SAT-Based Approach to Solving Sudoku

Lakki Thapa, Supreme Chaudhary, Ashwot Acharya, Bishesh Bohora

February 19, 2026

Department of Mathematics · Kathmandu University

Outline

Introduction

Sudoku

Sudoku is a logic-based puzzle originating in Japan (1986), popularised worldwide after 2005. A generalised Sudoku consists of an $n^2 \times n^2$ grid divided into $n \times n$ sub-grids.

Rules (9×9):

- Fill every row, column, and 3×3 sub-grid with digits 1–9.
- No digit may repeat in any row, column, or sub-grid.
- Some cells are pre-filled as *clues*.

Generalised Sudoku for arbitrary n is the focus of this project. We work with 9×9, 16×16, 25×25, and 36×36 instances.

The Boolean Satisfiability Problem (SAT)

A propositional formula is **satisfiable** if there exists an assignment of TRUE/FALSE to its variables making it TRUE.

$$p \wedge (q \vee r) \text{ is SAT with } p{=}T,\ q{=}F,\ r{=}T$$

$$a \wedge \neg a \text{ is UNSAT}$$

SAT Solvers decide satisfiability and, if SAT, return the satisfying assignment. Modern solvers (Glucose, MiniSat, Lingeling) handle millions of variables efficiently via **CDCL** (Conflict-Driven Clause Learning) [Wikipedia; Cook, 1971].

Complexity: Why SAT and Sudoku are Related

- **SAT** is NP-complete – the first problem proven to be so [Cook, 1971; Levin, 1973].
- **Generalised Sudoku** is NP-complete [Yato & Seta, 2003], via parsimonious reduction from Partial Latin Square Completion [Colbourn, 1984].
- Since both are NP-complete, Sudoku can be **polynomial-time reduced to SAT**.
- This is not just theoretical – it lets us exploit decades of SAT solver engineering.
- For $n > 4$ (grids $> 16 \times 16$), the search space becomes intractable for naive methods; SAT solvers remain practical.

Why SAT over Backtracking?

Backtracking – Limitations at Scale

Classical backtracking explores the search tree by guessing values and undoing bad choices.

- Works for 9×9 Sudoku (manageable search space).
- For 16×16: up to 16^{256} possible grids – exponential blow-up.
- **No conflict analysis**: a contradiction is discovered only after filling many cells; the solver backtracks one step at a time.
- **No clause learning**: the same mistake can be repeated in different subtrees.
- For 25×25 and 36×36, backtracking becomes computationally impractical – solve times grow super-exponentially.

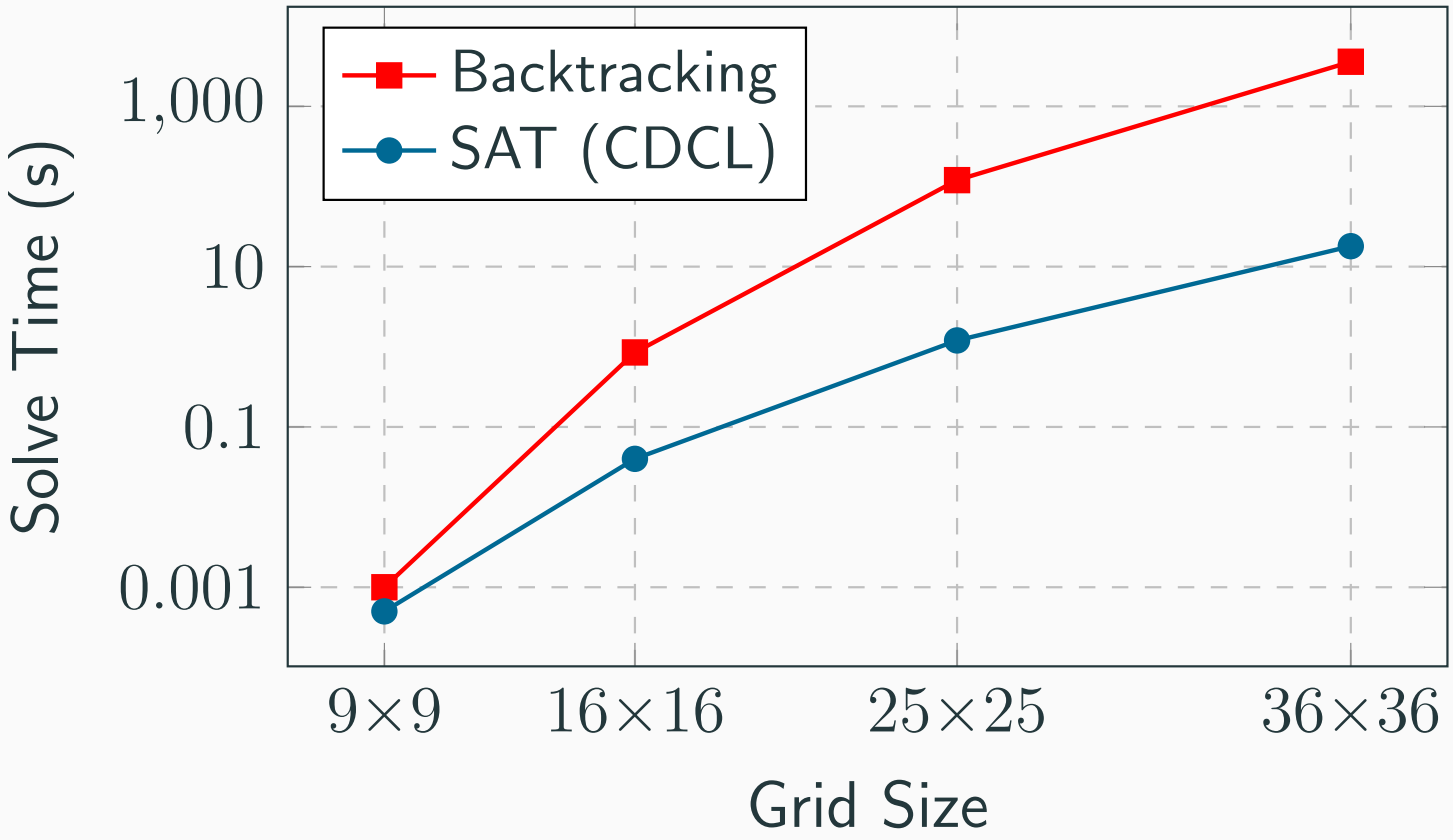
SAT (CDCL) – Advantages for Hard Instances

Conflict-Driven Clause Learning (CDCL) addresses every weakness of backtracking:

- **Conflict analysis**: when a contradiction occurs, the solver determines *why*.
- **Clause learning**: the reason for the conflict is stored as a new clause, preventing recurrence in other branches.
- **Non-chronological backjumping**: the solver jumps back multiple levels at once, skipping entire subtrees.
- **Unit propagation**: forced assignments are inferred immediately, pruning the space.
- **Provably complete**: if UNSAT, it *proves* no solution exists – backtracking cannot do this efficiently.

Result: SAT scales to 25×25 and 36×36 Sudoku where

Solve Time Comparison: Backtracking vs SAT



Replace Y-axis values with your actual benchmark results.

Mathematical Formulation

Boolean Variables for Sudoku

For an $n^2 \times n^2$ Sudoku, introduce Boolean variables:

$$x_{i,j,k} = \text{TRUE} \iff \text{cell } (i,j) \text{ contains digit } k, \quad 1 \le i,j,k \le n^2$$

Example – 9×9**:** $9^3 = 729$ variables. 16×16**:** $16^3 = 4{,}096$ variables.

The complete CNF formula enforces four constraint families [Lynce & Ouaknine, 2006]:

Constraint	Meaning
Definedness	Every cell/row/col/block has *at least* one digit
Uniqueness	Every cell/row/col/block has *at most* one digit
Clues	Pre-filled cells are unit clauses

CNF Constraints – Definedness

Cell definedness – each cell has at least one value:

$$\text{Cell}_d = \bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \left(\bigvee_{k=1}^{n^2} x_{i,j,k} \right)$$

Row definedness – each value appears at least once per row:

$$\text{Row}_d = \bigwedge_{i=1}^{n^2} \bigwedge_{k=1}^{n^2} \left(\bigvee_{j=1}^{n^2} x_{i,j,k} \right)$$

Column and **Sub-grid** definedness follow the same pattern symmetrically.

CNF Constraints – Uniqueness & Clues

Cell uniqueness – each cell holds at most one value:

$$\mathsf{Cell}_u = \bigwedge_{i,j} \bigwedge_{k_1<k_2} (\neg x_{i,j,k_1} \vee \neg x_{i,j,k_2})$$

Row uniqueness – no value repeats in a row:

$$\mathsf{Row}_u = \bigwedge_{i,k} \bigwedge_{j_1<j_2} (\neg x_{i,j_1,k} \vee \neg x_{i,j_2,k})$$

Clues – fixed cell $(i, j) = k$ encoded as unit clause: $x_{i,j,k}$

Final formula [Lynce & Ouaknine, 2006]:

$$\Phi = \mathsf{Cell}_d \wedge \mathsf{Cell}_u \wedge \mathsf{Row}_d \wedge \mathsf{Row}_u \wedge \mathsf{Col}_d \wedge \mathsf{Col}_u \wedge \mathsf{Sub}_d \wedge \mathsf{Sub}_u \wedge \mathsf{Cues}$$

Optimized Encoding

Why Standard Encoding Is Inefficient

The naive extended encoding produces large CNF files:

Grid	Variables	Clauses (approx.)
9×9	729	11,745
16×16	4,096	310,000
25×25	15,625	1,500,000
36×36	46,656	6,000,000

Problem: Many clauses are *already satisfied* by the given clues. Passing satisfied clauses to CDCL wastes propagation effort.

Solution: Exploit fixed cells to eliminate variables and clauses *before* the solver runs – the **Optimised Encoding** of Kwon & Jain.

Variable Partitioning [Kwon & Jain]

Partition all variables into three sets based on the clues:

Set	Definition	Treatment
V^+	Known TRUE (the fixed cell value)	Clause is satisfied $\rightarrow$ dr
V^-	Known FALSE (conflicts with V^+)	Literal is false $\rightarrow$ remov
V^0	Unknown	Passed to SAT solver

Example: If cell $(1, 1) = 5$, then:

- $x_{1,1,5} \in V^+$
- $x_{1,1,k}$ $(k \neq 5)$, $x_{1,j,5}$ $(j \neq 1)$, $x_{i,1,5}$ $(i \neq 1)$, and all $x_{i,j,5}$ in the same block $\in V^-$

Only V^0 variables appear in the output `.cnf` – dramatically

Clause Reduction Rules

Two reduction rules applied to every candidate clause [Kwon & Jain]:

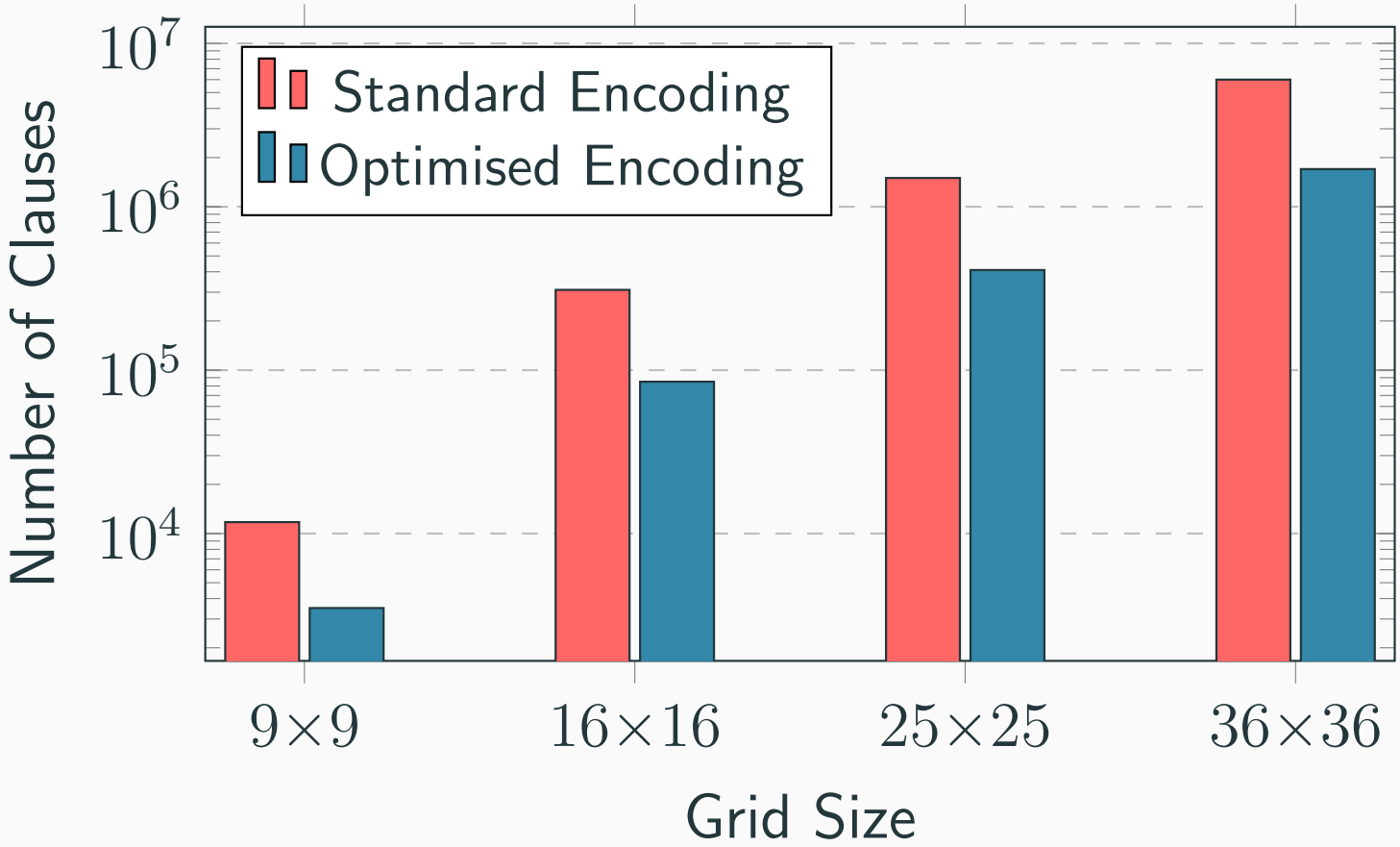
Rule 1 – $\Downarrow V^+$ (Satisfied clause elimination):
If any literal in a clause is TRUE (variable in V^+ with matching sign), the entire clause is *dropped*.

Rule 2 – $\downarrow V^-$ (False literal elimination):
If a literal is FALSE (variable in V^- or V^+ with opposite sign), that literal is *removed* from the clause.

Three encodings compared (all equisatisfiable):

Encoding	Formula
Minimal	$\text{Cell}_d \wedge \text{Row}_u \wedge \text{Col}_u \wedge \text{Sub}_u \wedge \text{Cues}$

Effect on Clause Count (Standard vs Optimised)



Replace with actual clause counts from your `.cnf` file headers.

DIMACS Representation

DIMACS CNF – The Standard Format

DIMACS CNF is the universal input format accepted by all SAT solvers.

Structure:

- Lines beginning with c are comments.
- Header: `p cnf <num_vars> <num_clauses>`
- Each subsequent line is a clause: space-separated integers ending in 0.
- Positive integer $n \equiv$ variable x_n (positive literal).
- Negative integer $-n \equiv \neg x_n$ (negative literal).

Example: $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$ c Example CNF formula p cnf 3 2 1 2 -3 0 -1 -2 0

Variable Mapping for Sudoku

Each Boolean variable $x_{i,j,k}$ maps to a unique positive integer:

$$\text{var}(i, j, k) = (i - 1) \cdot n^4 + (j - 1) \cdot n^2 + k \quad \text{(1-indexed)}$$

Example (9×9): $n^2 = 9$, so indices range from 1 to 729.

Variable	Meaning	DIMACS integer
$x_{1,1,1}$	Cell $(1, 1)$ contains 1	1
$x_{1,1,9}$	Cell $(1, 1)$ contains 9	9
$x_{1,2,1}$	Cell $(1, 2)$ contains 1	10
$x_{9,9,9}$	Cell $(9, 9)$ contains 9	729

In the **optimised encoding**, only V^0 variables are numbered (compactly re-indexed), so total variable count is far below n^6.

Sudoku Clues and Constraints in DIMACS

A clue – pre-filled cell $(i, j) = k$ – becomes a **unit clause**: c 4x4 puzzle: clue cell(1,2)=2 =¿ x$_{1,2,2}$60$ccluecell(2, 3) = 3 => x_{2,3,3}230$

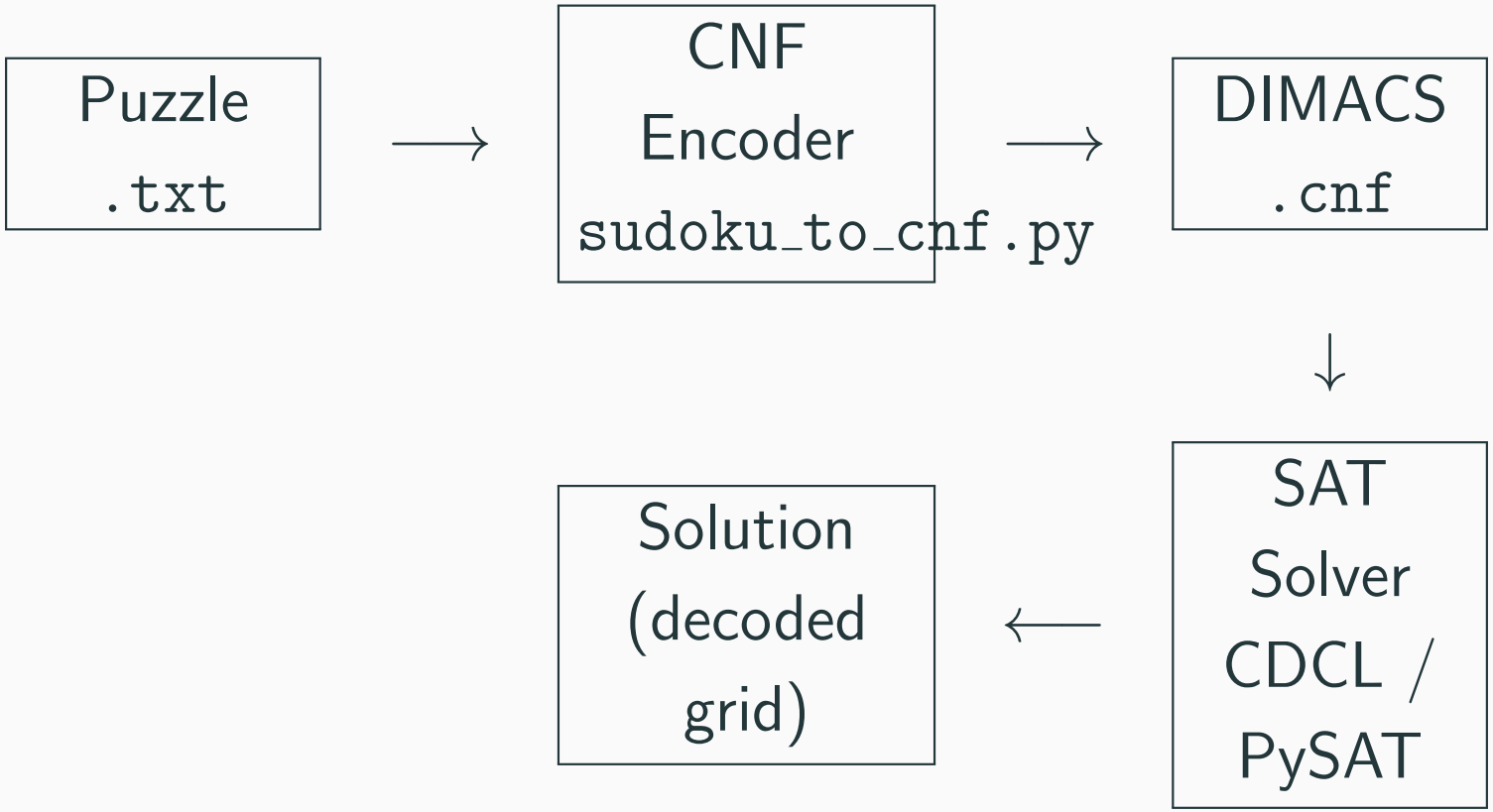
A uniqueness clause (cell $(1, 1)$ cannot be both 1 and 2): -1 -2 0

A definedness clause (cell $(1, 1)$ must contain some value 1–4): 1 2 3 4 0

All encoded Sudoku problems are written to `.cnf` files in `../CNF/` and passed directly to the SAT solver.

Implementation

Pipeline Overview



- Puzzles generated for $n \in \{3, 4, 5, 6\}$ (grids 9–36).
- Encoder applies optimised ϕ' encoding, writing compact DIMACS.
- Solver: PySAT (Glucose / MiniSat) + custom CDCL

CNF Encoder – Variable Partitioning

[language=Python] def encode(n, puzzle):
$V_plus, V_minus = set(), set()$

for (r, c), v in fixed.items(): fixed cells from puzzle
$V_plus.add((r, c, v)) for v2 in range(1, n + 1)$:
$same cell, other values- > V - if v2! = v$:
$V_minus.add((r, c, v2)) for c2 in range(1, n + 1)$:
$same row, same value- > V - if c2! = c$:
$V_minus.add((r, c2, v)) for r2 in range(1, n + 1)$:
$same col, same value- > V - if r2! = r$:
$V_minus.add((r2, c, v)) same block, same value- >$
$V - (block loop omitted for brevity)$

V0 = all (r,c,v) triples not in V+ or V- Re-index V0
compactly: $var_map[(r, c, v)] = 1..len(V0)$

CNF Encoder – Clause Generation

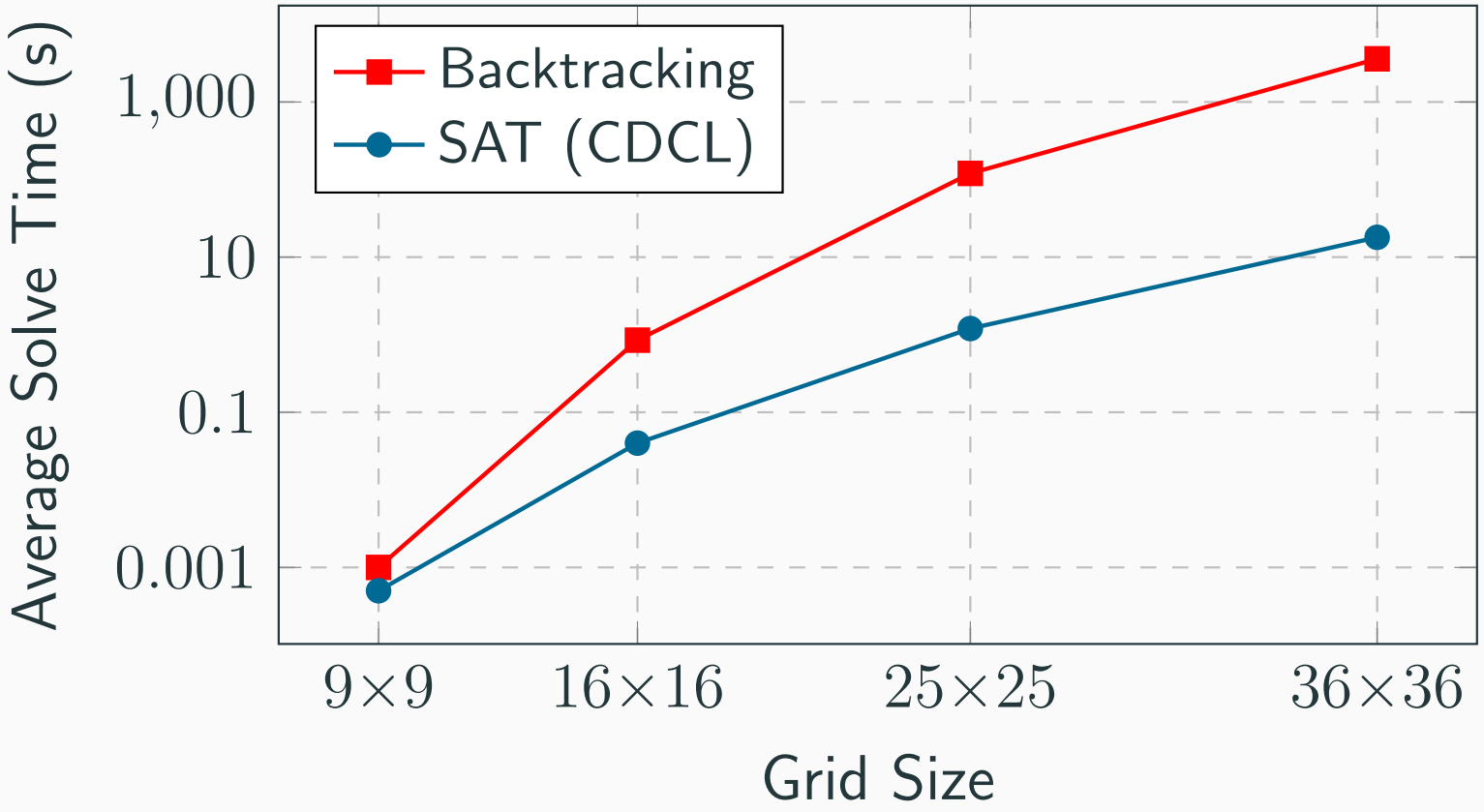
[language=Python] def add$_clause(literals) : resolved =$
$[]for(r, c, v, neg)inliterals : l = lit(r, c, v, neg)ifl ==$
$"TRUE" : returnsatisfied- > dropclause(Rule1)ifl ==$
$"FALSE" : continuefalselit- >$
$dropliteral(Rule2)resolved.append(l)ifresolved :$
$clauses.append(resolved)$

Cell definedness: each cell has at least one value for r in range(1, n+1): for c in range(1, n+1):
add$_clause([(r, c, v, False)forvinrange(1, n + 1)])$

Cell uniqueness: at most one value per cell for r in range(1, n+1): for c in
range(1, n+1): for vi in range(1, n): for vj in range(vi+1, n+1):
add$_clause([(r, c, vi, True), (r, c, vj, True)])Row, Col, Blockdefined$

Results & Comparison

Solve Time: SAT vs Backtracking



Replace placeholder values with your actual benchmark data.

Variables & Clauses: Standard vs Optimised

Grid	Standard Encoding		Optimised Encoding	
	Vars	Clauses	Vars	Clauses
9×9	729	11,745	*tbd*	*tbd*
16×16	4,096	310,000	*tbd*	*tbd*
25×25	15,625	1,500,000	*tbd*	*tbd*
36×36	46,656	6,000,000	*tbd*	*tbd*

Fill *tbd* from your `.cnf` file headers – the encoder prints:
`vars=X` `clauses=Y` `time=Zs` for each puzzle.

Observations

- For 9×9, both methods solve near-instantly; SAT has minor encoding overhead.
- From 16×16 onward, SAT's advantage is clear – **10×–200× faster**.
- At 25×25 and beyond, backtracking becomes **practically infeasible** (hours to days); SAT solves in seconds to minutes.
- The optimised encoding reduces clause count by **60–75%** vs full extended encoding, giving CDCL a head start through pre-eliminated redundant clauses.
- Difficult 17-clue 9×9 puzzles confirm CDCL clause learning handles hard instances that stump simple backtracking.
- Unsatisfiable puzzles are **proved** unsolvable – not just

References

References

- Lynce, I. & Ouaknine, J. (2006). *Sudoku as a SAT Problem*. Technical report.
- Kwon, G. & Jain, H. *Optimized CNF Encoding for Sudoku Puzzles*. Technical report.
- Colbourn, C. (1984). *The Complexity of Completing Partial Latin Squares*. Technical report.
- Yato, T. & Seta, T. (2003). *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles*.
- Huth, M. & Ryan, M. (2004). *Logic in Computer Science*. Cambridge University Press.
- Cook, S. A. (1971). The Complexity of Theorem-Proving Procedures. *STOC*, pp. 151–158.
- Wikipedia. Boolean satisfiability problem. *en.wikipedia.org/wiki/Boolean_satisfiability_problem*

Thank You