

Sudoku

Aayush Bajaj

July 10, 2026

Contents

1	csp formulation	1
1.1	constraint propagation and ac-3	1
1.2	search heuristics	2
2	the backtracking solver	2
3	integer programming	3
4	exact cover and algorithm x	5
5	generation, difficulty, and the number 17	5
6	see also	6

sudoku is the drosophila of constraint satisfaction: small enough to hold in your head, rich enough to demonstrate every solving paradigm that matters. this page works through four of them against my actual code — a backtracking solver with $O(1)$ constraint sets (`arcade/references/sudoku/solver.py`), a dart port that also *generates* puzzles (`arcade-mobile`), an integer-programming formulation solved for real with `scipy`, and the exact-cover view that leads to knuth’s algorithm x. every timing below is a real run on this machine.

1 csp formulation

- **variables:** the 81 cells (r, c) .
- **domains:** $\{1, \dots, 9\}$ for empties; singleton domains for the given clues.
- **constraints:** 27 `alldifferent` constraints — one per row, column, and 3×3 box. every cell sits in exactly three of them.

1.1 constraint propagation and ac-3

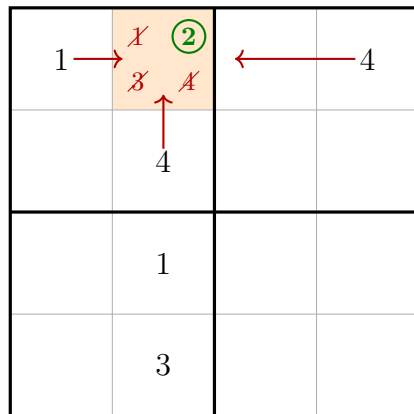
before searching, shrink domains. **arc consistency** (ac-3) repeatedly removes any domain value with no possible partner in a constraining cell; for sudoku’s **alldifferent** (decomposed into pairwise “not equal” arcs) this reduces to the familiar pencil-mark discipline: a placed digit deletes itself from the domains of the 20 peers it shares a unit with. cascades follow — when a domain hits one value (a **naked single**), place it and propagate again; when a value survives in only one cell of a unit (a **hidden single**), same. easy newspaper puzzles dissolve entirely under this fixpoint; harder ones leave a residue that search must finish.

1.2 search heuristics

when propagation stalls, backtracking begins, and two orderings dominate its cost:

- **mr_v** (minimum remaining values): branch on the cell with the *smallest* surviving domain — fail fast, prune early.
- **lc_v** (least constraining value): try the digit that eliminates the fewest options from peers — succeed early on satisfiable branches.

the experiments below put a number on how much this matters: my solver, which uses *neither*, pays a factor of a thousand on an adversarial instance.



row kills 1, 4; column kills 3, 4; box kills 4

Figure 1: candidate elimination in a 4×4 mini-sudoku. the shaded cell's candidates start as $\{1, 2, 3, 4\}$; the row kills 1 and 4, the column kills 3 (and re-kills 1 and 4), the box kills 4 (and 1 again). one candidate survives: a naked single.

2 the backtracking solver

`solver.py` is a leetcode-37-shaped dfs with one genuinely good idea: **incremental constraint sets**. instead of re-scanning a row, column and box (27 reads) to test a placement, it maintains one `set` per unit and updates them as moves are made and unmade — membership tests are $O(1)$, and the box index is the little arithmetic gem $(r // 3) * 3 + c // 3$, which flattens the 3×3 tiling of boxes into 0..8.

```
1 # from arcade/references/sudoku/solver.py (paraphrased)
2 rows = [set() for _ in range(N)]
3 cols = [set() for _ in range(N)]
4 boxes = [set() for _ in range(N)]          # index = (r//3)*3 + c//3
5
6 def validMove(board_state, pos):
7     r, c = pos
8     b = (r // 3) * 3 + (c // 3)
9     return [d for d in "123456789"
10             if d not in rows[r] and d not in cols[c] and d not in boxes[b]]
11
12 def dfs_rec(idx):
13     if idx == len(emptyies):
14         return True          # all empties filled legally
15     r, c = empties[idx]
16     b = (r // 3) * 3 + (c // 3)
17     for move in validMove(board, (r, c)):
18         board[r][c] = move   # place
19         rows[r].add(move); cols[c].add(move); boxes[b].add(move)
20         if dfs_rec(idx + 1):
```

```

21     return True
22     board[r][c] = '.' # undo
23     rows[r].remove(move); cols[c].remove(move); boxes[b].remove(move)
24     return False

```

make, recurse, unmake — the same chronological-backtracking skeleton as every dfs in this wiki (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009). what it *lacks* is ordering: **empties** is filled in row scan order and digits are tried "123456789" ascending, always. two real runs make the cost of that concrete. first, arto inkala’s 2012 “world’s hardest sudoku” (hard for *humans* — its solving path needs deeply nested forcing chains):

```

8 1 2 7 5 3 6 4 9
9 4 3 6 8 2 1 7 5
6 7 5 4 9 1 2 8 3
1 5 4 2 3 7 8 9 6
3 6 9 8 4 5 7 2 1
2 8 7 1 6 9 5 3 4
5 2 1 9 7 4 3 6 8
4 3 8 5 2 6 9 1 7
7 9 6 3 1 8 4 5 2
inkala solved in 0.024s

```

24 milliseconds — human difficulty means nothing to a backtracker. now the classic **anti-brute-force** puzzle, engineered so that a row-major, ascending-digit search commits to wrong digits in the nearly-empty top rows and must exhaust enormous subtrees to discover it:

```

9 8 7 6 5 4 3 2 1
2 4 6 1 7 3 9 8 5
3 5 1 9 2 8 7 4 6
1 2 8 5 3 7 6 9 4
6 3 4 8 9 2 1 5 7
7 9 5 4 6 1 8 3 2
5 1 9 2 8 6 4 7 3
4 7 2 3 1 9 5 6 8
8 6 3 7 4 5 2 1 9
anti-brute-force puzzle solved in 34.004s

```

same solver, same 81 cells, **three orders of magnitude slower** — note the solution’s first row is 987654321, maximally hostile to ascending trial order. an mrv cell choice (or even just trying the most-constrained rows first) collapses this instance back to milliseconds. that factor-of-1400 gap *is* the argument for search heuristics, measured rather than asserted.

3 integer programming

the page’s tag promises integer programming, so here is the full formulation. introduce 729 binary variables,

$$x_{rcd} = \begin{cases} 1 & \text{cell } (r, c) \text{ holds digit } d \\ 0 & \text{otherwise} \end{cases} \quad r, c, d \in \{1, \dots, 9\}, \quad (1)$$

and four families of exactly-one constraints (81 rows each, 324 total):

$$\begin{aligned}
\sum_d x_{r,c,d} &= 1 \quad \forall r, c && \text{every cell holds one digit} \\
\sum_c x_{r,c,d} &= 1 \quad \forall r, d && \text{every row contains each digit once} \\
\sum_r x_{r,c,d} &= 1 \quad \forall c, d && \text{every column contains each digit once} \\
\sum_{(r,c) \in B} x_{r,c,d} &= 1 \quad \forall B, d && \text{every box } B \text{ contains each digit once}
\end{aligned}$$

clues are pinned by forcing their variable's lower bound to 1. there is no objective — any feasible point is a solution, so we minimise the zero function. solved for real with scipy's `milp` (the highs branch-and-bound solver underneath):

```

1 import numpy as np
2 from scipy.optimize import milp, LinearConstraint, Bounds
3 from scipy.sparse import lil_matrix
4
5 idx = lambda r, c, d: 81*r + 9*c + d          # flatten x[r,c,d] -> 729 vars
6
7 A = lil_matrix((324, 729)); row = 0
8 for r in range(9):                          # each cell: exactly one digit
9     for c in range(9):
10         for d in range(9): A[row, idx(r,c,d)] = 1
11         row += 1
12 for r in range(9):                          # each row: each digit once
13     for d in range(9):
14         for c in range(9): A[row, idx(r,c,d)] = 1
15         row += 1
16 for c in range(9):                          # each column: each digit once
17     for d in range(9):
18         for r in range(9): A[row, idx(r,c,d)] = 1
19         row += 1
20 for br in range(3):                         # each box: each digit once
21     for bc in range(3):
22         for d in range(9):
23             for r in range(3*br, 3*br+3):
24                 for c in range(3*bc, 3*bc+3): A[row, idx(r,c,d)] = 1
25             row += 1
26
27 lb = np.zeros(729); ub = np.ones(729)
28 for r in range(9):                          # pin the clues
29     for c in range(9):
30         if puzzle[r][c] != '.':
31             lb[idx(r, c, int(puzzle[r][c]) - 1)] = 1
32
33 res = milp(c=np.zeros(729),                  # feasibility problem
34           constraints=LinearConstraint(A.tocsr(), np.ones(324), np.ones(324)),
35           integrality=np.ones(729), bounds=Bounds(lb, ub))

```

real output, on the same two puzzles that separated the backtracker by a factor of 1400:

```

inkala:          status: 0 (Optimization terminated successfully), milp time 0.073s
anti-brute-force: status: 0 (Optimization terminated successfully), milp time 0.005s

```

the adversarial instance is the ilp solver's *easiest* — highs' presolve is itself a constraint propagator, and the branching that remains uses proper pseudo-cost ordering.

the lesson runs both directions: the puzzle only ever punished the *ordering*, never the problem.

the lp relaxation of this system is not integral in general — fractional ‘superpositions’ of digits satisfy all 324 equalities — which is exactly why the integrality flag, i.e. branch and bound, is doing real work.

4 exact cover and algorithm x

the ilp’s constraint matrix has a special shape: every entry is 0/1, and every constraint says *exactly one*. that makes sudoku an **exact cover** problem — choose a subset of the 729 rows of a 729×324 incidence matrix so that each of the 324 columns is covered exactly once. each matrix row is a candidate placement “digit d at (r, c) ” and covers 4 columns: its cell-constraint, row-constraint, column-constraint and box-constraint.

knuth’s **algorithm x** solves exact cover by recursive elimination: pick the column with the fewest remaining rows (mrv again, wearing different clothes), try each row that covers it, delete every column that row covers and every row that clashes, recurse, undo. his **dancing links** implementation makes the undo spectacular — the matrix is a torus of doubly-linked nodes, and unlinking/relinking a node is two pointer writes each way, so backtracking costs almost nothing.¹ dlx solvers dispatch essentially any 9×9 sudoku in microseconds-to-milliseconds; it is the standard answer once you care about throughput.

5 generation, difficulty, and the number 17

the dart port in arcade-mobile (`lib/features/games/sudoku/sudoku_page.dart`) is the same backtracking idea run in reverse to *make* puzzles:

1. fill the three diagonal 3×3 boxes with independent shuffles (they share no units, so any permutations are mutually consistent);
2. complete the grid with the backtracking solver, trying digits in *shuffled* order (randomised value ordering = uniform-ish random completion);
3. store the completed grid as the hint oracle, then delete 45 random cells.

honest audit of that recipe: deleting cells without checking that the remainder still has a *unique* solution means some generated boards are ambiguous — the hint feature (which reveals the stored solution’s digit for a wrong or empty cell) can then contradict a player’s equally-valid line. the comment in the code even sketches a difficulty ladder (“easy: 40, medium: 50, hard: 55” removals) but the constant is hardcoded to 45. clue *count* is a weak difficulty proxy anyway — inkala’s monster has 21 clues, four more than the minimum — with difficulty really living in which *techniques* the solving path demands.

two structural facts worth pinning down precisely:

- **17 is the minimum.** no 16-clue 9×9 sudoku with a unique solution exists — settled by mcguire, tugemann and civario’s exhaustive computer search (roughly 800 processor-years), and tens of thousands of 17-clue puzzles are catalogued.²
- **generalised sudoku is np-complete.** on $n^2 \times n^2$ boards, deciding solvability

¹d. e. knuth (2000), *dancing links*, in *millennial perspectives in computer science*, arxiv:cs/0011047. the observation credited there to hitotumatu and noshita: a removed doubly-linked node still points at its old neighbours, so restoration needs no search.

²g. mcguire, b. tugemann, g. civario (2014), *there is no 16-clue sudoku: solving the sudoku minimum number of clues problem via hitting set enumeration*, *experimental mathematics* 23(2):190–217.

is np-complete (yato and seta, 2003), so no polynomial algorithm covers all sizes unless $p = np$.³ the 9×9 case is “just” a large constant.

6 see also

- [hashiwokakero solver](#) — the same propagate-then-backtrack architecture on a graph-shaped puzzle
- [constraint satisfaction problems](#) — parent section
- [arcade](#) — where the solver and the dart game live
- [classical algorithms](#) — backtracking in its natural habitat

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.

³t. yato and t. seta (2003), *complexity and completeness of finding another solution and its application to puzzles*, ieice transactions on fundamentals E86-A(5):1052–1060.