Università
della
Svizzera
italiana

Faculty
of
Informatics

Theory of Computation

Sudoku SAT Solver

Matteo Alberici, Michele Cattaneo, Edoardo Riggio

Spring Semester, May 27, 2022

Contents

1 Introduction

The Sudoku SAT Solver project aims at developing a way of solving the popular Japanese game "Sudoku" by using the SAT solver Z3. Sudoku is a combinatorial number placement puzzle, where all numbers from 1 to 9 must be placed in every row, column, and every 3×3 boxes of a 9×9 grid. More specifically, the game rules are implemented through a Boolean satisfiability problem, represented as a conjunction of clauses.

2 Encoding the problem into the SAT domain

2.1 Definition of the variables

We defined the variables as 3-tuples with the following form:

$$v\,(r, c, n)$$

which represents a number $n \in [1 \ldots 9]$ positioned at row $r \in [0 \ldots 8]$ and column $c \in [0 \ldots 8]$ on the grid of the Sudoku.
There is a total of $9^3 = 729$ variables. When the variables get assigned exactly one truth value for each variable we get an *interpolation* I, and we want to look for an assignment such that $I \vDash F$ where F is a well formed formula representing the rules of the game. There are $2^{729} \approx 0.2 \times 10^{220}$ possible assignments.

2.2 Definition of the clauses

We must define some constraints and generate a boolean formula F in a conjunctive normal form, such that formula evaluates to true under interpolation I, if and only if the interpolation represents a valid solution to a specific Sudoku game.
First of all we must enforce each cell of the grid to have one number, otherwise a valid solution would be an empty grid. We define the following conjunction of clauses to represent the constraint:

$$\bigwedge_{r=0}^{8} \bigwedge_{c=0}^{8} \bigvee_{n=1}^{9} v\,(r, c, n)$$

Furthermore, every number $n \in [1 \ldots 9]$ must be found in every row of the grid.

$$\bigwedge_{r=0}^{8} \bigwedge_{n=1}^{9} \bigvee_{c=0}^{8} v\,(r, c, n)$$

The same holds for the columns; every column must have every number $n \in [1 \ldots 9]$.

$$\bigwedge_{c=0}^{8} \bigwedge_{n=1}^{9} \bigvee_{r=0}^{8} v\,(r, c, n)$$

Additionally, every number $n \in [1 \ldots 9]$ must be found in all the 3×3 boxs. We can obtain this constraint by sub-dividing the indexing with two indices for each direction; one for the box and one for the row/column of the box. For example $b \in [0, 1, 2]$ represents one of the horizontal boxes and $r \in [0, 1, 2]$ represents one of the three rows in a specific box. The same works for p and c for the vertical boxes and columns.

$$\bigwedge_{b=0}^{2} \bigwedge_{p=0}^{2} \bigwedge_{n=1}^{9} \bigvee_{r=0}^{2} \bigvee_{c=0}^{2} v(3b+r, 3p+c, n)$$

There is one less obvious rule left, that we are not really thinking about while playing the game as we take it for granted. Every cell must also contain *at most* one number. The first rules we defined does not exclude that a cell can have more than one number in it, and if we were to put in every cell every possible number we would solve the problem, because every row, column and 3×3 box would have all the numbers needed. This also represents a rule we often use when playing the game; a number can not be found twice in the same column, row or box. Let's say our solution put the number 3 twice in the same row. The fact that we enforce each cell to only contain one number results in the second number 3 to take the spot of another number, and therefore the constraint on all numbers being present in every row would be violated, as one of those would be missing. The same reasoning holds for the columns and boxes.

$$\bigwedge_{r=0}^{8} \bigwedge_{c=0}^{8} \bigwedge_{x=1}^{8} \bigwedge_{y=x+1}^{9} (\neg v(r,c,x) \vee \neg v(r,c,y))$$

3 Python implementation

3.1 Mappings

The sat solver will require variables to have an integer number $[1, \dots, n]$ assigned to them. For every possible variable, which represents a number n at row r and column c, the following code assigns it its integer number and creates two mappings:
The first one from a human readable string name of the variable (obtained with `var_name(r,c,n)`) to its integer number:

$$str \rightarrow int$$

The second one from the integer number to a 3-tuple (r, c, n):

$$int \rightarrow (r, c, n)$$

```python
# Generating Variables
next_var = 1
var_map = {}
var_to_vals = {}

for row in range(ROWS):
    for col in range(COLUMNS):
        for n in range(1, NUMBERS+1):
            name = var_name(row, col, n)
            var_map[name] = next_var
            var_to_vals[next_var] = (row, col, n)
            next_var += 1
```

The first mapping is used while defining the constraints, to create the clauses that contain the integer representation of the variables. The second mapping is used after obtaining the solution from the sat solver and we want to populate a matrix with the numbers that the solver found. If a variable i is set to `True` by the solver, then the mapping $i \rightarrow (r, c, n)$ tells us that the solution will have number n at position (r, c) in the grid.

3.2 Definition of the clauses

Starting with an empty list `clauses=[]`, we can create the whole formula representing the constraints of the sudoku in CNF form. A CNF formula is a conjunction of clauses and a clause is a disjunction of literals. This means that `clauses` will be a list of lists, where each element is a list of disjunction and every list is in conjunction with the others.

$$[[a, b, c], [d], [e, f, g]] \Rightarrow [\underbrace{[a \vee b \vee c]}_{clause_1} \wedge \underbrace{[d]}_{clause_2} \wedge \underbrace{[e \vee f \vee g]}_{clause_3}]$$

In the following snippets of code, `variables` represents the mapping $str \rightarrow int$ that was called `var_map` in the previous section. Each of the snippets generates clauses to be added to the list.

The following code snippet generates the clause "*every cell contains at least one number*":

```python
# Every cell contains at least one number
for row in range(ROWS):
    for col in range(COLUMNS):
        clauses.append(
            [variables[var_name(row, col, n)] for n in range(1, NUMBERS+1)]
        )
```

The following code snippet generates the clause "*every cell contains at most one number*":

```
# Every cell contains at most one number
for row in range(ROWS):
    for col in range(COLUMNS):
        for n in range(1, NUMBERS):
            for m in range(n+1, NUMBERS+1):
                clause = [-variables[var_name(row, col, n)],
                  -variables[var_name(row, col, m)]]
                clauses.append(clause)
```

The following code snippet generates the clause "*every row contains all numbers*":

```
# Every row contains all numbers
for row in range(ROWS):
    for n in range(1, NUMBERS+1):
        clauses.append(
            [variables[var_name(row, col, n)] for col in range(COLUMNS)]
        )
```

The following code snippet generates the clause "*every column contains all numbers*":

```
# Every column contains all numbers
for col in range(COLUMNS):
        for n in range(1, NUMBERS+1):
            clauses.append(
                [variables[var_name(row, col, n)] for row in range(ROWS)]
            )
```

The following code snippet generates the clause "*every square contains every number*":

```
# Every square contains every number
for square_1 in range(SQUARE_SIDE):
        for square_2 in range(SQUARE_SIDE):
            for n in range(1, NUMBERS+1):
                clause = []

                for row in range(SQUARE_SIDE):
                    for col in range(SQUARE_SIDE):
                        clause.append(
                        variables[var_name(3*square_1+row, 3*square_2+col, n)]
                        )
                clauses.append(clause)
```

4 Graphical User Interface

4.1 Definition of the game

When the game starts, what the user sees consists of an empty grid, since the problem has not been solved yet.
Figure 1 shows the sudoku grid's state when the webpage is loaded.

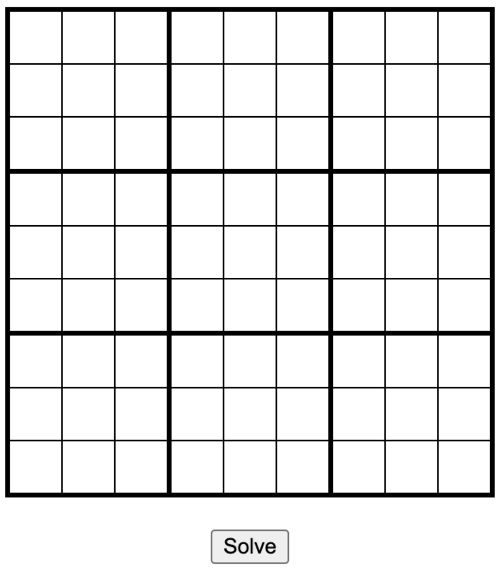
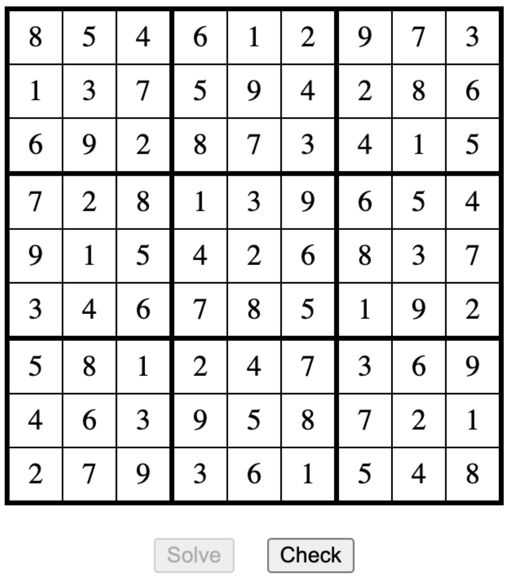


Figure 1: Not-solved sudoku

We now want to solve the Sudoku. In order to let the solver return the solution, we need to press the button *Solve*; each number will then appear after a regular amount of time.
Figure 2 shows the sudoku grid's state after every number has appeared, i.e. after the problem has been solved.



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

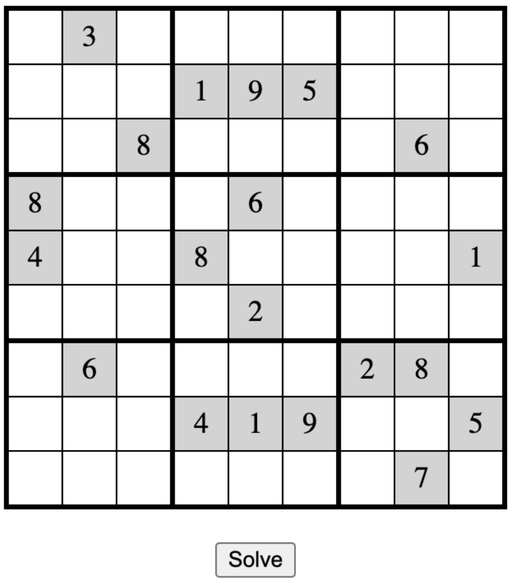
Figure 2: Solved sudoku

4.2 Using an input file

In order to manipulate the game, we implemented an additional feature which lets the user drag and drop a file onto the grid to fill some cells before the solver starts to execute. The input file must be one, more files at the same time will not be accepted. Moreover, it must be a text file, other file formats will be rejected.
Each line of the file must define a single variable using the following syntax:

$$r, c, n$$

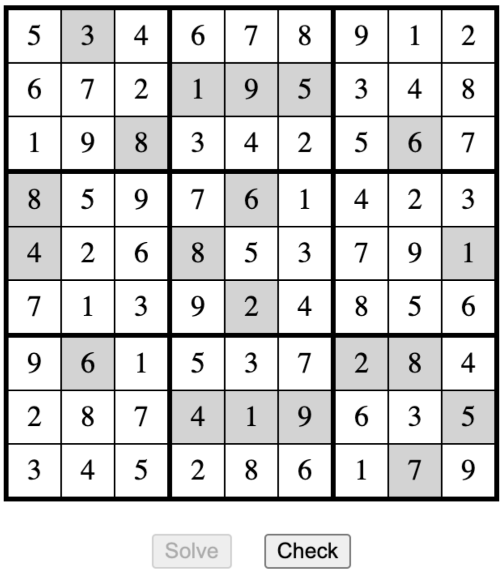
Components $n, r,$ and c are defined in section 2.1. The whole input file is then processed, letting the corresponding cells be filled with numbers on a gray background.
Figure 3 shows the sudoku grid's state after having loaded an input file and before the problem is solved.



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

Figure 3: Not-solved sudoku with an input file

By pressing the button "*Solve*" after having dropped a file into the webpage, numbers will start to fill the grid; the numbers written while processing the input file will not be modified.
Figure 2 shows the sudoku grid's state after the input file has been processed and every number has appeared, i.e. after the problem has been solved.

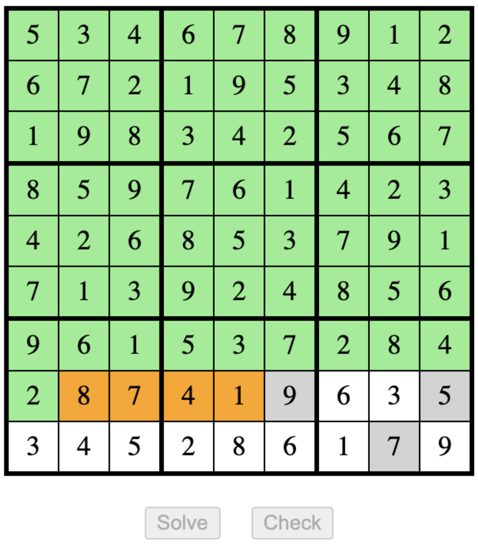


5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 4: Solved sudoku with an input file

4.3 Check the solution

After the game has been solved, a second button labelled *Check* appears right under the grid. By pressing it, the game solution will be checked in order to assert its correctness. The user can notice that the solution is being checked by an animation wherein each cell which is being checked obtains an orange background, while each already checked cell obtains a green one.
Figure 5 shows the sudoku grid's state while every cell is being checked.



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

Figure 5: Checking process executing

5 Conclusion

In this project we translated the sudoku problem in a SAT problem, by representing the various constraints and rules of the game in a boolean formula. This formula can then be fed to a SAT solver, which explores in a smart way the space of all possible assignments of variables, to then return an assignment that satisfies the formula and hence represents a solution to the sudoku game. We then created a Web application that allows users to visualise the solutions for any given sudoku problem, which can be loaded from a `.txt` file. The source code for the project can be found on its GitHub repository.
The result is a very quick sudoku solver, thanks to the power of the underlying z3 SAT solver, which produces a solution in an incurably small amount of time, considering how big the space of all possible assignments is.