How the solver turns blanks into certainty

The engine is a compact constraint solver: it models every empty cell as a bit mask, repeatedly applies deterministic Sudoku deductions, then searches only when deduction stalls.

plain text input · row / column / block units · u128 candidates · naked singles · hidden singles · guided backtracking

Parse
Read whitespace-separated rows and validate the shape.

Model
Precompute units and peers for each cell.

Mask
Track every candidate set as bits in a u128.

Deduce
Propagate placements, naked singles, hidden singles.

Branch
Pick the tightest unsolved cell and order values by impact.

Solve
Return the first complete state that survives constraints.

1. INPUT BECOMES A CONSTRAINT MODEL

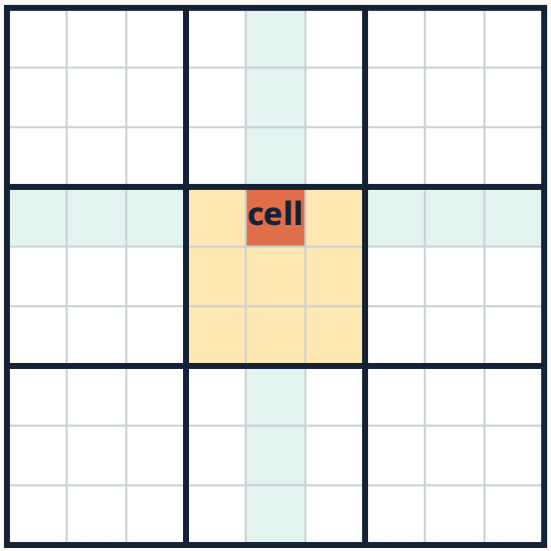
The grid is square, but the solver thinks in units and peers

After parsing, each cell knows the three units it belongs to: one row, one column, and one block. The union of those units, excluding the cell itself, becomes its peer list.

```
0 2 3 0
0 0 0 1
1 0 0 0
0 4 2 0
```

Blank cells are written as `0`. A 4x4 puzzle uses 2x2 blocks; a 9x9 puzzle uses 3x3 blocks; the same machinery handles both.

3	27	20	1
unit kinds per cell	units in a 9x9 puzzle	peers for a 9x9 cell	state object to search



row peers
same horizontal unit

column peers
same vertical unit

block peers
same sub-square

selected cell

Peers are precomputed once, so assigning a value later is just a quick walk over affected cells.

A cell's possible values fit into one `u128`

Each value maps to one bit. Eliminating a candidate clears its bit; placing a value replaces the whole mask with exactly that bit.

9x9 full mask

values 1 through 9 are initially possible

After peers place 1, 5, 8


```
mask & !value_bit(value)
```

Why masks work well here

Bit masks make candidate checks tiny: membership is `mask & bit != 0`, removal is `mask &= !bit`, and a naked single is `count_ones() == 1`.

Full mask

`(1 << size) - 1` turns on every legal value bit.

Placed value

Once a cell is assigned, its candidate mask becomes exactly `value_bit(value)`.

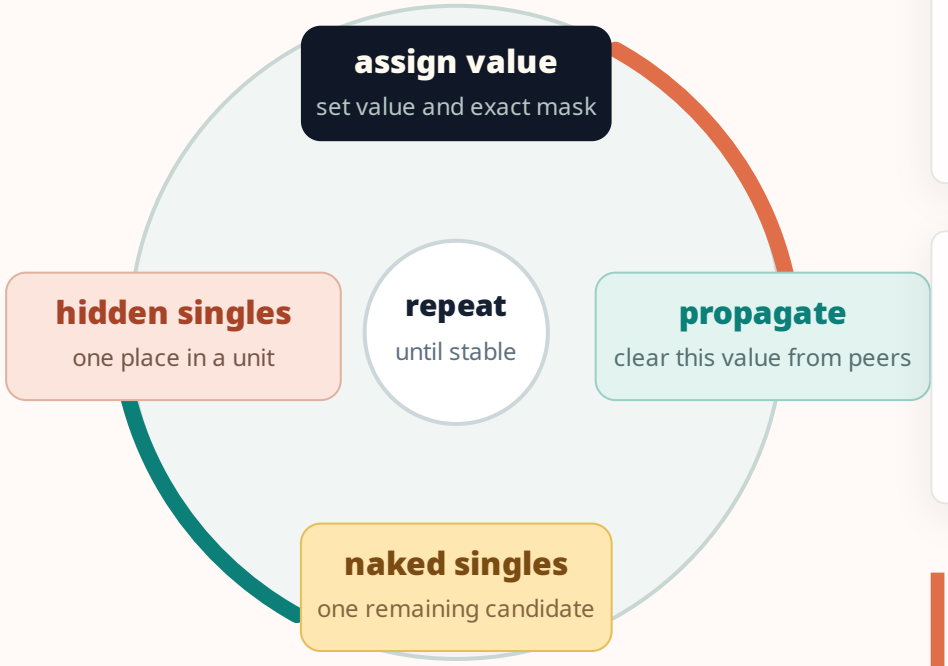
Contradiction

If an unsolved cell's mask becomes zero, that path is rejected.

3. DEDUCTION RUNS UNTIL IT STALLS

Every assignment immediately removes pressure from the board

The solver loops over simple, strong rules. If any rule places a value, it starts another pass because that placement may unlock more forced moves.



Naked single

A cell with one remaining candidate is forced. The solver detects this with `count_ones() == 1`, assigns it, and propagates again.

Hidden single

For each row, column, and block, the solver checks every value. If only one unsolved cell in that unit can still hold the value, that cell is forced.

If a unit has no legal place for a value, or a candidate mask goes empty, that path is rejected as a contradiction.

4. SEARCH BEGINS ONLY AFTER DEDUCTION STALLS

The branch point is chosen to make guessing as constrained as possible

When no rule can place another value, the solver picks one unresolved cell and tries each candidate in a cloned state.

Choose branch cell

Fewest candidates wins; ties prefer more unsolved peers.

{2,7}	{1,5,9}	{4,6}
2 candidates	3 candidates	2 candidates
13 open peers	18 open peers	17 open peers
		selected

Order selected candidates

Impact = peers that would lose this value as a candidate.

Cell heuristic

The solver scans unresolved cells and keeps the cell with the fewest candidate bits. If two cells are equally tight, it picks the one touching more unsolved peers.

Candidate heuristic

For that cell, values are sorted by impact. A value has higher impact when more peer cells currently contain that same value as a candidate.

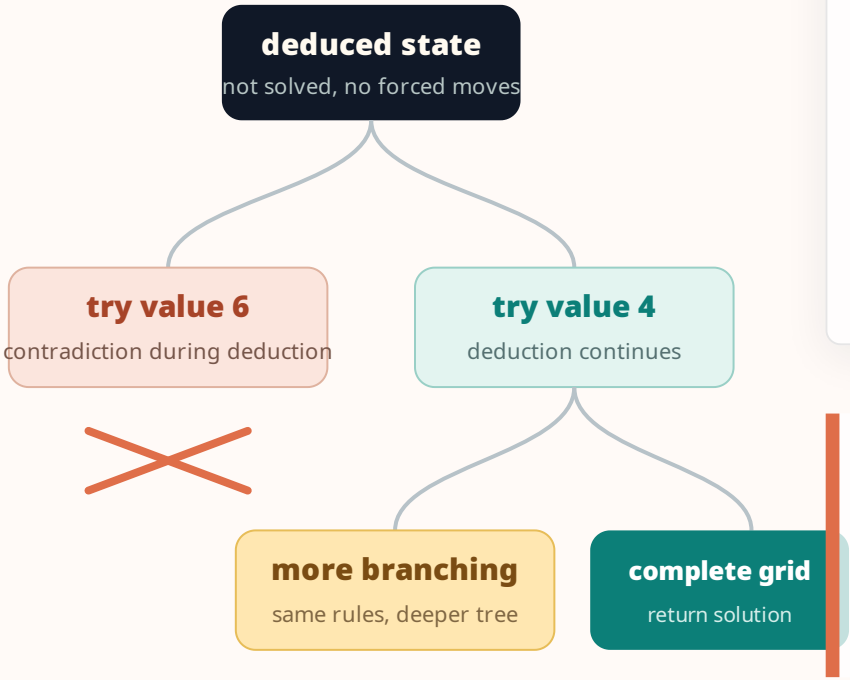
Tie break

If candidate impacts are equal, the smaller value is tried first for deterministic output.

5. BACKTRACKING IS JUST RECURSIVE STATE CLONING

Every assumption either solves the board or collapses quickly

Search clones the current state, assigns one candidate, and immediately re-enters deduction. Failed branches vanish; successful branches return the solved grid.



The core rhythm

- Run deduction before every search decision.
- If all cells are filled, return the state.
- Otherwise choose one branch cell.
- Try candidates in impact order.
- Discard any branch that hits a contradiction.

Key idea

Most of the “intelligence” is in shrinking the search tree before guessing and making each guess remove as much uncertainty as possible.

IMPLEMENTATION MAP

Where to look in the code

The solver is small enough that the visual model maps directly onto a few functions in `src/lib.rs`.

Parsing

`Puzzle::parse` validates square dimensions, block layout, and value ranges before solving starts.

Constraint model

`Solver::new` creates row, column, and block units; `build_peers` turns them into peer lists.

Candidate masks

`full_mask` and `value_bit` encode possible values into a single integer.

Propagation

`assign` places a value, checks duplicate peers, and clears that value from every unsolved peer.

Deduction

`deduce` repeatedly applies naked singles and hidden singles until no rule progresses.

Search

`search` clones states, tries ordered candidates, and returns the first branch that reaches a complete solution.

Correctness guard

Contradictions stop invalid givens and impossible branches early.

Performance trick

Peer lists and bit masks make the hot path small.

Search control

Fewest candidates narrows the branch factor.

Impact ordering

Higher-impact values test stronger assumptions first.