SAT-Based Approach to Solving Sudoku

Lakki Thapa, Supreme Chaudhary, Ashwot Acharya, Bishesh Bohora

February 20, 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.

The Sudoku Solving Algorithms

The Backtracking Algorithm- How it works

The backtracking algorithm is a DFS $DepthFirstSearch$ algorithm

- Firstly finds an empty cell
- Tries for a candidate number $1, 2, 3, 4 ... etc$ and checks the constraint
- if the number is valid it places on the cell if not it checks foranother number resursively
- If none of the number works it undos the last assignment $Backtracking$ and tries for a different number
- This is recursively continued untill the grid is completed or all possibilities are exhausted

The Backtracking Algorithm : Limitation

- Works for relatively nicely for small grid (i.e $4 \times 4, 9 \times 9, 16 \times 16$)
- For 16×16: up to 16^{256} possible grids i.e growing exponentially
- Contradiction is discovered only after filling many cells; the solver backtracks one step at a time.
- The same errors can be repeated in different subtrees.
- For 25×25 and 36×36, backtracking becomes computationally impractical – solving times grow super-exponentially.

Backtracking is a depth-first search with no memory of failures.

The CDCL Algorithm (Conflict-Driven Clause Learning)

CDCL is an advanced algorithm used in modern SAT solvers.

- **Decision step:** Choose a variable and assign a value (make a guess).
- **Unit propagation:** Automatically deduce forced assignments from constraints (Boolean Constraint Propagation).
- **Conflict detection:** If a contradiction occurs, identify the conflicting constraints.
- **Clause learning:** Analyze the conflict and learn a new clause that prevents repeating the same mistake.
- Repeat until:
  - A satisfying assignment is found, or
  - The formula is proven unsatisfiable.

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 and can easily be expanded to $n^2 \times n^2$ Sudoku where backtracking fails or takes a very long time.

Why SAT?

SAT is one of the most powerful problem-solving frameworks in computer science.

- **Expressiveness:** Many combinatorial problems can be encoded as SAT.
- **Powerful solvers:** Modern CDCL solvers handle millions of variables efficiently.
- **Reduction principle:** If we can encode a problem into SAT, we can leverage highly optimized SAT solvers instead of designing a new algorithm.
- **Theory support:** Yato (2003) showed that even restricted forms of Sudoku are NP-complete — meaning structured constraint problems can be naturally reduced to SAT.

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:

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

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

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

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

CNF Constraints – Uniqueness & Clues

Cell uniqueness – each cell holds at most one value:

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

Row uniqueness – no value repeats in a row:

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

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

Final formula [Lynce & Ouaknine, 2006]:

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

Optimized Encoding

The optimized encoding for sudoku

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 $\rightarrow$ drop it
V^-	Known FALSE (conflicts with V^+)	Literal is false $\rightarrow$ remove from clause
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 reducing problem size.

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	$\mathsf{Cell}_d \wedge \mathsf{Row}_u \wedge \mathsf{Col}_u \wedge \mathsf{Sub}_u \wedge \mathsf{Cues}$
Extended	$+\ \mathsf{Cell}_u$ (adds redundant but helpful clauses)
Optimised ϕ'	**Full Φ with V^+/V^- elimination applied**

Tests in [Lynce & Ouaknine, 2006] show the optimised encoding is fastest in practice.

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}
6 0
c clue cell(2,3)=3 => x_{2,3,3}
23 0
```

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

```
-1 -2 0
```

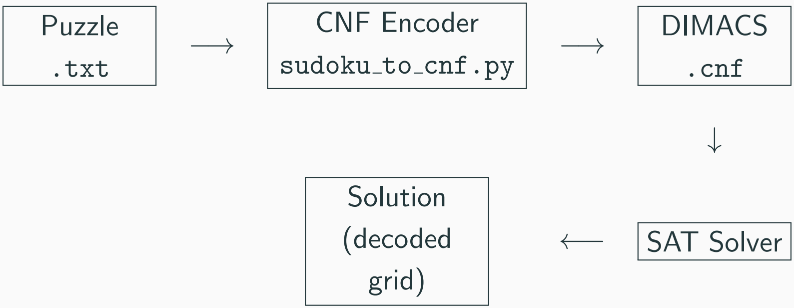
A definedness clause (cell $(1, 1)$ must contain some value 1–4):

```
1 2 3 4 0
```

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

Implementation

Overview



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

CNF Encoder

```
def encode(n, puzzle):
    V_plus, V_minus = set(), set()

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

    # 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

```
    def add_clause(literals):
        resolved = []
        for (r, c, v, neg) in literals:
            l = lit(r, c, v, neg)
            if l == "TRUE":  return       # satisfied -> drop clause
 (Rule 1)
            if l == "FALSE": continue     # false lit -> drop literal
            resolved.append(l)
        if resolved: 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) for v in range(1, n+1)])

    # Cell uniqueness: at most one value per cell
```

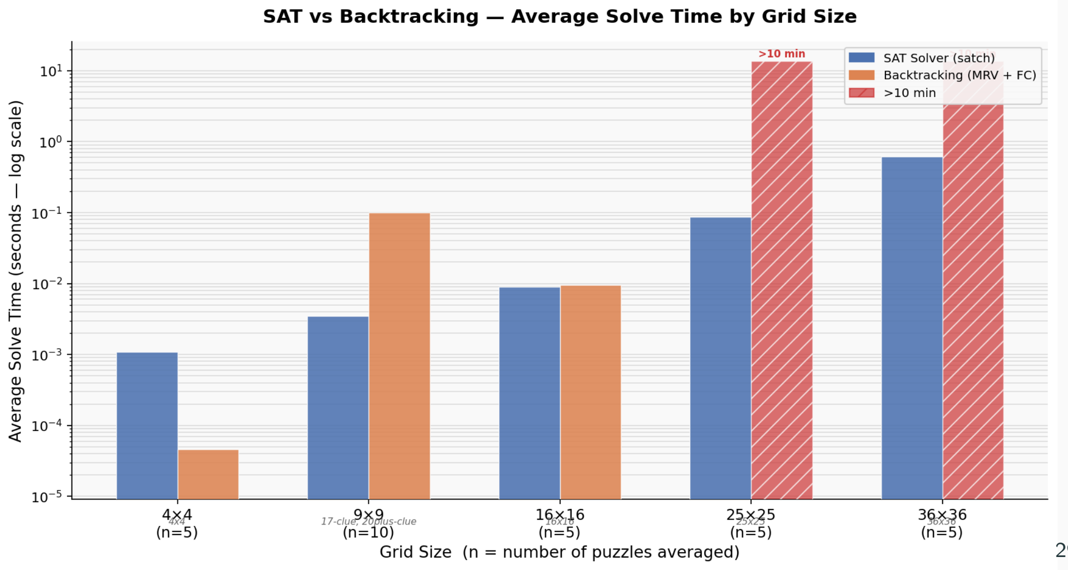
CDCL Implementation

We made our CDCL algorithm based on satch , a SAT solving tool avaible below is a code snippet:

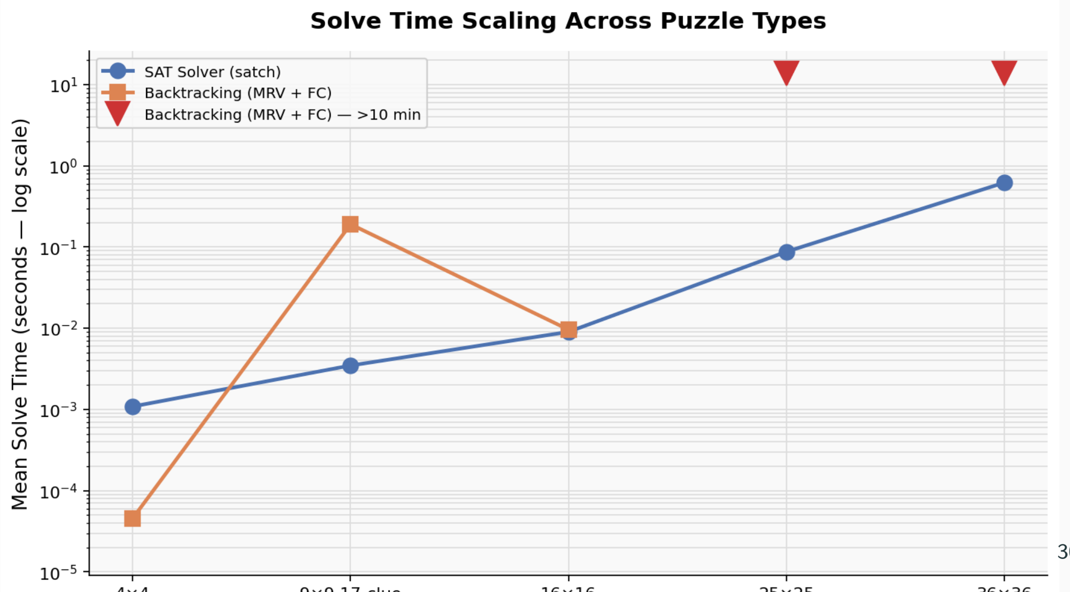
```c
static int solve(void) {
solver.level = 0;
for (;;) {
    int conflict = propagate();
    if (conflict >= 0) {
        if (solver.level == 0) return UNSAT;
        int bt = analyze(conflict);
        backtrack(bt);
        continue;
    }
    int var = decide();
    if (var == 0) return SAT;
    solver.level++;
    assign(var, solver.level, -1);
}
}
```

Results & Comparison

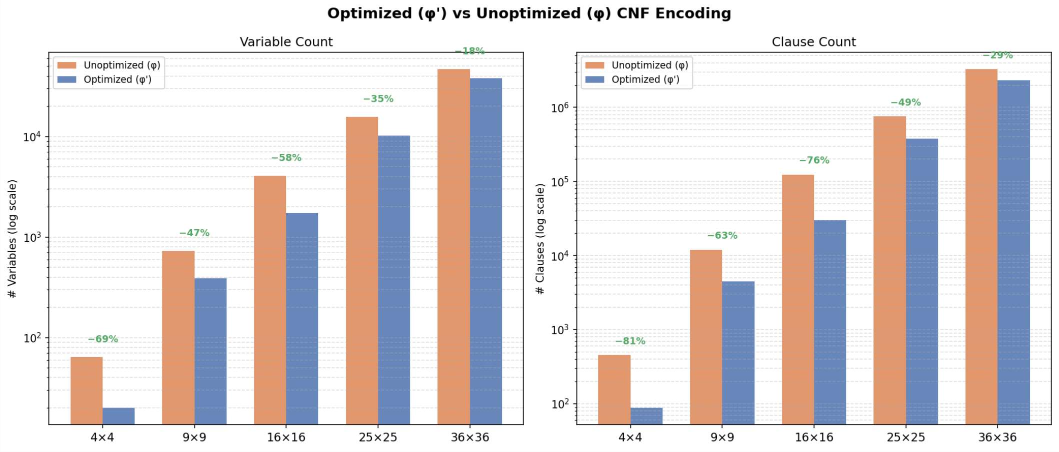
Solve Time Comparison: Backtracking vs SAT



Solve Time: SAT vs Backtracking



Comparision between standard Vs Optimized CNF



Optimized (φ') vs Unoptimized (φ) CNF Encoding
Variable Count
Unoptimized (φ)
Optimized (φ')
Variables (log scale)
10^2
10^3
10^4
−69%
−47%
−58%
−35%
−18%
4×4
9×9
16×16
25×25
36×36
Clause Count
Unoptimized (φ)
Optimized (φ')
Clauses (log scale)
10^2
10^3
10^4
10^5
10^6
−81%
−63%
−76%
−49%
−29%
4×4
9×9
16×16
25×25
36×36

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 where as 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 timed out.

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