

10,000 Sudoku Puzzles in 76 Milliseconds: Engineering a Fast Human-Strategy Solver

Sitan Chen

Changjun High School International Department

stanchenx@outlook.com

Abstract

Human-strategy Sudoku solvers use recognizable logical patterns without guessing or backtracking, but repeated candidate scans and deduction bookkeeping can make them slow. We present a C++ solver called HPS that retains 15 named strategies while reducing redundant operations. Based on measured bottlenecks, we design and implement a set of optimizations including compact representation of candidates, incremental index data structures, direct action output without full deduction records, and batched application of compatible actions. Additional optimizations including failure caching, delayed full-board validation, and cost-aware strategy scheduling further reduce work, while independent puzzles are distributed across worker threads. We evaluate the solver on 10,000 distinct puzzles from four difficulty categories in a public puzzle bank. The corpus contains puzzles completed by an earlier 15-strategy solver, so the evaluation measures runtime on supported puzzles rather than general solving coverage. Every returned grid matches its expected answer. On an Apple M3 Pro, mean runtime falls from 16.717 seconds for the serial C++ baseline to 0.475 seconds for serial HPS, indicating a 35.2x speedup. With eleven threads, HPS completes the corpus in 76 milliseconds, a 220.4x speedup over the same serial baseline. Ablation studies show that major performance gains are from faster strategy checks and candidate-position lookups.

1. Introduction

There are two types of solvers. One type uses search, trying out values and backtracking from dead-ends. Another type uses human-like strategies, making moves that are justified by the rules of Sudoku. Examples include finding a cell that must take a particular value or deleting candidates based on patterns like “naked pairs”. For problems where we care only about finding a solution, search is enough. But if we want to know how a solution was found, which types of human strategies were used, or the difficulty of the puzzle, then human strategies are useful. Puzzles solved using pencil and paper can be analyzed as algorithms and not just ad hoc techniques (1).

A naive implementation of the Sudoku problem is inefficient. A straight-forward implementation could restart its sequence of strategies after each move. It could revisit the same regions many times. It could update information that has not changed much. It could re-validate all 81 cells after each insertion/deletion. While each of these operations is cheap, together they may significantly impact solving time on a hard puzzle and over an entire puzzle set. This leads us to the research question: Can we implement human-style Sudoku strategies to achieve low single-puzzle latency and high batch throughput?

The approach used here is to fix the solving method for every puzzle to the same 15 human strategies, avoiding guessing and backtracking, and to optimize for the cost of implementing those strategies. The solver is then designed to track candidates more compactly, reuse unchanged positions, combine multiple valid deductions at once, and run multiple puzzles in parallel.

For simplicity, the optimized human-strategy solver will be referred to as HPS throughout this paper. There are three main contributions made by this study: a measurement of the baseline solving process and a report of redundant computation; an integrated design of the solver combining compact candidate tracking, reused position data, direct strategy detection, batched actions, incremental updating, and puzzle-level parallelism; and a set of correctness and performance tests including action-level auditing, controlled timing measurements, ablation analysis, single-puzzle measurements, thread count comparison, and benchmarking against search and logic solvers.

This paper is organized as follows. Section 2 provides background on the game of Sudoku and the reference solvers discussed; Section 4 introduces the baseline solver and measures its performance and bottleneck areas; Section 5 introduces the optimized solver and describes the individual components; Section 6 describes the correctness tests and performance benchmarks; Section 7 describes the performance results obtained; Section 8 discusses the findings and possible limitations; and Section 9 is the conclusion.

2. Background

2.1. Sudoku as a constraint problem

Sudoku has 81 cells, laid out in nine rows, nine columns, and nine 3×3 boxes. Each cell contains exactly one number from 1 through 9, and each of these nine numbers occurs exactly once in every row, column, and box. The potential numbers for an empty cell are those that are not excluded in its row, column, and box. The goal of solving the puzzle is to find a filling of the empty cells such that all cells have a single number and all numbers in each row, column, and box are unique.

We call a full row, column, or box a **unit**. If two cells share at least one unit, we call them **peers**; if one cell is assigned a number, that number cannot occur in any of its peers. Every cell has 20 distinct peers (eight in the same row, eight in the same column, and the remaining four cells in its 3×3 box that are not already in the row or column). A cell can be referenced using coordinates, e.g., $r1c1$ denotes row 1, column 1.

Sudoku is a constraint satisfaction problem: the cells are the variables, the numbers are their domains, and the row/column/box constraints define valid assignments. It is known that generalized Sudoku where the board size can vary is NP-complete (2). Note that this complexity result does not apply to the time taken to solve a 9×9 Sudoku puzzle, but rather describes how the time to solve it increases as the board size increases. This paper studies only the operations that are repeated while solving a fixed size puzzle.

2.2. State, actions, and solve rounds

The terms below distinguish the puzzle's logical meaning from its implementation:

- The **state** is the collection of all currently-fixed values along with the current candidate set of each unsolved cell.
- A **candidate** is a digit that is still possible in a cell under the current state.
- An **action** either **assigns** of a digit to a cell or **eliminates** a candidate. Assignments fill cells, while eliminations remove only a single possibility and may allow an assignment later on.
- A **deduction** is a justified action plus evidence for the justification. An answer-only solver might keep track of the action, but wouldn't build a text representation of the evidence.
- A **strategy check** looks at the state and returns deductions (or other actions) that are justified by a single named strategy. The part of the implementation that does this check is called a **detector**.

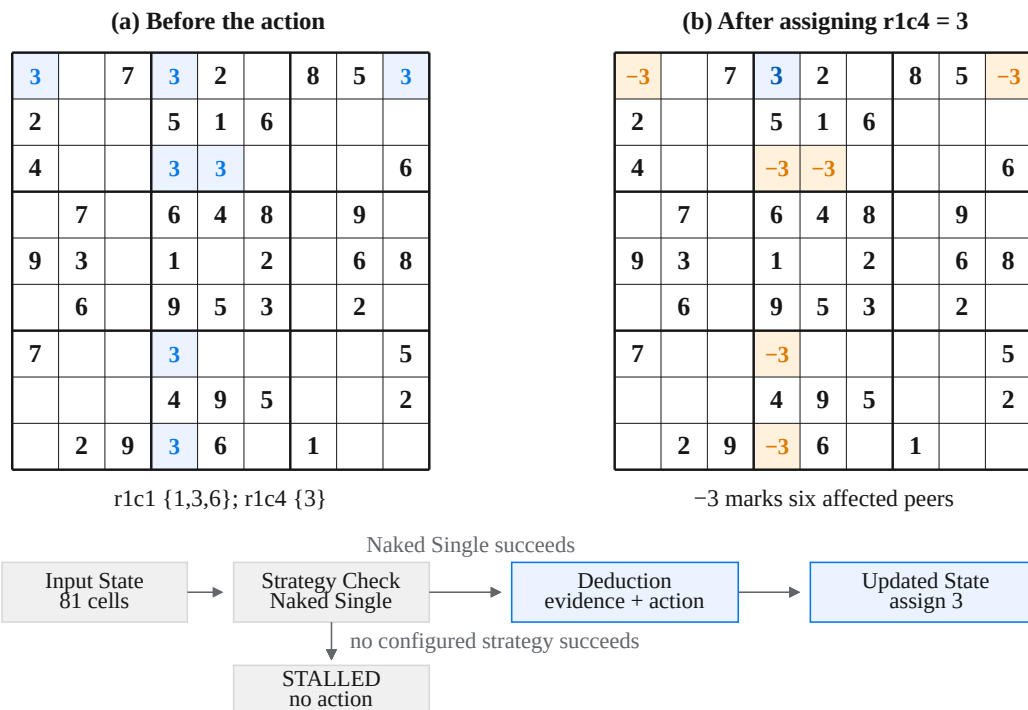


FIGURE 1. One solve round using Naked Single. In panel (a), r1c4 has only candidate 3. Panel (b) shows 3 assigned to that cell and removed as a candidate from six peers. Blue 3s in (a) mark candidates relevant to this step; orange -3s in (b) mark removals. Other candidates are omitted from the drawing, not from the solver state. The lower diagram also shows the alternative stalled outcome when no strategy finds an action.

- A **solve round** picks a strategy, gets one or more actions, and applies them, then determines what to check next.
- **Stalled** means that none of the configured strategies found a next action, not that the puzzle has no solution.

2.3. A complete solve round

Figure 1 shows one full solve round and links the definitions above. In the left grid, the input board and the candidate 3 of interest are shown while the full state also retains all other candidates. E.g. r1c1 has candidates {1, 3, 6}, while r1c4 has only {3}. A blue 3 in an empty cell indicates that 3 is still a candidate.

The Naked Single detector does a Strategy Check on this state. The detection concludes that r1c4 has only candidate 3 (evidence) and assigns 3 to r1c4 (action). Evidence and its corresponding action (a logical justification of that action) make up a deduction. Once the solver executes the action, 3 is no longer a candidate in any of its 6 unfilled peers that still had 3 as a candidate. The right grid shows the state after the action has been executed.

A solve round begins in the current state and iterates through the configured strategies until one yields a deduction. The solver then applies the actions that the deduction justifies in order to get the state for the next round. In this example, the action sets a digit to a cell, but another strategy could instead prune a candidate. If none of the strategies yields a deduction, the solver makes no changes and says it has stalled. This does not imply the puzzle has no solution, just that the configured strategies cannot find any more deductions in the current state.

2.4. Strategy families and their computation

There are 15 named strategies, which are grouped in Table 1 by the information that their detectors scan. Their underlying patterns are shown in Figures 2 and 3 and explained below. A *Naked* pattern is one that is directly observable from the candidates of specific cells; a *Hidden* pattern is one that is observed by identifying where specific digits may reside in a unit.

TABLE 1. The 15 implemented strategies grouped by the information their detectors inspect.

Group	Named strategies	Information inspected
Singles	Naked Single, Hidden Single	Candidates in one cell or unit
Subsets	Naked Pair, Hidden Pair, Naked Triple, Hidden Triple, Naked Quad	Candidate sets across a few cells
Intersections	Pointing Pair/Triple, Claiming	Digit positions shared by a box and a row or column
Fish	X-Wing, Swordfish	Digit positions across rows and columns
Coloring	Simple Coloring	Linked positions of one digit
Wings	Y-Wing, XYZ-Wing	Candidates in a small group of linked cells
Chains	XY-Chain	Paths through two-candidate cells

Figures 2 and 3 show examples of each strategy family. All boards are taken from states recorded immediately before a validated solver action. To keep the boards legible at page size, highlighted cells show only the digits relevant to each pattern. The complete candidate sets for the pattern cells are listed below each board.

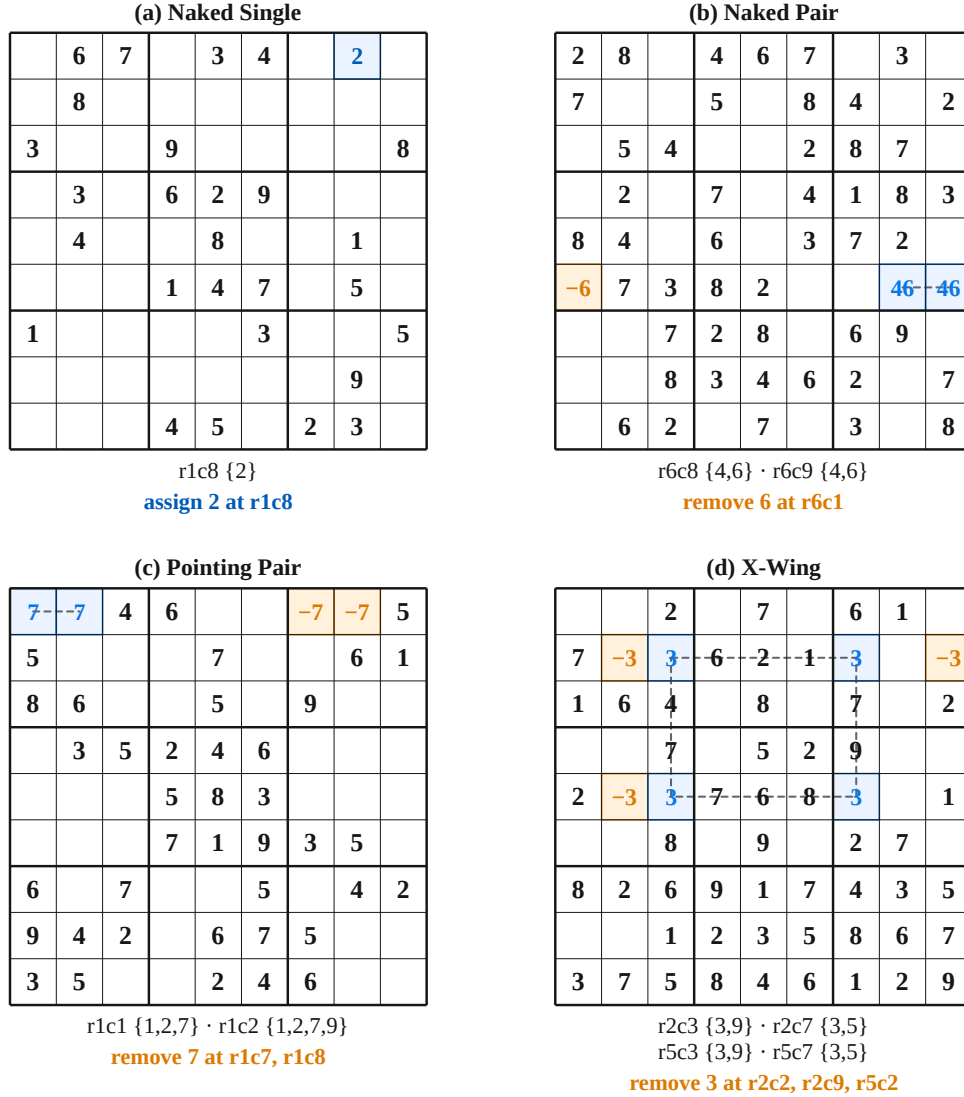


FIGURE 2. Verified examples of (a) Singles, (b) Subsets, (c) Intersections, and (d) Fish. Blue cells identify the pattern; orange -digits mark eliminations and a filled blue digit marks an assignment. Pattern-cell candidate sets and the resulting action appear below each board.

- **Singles (Figure 2a).** At this moment in the record, cell r1c8 contains only the candidate 2, and therefore the solver fills that cell with 2.
- **Subsets (Figure 2b).** Both cells r6c8 and r6c9 contain only candidates {4,6}, which blocks these values elsewhere in row 6, including r6c1, where we can remove candidate 6.
- **Intersections (Figure 2c).** In box 1 there are only two places where 7 could go, r1c1 and r1c2; these two cells constitute the *Pair* in **Pointing Pair**. Since both are in row

1, the 7 in box 1 is restricted to row 1; such restriction is *Pointing*. Prior to the above reasoning, both r1c7 and r1c8 also had 7 as a candidate. But since either r1c1 or r1c2 must be the real 7 in row 1, those 7s in r1c7 and r1c8 are candidates in error, and can be discarded. The two discarded candidates are not the *Pair*. The two cells of the pattern are not required to have the same pair of candidates, as in a *Naked Pair*; here the candidates for the two cells are {1, 2, 7} and {1, 2, 7, 9} respectively.

- **Fish (Figure 2d).** Column 3 has a single occurrence of digit 3 in r2c3 and r5c3. Similarly, column 7 has a single occurrence of digit 3 in r2c7 and r5c7. This set of four cells is the X-Wing rectangle. Since each column contains only one 3 and each row has only one 3, the 3s in each row and column must be placed in diagonal corners. That is, either r2c3 and r5c7, or r5c3 and r2c7. Therefore, rows 2 and 5 will have their 3s located in one of these two columns. The candidate 3 can thus be eliminated from the remaining cells of these two rows, in this case: r2c2, r2c9, and r5c2.

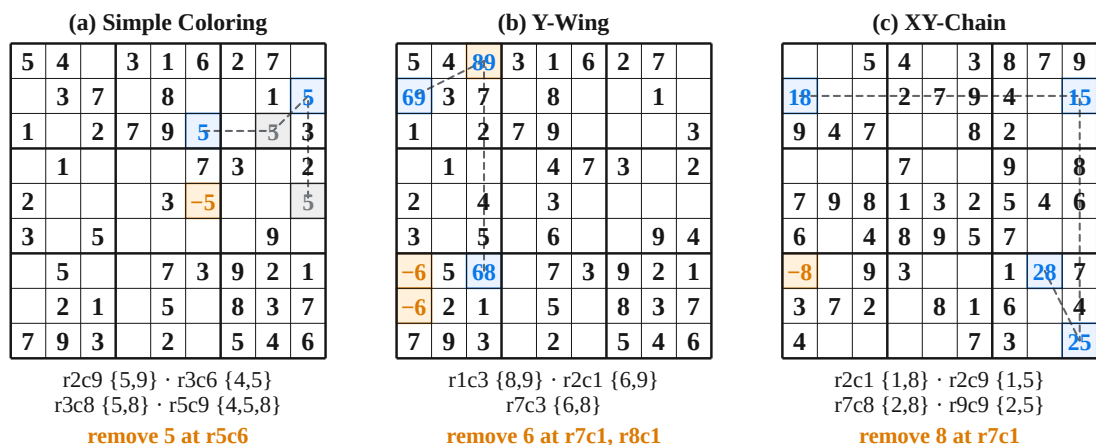


FIGURE 3. Verified examples of (a) Coloring, (b) Wings, and (c) Chains. Contrasting blue and gray cells or connecting lines show the relevant candidate relationships; full pattern-cell candidate sets and the action appear below each board.

- **Coloring (Figure 3a).** Coloring examines a single digit. If two cells are the only candidates for this digit in their row, column, or box, one of them is the correct cell. The figure links such cells together and shades them alternately. There are two possible cases: either the blue cells hold the digit and the gray cells do not, or the opposite. Thus any candidate that is in the same row, column, or box as both a blue cell and a gray cell can be eliminated: it would conflict with a correct candidate in one of the two cases. In Figure 3a, the connections for digit 5 go from r3c6 to r3c8 along row 3, then to r2c9 in the upper-right box, and then to r5c9 along column 9. Cell r5c6 sees blue r3c6 in column 6 and gray r5c9 in row 5. In either case, r5c6 cannot hold 5.

- **Wings (Figure 3b).** A Y-Wing uses three cells with exactly two candidates each and three distinct digits a , b , and c : a pivot $\{a, b\}$ and two wings $\{a, c\}$ and $\{b, c\}$. The pivot must see each wing. If the pivot is a , the $\{a, c\}$ wing must be c ; if it is b , the $\{b, c\}$ wing must be c . Therefore, no cell which sees both wing cells may contain the digit c . Here, the pivot is cell $r1c3 \{8, 9\}$. The two wings are $r7c3 \{6, 8\}$ and $r2c1 \{6, 9\}$. They are seen by the pivot through column 3 and box 1, respectively. Whether the pivot is 8 or 9, one wing must contain 6. Cell $r7c1$ sees both wings through row 7 and column 1; cell $r8c1$ sees both wings through column 1 and the lower-left box. The digit 6 can therefore be removed from these two cells.
- **Chains (Figure 3c).** An XY-Chain joins cells which have only two candidates each. Each pair of neighbors in the chain shares both a unit and one candidate digit: if one neighbor takes that digit, the other must take its alternative. The four cells here are $r2c1 \{1, 8\}$, $r2c9 \{1, 5\}$, $r9c9 \{2, 5\}$, and $r7c8 \{2, 8\}$. The three links involve digits 1, 5, and 2 in order in units row 2, column 9, and box 9 (lower right), respectively. The two end cells of the chain both have the candidate 8. If cell $r2c1$ is assigned 8, the 8 will be in the first end cell. If $r2c1$ is instead assigned 1, then $r2c9$ cannot be 1, so it must be 5; then $r9c9$ must be 2; and then $r7c8$ must be 8. Thus there is always an 8 in one or the other end cell. Cell $r7c1$ shares column 1 with the first end and row 7 with the last end, so it cannot contain 8.

All seven families search for solutions in the candidate state, but they do so in different ways. Singles look at the candidates in one cell or at where one digit can be placed in a unit. Subsets compare multiple candidate sets. Intersections and Fish consider how the possible placement of a digit lines up among rows, columns, and boxes. Coloring, Wings, and Chains trace logical connections between candidates in different cells. The solver must have a single representation of the puzzle and detection techniques that work well with each form of reasoning. The next section discusses the implications of this difference when analyzing and optimizing the repeated strategy scans of the baseline.

2.5. Search solvers as performance references

A human-strategy solver finds a solution via deductions similar to those shown in Fig. 2 and Fig. 3. A search solver works differently by making a tentative guess at a digit, continuing from that guess, and, if a contradiction is encountered, returning to make a different guess. Basic backtracking (Basic BT) finds the first empty cell and iteratively tests each legal digit for that cell. Minimum Remaining Values (MRV) uses the same backtracking process, but first determines which empty cell has the fewest remaining legal digits; this can result in detecting a wrong guess sooner, reducing the size of the search tree that needs to be examined (3). Another popular approach formulates Sudoku as an exact cover

problem and applies search techniques such as Dancing Links (4).

This solver does not use basic BT or MRV, not even when its 15 strategies cannot make any more progress. The other two search-based solvers are included in the final experiments solely as performance benchmarks. These three solvers all receive the exact same prepared puzzles via the same batch interface and give output in the exact same format. Thus we compare runtimes between specific implementations for the same inputs on the same dataset; we do not claim that all human-strategy based Sudoku solvers must be faster or slower than all search-based solvers.

2.6. Performance at two scales: one puzzle and a batch

We consider two distinct metrics. **Single-puzzle latency** is defined as the duration between when we receive a prepared grid and when we return its current status and final grid. **Batch throughput** refers to how many independent puzzles can be solved per second. Minimizing redundant computation while solving a puzzle will benefit both metrics. It is possible that introducing additional workers to a batch would improve throughput without reducing the length of the longest logical sequence for a given puzzle. Further, parallel speedup can be reduced by synchronization and sequential computation (5). The design of our workers is detailed in Section 5.9.

3. Related Work

3.1. Logical solving and efficient propagation

Some research related to human-style Sudoku solving considers the kinds of reasoning that humans employ to solve puzzles, and which puzzles those reasoning steps can solve. Crook’s pencil-and-paper algorithm relies on candidates and candidate subsets of cells to deduce new values and eliminations (1). Its complete procedure also permits trial choices and backtracking when logical deductions are insufficient. The work in (6) aims to model human solving in order to determine the difficulty of a puzzle, and evaluates both the difficulty of each step and the dependencies between steps. Nishikawa and Toda define *strategy-solvability* that is whether a puzzle can be completed using only a given set of strategies, and use this concept to generate clues (7). While these works relate to the use of a fixed set of strategies, their objectives are not to reduce the running time of the strategy implementation.

Research on constraint programming also studies efficient checking. Simonis represents Sudoku as row, column and box constraints, and describes a dual representation where the possible digits for a cell are related to the possible locations of a digit (8). HPS similarly stores both kinds of candidates, but with bitmasks, not as a general constraint solver. Schulte and Stuckey examine how a propagation engine can avoid doing checks

that cannot contribute new information, and which checks should be done first among the remaining ones (9). HPS similarly avoids some checks: a check that has been tried before and failed will not be tried again unless its inputs have changed; and checks are applied in a static order based on their costs and hit rates. Both of these differ from the event-driven approach and dynamic priority of a general propagation engine.

3.2. Related open-source solvers

HoDoKu2, a maintained fork of HoDoKu (10) and Sukaku Explainer (11) allow a wide catalogue of human techniques but focus on solution path, hints, analysis, and difficulty rating. The HappyCerberus (12) logic based solver is more similar to the problem considered here. It does not use guessing and includes a number of human techniques including Subsets, Intersections, Fish, and Chains, as well as a large batch benchmark. Finally, Tdoku (13) is a search oriented reference. It is an optimized search solver which has a library that can be used as well as a benchmark with other external solvers and data sets.

TABLE 2. Roles of the open-source systems considered when positioning and evaluating the solver.

System	Primary emphasis	Relevance to this study
HoDoKu2 / Sukaku Explainer	Hints, explanations, and ratings	Strategy-coverage examples
HappyCerberus Sudoku	Logic-only batch solving	External logic-solver comparison
Tdoku	Optimized search and benchmarks	Search-performance context

The different solvers cover different strategies, outputs, and stopping behaviors, so we cannot directly compare runtimes between the solvers without normalizing the benchmark setup. HappyCerberus uses some of these strategies, though its strategy list is not the same as ours. Therefore, we additionally compare on the fixed set of 7,319 puzzles where both solvers can finish, using the same batch completion criterion and timeout. We do not include Tdoku’s SIMD-enabled version in our benchmark since it uses x86 instructions that are not available on our ARM64 evaluation system. Our fixed internal baseline is the main point of comparison for our improvements, and basic BT and MRV provide search-based runtime references.

This study focuses on the implementation cost of a fixed set of 15 human strategies, rather than introducing new deduction rules or a difficulty model. It combines compact state representations, direct action output, fewer repeated checks, and puzzle-level parallelism. Profiling and ablation experiments evaluate these changes against a frozen C++ baseline, while external solvers provide separate performance references.

4. Baseline and Bottleneck Analysis

We first consider a naive implementation of the 15 rules in Table 1 before optimizing the solver. The naive version is a full C++ solver which takes the candidates from the puzzle, finds and performs the valid deductions, evaluates the state, and stores the chosen steps so that the solution path can be reproduced. It halts without resorting to guessing or backtracking when no more strategies can advance the state. By inspecting this implementation, we can gain insight into what parts of the solving process contribute to the execution time.

The design deliberately follows the solution process of a human player using the given strategies as closely as possible. Candidate sets are implemented with general-purpose ordered sets; each strategy scans through the current state separately; and a full record of available deductions is built up before the solver chooses one to follow. These decisions were adopted to make the code easy to read and to ensure that the process of solving a particular puzzle always follows the same sequence. They also create obvious places where we can improve the performance of the solver without altering the correctness of the strategies.

4.1. One baseline solve round

Figure 1 illustrated that r1c4 had candidate 3 only, and what happened when 3 was placed there. We now trace that round in the baseline. The input is a string of 81 digits, 0 representing an empty cell. The baseline builds the fixed lists of rows, columns, boxes, and peers, then records the value of each cell and the ordered set of its remaining candidates. Finally, it removes from each cell's candidate set any that are ruled out by the filled values of their peers. Before trying any strategy, the baseline checks that no digit is repeated within a row, column, or box and that every empty cell has at least one candidate. If either check fails, it rejects the input puzzle.

The baseline checks its 15 strategies in the fixed order given by Table 3. A detector reads the current values and candidates, but does not modify them. For every pattern that yields an action, it creates a *deduction*: an item recording the strategy and location, the facts about its candidates that led to it, the action(s) it produces, and a textual record of those changes. In this state, the first detector, Naked Single, produces four deductions: place 3 in r1c4, 7 in r5c5, 1 in r7c6, and 7 in r9c6. Figure 1 highlights only the first of these deductions; the candidates supporting the other three are not shown.

TABLE 3. Fixed strategy order used by the baseline. Later strategies are not checked in a round once an earlier one returns a deduction.

Priority	Strategies, in execution order
1-2	Naked Single, Hidden Single
3-4	Naked Pair, Hidden Pair
5-6	Pointing Pair/Triple, Claiming
7-9	Naked Triple, Hidden Triple, Naked Quad
10-11	X-Wing, Swordfish
12	Simple Coloring
13-14	Y-Wing, XYZ-Wing
15	XY-Chain

Once a strategy produces one or more deductions, the solver stops checking later strategies in that round. It then uses a fixed comparison key to select one of those deductions, making the choice reproducible. In the case of Naked Single, it picks the first cell in row-major order, i.e. $r1c4=3$. The remaining three full records are not utilized in this round. The chosen deduction includes the candidate fact $r1c4 \in \{3\}$ and the action of filling that cell with a 3. A single deduction can also involve multiple actions, e.g., the Pointing Pair deduction in Figure 2c eliminates two 7s in a single step.

Applying the selected deduction would place 3 in $r1c4$ and remove candidate 3 from each of the six peers that are still listing 3 as a candidate (Figure 1). At this point the current state is checked: no unit may have a repeated filled digit, all empty cells should have at least one candidate remaining, and no candidates in a cell should conflict with any digits filled in its peers. If all of the tests are passed, the solver adds the selected deduction to the solve history. It does not keep the other three deductions for later use. The next round starts again with Naked Single and scans the updated state, even though most cells have not changed.

Figure 4 illustrates the round we discussed. An elimination proceeds in a similar manner but modifies the set of candidates rather than filling a digit. The Naked Pair elimination in Figure 2b removes a 6 from $r6c1$, for instance. The baseline needs to persist the modification of candidates from round to round since they are not directly visible. The solver concludes successfully with SOLVED, unsuccessfully with EXHAUSTED when none of the strategies succeed in finding a deduction, or with RULE_ERROR if it reaches the safety cutoff at 10,000 rounds. It is guaranteed to neither guess nor backtrack.

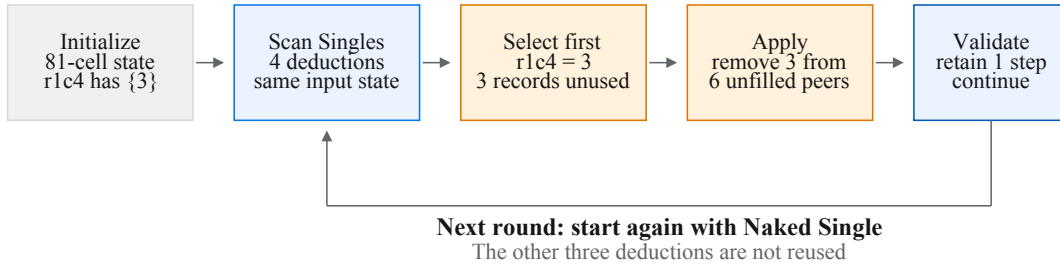


FIGURE 4. One baseline round from the Figure 1 state. Naked Single constructs four deductions; the minimum-key deduction assigns 3 to $r1c4$. The solver updates six peers, validates the state, records the selected step, and starts the next strategy scan.

4.2. Profiling the baseline and quantifying repeated work

To identify repeated work in the baseline, we instrumented a diagnostic build and ran it on our frozen set of 10,000 puzzles. We counted the deductions that the strategy detectors build and those they choose to apply, and measured the major phases of each solve. On every puzzle, the diagnostic build makes the same solving decisions as the unmodified baseline. Because frequent timing measurements have overhead, we use these phase measurements to find where the bottleneck is; Section 7 reports end-to-end runtimes from uninstrumented builds.

The detectors made 2,470,363 deductions across the 10,000 puzzles, but the solver used only 556,541. Out of these deductions, 1,913,822, which accounts for 77.5% of the total, were not used in the round where they were built. However, an unselected deduction is not necessarily incorrect. In Figure 4, Naked Single finds four possible placements, but the solver applies only one in that round. The baseline still builds a complete record for each of the four. After applying the selected placement, it scans the updated board again rather than reusing the other three records.

4.3. Where the baseline spends time

The measurements in Section 4.2 show how much work the baseline repeats. We now ask where that work takes time. Figure 5 separates the measured runtime into the main phases of a solve round.

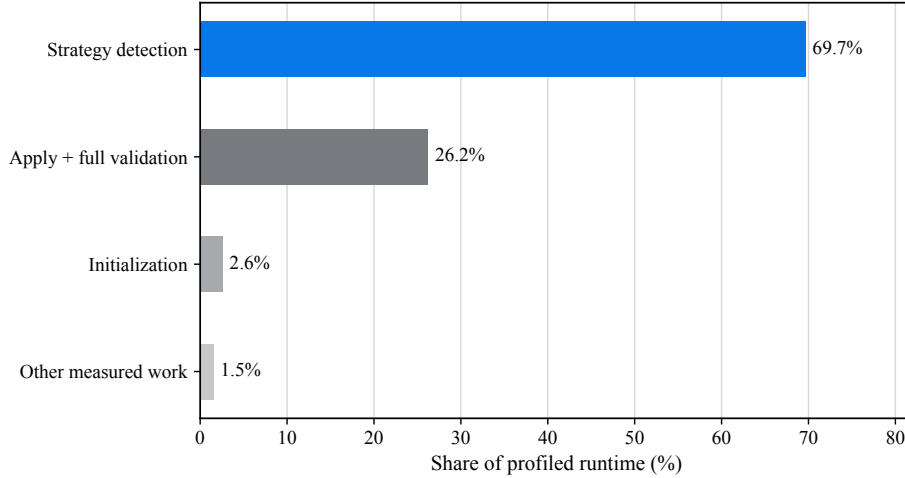


FIGURE 5. Share of instrumented baseline runtime by phase. Detection includes the complete construction of deductions and their evidence. “Apply + full validation” includes state mutation followed by a scan of the complete state. The profile is diagnostic; final runtime comparisons use the uninstrumented build.

The Strategy Detection phase accounts for 69.7% of the instrumented runtime, while applying actions and validating the board after each action together take 26.2%. Therefore, they make up 95.9% of the runtime that we measure, with the remaining 4.1% being initialization and other work. However, knowing these percentages does not allow us to estimate the actual savings provided by any given optimization.

The detection step consists of two parts: identifying a valid pattern and building the corresponding deduction record. As we saw in section 4.2, most records which are finished are not picked in the round they are created. The timer does not distinguish between pattern search and record construction, so the 77.5% of the total number of records which are not chosen *is not* a proportion of runtime which could be eliminated in any obvious way.

The cost of checking a strategy also depends on which detector is used. A *call* in Table 4 represents one check of a strategy, while the *hit rate* represents the fraction of these calls in which at least one deduction is found. We show just a few contrasting detectors instead of all fifteen strategies in the table.

The most common calls are Naked Single and Hidden Single. Though Naked Single runs more often, its lower cost per call means that Hidden Single has a much bigger share of total detection time. Hidden Triple and Naked Quad take a deduction in less than 1% of their calls on this workload, so the vast majority of their time is spent in unsuccessful checks. XY-Chain follows the opposite pattern: it runs rarely, but each call is expensive. Since XY-Chain only runs after all of the other 14 checks fail, its 100% hit rate is conditional on reaching the last position in this strategy order. It cannot be used to predict its hit rate

in any other order.

TABLE 4. Contrasting strategy checks in the diagnostic profile. Hit rates are conditional on the fixed strategy order and the evaluated puzzles.

Strategy	Calls	Hit rate	Mean time per call (μ s)	Share of detection time
Naked Single	556,541	77.2%	1.05	4.8%
Hidden Single	127,056	75.4%	19.27	20.0%
Hidden Triple	11,133	0.8%	60.00	5.5%
Naked Quad	11,040	0.3%	55.70	5.0%
XY-Chain	4,962	100.0%	608.77	24.7%

Another large phase in Figure 5 groups updates to the candidate state and a full validation step on each action. The timer did not differentiate these steps, so we have no indication as to which consumes more of the 26.2%. In summary, the profiling suggests two main categories of work: finding the next deduction and processing the resulting action. It also shows that different strategies have different costs and different frequencies of producing a new deduction. We next make these observations into hypotheses to test, not conclusions about optimization made from the profile.

4.4. Design implications

The profile identifies costly work but does not establish which change will remove it. We therefore test three related design goals:

- **Make each strategy check cheaper.** Detection makes up 69.7% of the measured time. In the baseline, the detector looks through the current candidates to check whether a pattern is satisfied, and produces a full record for each such deduction, even though only one is chosen during a given round. We have investigated two different mechanisms for reducing this workload: using more efficient data structures to make pattern-matching faster, and allowing the detector to produce a justification for its choices without producing a full record that is unnecessary for answer-only solving. The checks themselves also differ: Singles simply count candidates in a cell or unit, while Fish needs to know all locations in which a digit can still occur.
- **Avoid unnecessary checks.** Once a deduction is made, the baseline begins checking for applicable strategies again from the beginning. We propose three optimizations for mitigating this repeated checking. First, multiple applicable strategies discovered at the same state could potentially be applied simultaneously before performing another

scan. Second, if a previous check identified no applicable strategies, and none of candidate information relevant to the check has changed, there is no need to use the same strategy again. Third, as different strategies have different costs and probabilities of yielding an applicable strategy (Table 4), we can also explore the use of alternative fixed scanning orders.

- **Reduce work after an action.** The baseline already updates only candidate sets affected by an action; it does not rebuild all candidates. It then validates the entire board. Together, application and validation take 26.2% of profiled time, but the timer does not separate them. The optimized state adds cell masks and unit-position indexes. We maintain each index as candidates change, rather than reconstructing it from all cells after every action, and test postponing full-board validation until a result is returned. Incremental index maintenance adds writes, while delayed validation can postpone error detection. Section 5.6 explains both choices and their safeguards; their net runtime effect must be measured.

5. Optimized Solver Design

Section 4 identified two major costs in the baseline: repeated strategy checks and repeated whole-board validation. This section follows one solve round from the state in Figure 1 to show where the optimized solver avoids that work.

5.1. The optimized solve loop

Figure 6 begins with the recorded state S from Figure 1. The four cells at the top each have one candidate, so both solvers find the same four Naked Single placements. The branches differ in how these findings are recorded and used, not in the logic that establishes them.

The baseline detector builds one full deduction record per placement. Each record contains the strategy, supporting candidate facts, the assignment, and explanatory text. The optimized detector performs the same strategy check but returns only its result, such as $(\text{assign}, r1c4, 3)$, meaning “place 3 in $r1c4$.” This short *action* identifies what to change; it does not repeat the evidence or prepare an explanation for each of the four placements.

The baseline applies only $r1c4=3$, discards the other three records, and scans again. The optimized solver applies all four assignments before another scan because they are compatible. The four cell cards in each branch show the difference: braces mark candidates in an empty cell, while an unbraced blue digit marks a filled cell. Figure 11 shows the same comparison on the full board.

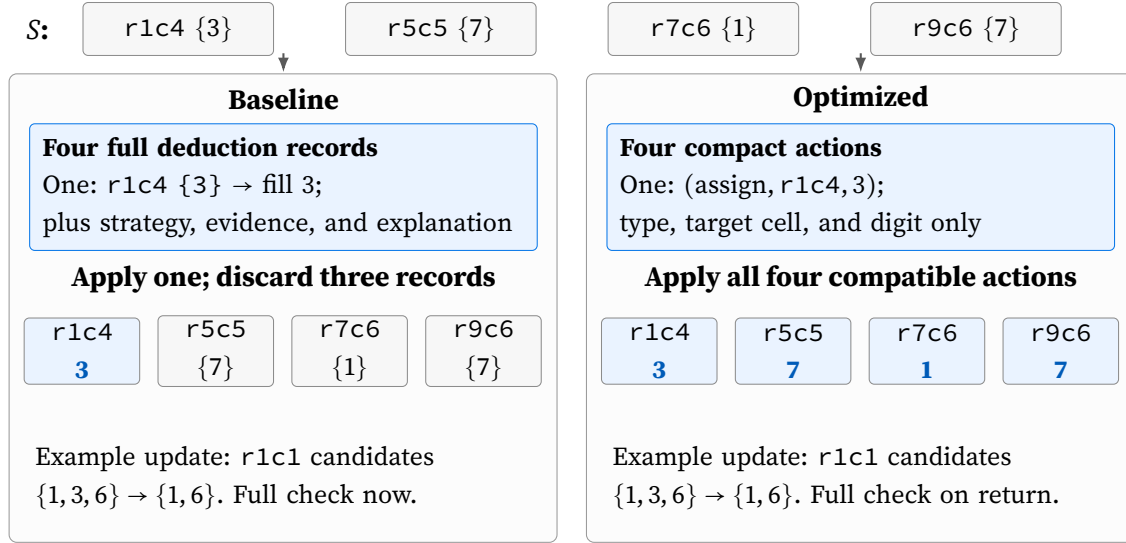


FIGURE 6. The same four Naked Singles processed in one round by each solver. Braces show candidates in an empty cell; blue, unbraced digits are filled values. Each branch shows one example detector output, the four cells after its round, and one state update caused by $r1c4=3$. Other affected cells are omitted.

Before taking an action, the optimized solver verifies it applies in the current state: for $r1c4=3$, $r1c4$ must still be empty, and 3 must still be a candidate there. Placing 3 marks $r1c4$ as filled, and eliminates candidate 3 from the remaining candidates of all its unfilled neighbors (including $r1c1$ and $r1c9$). In both branches of Figure 6, the candidates of $r1c1$ change from $\{1, 3, 6\}$ to $\{1, 6\}$ (see bottom of Figure 6). The *full-board check* detects if any row, column, or box contains a repeated filled digit, or an empty cell with no remaining candidates, or a candidate that disagrees with a filled neighbor. After its single placement in each round, the baseline run performs this check. The optimized solver can take multiple actions per round, and delays the full-board check until it is about to return an answer or stop status. Both solvers start the next round from their updated states. If no strategy can find an action, the solver returns *Stalled*.

5.2. Values and cell candidate masks

While Figure 6 shows the filled digits and a few candidate sets, it does not show how the optimized solver actually stores them. In fact, for each cell c it stores both a single-value component $V(c)$ (which is zero if the cell is empty), and a 9-bit component $M(c)$ (a candidate mask, which has one bit set for each remaining possible digit). Figure 7 shows both components of the state on the board of Figure 1: Cell $r1c3$ is filled, so $V(r1c3) = 7$. Cell $r1c1$ is not filled, so $V(r1c1) = 0$, but it contains the candidates $\{1, 3, 6\}$, stored separately in $M(r1c1)$.

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

Values $V(c)$

$V(r1c3) = 7$ (filled); $V(r1c1) = 0$ (empty)

Cell mask $M(r1c1)$

Candidates $\{1, 3, 6\}$; bit $d - 1$ represents digit d

1	2	3	4	5	6	7	8	9
1	0	1	0	0	1	0	0	0

Digits 1 to 9, left to right; $M = 0b000100101$.

Position index $P(\text{row } 1, 3)$

Candidate 3 occurs at columns 1, 4, and 9

c1	c2	c3	c4	c5	c6	c7	c8	c9
1	0	0	1	0	0	0	0	1

Columns 1 to 9, left to right; $P = 0b100001001$.

Filled digits are black; blue 3s are candidates in row 1.

FIGURE 7. Three readings of the same pre-action state. The board shows filled digits and blue candidate positions for 3. The upper bit row encodes $r1c1$'s candidates $\{1, 3, 6\}$; the lower encodes the row-1 positions $\{c1, c4, c9\}$ for digit 3. Bit boxes run from 1 to 9 left to right; binary numerals below them run from high to low.

The upper bit row of Figure 7 shows $M(r1c1)$: only bits for 1, 3, and 6 are set. The baseline stores those candidates in an ordered set; the optimized solver uses a nine-bit mask, where bit $d - 1$ represents digit d . Writing $C(c)$ for the candidates of cell c , this rule is

$$M(c) = \sum_{d \in C(c)} 2^{d-1}.$$

So we get $M(r1c1) = 2^0 + 2^2 + 2^5 = 0b000100101$. Counting the set bits of $M(r1c4)$ in Figure 6 gives one, for digit 3: a Naked Single. Figure 2b shows two more operations. The two row-6 cells $r6c8$ and $r6c9$ both have $\{4, 6\}$. The OR of their masks still has two set bits, thus forming a Naked Pair. The AND of this pair mask with the mask for $r6c1$ (candidates $\{1, 5, 6, 9\}$) gives one set bit, for 6: this is the candidate to remove from $r6c1$. All these fixed size operations avoid constructing temporary sets.

The mask only alters the cost of reading candidates, not which digits are allowed. The lower bit row of Figure 7 introduces the position index P : instead of keeping track of which digits remain possible in one cell, it keeps track of where one digit remains possible in a unit.

5.3. Fixed units and position indexes

The cell mask asks “Which digits can go in this cell?” The useful question for Pointing is the opposite: “Where can this digit go in a row, column, or box?” In Figure 2c, 7 can only be placed in $r1c1$ or $r1c2$ in the upper-left box. Both cells are in row 1, so we remove 7 as a candidate from $r1c7$ and $r1c8$. Figure 7 shows the same kind of lookup for a different

digit: the lower bit row records that 3 can be placed in columns 1, 4, and 9 in row 1. Hidden Single and Fish also query the positions of a digit in a unit.

Each row, column, and box is a nine-cell *unit*, giving a total of 27 units. A cell belongs to one row, one column, and one box. The other eight cells in each of these three units form its 20 distinct *peers*. These relationships do not change; only the cell values and candidates do. If we were to find out which units could accept a given digit, we would have to look at each of the nine cells in turn. Because this lookup will happen many times, we instead maintain a nine-bit mask $P(u, d)$ for every unit u and every digit d . We label the cells of each unit with an index $j \in \{0, \dots, 8\}$, going left-to-right for a row. We denote the cell in unit u at position j by u_j . The rule is:

$$\text{bit}_j(P(u, d)) = 1 \iff d \in C(u_j), \quad j = 0, \dots, 8.$$

In Figure 7, 3 could be in row 1, columns 1, 4, and 9, which are positions 0, 3, and 8. Therefore, $P(\text{row } 1, 3) = 2^0 + 2^3 + 2^8 = 0b100001001$. The three set bits indicate to a Hidden Single detector that 3 is not restricted to a single cell in the row. In the Pointing example earlier, $P(\text{box } 1, 7)$ instead identifies two cells in the same row, and we make the eliminations. The two masks answer different questions about one state: $M(r1c1)$ gives us the digits that can go into the cell; $P(\text{row } 1, 3)$ gives us the cells where 3 can go in the row.

After r1c4=3 (Figure 1(b))									Selected state fields		
									Field	Before	After
-3		7	3	2		8	5	-3	$V(r1c4)$	0	3
2			5	1	6				$M(r1c4)$	{3}	$\emptyset$
4			-3	-3				6	$M(r1c1)$	{1, 3, 6}	{1, 6}
	7		6	4	8		9		$M(r1c9)$: bit for 3	1	0
9	3		1		2		6	8	$P(\text{row } 1, 3)$	{c1, c4, c9}	$\emptyset$
	6		9	5	3		2		$M(r3c4)$: bit for 3	1	0
7			-3					5	$P(\text{row } 3, 3)$: r3c4 bit	1	0
			4	9	5			2			
	2	9	-3	6		1					

Blue: placed 3. Orange: candidate 3 removed from six peers.

FIGURE 8. The $r1c4=3$ assignment from Figure 1 on the full board. The left panel repeats the resulting board: blue 3 is the assigned value, and orange -3 marks the six peers that lost candidate 3. The table gives selected before-and-after entries in V , M , and P ; it does not list every changed position index.

Figure 8 shows the placement $r1c4=3$ from Figure 1 on the entire board. The value at $r1c4$ changes from 0 to 3, and its candidate mask becomes empty because the cell is filled. Candidatend 3 is also removed from six unfilled peers, shown in orange in the

figure. Specifically, two peers are on row 1: $M(r1c1)$ goes from $\{1, 3, 6\}$ to $\{1, 6\}$, and bit 3 in $M(r1c9)$ is cleared. Since our newly placed $r1c4$ now has no candidates either, $P(\text{row } 1, 3)$ loses all three of its set bits.

This assignment also removes 3 from four peers outside row 1, as can be seen in both Figure 1(b) and the left half of Figure 8. One such peer is $r3c4$. In the table, the bit for 3 in $M(r3c4)$ becomes 0, while the bit for $r3c4$ in $P(\text{row } 3, 3)$ becomes 0. The solver must also remove $r3c4$ from $P(\text{column } 4, 3)$ and $P(\text{box } 2, 3)$, though those two elements are not displayed in the table. Those two entries are determined by the fixed peer list and unit lists. Tests rebuild P from all 81 cell masks to ensure that no update is missed.

Note that M and P provide two perspectives on the same candidates, not two steps of solving. In Figure 8, the update $M(r3c4)$ removes 3 from that cell's candidate list, and the corresponding update $P(\text{row } 3, 3)$ removes $r3c4$ from the list of cells where 3 could go in row 3. To keep P accurate, we need to turn off one bit in each of $r3c4$'s row, column, and box indexes. On the other hand, if we later perform a Hidden Single check for 3 in row 3, we can look at how many bits are still set in $P(\text{row } 3, 3)$ rather than examining all nine cells. Should the check require us to know where those candidates are located, we can find them immediately in P without first copying them to a temporary list.

Section 5.6 returns to these updates after showing how strategies produce and group actions. It distinguishes reading V , M , and P to find an action from changing them when that action is applied.

5.4. Direct action detection

In the Detect step of Figure 6, a strategy examines the current filled values and remaining candidates in order to determine any justifiable assignments or eliminations. It returns these actions without applying them, and the solver applies them in the next step. Our optimized solver checks the same 15 logical patterns as our baseline solver. To determine if a pattern is applicable, the detector checks the current values for the cells in cell values V , candidate masks M , position indexes P , or combinations thereof. An action records only its type, target cell, and digit, such as (assign, $r1c4, 3$) or (eliminate, $r6c1, 6$); it does not contain a textual explanation. Table 5 shows the mapping between the strategy families and the state they read, along with examples which we will develop further in the following sections.

TABLE 5. State reads used by direct detectors. Each row points to an example of the same read-test-action interface.

Strategy family	State read and pattern test	Example
Singles	Count set bits in M or P	Figure 6
Subsets	Combine a few M or P masks	Figure 9
Intersections and Fish	Compare digit positions in P	Figure 10(a)
Coloring	Follow links derived from P	Figure 10(b)
Wings	Compare cell masks and peers	Figure 10(c)
Chains	Follow two-candidate links and peers	Figure 10(d)

In the Naked Single from Figure 6, the detector sees $V(r1c4) = 0$ and has just 3 in $M(r1c4)$. So it returns (assign, $r1c4, 3$) and discovers three additional assignments while not generating full deduction records.

Figure 9 corresponds to the Subsets line of Table 5 on a second recorded puzzle. Both $r6c8$ and $r6c9$ have candidates $\{4, 6\}$. Thus, if we add their cell masks together, we have exactly two digits for two cells. Hence 6 can be eliminated from $r6c1$. The detector returns (eliminate, $r6c1, 6$); the later update to $M(r6c1)$ belongs to another phase.

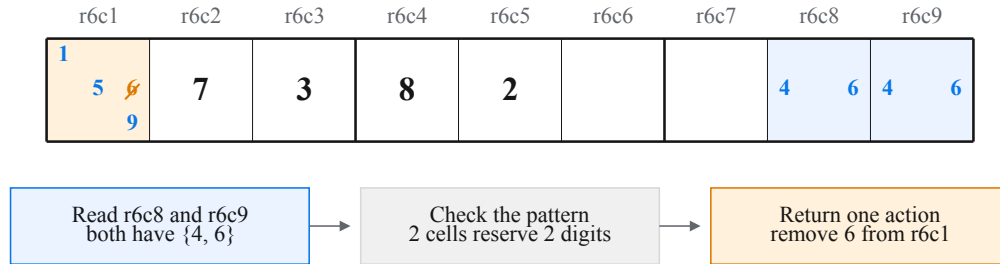


FIGURE 9. The Subsets example in Table 5. The blue cells form a Naked Pair; the crossed-out orange 6 is the action target. The boxes below the row show the state read, the pattern test, and the returned action.

The remaining rows of Table 5 are illustrated in Figure 10. Each panel repeats its recorded pre-action Sudoku board, then follows the same sequence: read the relevant M or P masks, test a pattern in those candidates, and return an elimination. The detector does not alter the board at this stage.

(a) X-Wing (Figure 2d)

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

Read state: $P(\text{col } 3, 3) = P(\text{col } 7, 3)$; both have set bits at rows 2 and 5.

Test pattern: Equal two-row P masks form an X-Wing; all other 3s in those rows are excluded.

Return action: Remove 3 at $r2c2$, $r2c9$, $r5c2$.

(b) Simple Coloring (Figure 3a)

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

Read state: Set bits of $P(\text{row } 3, 5)$: $c6, c8$; $P(\text{box } 3, 5)$: $r3c8, r2c9$; $P(\text{col } 9, 5)$: $r2, r5$.

Test pattern: Each P has two set bits: alternate colors along the links. $r5c6$ sees both colors.

Return action: Remove 5 at $r5c6$.

(c) Y-Wing (Figure 3b)

5	4	89	3	1	6	2	7	
69	3	7		8			1	
1		2	7	9				3
	1			4	7	3		2
2		4		3				
3		5		6			9	4
-6	5	68		7	3	9	2	1
-6	2	1		5		8	3	7
7	9	3		2		5	4	6

Read state: $M(r1c3) = \{8, 9\}$; $M(r2c1) = \{6, 9\}$; $M(r7c3) = \{6, 8\}$.

Test pattern: Intersect pivot/wing masks: $\{9\}$ and $\{8\}$. The wings share 6, so one must hold it.

Return action: Remove 6 at $r7c1$ and $r8c1$.

(d) XY-Chain (Figure 3c)

		5	4		3	8	7	9
18	-2	-7	-9	-4	-15			
9	4	7			8	2		
			7			9		8
7	9	8	1	3	2	5	4	6
6		4	8	9	5	7		
-8		9	3			1	28	7
3	7	2		8	1	6		4
4					7	3		25

Read state: Read M for the four blue pair cells: $\{1, 8\}$, $\{1, 5\}$, $\{2, 5\}$, $\{2, 8\}$.

Test pattern: Intersections of adjacent M masks are $\{1\}$, $\{5\}$, $\{2\}$; one endpoint must hold 8.

Return action: Remove 8 at $r7c1$.

FIGURE 10. Four direct detectors on the recorded boards from Figures 2 and 3. Blue marks pattern candidates; orange $-d$ marks elimination targets, while the orange pivot in (c) is part of the pattern. Below each board, Read state names the M or P data inspected, Test pattern gives the logical check, and Return action gives the result without applying it.

In Figure 10(a), the detector first reads the position masks of digit 3 in columns 3 and 7. They both have rows 2 and 5. The detector checks if those two columns can restrict 3 to the same two rows; the four blue cells form an X-Wing, so the other 3s in rows 2 and 5 cannot remain. It returns eliminations in the three orange cells, $r2c2$, $r2c9$, and $r5c2$. Swordfish performs a similar comparison among three units. The stored P masks provide the positions without scanning the nine cells of each column.

In Figure 10(b), the detector begins by reading three position masks for 5: Row 3 contains only $r3c6$ and $r3c8$; the upper-right box contains only $r3c8$ and $r2c9$; and column 9 contains only $r2c9$ and $r5c9$. Each of the two-position masks defines a linkage

where exactly one of the two cells is 5 and the other is not. Coloring alternately along these links, we see that $r3c6$ and $r2c9$ will have the same color, while $r3c8$ and $r5c9$ will have another color. The orange cell $r5c6$ sees $r3c6$ in its column and $r5c9$ in its row. This implies that $r5c6$ cannot be 5 under either coloring. Thus the detector outputs (eliminate, $r5c6, 5$).

In Figure 10(c), the Y-Wing detector reads 3 cell masks, the pivot $r1c3$ containing $\{8,9\}$, and the wing $r2c1$ containing $\{6,9\}$ and the wing $r7c3$ containing $\{6,8\}$. It tests all shared digits between them. If the pivot is 9, the first wing would be 6, and if the pivot is 8, the second wing would be 6. At least one wing must contain 6. The orange cells $r7c1$ and $r8c1$ can both see the two wings, therefore the detector returns an elimination of 6 at both cells.

The XY-Chain detector in Figure 10(d) reads the four blue cell masks in chain order: $\{1, 8\}$, $\{1, 5\}$, $\{2, 5\}$ and $\{2, 8\}$. It tests the links between adjacent cells sharing 1, then 5, then 2. Following those links means at least one endpoint must be 8. Since the orange target $r7c1$ can see both such endpoints, the detector returns (eliminate, $r7c1, 8$). In both panels, M provides the candidate pairs to test.

Figure 10 shows only one match per strategy, but a detector call searches for matches beyond the shown ones. With the state S unchanged, each detector visits cells and/or combinations specific to its own rule; not all strategies perform the same row-by-row scans. Singles check cells and/or unit-digit pairs. Subsets look at small combinations within each row, column, and box. Intersections and Fish look at all possible locations of a digit from P in the units. Coloring joins pairs of positions (found from P) into links. Wings look at the M masks of a pivot and its peers. XY-Chains traverse links between two-candidate cells (using M to determine their common digits). For any match that results in a change, the detector reports the target cell and records either an assignment or an elimination. It leaves S unchanged while accumulating such actions.

For X-Wing, we have to check each digit in both directions. If we take the rows as our start units, the detector takes the P mask from each row, looks for pairs of rows with the same two potential columns, and finds in those columns the cells to remove outside the pair. We repeat the process with the columns as the start units. Figure 10(a) shows a single instance found during the column-based pass; it does not show the entire pass. The detector collects the actions from each match, filters out duplicates, and returns the final list of actions.

The solve loop terminates on the first strategy for which the complete detector invocation returns actions. The strategy's compatible actions are applied and we move to the next step in a new round, which means that the subsequent strategies are not tested against the original S . We compare each direct detector's deduplicated output with the reference detector on the same reached states to check that no assignment or elimination

is missing or added.

5.5. Batched actions

Back to state S in Figure 6, one Naked Single scan would read V and M , yielding four assignments, namely 3 at r1c4, 7 at r5c5, 1 at r7c6 and 7 at r9c6. All four are warranted by this state, and all are mutually consistent: their target cells differ, and those targeted with 7 are not peers. The solver sorts and deduplicates the actions; it never mixes these with those from some other strategy or some later state.

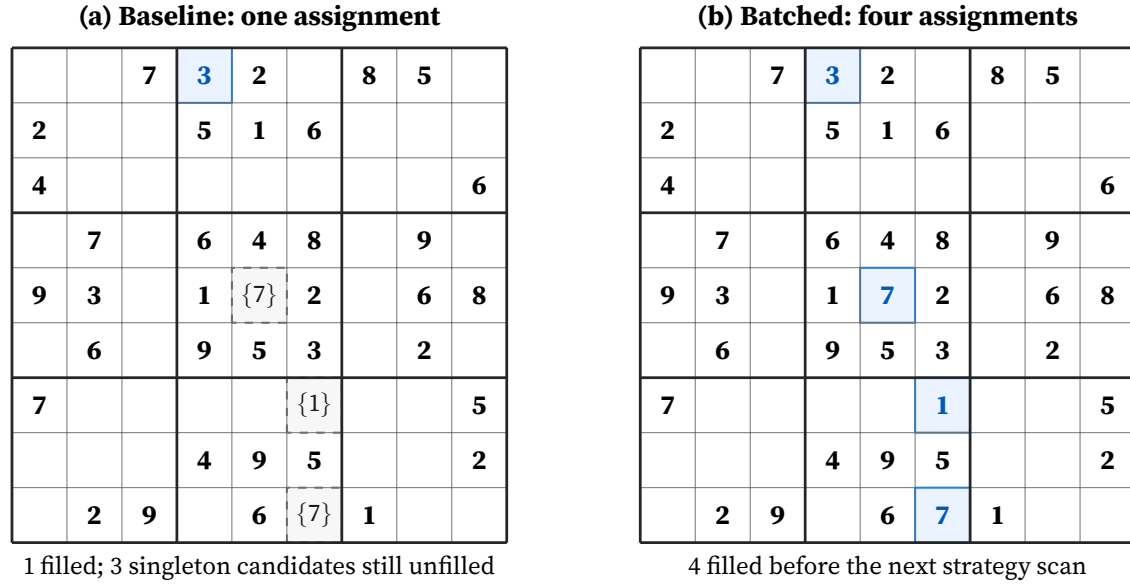


FIGURE 11. Two results of one round from the same state S . Solid blue cells contain assigned values; braces in outlined cells denote candidates in cells that remain empty. The baseline fills one cell, whereas batching fills all four before the next strategy scan. Other candidate changes are omitted.

In Fig. 11(a) the baseline can only fill r1c4 since the other 3 braced numbers still remain as candidates in unfilled cells and will have to be found in later rounds. In Fig. 11(b) our optimized solver has found all 4 assignments in a fixed order before scanning again. This avoids rediscovering those three singles. Batching is beneficial only if one detector produces multiple compatible actions.

The solver does not compare every pair of actions in advance. It applies them one by one, and checks each time whether the target cell is still empty and whether the digit is still a candidate. If any action invalidates a subsequent action, the solver fails with a stale-action error. Otherwise, the next strategy scan begins only after the state has been updated according to all four actions. The details of these updates are covered in Section 5.6.

5.6. Incremental update and delayed validation

The baseline already does a local propagation of an assignment: for $r1c4=3$ it populates the cell, resets the ordered candidates of the cell, and deletes 3 from six peer sets (see Figure 1). The optimized solver makes the same logical updates via cell masks M , but also maintains position indexes P . For example, when $r1c1$ drops 3, its candidates go from $\{1, 3, 6\}$ to $\{1, 6\}$ in M , and its position is cleared in the row-1, column-1, and box-1 indexes for 3 (see Figure 8). Updating these bits upon removing the candidate rather than recomputing P from all cells is the *incremental update*. The baseline did not recompute all candidates, either.

The second change is *delayed validation*. The baseline performs a full board check for any inconsistencies after every deduction is chosen. To prevent this repeated check of the board, our optimized solver checks each action against the current state during application and performs a single full board check just before returning a result. Since the full board check only reads the state, removing these intermediate checks does not alter the deductions or state updates along a valid solving path. However, before a completed grid can be reported as solved, a full board check ensures that there are no duplicate filled digits or any other state inconsistencies. If the resulting grid is invalid, we do not return it as a solution.

5.7. Exact failure caches

A subset detector examines one row, column, or box at a time. For example, if a Naked Triple check finds no action in row 6, the solver stores that fact along with the value V and candidate mask M for all nine cells of row 6. When a future pass encounters the same check with the same nine pairs, the detector sees identical input and finds no action, so the solver does not waste time searching. But if any candidate were deleted from $r6c1$ then the stored state would not match and row 6 is once again checked.

The cache only contains these "no action" results; it does not reuse a successful action. For Hidden Subsets, the position indexes P are determined by the same candidate masks M , so if V and M do not change, the detector input does not change either. Removing this cache caused a 3.0% increase in runtime in the final-configuration ablation (Figure 13).

5.8. Strategy scheduling

At the detection phase of Figure 6, the solver evaluates strategies sequentially, terminating once a strategy yields an action. Consequently, a strategy appearing later is evaluated only if all preceding strategies fail to produce an action, so its effective utility depends on both its cost of evaluation and its probability of yielding an action in such cases. Reordering

the strategies does not alter each strategy’s test logics, but it may change which action is returned first and, consequently, the path taken through the solver.

We compare two fixed orders. Both retain the first nine strategies in Table 3; Table 6 shows how they order the remaining six.

TABLE 6. Advanced strategy positions in the two evaluated orders. Positions 1–9 are identical.

Check position	Canonical order	Cost-aware order
10	X-Wing	XY-Chain
11	Swordfish	XYZ-Wing
12	Simple Coloring	X-Wing
13	Y-Wing	Y-Wing
14	XYZ-Wing	Simple Coloring
15	XY-Chain	Swordfish

Table 6 shows an example of how order impacts a round. Consider the case where all of the first 9 strategies return no action, and the only one that finds an action in the current state is XY-Chain. In canonical order, XY-Chain is checked after all of the other 6 advanced strategies. In the cost-aware order, it is checked first. If only Swordfish can find an action, it would be reached on the 2nd advanced check for canonical, and on the 6th advanced check for cost-aware. So, placing a strategy earlier reduces the number of checks made when it finds an action, and increases the number of checks made when another strategy needs to be used.

This order is selected based on detector costs and hit rates on the 10,000-puzzle corpus. Note that a detector’s hit rate may drop if it is shifted to an earlier position. For example, XY-Chain’s 100% hit rate in Table 4 was measured after all fourteen earlier strategies had returned no action. If we move XY-Chain to the 10th position, then it will run before 5 of the previous detectors and might find no action for some puzzles on which the first 9 strategies also fail. Therefore, we compare in Sec. 7 the times taken to solve the entire corpus in each ordering. The detector costs and success patterns from the 10,000 puzzles thus provide an empirical basis for applying this order to other puzzle sets, even if the benefit varies according to workload.

5.9. Puzzle-level parallelism

A round in a specific puzzle is determined by the outcome of prior moves. If you do many strategy checks on different threads, they need to be coordinated on any state change. Each puzzle is entirely independent from every other puzzle, so the executor could execute

concurrently puzzles at the same time. A fixed set of workers keeps pulling the next input index:

```
shared next_index = 0
each worker:
    while an input index i can be claimed:
        local_state = initialize(input[i])
        output[i] = solve(local_state)
```

Since the index claim is atomic, two workers will never attempt to execute the same puzzle. A worker that finishes an easier puzzle will pick up another puzzle right away, while a worker that is solving a harder puzzle keeps solving it. So we do not have cores waiting for a statically partitioned workload that does not distribute complexity evenly. And writing the solution at index i makes sure the input order is maintained even if the puzzles finish out of order.

Parallelizing at the puzzle level improves throughput as more inputs are available, but it does not reduce the time spent deducing the answer to a single puzzle, and it adds the cost of claiming and synchronization to each worker. Thus we report both single-puzzle latency and batch throughput separately, including the synchronization overhead in the latter.

6. Evaluation Methodology

The evaluation verifies deductions and final answers, then compares runtime on a fixed puzzle corpus.

6.1. Correctness validation

Optimized detectors are compared against the original full-record detectors on the same states that the latter reached when solving the corpus of 10,000 puzzles. Their assigned and eliminated cells must be the same after duplicates are eliminated. We verify that the profiled baseline is correctly reproducing all 556,541 deduced cells of the original unoptimized baseline.

After each performance run, returned grids are compared with stored expected answers. A run is rejected if any puzzle stalls, returns an error, or produces a different grid. Detector comparisons and expected-answer checks are outside performance timing.

6.2. Platform, build, and dataset

For the measurements we used an Apple M3 Pro MacBook Pro (with five performance and six efficiency cores and 18 GB memory) running Apple Clang 17.0.0 and compiled with the

Release build settings (-O3 -DNDEBUG). We did not control for processor frequency or background processes.

The corpus consists of 10,000 unique puzzles taken from the public-domain Sudoku Exchange puzzle bank (14): the first 2,500 from each of its easy, medium, hard, and diabolical sets that were solved by a previous fifteen-strategy solver. We verified that those solutions maintained the given values and satisfied the rules of Sudoku. We then verified that both the frozen baseline and the final solver were able to solve all the puzzles retained in the corpus. The comparison with HappyCerberus uses only the 7,319 puzzles that both solvers solve correctly.

6.3. Timing and comparison protocol

We time a full set of puzzles with a common serial or parallel execution wrapper that handles initialization, solving, final validation, writing results, and coordinating tasks and workers. We exclude input parsing, output buffer allocation, warm-up, worker creation, and checking expected answers from our timings. Also, profiling is turned off when timing.

For whole-set experiments, we perform five rounds. For the BT/MRV comparison, we perform three rounds. All rounds within an experiment use the same shuffled inputs and randomize the order of configurations. We report the mean runtime for each experiment, as well as the speedup against the reference provided in the description of each experiment. We measure single-puzzle latency as the time to execute one solve. We perform three rounds of single-puzzle latency measurement, providing 30,000 solve calls for each configuration to compute the mean, median, P95, and P99.

7. Experimental Results and Comparisons

Every run we report returns the correct final grids. The results we report below are for the 10,000 puzzle corpus unless otherwise noted, and the reported time is the mean over the measurement rounds described in Section 6. In Section 7.1, we present a basic comparison with the baseline, and later sub-sections answer each of our performance questions in turn via separate experiments. All speedups we report are computed from times measured in the same experiment.

7.1. Overall performance

Table 7 shows a comparison between the baseline frozen serial C++ version and the final version of HPS, which is solved serially and using eleven worker threads. In both cases all of the solver optimizations introduced in Section 5 are enabled (including the cost-aware strategy ordering), and the version with eleven worker threads uses puzzle level

parallelism. All three versions solve the same set of 10,000 puzzles with the same timing protocol as described in Section 6.3.

TABLE 7. Overall runtime for 10,000 puzzles, averaged over five measurement rounds. HPS enables all solver optimizations in both execution modes. Every speedup uses the serial C++ baseline as its reference and is calculated from unrounded mean times.

Configuration	Execution	Time (s)	Speedup vs serial baseline
C++ baseline	Serial	16.717	1.0×
Serial HPS	Serial	0.475	35.2×
Parallel HPS	11 worker threads	0.076	220.4×

Solver optimizations reduce the serial runtime from 16.717 seconds down to 0.475 seconds (a 35.2× speedup). Adding in the eleven worker threads reduces the runtime further to 0.076 seconds, for a total speedup over the serial version of 220.4×. The combined gain is due to both reduced work per solve, and concurrent solving of multiple independent puzzles. We go into more detail about the implementation changes, and the thread scaling, in the next two sections.

7.2. Performance across cumulative configurations

We compare these five cumulative serial configurations in a separate five-round experiment on the 10,000-puzzle corpus in Figure 12. The first optimized configuration, which includes compact state, position indexes, and failure caches (Sections 5.2, 5.3, and 5.7), as well as other implementation changes such as reduced output bookkeeping, drops runtime from 17.440 to 2.389 seconds. The lightweight solve loop with delayed validation (Sections 5.1 and 5.6) brings it down to 2.124 seconds. Direct detection and batched actions (Sections 5.4 and 5.5) drop runtime further to 0.538 seconds; cost-aware scheduling (Section 5.8) brings it to 0.476 seconds.

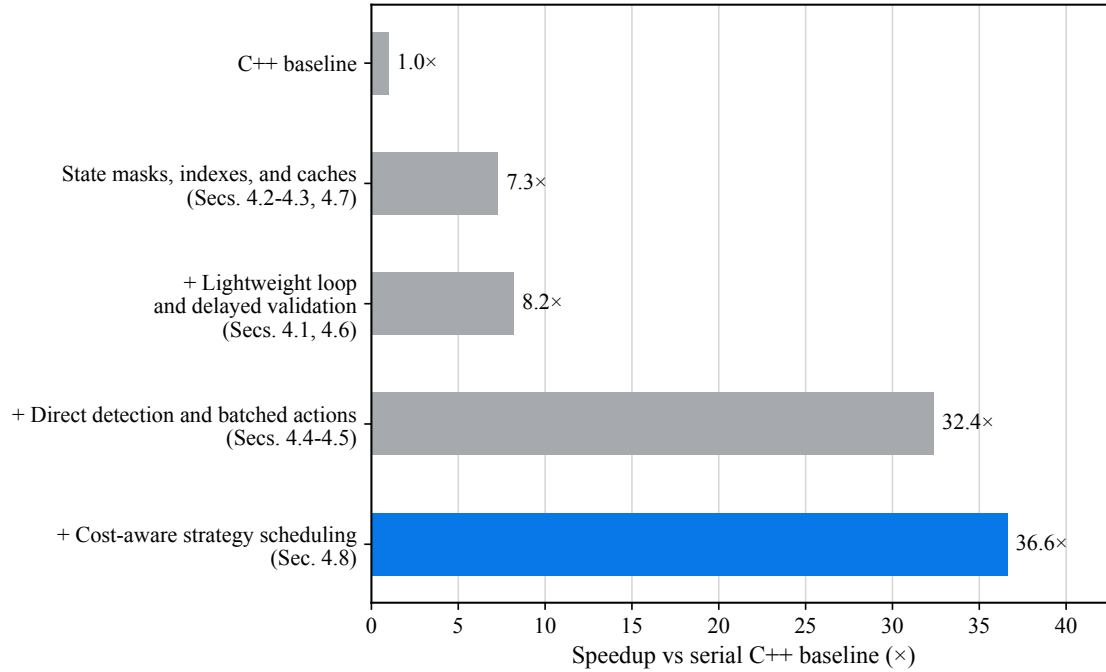


FIGURE 12. Cumulative serial performance on 10,000 puzzles. Each speedup uses the baseline measured in this experiment. Later configurations retain the preceding changes; some stages combine multiple optimizations.

Note that the bars report the performance of the cumulative configurations, and not the separate contribution of every optimization. For example, failure caching is already enabled in the first optimized configuration, so there is no separate step for it here. We examine selected mechanisms by disabling them in the final implementation in Section 7.3. Finally, puzzle-level parallelism is omitted from this serial comparison and evaluated in Section 7.4.

7.3. Effects of disabling optimizations

Unlike Section 7.2 where changes were added cumulatively, Figure 13 begins with the frozen final implementation and removes a single mechanism or a stated group at a time. All configurations use serial execution, the canonical strategy order, and solve the same 10,000 puzzles over five rounds of measurement. The percentages represent the increase in runtime relative to the complete configuration used in this experiment, not the C++ baseline.

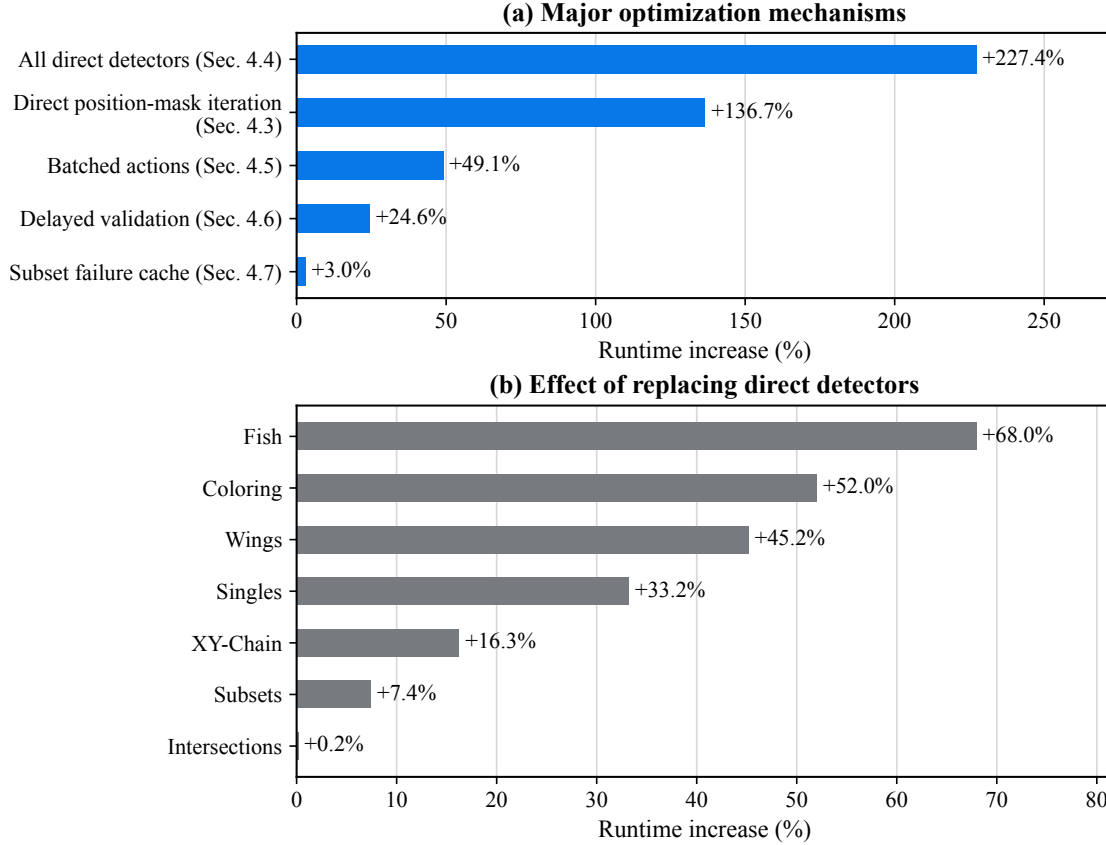


FIGURE 13. Runtime increase when disabling optimizations in the final serial implementation with canonical strategy order. Panel (a) compares major mechanisms from Sections 5.3–5.7; panel (b) replaces each strategy family’s direct detector with its original full-record detector. All strategies and their checking order remain unchanged. The panels have separate linear scales, and the effects are not additive.

In Figure 13(a), replacing all direct detectors with the original full-record detectors increases runtime by 227.4%. When building a temporary list for each position query rather than reading the positions directly from P as in Section 5.3, runtime increases by 136.7%. In both cases, P is retained. Disabling batched actions increases runtime by 49.1%. Restoring full-board checks after each successful round increases runtime by 24.6%. Disabling the Subset failure cache increases runtime by 3.0%.

Figure 13(b) determines which direct detector implementations provide the larger benefit as shown in Figure 13(a). We take each strategy family’s direct detector and replace it with its original detector and extract the actions from the deduction records that it builds. All strategies, their order, and their batching are kept the same. Because Fish, Coloring, and Wings have greater increases in runtime, they benefit more from the direct implementations than Subsets and Intersections do. We compare implementations rather than schedules, and measure changes to both the pattern searches and the actions

generated, not just to the record construction.

7.4. Single-puzzle latency and thread scaling

First we look at how scheduling of strategies (Section 5.8) influences time taken to solve a puzzle. The results of serial HPS calls in the time protocol of Section 6.3 are given in Table 8. We keep all other optimizations for the solver enabled and change only the order in which strategies are called.

TABLE 8. Serial single-puzzle latency for two HPS strategy orders. Each row pools 30,000 calls over three rounds on the 10,000-puzzle corpus. P95 and P99 are the times within which 95% and 99% of measured calls finish.

Strategy order	Mean (μ s)	Median (μ s)	P95 (μ s)	P99 (μ s)
Canonical	50.51	26.92	137.00	195.75
Cost-aware	45.69	26.83	119.79	171.63

Cost-aware order reduces the mean latency from 50.51 to 45.69 μ s, a 9.5% reduction. The median is almost unaffected, and P95 and P99 drop by approximately 12%, giving an improvement at the higher end of the measured latency distribution. The mean latency is reduced from 108.61 to 86.42 μ s for diabolical puzzles, but increased from 49.05 to 52.02 μ s for hard puzzles. Overall, these results confirm that this strategy scheduling works better on our corpus, though it might not improve performance for all types of puzzle distributions.

In Figure 14, we perform another thread-scaling experiment that explores puzzle-level parallelism (Section 5.9). The worker count is the only thing varied here for cost-aware HPS. Speedups are relative to the serial HPS in this experiment instead of the serial C++ baseline from Section 7.1.



FIGURE 14. Thread scaling of cost-aware HPS on 10,000 puzzles, relative to serial HPS without workers. Points show mean paired per-round speedups over five rounds; error bars show one sample standard deviation. The dashed line shows ideal linear scaling.

The results show that one worker yields 0.99x speedup, because it also evaluates puzzles sequentially. The two-, four-, eight-, and eleven-worker speedups are 1.87x, 3.53x, 5.59x, and 6.91x, respectively. None are close to the ideal linear speedup. These results measure how much faster adding workers makes HPS in this experiment. For more information on HPS speedup overall, see Section 7.1.

7.5. Backtracking references

Figure 15 compares cost-aware HPS to Basic BT and ordinary MRV. In the Basic BT implementation, we just branch to the first empty cell available. In the standard MRV implementation, we recompute the legal digits for each cell when choosing where to branch. This is a separate experiment with three rounds of execution on the same set of 10,000 puzzles. In each panel, we divide all runtimes by the mean HPS runtime across executions in that mode, so HPS will be 1 and smaller values indicate faster execution.

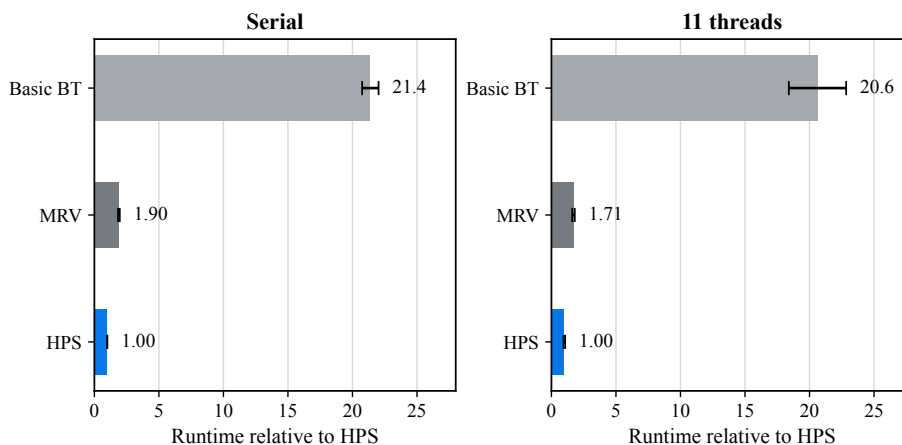


FIGURE 15. Runtime relative to HPS for 10,000 puzzles with the same answer output and puzzle-level execution wrapper. Both panels use the same linear scale. Bars and sample standard deviations over three rounds are divided by the mean HPS time in each panel. These comparisons use this experiment’s measurements, not the overall runtimes in Section 7.1.

For serial execution, Basic BT takes 21.4 times as long and MRV takes 1.90 times as long as HPS. With eleven workers, Basic BT takes 20.6 times as long and MRV takes 1.71 times as long as HPS. These results suggest that HPS is faster than both of these implementations on this particular puzzle corpus. However, this comparison is not intended to show that human strategies are inherently faster than pure search-based strategies.

7.6. External logic-solver comparison

We compare the cost-aware HPS with HappyCerberus, an external logic-based solver, on the 7,319 puzzles that both solve correctly (Section 6.2). HappyCerberus uses the same shuffled input as HPS with the same timing boundary. It also has the same puzzle-level execution wrapper. Table 9 reports the mean runtimes of both solvers over five runs.

TABLE 9. Mean runtime over five rounds on the 7,319 puzzles completed correctly by both solvers. This experiment uses a smaller, selected subset than the primary 10,000-puzzle comparison.

Solver	Serial time (s)	11-worker time (s)
HappyCerberus	2.174	0.343
HPS	0.223	0.0335

HPS solves these 7,319 puzzles 9.74 times faster in serial execution and 10.22 times faster using eleven workers, comparing the two solvers in the same execution mode. Note that we compare only the performance of the two solvers on a set of puzzles both can solve (i.e., execution speed), rather than comparing how many puzzles each solver can solve (i.e., coverage). Since their strategies and checking orders are different, we cannot attribute the speedup difference purely to our optimizations in Section 5.

8. Limitations

There are three primary limitations in our conclusions. 1) We solved our puzzles with a 15-strategy variant, and our alternate strategy ordering was created and evaluated on that corpus. It is unknown whether we have enough coverage of solves to provide a solve-ordering benefit on any arbitrary set of puzzles, nor if that coverage provides any advantage. 2) Our results were measured using a single processor and answer-only interface, and thus do not generalize to other processors or when full explanations are needed. 3) Before reporting a puzzle as solved, the solver validates the completed grid against the givens and the row, column, and box constraints. The limitation is strategy coverage: puzzles requiring deductions beyond the configured 15 strategies may remain unsolved.

9. Conclusion

In conclusion, this work demonstrates that a solver built with the help of human-named strategies can be made significantly faster without resorting to guessing or backtracking. The significant speedups were not due to any single change, but rather a combination of eliminating repeated candidate scans, eliminating temporary objects created to store

deductions, and eliminating repeated full-board scans after minor state changes. In its final form, our solver reproduced all expected answers for each puzzle in the 10,000-puzzle corpus, reducing the total serial batch runtime from 16.717 seconds down to 0.475 seconds, for a 35.2x speedup. The scope of this result is limited to puzzles solvable with the 15 strategies studied here, on the processor tested here, with answer-only output. We do not yet know if our solver can complete every arbitrary Sudoku puzzle, nor do we know the cost of producing an explanation. Future work involves testing additional strategy combinations, and different output modes to assess how broadly the design applies.

References

- [1] J. F. Crook. A pencil-and-paper algorithm for solving sudoku puzzles. *Notices of the American Mathematical Society*, 56(4):460–468, 2009. URL <https://www.ams.org/notices/200904/tx090400460p.pdf>.
- [2] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5):1052–1060, 2003. URL https://search.ieice.org/bin/summary.php?id=e86-a_5_1052.
- [3] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980. doi: 10.1016/0004-3702(80)90051-X.
- [4] Donald E. Knuth. Dancing links. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, pages 187–214. Palgrave, Basingstoke, UK, 2000. URL <https://arxiv.org/abs/cs/0011047>.
- [5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967 Spring Joint Computer Conference*, pages 483–485, 1967. doi: 10.1145/1465482.1465560.
- [6] Radek Pelánek. Difficulty rating of Sudoku puzzles by a computational model. In *Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference*, pages 434–439. AAAI Press, 2011. URL <https://cdn.aaai.org/ocs/2517/2517-11201-1-PB.pdf>.
- [7] Kohei Nishikawa and Takahisa Toda. Exact method for generating strategy-solvable Sudoku clues. *Algorithms*, 13(7):171, 2020. doi: 10.3390/a13070171. URL <https://arxiv.org/abs/2005.14098>.
- [8] Helmut Simonis. Sudoku as a constraint problem. In *Proceedings of the 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2005. URL <https://research.ucc.ie/en/publications/sudoku-as-a-constraint-problem/>.
- [9] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, 2008. doi: 10.1145/1452044.1452046. URL https://www.gecode.org/papers/SchulteStuckey_TOPLAS_2008.pdf.

- [10] HoDoKu2 Contributors. HoDoKu2. GitHub repository, <https://github.com/wyzelli/Hodoku2>. Accessed September 19, 2026.
- [11] SudokuMonster Contributors. Sukaku explainer. GitHub repository, <https://github.com/SudokuMonster/SukakuExplainer>. Accessed September 19, 2026.
- [12] HappyCerberus. Logic-based sudoku solver. GitHub repository, <https://github.com/HappyCerberus/sudoku>. Accessed September 19, 2026.
- [13] Tom Dillon. Tdoku: A fast sudoku solver and generator. GitHub repository, <https://github.com/t-dillon/tdoku>. Accessed September 19, 2026.
- [14] Grant McLean. Sudoku exchange puzzle bank. GitHub repository, <https://github.com/grantm/sudoku-exchange-puzzle-bank>. Accessed September 19, 2026.