SAT Based Approach To Solving Sudoku

THIRD YEAR PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF BSC. IN COMPUTATIONAL MATHEMATICS

BY

Ashwot Acharya, Bishesh Bohora, Supreme Chaudhary, Bishesh Bohora

DEPARTMENT OF MATHEMATICS
KATHMANDU UNIVERSITY
DHULIKHEL, NEPAL

CERTIFICATION

This project entitled "**SAT Based Approach To Solving Sudoku**" is carried out under our supervision for the specified entire period satisfactorily and is hereby certified as a work done by the following students:

1. Ashwot Acharya
2. Bishesh Bohora
3. Supreme Chaudhary
4. Lakki Thapa

in partial fulfillment of the requirements for the degree of B.Sc. in Computational Mathematics, Department of Mathematics, Kathmandu University, Dhulikhel, Nepal.

M.r K.B Manandhar
Assistant Professor
Department of Mathematics,
School of Science
Kathmandu University,
Dhulikhel, Kavre, Nepal
Date: August, 2025

APPROVED BY:
I hereby declare that the candidate qualifies to submit this report of **SAT Based Approach To Solving Sudoku** to the Department of Mathematics.

Head of Department
Department of Mathematics
School of Science
Kathmandu University
Date: August, 2025

ACKNOWLEDGMENTS

First and foremost, we would like to thank our supervisor Mr K.B Manandhar sir for guiding us through this research project. Their feedback and suggestions helped us overcome numerous challenges and shaped our understanding of the subject matter.

Abstract

Sudoku is a well-known puzzle whose generalized version can be formulated as a decision and search problem over an $n^2 \times n^2$ grid. From a computational perspective, generalized Sudoku is NP-complete, and finding a valid completion corresponds to solving a function problem in FNP. This project presents a Boolean satisfiability (SAT) based approach for solving generalized Sudoku instances by reducing the puzzle to a propositional formula in Conjunctive Normal Form (CNF).

We formally encode the structural constraints of Sudoku—definedness and uniqueness across cells, rows, columns, and sub-grids—into Boolean variables of the form $x_{i,j,k}$, representing the assignment of digit k to cell (i, j). The resulting constraints are translated into CNF and expressed in DIMACS format, enabling compatibility with standard SAT solvers. We adopt an optimized encoding strategy to improve propagation efficiency and conflict detection in modern Conflict-Driven Clause Learning (CDCL) solvers.

The implementation consists of generating CNF files using Python, mapping logical variables to integer representations, and solving the instances using both existing SAT solvers and a basic SAT solver implemented in C. We also compare the SAT-based method with a classical backtracking approach to evaluate performance differences. Experimental observations demonstrate that SAT-based techniques provide a systematic and scalable framework for solving larger generalized Sudoku instances, highlighting the practical strength of reductions to SAT for combinatorial search problems.

Contents

List of Acronyms

SAT	Boolean Satisfiability Problem
CNF	Conjunctive Normal Form
CDCL	Conflict-Driven Clause Learning
DPLL	Davis–Putnam–Logemann–Loveland Algorithm
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science
MRV	Minimum Remaining Values
VSIDS	Variable State Independent Decaying Sum
UNSAT	Unsatisfiable
SAT (result)	Satisfiable

1 Introduction

1.1 Sudoku

A puzzle in which a $n^2 \times n^2$ grid consisting of n "$n \times n$" sub-grids is to be filled with numbers from 1 to n^2 so that every row, column, and region contains only one instance of each number, $n \geq 1$. In a sudoku puzzle, few cells are already filled, those are called clues. The **Sudoku Problem** is to determine whether there exist a completion such that the forementioned conditions are satisfied.

The most common format for sudoku is of 9×9 grid. In this project we worked upon larger variants. For arbitary positve integer n, we call it **Generalized Sudoku**.

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

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

Figure 1: 9×9 Sudoku and its solution [Arizona]

1.1.1 Complexity of Generalized Sudoku Problem

The Partial Latin Square Completion problem was shown to be NP-complete [Colbourn]. Yato and Seta [Yato, 2003] demonstrated that Partial Latin Square can be parsimoniously reduced to Number Place (generalized Sudoku), thereby establishing the ASP-completeness of Sudoku, which captures the computational hardness of finding another solution. Because ASP-completeness implies NP-completeness, generalized Sudoku is **NP-complete**.

1.1.2 Sudoku as CSP

Sudoku can naturally be formulated as a Constraint Satisfaction Problem (CSP), where each cell is a variable, the domain consists of the numbers $\{1, \ldots, n^2\}$, and the constraints enforce row, column, and subgrid consistency. Classical approaches solve this CSP using backtracking search [Russell and Norvig, 2010, **?**].

1.2 The Boolean Satisfiability Problem

In logic and computer science, the Boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated SATISFIABILITY, SAT or B-SAT) asks whether there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the formula's variables can be consistently replaced by the values TRUE or FALSE to make the formula evaluate to TRUE. If this is the case, the formula is called satisfiable, else unsatisfiable. [Wikipedia] We refer it as SAT.
$a \vee \neg b$ is SATISFIABLE with $a = 1$ and $b = 0$.
$a \wedge \neg a$ is UNSATISFIABLE .

1.2.1 Complexity of SAT

SAT is NP complete. In fact, it was the first problem to be known as such. As proved by Stephen Cook [Cook, 1971] at the University of Toronto in 1971 and independently by Leonid Levin at the Russian Academy of Sciences in 1973. [Wikipedia].

1.2.2 SAT Solvers

SAT Solvers are computer programs or algorithms that check if there exist some assignment of boolean variables in the given problem such that it evaluates to True. SAT solvers also provide us the assignments. Therefore dealing with the search if satisfiable.

1.3 Justification for SAT approach

Since generalized Sudoku is NP-complete (and thus belongs to NP), it can be reduced in polynomial time to any other NP-complete problem, such as Boolean satisfiability (SAT). Note that forementioned complexity is for when n grows without bound, but in practice we have a finite grid. This reduction provides a formal justification for using SAT solvers to find Sudoku solutions: by converting a Sudoku instance into an equivalent SAT instance, we can leverage modern SAT-solving algorithms to efficiently compute a valid completed grid. This approach is theoretically sound, because solving the SAT instance is guaranteed to yield a solution to the original Sudoku problem whenever one exists. It is also found SAT solving aids in efficient solving of CSPs.[Lardeux et al., 2020]

2 Mathematical Formulation

2.1 Conjuctive Normal Form

A boolean formula is said to be in a conjective normal form if it is a conjuction of clauses , where each clause is a disjunction of literals.[Huth and Ryan, 2004]
The following expression is in CNF

$$((x_1 \vee x_2) \wedge (x_2 \vee \neg x_3))$$

Every boolean expression can be converted to CNF.

2.2 Representation of Sudoku As Boolean Expression (CNF)

Our goal is to represent the constraints of the puzzle as boolean expression and obtain a formula. For this we take the naive approach i.e each cell is visited and all constraint are enforced for every possible digit.

	2		
		3	
	3		
		1	

Figure 2: 4×4 Sudoku

Each cell in a $n^2 \times n^2$ sudoku has a total of n^2 different possible digit it can contain. Suppose,
i: row index
j: column index
k: possible digit
$1 \leq i, j, k \leq$
Then the variable x_{ijk} is true whenever the cell (1,j) contains the digit k.

2.2.1 Definedness

Every cell must contain atleast one digit. Every row, column and sub-grid must contain digits from 1 to n^2. This property is called **Definedness**.

Each cell contains atleast one digit

$$Cell_d = \bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \left(\bigvee_{k=1}^{n^2} x_{ijk} \right)$$

Each value appears atleast once in each row

$$Row_d = \bigwedge_{i=1}^{n^2} \bigwedge_{k=1}^{n^2} \left(\bigvee_{j=1}^{n^2} x_{ijk} \right)$$

Each value appears atleast once in each column

$$Col_d = \bigwedge_{j=1}^{n^2} \bigwedge_{k=1}^{n^2} \left(\bigvee_{i=1}^{n^2} x_{ijk} \right)$$

Each value appears atleast once in each sub-grid

$$Subgrid_d = \bigwedge_{a=0}^{n-1} \bigwedge_{b=0}^{n-1} \bigwedge_{k=1}^{n^2} \left(\bigvee_{i=1}^{n} \bigvee_{j=1}^{n} x_{an+i,\, bn+j,\, k} \right)$$

2.2.2 Uniqueness

A cell can contain atmost one digit and different cells in any row/column/sub-grid cannot attain same value, these are the **Uniqueness** Constraints.

Each cell contains at most one value:

$$Cell_u = \bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \bigwedge_{1 \le k_1 < k_2 \le n^2} (\neg x_{i,j,k_1} \vee \neg x_{i,j,k_2})$$

Each value appears at most once in every row:

$$Row_u = \bigwedge_{i=1}^{n^2} \bigwedge_{k=1}^{n^2} \bigwedge_{1 \le j_1 < j_2 \le n^2} (\neg x_{i,j_1,k} \vee \neg x_{i,j_2,k})$$

Each value appears at most once in every column:

$$Col_u = \bigwedge_{j=1}^{n^2} \bigwedge_{k=1}^{n^2} \bigwedge_{1 \le i_1 < i_2 \le n^2} (\neg x_{i_1,j,k} \vee \neg x_{i_2,j,k})$$

Each value appears at most once in every subgrid:

$$Subgrid_u = \bigwedge_{a=0}^{n-1} \bigwedge_{b=0}^{n-1} \bigwedge_{k=1}^{n^2} \bigwedge_{\substack{(i_1,j_1),(i_2,j_2) \\ 1 \le i_1,j_1,i_2,j_2 \le n \\ i_1 < i_2, j_1 < j_2}} (\neg x_{an+i_1,\, bn+j_1,\, k} \vee \neg x_{an+i_2,\, bn+j_2,\, k})$$

For clues

If the puzzle contains a fixed entry $(i, j) = k$, it is encoded as a unit clause:

$$\textbf{clues} = \bigwedge_{(i,j,k) \in G} x_{i,j,k}$$

which forces them to be true.

Example

For given 4×4 Sudoku in Figure 2, we introduce Boolean variables

$$x_{i,j,k}$$

with $i, j, k \in \{1, 2, 3, 4\}$, where

$$x_{i,j,k} = 1 \quad \text{iff cell } (i, j) \text{ contains number } k.$$

Thus, we obtain $4^3 = 64$ Boolean variables.
Each cell must contain at least one number:

$$\bigvee_{k=1}^{4} x_{i,j,k} \quad \text{for all } i, j.$$

for cell $(1, 1)$:

$$x_{1,1,1} \vee x_{1,1,2} \vee x_{1,1,3} \vee x_{1,1,4}.$$

Each cell contains at most one number:

$$\neg x_{i,j,k} \vee \neg x_{i,j,\ell} \quad \text{for all } k \neq \ell.$$

$$\neg x_{1,1,1} \vee \neg x_{1,1,2}.$$

Each number appears exactly once in every row.
At least once:

$$\bigvee_{j=1}^{4} x_{i,j,k} \quad \text{for all } i, k.$$

for number 1 in row 1:

$$x_{1,1,1} \vee x_{1,2,1} \vee x_{1,3,1} \vee x_{1,4,1}.$$

At most once:

$$\neg x_{i,j,k} \vee \neg x_{i,\ell,k} \quad \text{for } j \neq \ell.$$

Each number appears exactly once in every column.
At least once:

$$\bigvee_{i=1}^{4} x_{i,j,k} \quad \text{for all } j, k.$$

for number 2 in column 2:

$$x_{1,2,2} \vee x_{2,2,2} \vee x_{3,2,2} \vee x_{4,2,2}.$$

At most once:

$$\neg x_{i,j,k} \vee \neg x_{\ell,j,k} \quad \text{for } i \neq \ell.$$

Each number appears exactly once in every 2×2 subgrid.
For example, in the upper-left block (rows 1–2, columns 1–2):
At least once (number 1):

$$x_{1,1,1} \vee x_{1,2,1} \vee x_{2,1,1} \vee x_{2,2,1}.$$

At most once:

$$\neg x_{1,1,1} \vee \neg x_{2,2,1},$$

and similarly for all other pairs.

Encoding the Given Puzzle

The fixed entries of the puzzle are encoded as unit clauses.

From the grid:

$$(1,2) = 2 \quad \Rightarrow \quad x_{1,2,2}$$

$$(2,3) = 3 \quad \Rightarrow \quad x_{2,3,3}$$

$$(3,2) = 3 \quad \Rightarrow \quad x_{3,2,3}$$

$$(4,3) = 1 \quad \Rightarrow \quad x_{4,3,1}$$

Each of these is added as a unit clause to the CNF formula.

2.2.3 Final Formula

We will be using the encoding proposed in [Lynce and Ouaknine]. Therefore, The complete CNF encoding is:

$$\phi = \text{Clues} \Downarrow V^+ \cup (\text{Cell}_u \cup \text{Row}_u \cup \text{Col}_u \cup \text{Subgrid}_u) \Downarrow V^- \cup (\text{Cell}_d \cup \text{Row}_d \cup \text{Col}_d \cup \text{Subgrid}_d) \downarrow V^-$$

where,
V^+ = Set of variables with true assignment
V^- = Set of variables with false assignments
$\Downarrow$ eliminates clauses if they are true
$\downarrow$ eliminates literals if they are false
Note that each constraint here is treated as a set of clauses, the formula is in CNF.

3 Implementation

3.1 Computer Representation

The sudoku is be first represented in a plain text file. With the decided encoding, the problem will be converted to CNF, for which we use **DIMACS** format.

3.1.1 DIMACS Format

It is a textual representation of a formula in CNF form, where each line ending with 0 represents a clause. Each literal and its negation is represented by a positive and negative integer respectively. A DIMACS CNF file starts with a header p cnf (variables) (clauses) where (variables) and (clauses) are replaced with number of variables and clauses respectively. Any line beginning with c is a comment. The formula

$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y)$$

in DIMACS format would be

```
p cnf 3 2
1 2 3 0
-1 -2 0
```

where variables x, y, z are represented by $1, 2, 3$. All encoded Sudoku Problem is written in DIMACS and stored as a .cnf file.

3.2 Mapping variables to integers

To obtain a valid DIMACS format each triplet (i,j,k), $1 \leq i, j, k \leq n^2$ should be mapped to a positive integer. For this we used the following relation

$$V(i, j, k) = (i - 1)n^4 + (j - 1)n^2 + k$$

We also needed to get the sudoku assignments (i,j,k) from the satisfying assignments
Given a DIMACS variable number V, the inverse mapping (i, j, k) can be recovered as follows:

$$i = \left\lfloor \frac{V - 1}{n^4} \right\rfloor + 1$$

$$j = \left\lfloor \frac{(V - 1) \bmod n^4}{n^2} \right\rfloor + 1$$

$$k = \big((V - 1) \bmod n^2\big) + 1.$$

3.3 Conversion to DIMACS using python

In the mathematical formulation we use indices (i, j, k) to denote row, column, and value respectively. In the implementation, we use the equivalent notation (r, c, v) (row, column, value).Also The Sudoku grid is defined as an $n^2 \times n^2$ grid with subgrids of size $n \times n$. However, in the implementation we directly use n to denote the grid dimension.

For generating variables and decoding of assignments provided by the SAT solver, We used the relation above. And for generating clauses and literals we implemented the formulation using loops. See **Appendix A** for code.

3.4 Solution Using PySAT and Existing SAT solvers

First, we used the PySAT library in Python, which provides a unified interface to several modern SAT solvers. The generated clauses were added to a PySAT solver instance, and the solver was invoked to determine satisfiability. This was used for debugging of our conversion script. After verification, we used **Satch** as our primary solver with which further testing was done.

If the formula is satisfiable, the solver returns a model, which is a satisfying assignment of all Boolean variables. Using the inverse mapping described earlier, each positive literal corresponding to $x_{i,j,k}$ was decoded to reconstruct the completed Sudoku grid.

3.5 Implementation of CDCL

3.5.1 Overview of The Algorithm

Modern SAT solvers typically use Conflict-Driven Clause Learning (CDCL), an extension of the DPLL procedure that improves search through conflict analysis and clause learning.

The algorithm repeatedly:

1. selects a decision variable,
2. performs unit propagation,
3. analyzes conflicts to learn a new clause, and
4. backjumps non-chronologically based on the learned clause.

Clause learning enables additional unit propagations after backtracking, allowing CDCL to prune large parts of the search space. A high-level description follows [Junttila, 2020].

Algorithm 1 Conflict-Driven Clause Learning (CDCL)

Require: CNF formula ϕ
$\tau \leftarrow \emptyset$ {Partial assignment}
while true **do**
  $\tau \leftarrow$ unit-propagate(ϕ, τ)
  if τ falsifies a clause **then**
    if decision level $= 0$ **then**
      return UNSAT
    end if
    $C \leftarrow$ analyze-conflict(ϕ, τ)
    $\phi \leftarrow \phi \wedge C$ {Learned clause}
    backjump to decision level determined by C
  else
    if all variables assigned **then**
      return SAT
    end if
    start new decision level
    choose unassigned literal l
    $\tau \leftarrow \tau \cup \{l\}$ {Decision assignment}
  end if
end while

3.5.2 Implementation

Our implementation follows the standard CDCL architecture: decision making, unit propagation, conflict detection, clause learning via resolution, and non-chronological backtracking.

For guidance, we referred to the implementation of the SATCH SAT solver and adopted its overall control flow structure while keeping the design minimal.

The solver successfully handles small and medium-sized Sudoku instances, although it does not include advanced industrial optimizations such as VSIDS heuristics or clause database reduction. Find the code in **Appendix B**.

4 Comparison with Backtracking

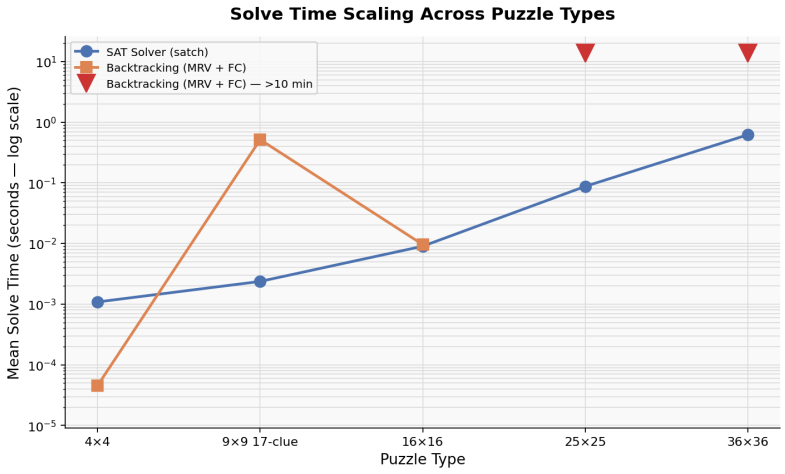


Figure 3: Performance over different puzzles

Figure 3 shows mean solve time (log scale) for a SAT solver and a backtracking solver (MRV + forward checking) across increasing Sudoku sizes. The SAT solver scales steadily, remaining under one second even for 36×36 puzzles. In contrast, backtracking performs well on very small

grids but scales poorly, becoming impractical beyond 16×16 and exceeding 10 minutes for larger puzzles. Overall, while backtracking is effective for small instances, SAT solving scales much more efficiently for larger Sudoku problems.

The test was conducted with puzzles of other sizes and difficulty, the respective graphs are at **Appendix C**

5 Conclusion

This project investigated the use of Boolean satisfiability (SAT) solving as an approach to solving Sudoku puzzles by encoding the problem as a formula in conjunctive normal form (CNF). The Sudoku constraints were translated into logical clauses, and solutions were obtained using both existing SAT solvers and a custom implementation of a Conflict-Driven Clause Learning (CDCL) solver.

Experimental results show that SAT-based solving scales more effectively than classical backtracking approaches, particularly as puzzle size increases. While backtracking performs efficiently on small grids, its performance degrades rapidly for larger instances. In contrast, SAT solving maintains stable performance due to efficient unit propagation, clause learning, and non-chronological backtracking. These mechanisms allow large portions of the search space to be pruned systematically.

The custom CDCL implementation successfully solved small and medium-sized Sudoku instances, demonstrating the core principles of modern SAT solving. However, it lacks several advanced optimizations used in industrial solvers, such as dynamic branching heuristics and clause database management, which limits performance on larger or more complex instances.

Appendix A

Encoding:

```
def encode(n, puzzle):

    box = int(math.sqrt(n))


    V_plus  = set()
    V_minus = set()

    fixed = {}
    for r in range(1, n + 1):
        for c in range(1, n + 1):
            v = puzzle[r - 1][c - 1]
            if v != 0:
                fixed[(r, c)] = v
                V_plus.add((r, c, v))

    def luc(i):
        return ((i - 1) // box) * box + 1

    for (r, c, v) in list(V_plus):
        for v2 in range(1, n + 1):
            if v2 != v:
                V_minus.add((r, c, v2))
        for c2 in range(1, n + 1):
            if c2 != c:
                V_minus.add((r, c2, v))
        for r2 in range(1, n + 1):
            if r2 != r:
                V_minus.add((r2, c, v))
        br, bc = luc(r), luc(c)
        for r2 in range(br, br + box):
            for c2 in range(bc, bc + box):
                if (r2, c2) != (r, c):
                    V_minus.add((r2, c2, v))

    V0 = set()
    for r in range(1, n + 1):
        for c in range(1, n + 1):
            for v in range(1, n + 1):
                triple = (r, c, v)
                if triple not in V_plus and triple not in V_minus:
                    V0.add(triple)

    V0_list  = sorted(V0)
    var_map  = {triple: idx + 1 for idx, triple in enumerate(V0_list)}
    num_vars = len(var_map)

    def lit(r, c, v, neg=False):

        if (r, c, v) in V_plus:
            return "FALSE" if neg else "TRUE"
        if (r, c, v) in V_minus:
            return "TRUE" if neg else "FALSE"
        idx = var_map[(r, c, v)]
```

```
    return -idx if neg else idx

clauses = []

def add_clause(literals):
    resolved = []
    for (r, c, v, neg) in literals:
        l = lit(r, c, v, neg)
        if l == "TRUE":
            return
        if l == "FALSE":
            continue
        resolved.append(l)
    if resolved:
        clauses.append(resolved)

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)])

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)])

for r in range(1, n + 1):
    for v in range(1, n + 1):
        add_clause([(r, c, v, False) for c in range(1, n + 1)])

for r in range(1, n + 1):
    for v in range(1, n + 1):
        for ci in range(1, n):
            for cj in range(ci + 1, n + 1):
                add_clause([(r, ci, v, True), (r, cj, v, True)])

for c in range(1, n + 1):
    for v in range(1, n + 1):
        add_clause([(r, c, v, False) for r in range(1, n + 1)])

for c in range(1, n + 1):
    for v in range(1, n + 1):
        for ri in range(1, n):
            for rj in range(ri + 1, n + 1):
                add_clause([(ri, c, v, True), (rj, c, v, True)])

for roffs in range(0, n, box):
    for coffs in range(0, n, box):
        for v in range(1, n + 1):
            add_clause([
                (roffs + dr, coffs + dc, v, False)
                for dr in range(1, box + 1)
                for dc in range(1, box + 1)
            ])

for roffs in range(0, n, box):
    for coffs in range(0, n, box):
```

```
            for v in range(1, n + 1):
                cells = [
                    (roffs + dr, coffs + dc)
                    for dr in range(1, box + 1)
                    for dc in range(1, box + 1)
                ]
                for i in range(len(cells)):
                    for j in range(i + 1, len(cells)):
                        r1, c1 = cells[i]
                        r2, c2 = cells[j]
                        add_clause([(r1, c1, v, True), (r2, c2, v, True)])

    return clauses, num_vars, var_map, V0_list
```

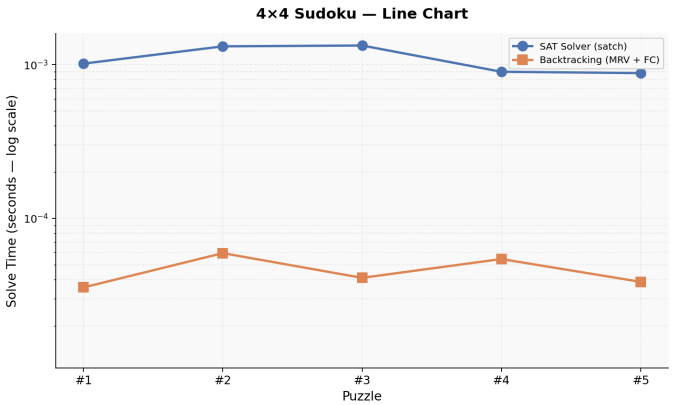
Decoding model returned by SAT solver

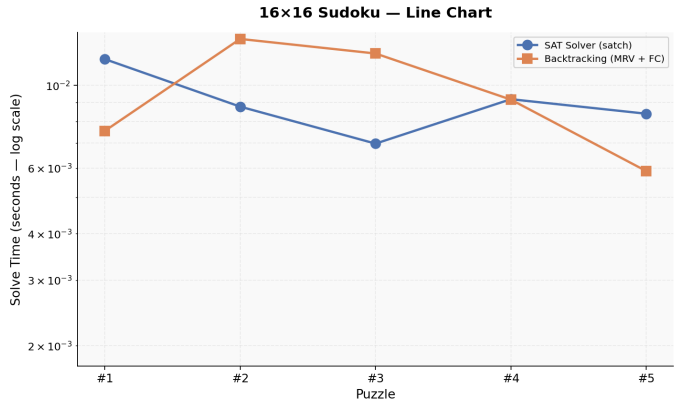
```
  def decode_solution(assignment, puzzle_path):
    sys.path.insert(0, SCRIPT_DIR)
    from sudoku_to_cnf import read_puzzle, encode

    n, puzzle    = read_puzzle(puzzle_path)
    _, _, var_map, _ = encode(n, puzzle)
    inv_map      = {idx: triple for triple, idx in var_map.items()}
    true_set     = set(assignment)

    grid = [row[:] for row in puzzle]
    for var_idx in true_set:
        if var_idx in inv_map:
            r, c, v = inv_map[var_idx]
            grid[r - 1][c - 1] = v
    return grid, n

def save_solution(grid, n, basename, sat_time, assignment):
    out = os.path.join(SOL_DIR, basename + "_SAT_solved.txt")
    with open(out, "w+") as f:
        f.write(f"SIZE {n}\nSOLVE_TIME_SEC {sat_time:.4f}\n")
        f.write("METHOD SAT\nSTATUS SOLVED\nSOLUTION\n")
        for row in grid:
            f.write(" ".join(str(v) for v in row) + "\n")
        for x in assignment:
            f.write(str(x))
    return out
```

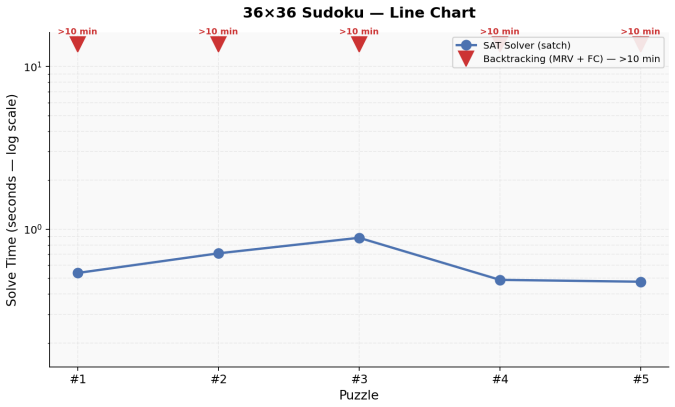
Appendix B

```
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);
    }
}
```

Appendix C


4×4 Sudoku — Line Chart
SAT Solver (satch)
Backtracking (MRV + FC)
Solve Time (seconds — log scale)
10^{-3}
10^{-4}
#1
#2
#3
#4
#5
Puzzle


16×16 Sudoku — Line Chart
SAT Solver (satch)
Backtracking (MRV + FC)
Solve Time (seconds — log scale)
10^{-2}
6×10^{-3}
4×10^{-3}
3×10^{-3}
2×10^{-3}
#1
#2
#3
#4
#5
Puzzle


36×36 Sudoku — Line Chart
>10 min
>10 min
>10 min
>10 min
>10 min
SAT Solver (satch)
Backtracking (MRV + FC) — >10 min
Solve Time (seconds — log scale)
10^{1}
10^{0}
#1
#2
#3
#4
#5
Puzzle

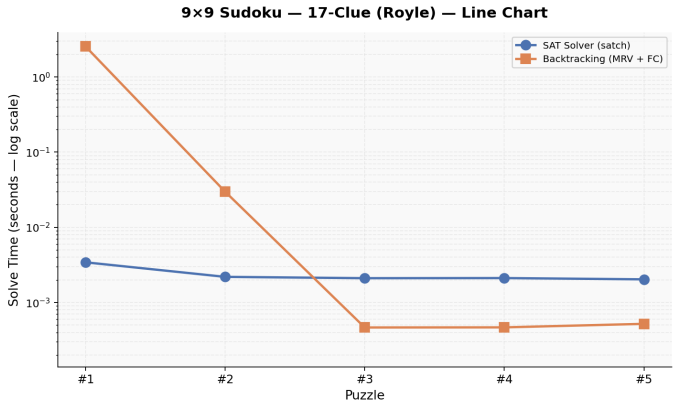


Figure 4: 17-clue puzzle

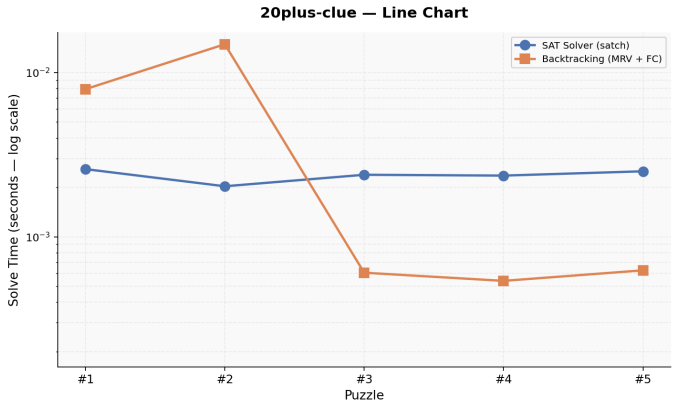


Figure 5: Performance comparison of SAT solver and backtracking solver across Sudoku sizes and clue densities.

References

Sanidway Arizona. https://sandiway.arizona.edu/sudoku/examples.html.

Charles J Colbourn. The complexity of completing partial latin squares.

Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, 2004. ISBN 978-0-521-54310-1.

Tommi Junttila. Conflict-driven clause learning (cdcl) sat solvers, 2020. URL https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html. Accessed: 2026-02-19.

Frédéric Lardeux, Éric Monfroy, Eduardo Rodriguez-Tello, Broderick Crawford, and Ricardo Soto. Solving complex problems using model transformations: from set constraint modeling to sat instance solving. *Expert Systems with Applications*, 149:113243, 2020. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2020.113243. URL https://www.sciencedirect.com/science/article/pii/S0957417420300695.

Inês Lynce and Joël Ouaknine. Sudoku as a sat problem.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010.

Wikipedia. Boolean satisfiability problem — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Boolean\%20satisfiability\%20problem&oldid=1335700096.

Takayushi Yato. Complexity and completeness of finding another solution and its application to puzzles. 2003. URL http://www-imai.is.s.u-tokyo.ac.jp/{\~{}}yato/data2/MasterThesis.pdf.