

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, Lakki Thapa

**DEPARTMENT OF MATHEMATICS
KATHMANDU UNIVERSITY
DHULIKHEL, NEPAL**

Contents

1	Introduction	3
1.1	Sudoku	3
1.1.1	Complexity of Generalized Sudoku Problem	3
1.2	The Boolean Satisfiability Problem	3
1.2.1	Complexity of SAT	3
1.2.2	SAT Solvers	3
1.3	Justification for SAT approach	3
2	Mathematical Formulation	4
2.1	Conjunctive Normal Form	4
2.2	Representation of Sudoku As Boolean Expression (CNF)	4
2.2.1	Definedness	4
2.2.2	Uniqueness	5
2.2.3	Final Formula	7
2.3	Encoding	7
3	Implementation	7
3.1	Computer Representation	7
3.2	DIMACS CNF	7
3.3	Mapping variables to integers	7
3.4	Conversion to DIMACS using python	7
3.5	Solution Using Existing SAT solvers and PySAT	7
3.6	Building a SAT solver in C	7
3.6.1	CDCL	7
3.6.2	Implementation of CDCL	7
4	Comparison with Backtracking	7

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 extended encoding strategy that introduces certain redundant clauses 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.

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 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 cues. 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 arbitrary positive integer n , we call it **Generalized Sudoku**.

1.1.1 Complexity of Generalized Sudoku Problem

The Partial Latin Square Completion problem was shown to be NP-complete [1]. Yato and Seta [5] 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. Since Our Project also consists of finding a solution, we also consider the FNP (as defined in [5]) version of the problem.

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. [4] 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 at the University of Toronto in 1971 and independently by Leonid Levin at the Russian Academy of Sciences in 1973. [4]. This project deals with finding out the assignments if they exist, which for the general case is in FNP.

1.2.2 SAT Solvers

SAT Solvers are computer programs 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.

2 Mathematical Formulation

2.1 Conjunctive Normal Form

A boolean formula is said to be in a conjunctive normal form if it is a conjunction of clauses , where each clause is a disjunction of literals.[2]

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 1: 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 (i,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 **For 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:

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

Each value appears at most once in every row:

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

Each value appears at most once in every column:

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

Each value appears at most once in every subgrid:

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

For Cues

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

$$\mathbf{Cues} = \bigwedge_{(i,j,k) \in G} x_{i,j,k}$$

Example

For given 4×4 Sudoku in Figure 1, 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

The complete CNF encoding is:

$$\Phi = Cell_d \wedge Cell_u \wedge Row_d \wedge Row_u \wedge Col_d \wedge Col_u \wedge Subgrid_d \wedge Subgrid_u \wedge Cues.$$

2.3 Encoding

For the final formula we choose the extended encoding (See [3]) where some clauses are redundant, the tests by [3] show that this is the fastest encoding for solving sudoku puzzles. Furthermore, making the formula more explicit assists CDCL due to conflicts being detected earlier. The other possible encodings which were compared in those tests are as follows:

$$\Phi = Cell_d \wedge Row_u \wedge Col_u \wedge Subgrid_u \wedge Cues.$$

$$\Phi = Cell_d \wedge Cell_u \wedge Row_u \wedge Col_u \wedge Subgrid_u \wedge Cues.$$

All three formulas are Equisatisfiable.

3 Implementation

3.1 Computer Representation

The sudoku will be first represented as a conventional array. With the decided encoding, the problem will be converted to CNF, for which we use **DIMACS** format.

3.2 DIMACS CNF

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
```

All encoded Sudoku Problem is written in DIMACS and stored as a .cnf file.

3.3 Mapping variables to integers

3.4 Conversion to DIMACS using python

3.5 Solution Using Existing SAT solvers and PySAT

3.6 Building a SAT solver in C

3.6.1 CDCL

3.6.2 Implementation of CDCL

4 Comparison with Backtracking

Graphs and stuffs go here

References

- [1] Charles J Colbourn. The complexity of completing partial latin squares. Technical report.
- [2] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, 2004.
- [3] Inês Lynce and Joël Ouaknine. Sudoku as a sat problem. Technical report.
- [4] Wikipedia. Boolean satisfiability problem — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Boolean%20satisfiability%20problem&oldid=1335700096>.
- [5] Takayushi Yato. Complexity and completeness of finding another solution and its application to puzzles. 2003.