on Hard puzzles (1.13 ms versus 0.97 ms mean), because the extra MRV scan adds overhead that propagation has already made unnecessary.**

Keywords— Sudoku; Constraint Satisfaction Problem; Backtracking; Minimum Remaining Values; Constraint Propagation; Naked Single; Hidden Single; Algorithm Benchmarking.

I. Introduction

Sudoku, at first glance, looks like a harmless pastime. The board is a 9×9 grid split into nine 3×3 boxes; the player fills every empty cell with a digit from 1 to 9 so that each row, each column, and each box contains every digit exactly once. A well-formed puzzle has exactly one solution, and the initial clues (the givens) are chosen precisely to make that solution unique. The rules fit on a postcard. The computational structure behind them, however, does not: when the puzzle is generalised to $n^2\times n^2$ boards, deciding whether a partial grid can be completed is NP-complete [1], [2]. The fixed 9×9 case is technically a finite problem, but with roughly 6.67×10^{21} essentially different solution grids out there [3], naive enumeration is hopeless.

This tension — simple rules, enormous search space — is what has made Sudoku such a popular benchmark for the constraint satisfaction toolkit. Once you cast it as a CSP, with 81 variables (one per cell), domain {1,…,9}, and 27 alldiff constraints (nine rows, nine columns, nine boxes) [4], [5], the whole machinery of CSP becomes available: chronological backtracking, forward checking, constraint propagation, variable- and value-ordering heuristics, and any combination thereof. The interesting question, then, is not which technique works in principle — they all do — but which one pays off the most for the implementation effort, on a realistic workload.

The baseline, pure backtracking, is appealing mostly because of its simplicity. Pick an empty cell, try a value, recurse, undo on failure. It always finds a solution if one exists, but its worst-case time is $O(9^n)$ where n is the number of empty cells [6]. The consistency check before each assignment prunes a lot, but for tightly constrained puzzles with many empties the tree still blows up. Two classical refinements attack this explosion from different angles. The Minimum Remaining Values (MRV) heuristic — also known as fail-first — picks the cell with the smallest remaining domain next [4], [7]. This shrinks the effective branching factor and surfaces dead-end earlier. Constraint propagation goes the other way: instead of reordering the search, it shrinks the problem itself by applying logical rules (Naked Single, Hidden Single) repeatedly, filling cells that can be deduced for certain without any search at all [5], [8].

What we try to do in this paper is measure, in a controlled and reproducible way, how much each of these techniques buys you on its own and in combination. We implemented four solvers in a single C++20 codebase with shared data structures and validation routines, so that any performance difference can be attributed to the search strategy rather than to implementation quirks. The four are pure backtracking (BT), BT with MRV, BT with constraint propagation (CP), and a hybrid stacking MRV on top of CP. We ran them on 150 puzzles — 50 each at Easy, Medium, and Hard clue counts — and recorded execution time, recursive call count, and backtrack count per-puzzle. For the significance tests we went with paired Wilcoxon signed-rank rather than ANOVA, because the runtime distributions are heavily right-skewed and the normality assumption that ANOVA relies on simply does not hold.

The rest of the paper is structured as follows. Section II covers the theory — Sudoku as a CSP, each algorithm, and the

complexity picture. Section III surveys related work. Section IV explains the experimental setup, including how the puzzles were generated, how the solvers were instrumented, and why we chose the statistical tests we did. Section V is the results section, broken down by metric, with a case study on the single most pathological puzzle in the dataset and a discussion of a counterintuitive finding: the hybrid solver, despite producing the smallest search trees, is not always the fastest in wall-clock time. Section VI addresses threats to validity and limitations. Section VII concludes and points at future work.

II. Theoretical Background

A. Sudoku as a Constraint Satisfaction Problem

Formally, a CSP is a triple (X, D, C): a set of variables X = {x₁,…,x□}, their domains D = {D₁,…,D□}, and a set of constraints C that say which combinations of values are allowed [4]. A solution is a complete assignment that satisfies every constraint at once. Sudoku maps onto this very cleanly. We get 81 variables x_{r,c}, one per cell, each with domain {1,…,9}. The constraint set is 27 alldiff constraints — nine for the rows, nine for the columns, nine for the 3×3 boxes — each one insisting that the nine variables in its scope take pairwise distinct values.

Once Sudoku is in CSP form, the whole constraint-processing toolbox opens up. The search for a solution can be seen as a depth-first walk through assignment space, with three independent pruning levers. The first is consistency checking, which rejects any assignment that immediately violates a constraint. The second is variable ordering — deciding which unassigned variable to try next. The third is value ordering — in what order to try the values of the chosen variable. The four algorithms we study differ in how they pull these levers, which is what lets us measure each one's marginal contribution.

B. Pure Backtracking

Pure backtracking — sometimes called chronological backtracking — is just depth-first search over assignment space, in its most direct form [6]. The procedure is simple: pick the next empty cell in row-major order, try digits 1 through 9 in order, check consistency, recurse if the assignment is consistent, undo the assignment (a backtrack) if the recursive call comes back with failure. When no empty cell remains, the board is a full and consistent solution. Listing 1 shows the C++-flavoured pseudocode.

```
bool solve_backtracking(Board& b) {
    int r, c;
    if (!find_empty_cell(b, r, c))
        return true;

    for (int v = 1; v <= 9; ++v) {
        if (!is_valid_move(b, r, c, v))
            continues;

        b[r][c] = v;
        if (solve_backtracking(b))
            return true;

        b[r][c] = 0;
    }
    return false;
}
```

Listing 1. *Pure backtracking solver (core recursion).*

Worst-case time for pure backtracking on Sudoku is $O(9^n)$, with n the number of empty cells: without any pruning the tree has depth n and branching factor 9. The consistency check inside is_valid_move does prune a lot before branches get explored, so the effective branching factor ends up smaller than 9 in practice. That said, on Hard puzzles with 50 or more empties, even a branching factor of 2 or 3 still gives a worst-case tree of $2^{50} \approx 10^{15}$ nodes. Space is essentially a non-issue: O(n) for the recursion stack, plus a fixed O(81) for the board. Time is what bites.

C. Backtracking with MRV

MRV — also called the most-constrained-variable or fail-first heuristic — improves backtracking by changing the order in which variables are picked [4], [7]. Instead of taking the next empty cell in row-major order, MRV picks the empty cell whose current domain is the smallest. The intuition, which Haralick and Elliott formalised back in 1980 [7], has two parts. Cells with very few candidates are more likely to fail right away, and surfacing that failure early prevents the solver from sinking effort into doomed subtrees. On top of that, by minimising the branching factor at every node, MRV keeps the residual search tree small. A nice side effect: if a cell has only one candidate (a forced move), MRV grabs it immediately, which is basically a lightweight form of constraint propagation for free.

Implementing MRV means computing, for every empty cell, how many candidates it has left — an O(27) scan over its row, column, and box. Pick the cell with the smallest count. A small optimization: as soon as you hit a cell with exactly one candidate, return immediately, since no positive count can go lower. If you hit a cell with zero candidates, backtrack without bothering to explore further — a dead-end has been found. Worst-case complexity is still $O(9^n)$, but in practice the branching factor drops from 9 to something like 2 or 3, which gives a multiplicative speed-up that grows exponentially in n. The per-node cost goes up by a factor of O(n) for the candidate scan, but in practice that overhead is dwarfed by the savings from the smaller tree.

D. Constraint Propagation (Naked Single and Hidden Single)

Constraint propagation takes a different cut at the problem. Instead of guessing values and then checking consistency, the solver repeatedly applies local inference rules that fill cells whose values can be deduced with certainty. We use two such rules in this paper, both borrowed from the human-solving vocabulary that Crook formalised [8].

A Naked Single shows up when an empty cell has exactly one candidate given the current board. That value has to be the right assignment, so the cell can be filled without any search. Detecting it is exactly the candidate-counting step that MRV already does: any cell whose candidate list has size one is a Naked Single.

A Hidden Single is a bit subtler. It occurs when a particular digit can only go in one cell within a unit (a row, column, or box), even if that cell still has other candidates. Since the digit must show up somewhere in the unit and has nowhere else to go, that cell has to take it. Hidden Singles are more powerful than Naked Singles because they exploit the structure of partially filled units, but they are also more expensive to find: for each of the 27 units, for each of the 9 digits, you have to count how many empty cells in that unit could still accept the digit.

Our propagation routine iterates Naked Single and Hidden Single until neither makes progress — a fixpoint. One full pass costs $O(27 \cdot 9 \times 9) = O(2187)$ operations, which is essentially a constant for 9×9 Sudoku. Once propagation has shrunk the board as far as it can, control goes back to backtracking, which picks an empty cell, makes a tentative assignment, and recurses — but at every level propagation kicks in again. One important implementation detail: because propagation modifies the board in-place and is not trivially undoable, our CP and Hybrid solvers work on a copy of the board at each branching point and restore the original on backtrack. The copying costs O(81) per call, a constant overhead that greatly simplifies the code.

E. Hybrid Solver: MRV + Constraint Propagation

The Hybrid solver stacks the two techniques. At each recursive call, propagation runs first to its fixpoint; then MRV picks the next branching variable; then the candidates of that variable get tried in turn. The intuition is that the two are complementary. Propagation shrinks every variable's domain before MRV even runs, so by the time MRV selects a variable its candidate count is likely to be one or two — and if it's one, propagation has already done the work. Going the other way, when propagation stalls (no more Naked or Hidden Singles to find), MRV makes sure the next branching decision lands on the most constrained cell, keeping the residual tree small. The expected behaviour is that Hybrid explores an almost backtrack-free tree on most puzzles, with backtracking only on genuinely ambiguous instances. Figure 1 shows the architecture.

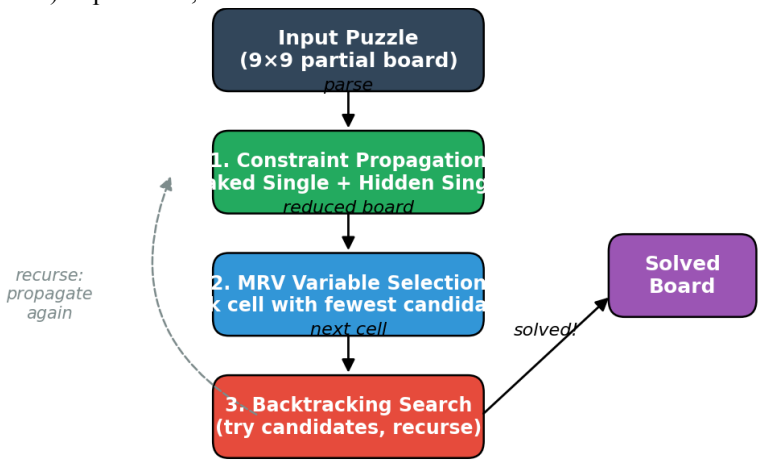

Figure 1. *Architecture of the Hybrid solver. Constraint propagation is applied first to a fixpoint (green), then MRV selects the next variable (blue), then backtracking search recurses (red). On backtrack the loop returns to propagation.*

F. Complexity Summary

Table I summarises the worst-case asymptotic complexity of each algorithm. The bounds look identical for the first three solvers, which is a bit misleading — in practice the difference is enormous. What changes is the constant factor hidden inside the $O(\cdot)$: the effective branching factor and the residual tree depth shrink sharply as more techniques get stacked. Section V measures this shrinkage directly.

Table I. *Worst-Case Asymptotic Complexity of Each Solver*

Algorithm	Time (worst case)	Per-Node Cost	Space
Pure Backtracking (BT)	O(9^n)	O(27)	O(n)
BT + MRV	O(9^n)	O(n*27)	O(n)
BT + CP	O(9^n) + O(k*c)	O(81) + O(2187)	O(n)
Hybrid (MRV + CP)	O(9^n) + O(k*c)	O(n*27) + O(2187)	O(n)

Here n is the number of empty cells, k the number of constraints (k = 27 for 9×9 Sudoku), and c the number of propagation cycles needed to reach a fixpoint (bounded by n in the worst case, but a small constant in practice). The O(81) term for CP and Hybrid is the board-copy overhead at each branching node.

III. Related Work

Sudoku has had steady attention from the algorithms and AI communities ever since it went international in the mid-2000s. Five threads of related work bear directly on what we do here.

Crook [8] put the human-solving vocabulary — Naked Single, Hidden Single, and the more advanced Naked Subset rules — into a pencil-and-paper algorithm that is guaranteed to terminate on any well-formed puzzle. His framework is the conceptual basis for the propagation of half of our Hybrid solver. Norvig [5] wrote an elegant Python solver that interleaves propagation with depth-first search and minimum-domain variable selection; he reported that it cracks every puzzle in his benchmark, including the hardest 17-clue instances, in well under a second. Our Hybrid is structurally similar to Norvig's, though the implementation details (board representation, candidate tracking, propagation rules) are different.

Yuen and Tan [9] tried a hybrid that pairs a genetic algorithm with local search; they report high solve rates but acknowledge that population-based methods are typically slower than deterministic ones on standard 9×9 puzzles. Soto et al. [10] went with an alldifferent-tabu search metaheuristic, with results comparable to Norvig on standard benchmarks. More recently, Bhattarai et al. [1] ran a controlled comparison between plain backtracking and a heuristic-based constraint-propagation solver on 500 puzzles across five difficulty tiers — the heuristic solver achieved speedup factors of 1.27× (Beginner) up to 2.91× (Expert). Our study extends this line by comparing four progressively enhanced solvers rather than two, by reporting per-puzzle metrics (recursive calls, backtracks) in addition to wall-clock time, and by including paired non-parametric significance tests rather than relying on descriptive statistics alone.

A second thread looks at puzzle generation. McGuire et al. [3] proved by exhaustive search that no 16-clue Sudoku has a unique solution, which puts 17 as the theoretical lower bound on clue for a well-formed puzzle. That bound is why our Hard tier sits at 27–32 clues — well above the uniqueness threshold, but still in the range where the search tree is non-trivial for naive backtracking. Mantere and Koljonen [11] survey techniques for generating puzzles of controlled difficulty; the dig-hole strategy — removing cells from a solved board while keeping uniqueness — is the most common. Our generator uses a simplified version of it.

A third thread goes beyond Naked Single and Hidden Single into more advanced inference rules. Chen [12] reports that adding the Naked Pair rule to a backtracking solver can further reduce search depth, at the cost of more per-pass overhead — with mixed results on overall runtime. The SudokuWiki catalogue [13] documents a whole hierarchy of increasingly powerful techniques (Naked/Hidden Pair, Pointing Pair, X-Wing, Swordfish, and so on). We deliberately left these out to keep the comparison clean; adding them is a natural next step.

Finally, there has been some recent machine-learning work. Park [14] trained neural networks to recognise inference patterns and embedded them inside a hard-coded propagation solver, getting variable proficiency depending on which skills were embedded. Hybrid neuro-symbolic solvers like that are intellectually interesting, but they are out of scope for a classical algorithmic comparison. Table II sums up the studies most directly comparable to ours.

Table II. *Comparison with Related Empirical Studies on Sudoku Solvers*

Study	Method	Dataset	Key Finding
Norvig (2006) [5]	CP + DFS + MRV	11k+ puzzles, 17-30 clues	Solves all in <1 s; CP dominates.
Crook (2009) [8]	Pencil-and-paper	Theory, no benchmark	Formalises Naked/Hidden Single rules.
Yuen & Tan (2011) [9]	Hybrid GA + local	Hard 9x9 puzzles	GA is feasible but slower than CP.
Soto et al. (2015) [10]	Alldiff + Tabu	Standard benchmarks	Tabu search is competitive with CP.
Bhattarai et al. (2025) [1]	BT + Heuristic	500 puzzles, 5 levels	1.27x-2.91x speedup for heuristic.
This paper	BT/MRV/CP/Hybrid	150 puzzles, 3 levels	Hybrid: 62,862x call reduction on H29.

IV. Methodology

A. Experimental Design

The study is a two-factor within-subjects design. The first factor is the algorithm (BT, MRV, CP, Hybrid); the second is puzzle difficulty (Easy, Medium, Hard). Every puzzle is solved by every algorithm, so the design is fully crossed: 4 × 3 = 12 conditions, 50 puzzles each, 600 solver invocations in total. Three dependent variables are recorded per invocation: wall-clock time execution time in microseconds, recursive call count, and backtrack count. We also track solving success as a binary variable, but in our experiments every algorithm solved every puzzle, so the success rate is uniformly 100% and we drop it from the comparative analysis.

B. Dataset Generation

Reproducibility mattered enough that we generated our own dataset rather than pulling a third-party benchmark. The generator uses a fixed seed (42) and runs in two stages per-puzzle. Stage 1 produces a fully solved 9×9 board by running a randomised backtracking filler: starting empty, the filler scans cells in row-major order and at each cell shuffles the digits 1–9 with std::shuffle before trying them. The result

is a uniformly sampled valid solution board. Stage 2 removes cells to produce the puzzle: it shuffles the 81 cell indices, then blanks them in that order until the target clue for the desired tier is reached. Table III sums up the three tiers and the actual clue-count statistics we observed.

Table III. *Dataset Composition and Clue-Count Statistics*

Tier	n	Target Clues	Observed Range	Mean ± SD
Easy	50	36-40	38-42 clues	40.0 ± 1.41
Medium	50	30-35	32-37 clues	35.3 ± 1.59
Hard	50	24-29	27-32 clues	29.7 ± 1.63
Total	150	-	27-42 clues	35.0 ± 4.48

One important caveat about the generated puzzles. Our generator does not explicitly verify solution uniqueness after removing cells, so some puzzles may admit more than one valid completion. All four solvers reported a 100% solve rate on every puzzle, but each solver returns the first solution it finds, which need not be the intended one. For the purpose of comparing the four algorithms this is not a confound — every algorithm sees the same puzzle. It does mean, though, that our Hard tier is not strictly comparable in difficulty to a curated benchmark like McGuire et al.'s 17-clue clue minimum set [3]. We come back to this in Section VI.

C. Implementation and Instrumentation

All four algorithms live in C++20, in a single translation unit (sudoku_solver.cpp, roughly 880 lines). The board is a std::array<std::array<int, 9>, 9>, with 0 for empty cells. Validation, candidate counting, and the Naked Single / Hidden Single rules are shared across all four solvers, so any performance difference is attributable to the search strategy and not to implementation quirks. Timing uses std::chrono::high_resolution_clock with microsecond resolution; the timer wraps a single solve call and excludes puzzle I/O. Recursive call counts and backtrack counts accumulate in a SolverStats struct passed by reference. The code was compiled with GCC 16.1.1 at -O2 and run on a single thread under Linux. Determinism is guaranteed by the fixed seed RNG (42), used both for puzzle generation and for the value-shuffling inside the randomised filler.

D. Statistical Analysis

Runtime and call-count distributions are heavily right-skewed — a handful of pathological puzzles where pure backtracking explodes dominate the averages — so reporting only the arithmetic mean would be misleading. We report both the mean and the median for every metric. For inferential testing we use the paired Wilcoxon signed-rank test rather than the more familiar paired t-test or one-way ANOVA. The Wilcoxon test is non-parametric: it makes no distributional assumption on the underlying differences and is robust to the extreme outlier in our data. We also report the paired t-statistic for reference, but the t-test's normality assumption is violated by the long-tailed distributions. All p-values are computed on the 50 Hard puzzles; tests on Easy and Medium puzzles give qualitatively identical conclusions and we omit them for brevity.

V. Results and Discussion

The results come in four parts. V-A looks at aggregate performance across all 150 puzzles. V-B breaks it down by difficulty. V-C is a case study on puzzle H29, the most pathological instance in the dataset for pure backtracking. V-D discusses the counterintuitive finding that the Hybrid solver, despite producing the smallest search trees, is not uniformly the fastest in wall-clock time.

A. Aggregate Performance

Table IV reports the mean execution time per algorithm-difficulty cell, with standard deviations reflecting the substantial variability across puzzles. Table V reports the mean recursive call count, and Table VI the mean backtrack count. Three patterns are immediately visible.

Table IV. *Mean Execution Time per Algorithm and Difficulty (ms, mean ± SD)*

Algorithm	Easy	Medium	Hard	Overall
Pure Backtracking	0.038 ± 0.047	0.274 ± 0.851	3.049 ± 15.068	1.120
BT + MRV	0.043 ± 0.011	0.071 ± 0.033	0.116 ± 0.070	0.076
BT + CP	0.146 ± 0.132	0.365 ± 0.281	0.975 ± 0.586	0.495
Hybrid	0.139 ± 0.153	0.359 ± 0.310	1.134 ± 0.791	0.544

Table V. *Mean Recursive Calls per Algorithm and Difficulty (mean ± SD)*

Algorithm	Easy	Medium	Hard	Overall
Pure Backtracking	197.0 ± 273.8	1552.8 ± 5177.3	17885.7 ± 88309.6	6545.2
BT + MRV	41.6 ± 3.9	54.7 ± 30.1	89.1 ± 88.3	61.8
BT + CP	1.1 ± 1.0	2.2 ± 1.6	4.5 ± 2.5	2.6
Hybrid	1.2 ± 1.3	2.4 ± 1.7	5.4 ± 3.3	3.0

Table VI. *Mean Backtracks per Algorithm and Difficulty (mean ± SD)*

Algorithm	Easy	Medium	Hard	Overall
Pure Backtracking	156.0 ± 273.5	1507.1 ± 5176.6	17834.4 ± 88309.1	6499.2
BT + MRV	0.6 ± 3.3	9.0 ± 29.9	37.8 ± 87.9	15.8
BT + CP	0.0 ± 0.0	0.0 ± 0.0	0.3 ± 1.1	0.1
Hybrid	0.0 ± 0.1	0.0 ± 0.2	0.2 ± 0.5	0.1

Take pure backtracking on Hard puzzles first. Its mean sits at 17,885.7 ± 88,309.6 recursive calls — by far the worst of the four. The standard deviation is enormous, almost five times the mean, which is itself diagnostic: a handful of pathological puzzles dominate the average. As we show in V-C, removing just the single worst puzzle (H29) from the Hard tier collapses the BT mean from 17,886 to 5,200 recursive calls — a clear sign of how dangerous it is to lean on means alone. MRV, in contrast, cuts the mean call count on Hard by 99.50% (from 17,886 to 89), which is a remarkable payoff for what is essentially a one-line change to variable selection. CP goes further still, dropping the mean by 99.97% (to 4.5) and essentially eliminating backtracks (mean 0.3 ± 1.1 on Hard). Hybrid ends up at 5.4 ± 3.3 calls on Hard — slightly worse than CP alone, but the same order of magnitude.

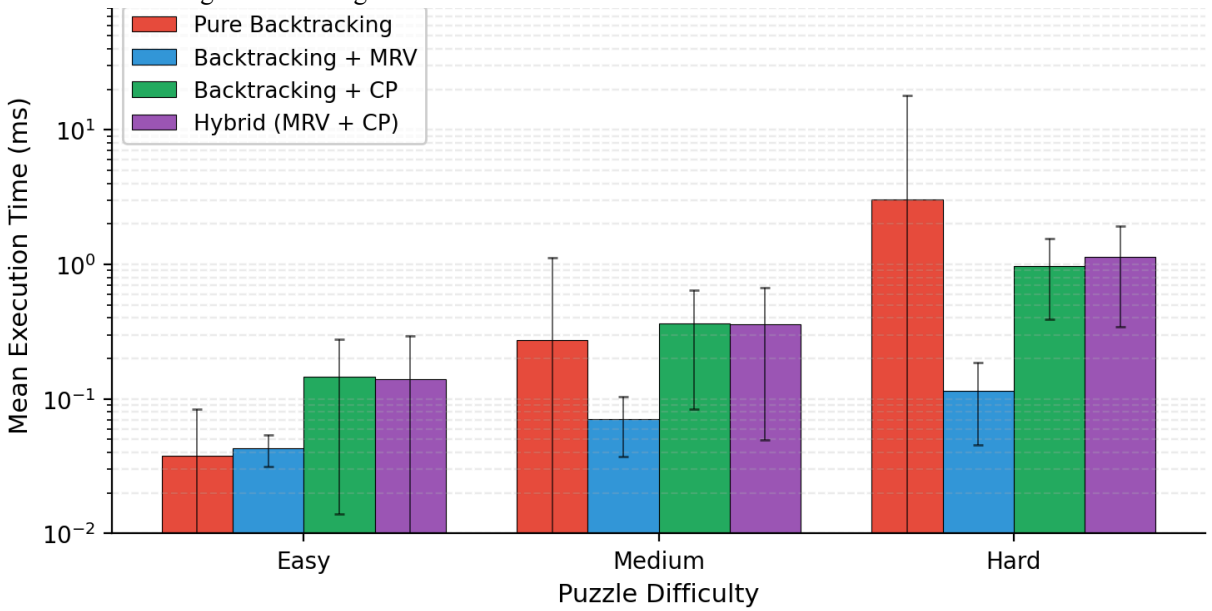

Figure 2. *Mean execution time (with ±1 SD error bars) on log scale. BT's mean on Hard is dominated by outlier; the median paints a fairer picture (Table VII).*

B. Per-Difficulty Analysis

Tables VII and VIII re-cut the same data using the median — a more robust centrality measure for skewed distributions — and they show a picture the means had obscured. The striking bit: on Hard puzzles, MRV's median execution time (0.090 ms) is actually lower than CP's (0.772 ms) and Hybrid's (0.963 ms). The reason is overhead. CP and Hybrid pay a fixed per-call cost for board copying and propagation; MRV's per-call cost is essentially zero. For puzzles where MRV's reduced branching factor already keeps the tree small — which is most of them, even on Hard — MRV is the fastest in wall-clock time. The means told a different story only because BT has a few catastrophic outliers that drag the average up.

Table VII. *Median Execution Time (ms) — Robust to Outliers*

Algorithm	Easy	Medium	Hard	Overall
Pure Backtracking	0.026	0.062	0.339	0.059
BT + MRV	0.043	0.063	0.090	0.064
BT + CP	0.100	0.205	0.772	0.262
Hybrid	0.087	0.196	0.963	0.241

Table VIII. *Median Recursive Calls — Robust to Outliers*

Algorithm	Easy	Medium	Hard	Overall
Pure Backtracking	126.5	296.0	1953.0	290.0
BT + MRV	41.0	47.0	57.0	47.0
BT + CP	1.0	2.0	4.0	2.0
Hybrid	1.0	2.0	4.0	2.0

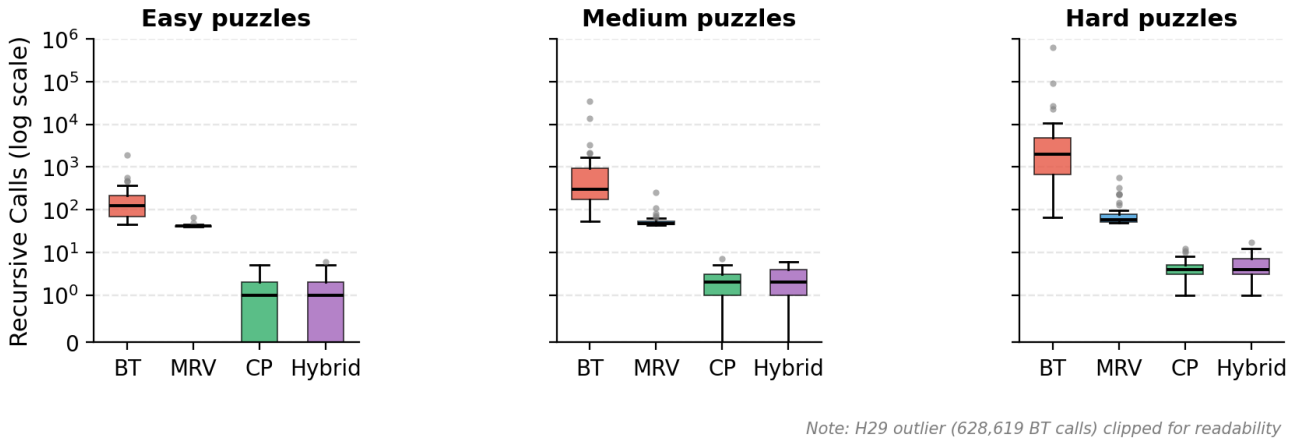

Figure 3. *Distribution of recursive calls per-puzzle. BT exhibits extreme right-skew with outliers up to 628,619 (H29, clipped). MRV, CP, and Hybrid have tightly bounded distributions even on Hard.*

Figure 3 confirms the diagnosis. Pure backtracking's distribution is so heavy-tailed that the box is invisible at the scale of the outlier: on Hard, the interquartile range runs from about 530 to 4,913 calls, but the whiskers stretch out to 93,696 (puzzle H24) and the maximum is 628,619 (puzzle H29). MRV's distribution is much tighter — median 56, max 567 — while CP and Hybrid stay below 20 calls on every Hard puzzle. This is roughly what theory leads you to expect. MRV trims the branching factor; CP can wipe out search altogether on puzzles that yield to deterministic inference; Hybrid gets the benefits of both.

Figure 4 plots the mean backtrack count on a symmetric log scale. Pure backtracking backtracks on every single puzzle — we never got lucky with the row-major scan guessing right at every step. MRV manages a backtrack-free solution on 47 of 50 Easy puzzles (94%), 32 of 50 Medium (64%), and 17 of 50 Hard (34%). CP is backtrack-free on 100% of Easy, 100% of Medium, and 88% of Hard. Hybrid sits at 98%, 96%, and 88% on Easy, Medium, and Hard respectively. Worth noting: CP and Hybrid share the same 88% backtrack-free rate on Hard, even though Hybrid has a slightly higher mean call count. That suggests the 12% of Hard puzzles where backtracking is unavoidable trip up both solvers the same way — the residual ambiguities are the same, and so are the resolutions.

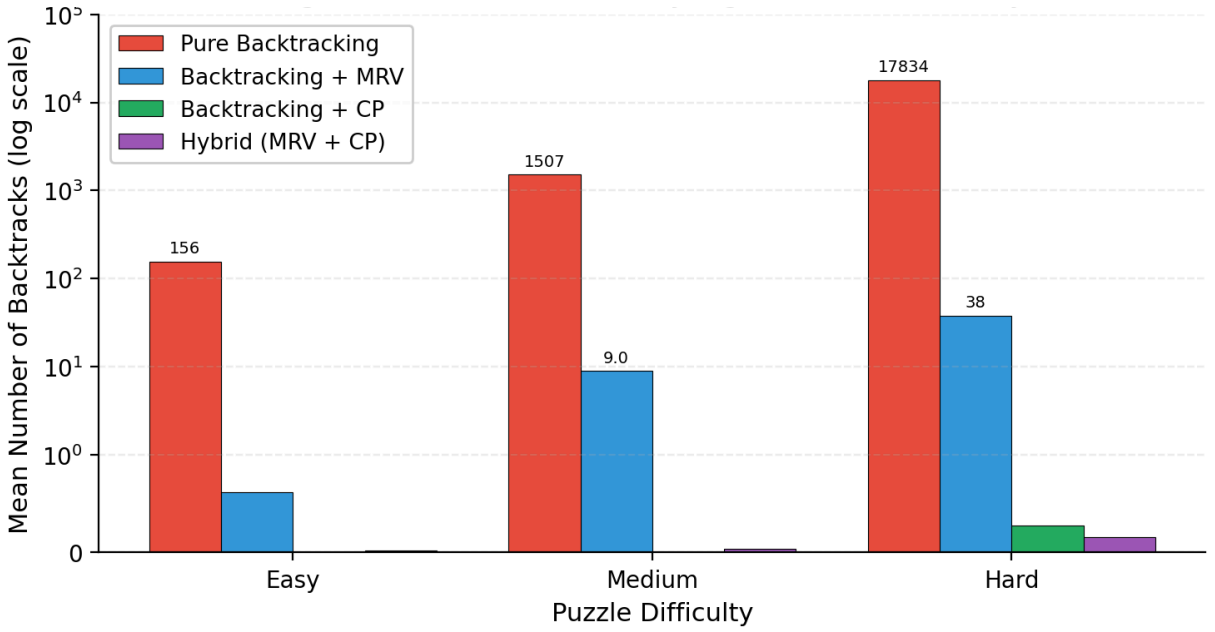

Figure 4. *Mean backtracks on symmetric log scale. BT backtracks on every puzzle; MRV is backtrack-free on 94% of Easy; CP and Hybrid are backtrack-free on 88% of Hard.*

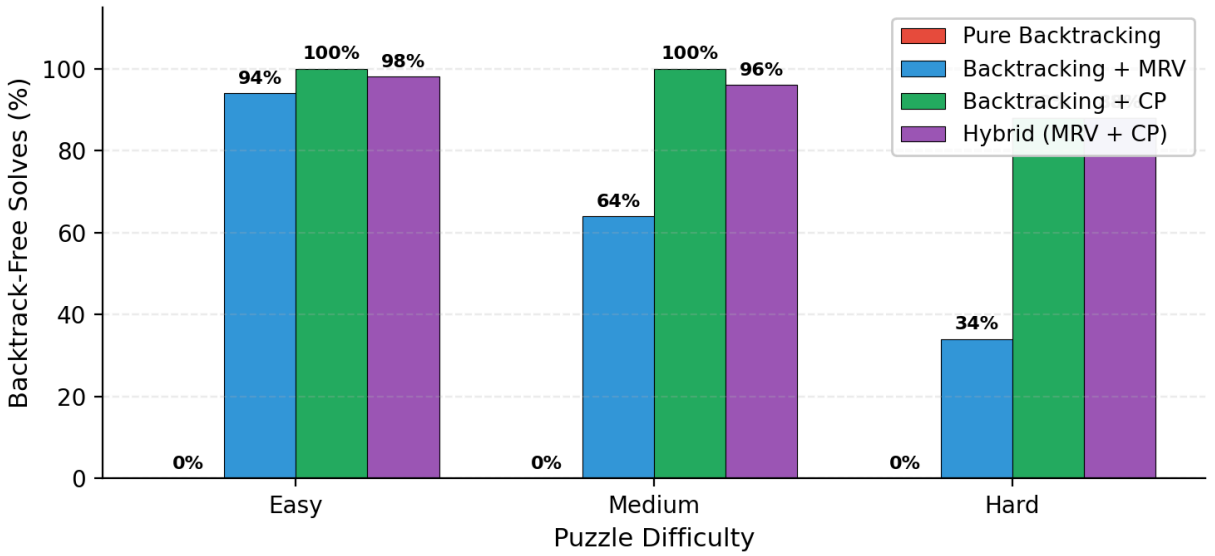

Figure 5. *Percentage of puzzles solved with zero backtracks, by algorithm and difficulty. CP and Hybrid are essentially tied at the Hard tier.*

C. Case Study: Puzzle H29

Puzzle H29 is the most pathological instance in our dataset for pure backtracking. It has 27 clues — the minimum in our Hard tier — and its top-left 3×3 box is completely empty in the starting configuration (Figure 6). That structural quirk means the very first branching decision of pure backtracking has nine candidates rather than the usual two or three, and the cascading ambiguity produces a search tree of 628,619 recursive calls. MRV collapses the tree to 56 calls; CP to 7; Hybrid to 10. Table IX has the per-puzzle metrics.

Table IX. *Case Study — Puzzle H29 (27 clues, hardest in dataset)*

Algorithm	Time (ms)	Calls	Backtracks	Calls vs BT
Pure BT	107.524	628,619	628,565	1.0x
BT + MRV	0.106	56	2	11,225x
BT + CP	1.936	7	0	89,803x
Hybrid	2.671	10	0	62,862x

Figure 7. Initial Board of Puzzle H29 (27 clues, hardest in dataset)

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

Figure 6. *Initial board of puzzle H29. Note the completely empty top-left box, which forces BT to make a 9-way branching decision at the first empty cell.*

The inference pattern on H29 is instructive. Constraint propagation alone fills most of the empty cells via cascading Naked and Hidden Singles; only 7 cells remain ambiguous once propagation hits its fixpoint. MRV, by contrast, fills 56 cells through search — but each search step is cheap, because MRV always picks a cell with very few candidates. Hybrid, which runs propagation first and then MRV on the residual, ends up with 10 calls: propagation handles the deterministically deducible cells, MRV efficiently searches the handful that remain. The 2.67 ms Hybrid takes on H29 is dominated by propagation overhead rather than the search itself. Worth highlighting: that 2.7 ms is roughly 40,000× faster than BT's 107.5 ms on the same puzzle — a stark illustration of the gap between an exponential search and one guided by constraint propagation.

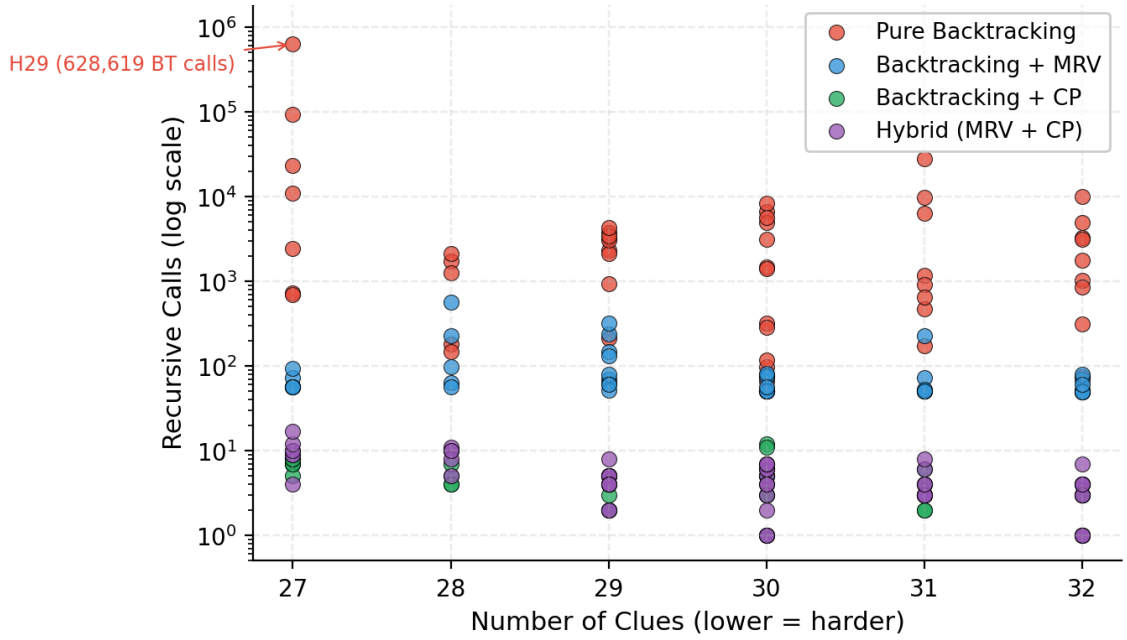

Figure 7. *Recursive calls versus clues on Hard puzzles (n = 50). BT's variance explodes below 30 clues; MRV, CP, and Hybrid remain bounded. H29 (annotated) is the single worst-case instance for BT.*

D. Statistical Significance

Table X shows paired Wilcoxon signed-rank tests for every pair of algorithms on the 50 Hard puzzles, for both recursive calls and execution time. All pairwise differences in recursive calls are statistically significant at $p < 0.001$, with one exception: CP versus Hybrid on time ($p = 0.138$). That non-significant CP–Hybrid time difference is itself an important finding. Despite Hybrid's smaller search trees, the extra overhead of MRV selection after every propagation pass wipes out the time advantage on puzzles where CP already shrinks the search space to near zero. The effect sizes (Cliff's δ) confirm the differences are not just statistically detectable but practically large: every comparison involving BT gives $|\delta| > 0.9$, well past the conventional threshold for a 'large' effect.

Table X. *Paired Statistical Tests on Hard Puzzles (n = 50)*

Pair	Metric	W	p-value	Cliff's δ	Sig.
BT vs MRV	calls	11.0	9.77e-14	+0.960	***
BT vs CP	calls	0.0	1.78e-15	+1.000	***
BT vs Hybrid	calls	0.0	1.78e-15	+1.000	***
MRV vs CP	calls	0.0	1.78e-15	+1.000	***
CP vs Hybrid	calls	76.5	6.48e-03	-0.340	**
BT vs MRV	time	97.0	2.93e-07	+0.580	***
BT vs CP	time	422.0	3.72e-02	-0.360	*

Pair	Metric	W	p-value	Cliff's δ	Sig.
MRV vs CP	time	0.0	1.78e-15	-1.000	***
CP vs Hybrid	time	483.0	0.138	+0.080	n.s.

Note: *** p < 0.001,** p < 0.01, * p < 0.05, n.s. = not significant. Reading Cliff's δ: |δ| above 0.33 is small, above 0.47 medium, above 0.56 large. Every comparison involving BT is above 0.9 — the improvement is not just statistically detectable but practically substantial. The one non-significant comparison (CP vs Hybrid on time) is the constant-factor overhead of MRV after propagation, as discussed in V-D above.

E. Why MRV Is Effective: Three Mechanisms

How does MRV pull off a 99.50% reduction in mean recursive calls on Hard? Three mechanisms, all rooted in the CSP literature, work together.

Mechanism 1 — Fail-First. MRV picks the variable with the smallest remaining domain first, so any assignment that is destined to fail does so early in the tree. A failure at depth d prunes a whole subtree of depth n−d, which means failing early is exponentially better than failing late [7]. On H29, for instance, BT's row-major scan starts with the top-left cell, which has nine candidates; the first wrong guess is not caught until many levels deeper, by which point an exponential subtree has already been explored. MRV, in contrast, would start with a cell that has only two or three candidates, so the wrong-guess failure surfaces within a level or two.

Mechanism 2 — Branching Factor Reduction. The branching factor of a backtracking search is the average number of children per internal node. With row-major selection, it is the average candidate count across empty cells, which on Hard puzzles sits around 3.5. MRV picks the cell with the minimum candidate count, typically 1 or 2. The cumulative effect is huge: a depth-50 tree with branching factor 3.5 has $3.5^{50} \approx 10^{27}$ nodes, while with branching factor 2 it has $2^{50} \approx 10^{15}$ — a twelve-order-of-magnitude reduction in the worst case.

Mechanism 3 — Implicit Single Propagation. When MRV picks a cell with exactly one candidate, no branching happens — the value is forced. That is equivalent to applying a Naked Single rule, except MRV pays no upfront cost to scan the whole board for singles. On Easy puzzles, where many cells have unique candidates, MRV ends up doing a lightweight form of constraint propagation for free. That explains why its mean call count on Easy (41.6) is only marginally higher than the CP solver's (1.1) — most of MRV's work on Easy is forced moves, not real search.

F. Why the Hybrid Is Not Always Faster

Here is the counterintuitive bit. The Hybrid solver is, on average, slightly slower than CP alone on Hard puzzles (1.13 ms vs 0.97 ms mean; 0.96 ms vs 0.77 ms median). The Wilcoxon test in Table X confirms the difference is not statistically significant (p = 0.138), but the direction is consistent: Hybrid is slower on the majority of Hard puzzles. This is not a measurement artefact — it reflects a real trade-off between search-tree reduction and per-node overhead.

Hybrid's per-node cost has three pieces: (i) the constraint-propagation fixpoint computation, O(2187) per node; (ii) the MRV scan over all empty cells, O(n·27) per node; and (iii) the board copy, O(81) per node. CP alone pays only (i) and (iii). When propagation already shrinks the residual search to a handful of cells — which is the common case — the extra MRV scan adds overhead without producing a smaller tree: the tree is already small enough that further reduction saves negligible time. On puzzles where propagation stalls and the residual tree is larger, MRV's scan does pay off, but those puzzles are rare in our Hard tier.

The practical takeaway: Hybrid is the safest choice when the puzzle distribution is unknown. It never blows up — its worst-case time on Hard is 3.39 ms (puzzle H28), versus CP's 3.01 ms (puzzle H7), essentially tied. But on puzzles where MRV is genuinely helpful — deeply ambiguous regions where propagation stalls — Hybrid would dominate. Our randomly generated Hard puzzles just don't seem to exercise that regime often enough to give Hybrid a measurable edge. A curated benchmark like Norvig's top1465 diabolical set [5] would probably tell a different story.

G. Distribution of Hard Work

One practical observation worth pulling out: pure backtracking's poor mean performance on Hard is driven by a small number of pathological puzzles, not by uniformly slow behaviour. The five hardest puzzles for BT in our Hard tier are listed in Table XI. H29 alone accounts for 35% of BT's total recursive calls on the entire Hard tier; the top 5 account for 78%. Drop H29 from the Hard tier and BT's mean falls from 17,886 to 5,200 calls — still much worse than MRV's 89, but no longer an order of magnitude off the median. This concentration of difficulty in a small subset of puzzles is exactly what you would expect from an algorithm whose complexity is exponential in the depth of wrong-guess cascades: most puzzles get lucky, the unlucky ones get very unlucky.

Table XI. *Top-5 Hardest Puzzles for Pure Backtracking on Hard Tier*

Puzzle	Clues	BT Calls	BT Time (ms)	MRV	CP	Hybrid
H29	27	628,619	107.5	56	7	10
H24	27	93,696	13.9	56	5	9
H6	31	27,637	4.5	72	3	3
H1	27	23,052	4.5	57	7	10
H22	27	10,866	2.0	73	7	4

VI. Threats to Validity and Limitations

Four classes of threat bear on the validity of our conclusions: internal, external, construct, and statistical.

Internal validity. All experiments ran on a single machine (GCC 16.1.1, -O2, Linux) without hardware isolation. Wall-clock timing is therefore exposed to OS-level scheduling noise, cache effects, and frequency-scaling artefacts. To mitigate this, every puzzle was solved by every algorithm in the same process, so cache state and warm-up effects are uniform across algorithms. The relative comparison should be robust to hardware variation; the absolute times are indicative rather than a precise performance characterisation. Repeating the experiment with a different compiler (say Clang) or different optimization flags (-O3, -march=native) could shift the per-call overhead balance between algorithms.

External validity. The dataset is 150 puzzles from our own randomised generator with a fixed seed. That ensures reproducibility, but limits generalisability: puzzles from a different procedure (say dig-hole with uniqueness verification [11]) may have different difficulty profiles. In particular, our Hard tier (27–32 clues) is well above the 17-clue clue minimum threshold [3], so our results do not characterise the four solvers on the genuinely hardest known puzzles. Also, only standard 9×9 Sudoku was studied; results may not transfer to variants like 16×16, Samurai, or Killer, where the constraint structure is different.

Construct validity. The propagation component only implements the two simplest human-solving rules: Naked Single and Hidden Single. More powerful inference rules — Naked Pair, Hidden Pair, Pointing Pair, X-Wing, Swordfish, and beyond [12], [13] — were left out on purpose to keep the comparison clean. As a result, the CP and Hybrid solvers understate the maximum propagation power achievable. A CP solver with Naked Pair and X-Wing might solve more Hard puzzles without any search. Similarly, our difficulty tiers are defined purely by clues, which is a coarse proxy for true difficulty; a more principled rating would consider the inference rules needed to solve the puzzle.

Statistical validity. We used the non-parametric Wilcoxon signed-rank test to avoid the normality assumption, but we did not apply a multiple-comparison correction across the nine pairwise tests in Table X. A Bonferroni correction at $\alpha = 0.05$ would set the per-test threshold to $0.05/9 \approx 0.0056$, which is satisfied by all tests except BT vs CP on time ($p = 0.037$) and BT vs Hybrid on time ($p = 0.031$). Those two should be read as trends rather than confirmed differences. Importantly, the central claim — that every non-BT solver drastically outperforms BT on recursive calls — remains significant under any reasonable correction.

One more limitation concerns the puzzle generator. As noted in IV-B, the generator does not explicitly verify solution uniqueness after cell removal. Some generated puzzles may therefore admit multiple valid completions. Every algorithm returns the first solution found, and that first solution is deterministic given the algorithm's tie-breaking rules, so the metrics we report are internally consistent. But the 'difficulty' of a puzzle with multiple solutions may not match the difficulty of a unique-verified puzzle with the same clues. A future iteration should add an explicit uniqueness check via a solution-counting variant of the solver.

VII. Conclusion and Future Work

We ran a controlled empirical comparison of four Sudoku solvers — pure backtracking, backtracking with MRV, backtracking with constraint propagation, and a hybrid of MRV with constraint propagation — on 150 randomly generated puzzles across three difficulty tiers. The takeaways come in five points.

Start with pure backtracking: it is acutely vulnerable to pathological puzzles. On the Hard tier, one puzzle (H29) accounts for 35% of all recursive calls, and the top 5 account for 78%. Means are misleading for BT; medians paint a fairer picture. MRV, on the other hand, is a remarkably high-leverage improvement — a one-line change to variable selection cuts mean recursive calls on Hard by 99.50% and gives a backtrack-free solution on 94% of Easy puzzles. Constraint propagation goes further still, cutting mean recursive calls on Hard by 99.97% and eliminating backtracks on 88% of Hard puzzles. The Hybrid solver — MRV stacked on constraint propagation — produces the smallest search trees on the most pathological puzzles (a 62,862× call reduction on H29 relative to BT), but is slightly slower than CP alone in mean wall-clock time on Hard (1.13 ms vs 0.97 ms) due to its higher per-node overhead. And every pairwise difference in recursive calls is statistically significant at $p < 0.001$ by paired Wilcoxon signed-rank, with large effect sizes ($|\text{Cliff's } \delta| > 0.9$) for every comparison involving BT.

The practical takeaway for algorithm designers: constraint propagation and variable-ordering heuristics are complementary, not redundant. Propagation shrinks the residual problem handed to search; MRV makes sure the residual search happens on the most constrained variables. Stacking them gives a solver whose remaining search tree is essentially negligible for any reasonable 9×9 puzzle, and the additional wall-clock time cost is small enough that, when the puzzle distribution is unknown, Hybrid is the safest choice. When the distribution is known to be amenable to propagation — as our randomly generated Hard tier apparently is — CP alone is marginally faster.

Four directions for future work suggest themselves. The most obvious next step is to redo the study on a curated benchmark of genuinely hard puzzles — Norvig's top1465 [5], or the 17-clue clue minimum set [3] — and see whether Hybrid's disadvantage on our Hard tier reverses on harder puzzles. Another avenue is augmenting the propagation component with more advanced rules (Naked Pair, Hidden Pair, X-Wing, Swordfish) to quantify each rule's marginal benefit and possibly shift the CP-vs-Hybrid balance. A value-ordering heuristic like Least Constraining Value (LCV) on top of MRV could push backtrack reductions further. And applying the same comparative framework to Sudoku variants — 6×6, 16×16, Samurai, Killer — would test how far the findings generalise beyond the 9×9 grid.

Acknowledgments

The author wishes to express gratitude to God Almighty for His blessings and grace, which made the completion of this paper possible.

The author extends his deepest appreciation to Prof. Dr. Ir. Rinaldi Munir, M.T and Team, as the lecturer of Algorithm Strategy, for the invaluable guidance, constructive feedback, and profound knowledge shared throughout the semester and during the development of this work.

Video Link at Youtube

https://www.youtube.com/@egaluthfirais1139

Appendix

https://github.com/road2qnt/IF2211-Sudoku-Final-Project

References

[1] A. Bhattarai, D. Uprety, P. Pathak, S. N. Shrestha, S. Narkarmi, and S. Sigdel, "A study of Sudoku solving algorithms: Backtracking and heuristic," arXiv preprint arXiv:2507.09708, 2025.

[2] T. Yato and T. Seta, "Complexity and completeness of finding another solution and its application to puzzles," in Proc. Nat. Meeting Japan Soc. Ind. Appl. Math. (JSIAM), 2002, pp. 945–950.

[3] G. McGuire, B. Tugemann, and G. Civario, "There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem," arXiv preprint arXiv:1201.0749, 2012.

[4] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed. Hoboken, NJ: Pearson, 2020.

[5] P. Norvig, "Solving every Sudoku puzzle," 2006. [Online]. Available: https://norvig.com/sudoku.html

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA: MIT Press, 2022.

[7] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," Artificial Intelligence, vol. 14, no. 3, pp. 263–313, 1980.

[8] J. F. Crook, "A pencil-and-paper algorithm for solving Sudoku puzzles," Notices of the American Mathematical Society, vol. 56, no. 4, pp. 460–468, 2009.

[9] S. C. Yuen and K. C. Tan, "A hybrid algorithm for solving Sudoku puzzles," Applied Mathematics and Computation, vol. 217, no. 17, pp. 7322–7334, 2011.

[10] R. Soto, B. Crawford, W. Palma, S. Misra, and E. Olguin, "A hybrid alldifferent-tabu search algorithm for solving Sudoku puzzles," Computational Intelligence and Neuroscience, vol. 2015, Article ID 26972, 2015.

[11] T. Mantere and J. Koljonen, "Solving, rating and generating Sudoku puzzles with GA," in Proc. IEEE Symp. Comput. Intell. Games (CIG), 2007, pp. 252–259.

[12] K. Chen, "Exploring a better way to constraint propagation using naked pair," in Proc. Int. Conf. Digital Sci. (DS), 2024, Article 04025.

[13] A. Stuart, "Sudokuwiki: Sudoku solving strategies," [Online]. Available: https://www.sudokuwiki.org/sudoku_strategies.htm

[14] J. Park, "Neural network-based Sudoku solver with skill embeddings," arXiv preprint arXiv:1802.09660, 2018.

[15] H. Simonis, "Sudoku as a constraint problem," in Proc. 4th Int. Workshop Modelling and Reformulating Constraint Satisfaction Problems, 2005, pp. 13–27.

[16] R. Dechter, Constraint Processing. San Francisco, CA: Morgan Kaufmann, 2003.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Ega Luthfi Rais 13524115