PROJECT REPORT

May 11, 2023

COLORADO STATE UNIVERSITY

Richi Rodriguez

Colorado State University

CS-557 Network Verification & Synthesis

Spring 2023

1 Introduction

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

Figure 1 An example of a typical Sudoku puzzle [1].

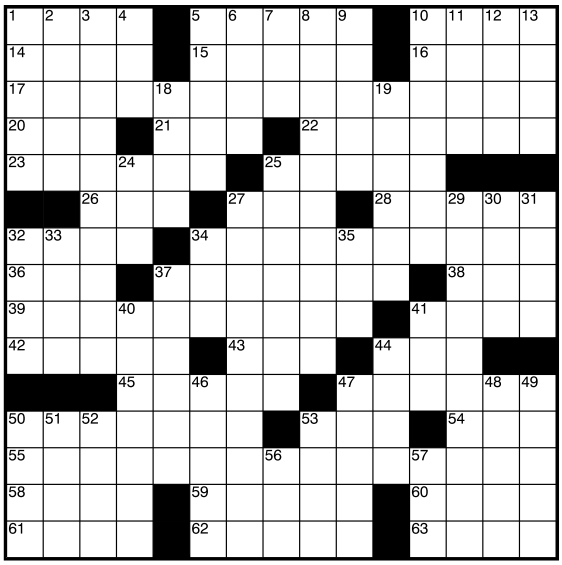

Figure 2 An example of a crossword puzzle [2].

Sudoku (see Figure 1) is a logic-based puzzle that originated in Japan. It is appealing to people because the level of difficulty can be chosen by the user. The levels of difficulty range from super easy to super difficult. Acceptance grew rapidly once it was introduced outside of Japan, partially because upon a quick look at a Sudoku puzzle, one can easily confuse it with a crossword puzzle (see Figure 2).

A typical Sudoku puzzle consists of a 9x9 grid that features the numbers 1 to 9. As shown in Figure 1, there is a main box around the 9x9 grid, but there are mini boxes of size 3x3, as seen in the same grid.

In order to win, a user must follow specific rules. The main rules are:

- The user can only use the numbers 1 to 9.
- Every mini box (grid size 3x3) can only contain the numbers 1 to 9.
- Every mini box cannot have duplicate numbers.
- Every vertical column can only contain the numbers 1 to 9.
- Every vertical column cannot have duplicate numbers.
- Every horizontal row can only contain the numbers 1 to 9.
- Every horizontal row cannot have duplicate numbers.

Failure to follow the rules will result in duplicate numbers in either a mini box, a vertical column, and/or a horizontal row, which renders the puzzle unsolvable. If the rules are

followed, then the Sudoku puzzle is solved. As noted by the rules, they seem to be very similar and straightforward, which makes this a perfect problem for SMT.

2 Examples

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

Figure 3 Another Sudoku Example, but unsolved [1].

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 Same puzzle (from Figure 2) but solved [1].

Figure 3 shows a Sudoku puzzle that could be classified with an easy difficulty. Figure 4 shows the same Sudoku puzzle but solved. As an example, let us solve the middle vertical column from Figure 3:

1) The numbers 1, 2, 6, 7, 8, 9 are given; therefore, we need to fill the empty spaces of that column with the missing numbers: 3, 4, and 5.
2) The number 3 is not a candidate in the middle empty space because it's already feature in the mini box in Figure 3, but it's a candidate in the lower and upper empty spaces (we'll keep this on hold until we can get more insights on the placement of other numbers).
3) The number 4 is not a candidate in the lower empty space because the lower-middle mini box from Figure 3 already features the number 4, and it's not a candidate for the middle empty space because it's already feature on the middle horizontal row. Since there is only one possible space left, at the top, then the number 4 can be placed there.
4) We now have two choices left: 3 and 5. Recall that the number 3 (we placed this on hold on step 2), could go on the top and lower empty spaces. Due to us assigning the number 4 at the top, it reduced the possible options of number 3 to the lower empty space. Therefore, we assign the number 3 to the lower empty space.

5) That only leaves one number left, number 5, and one empty space on that column from Figure 3; therefore, we insert the only possible number that can go into that empty space: number 5.

The rows are solved in the same exact manner. Similarly, the mini boxes are solved by checking that there are no duplicates within the mini box and that there are no duplicates when comparing the row and column that intersect with each of the cells within the mini box.

3 Proposed Approach

My proposed approach is to use the Sudoku rules mentioned in the introduction. Looking at all the rules at the same time can be stressing, but if I decompose the problems into smaller ones, then the problem is approachable as follows:

Let ψ represent all the columns, such that

$$\psi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_8 \wedge \psi_9$$

Let ϕ represent all the rows, such that

$$\phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi\_8 \wedge \phi_9$$

Let η represent all the mini boxes, such that

$$\eta = \eta_1 \wedge \eta_2 \wedge \cdots \wedge \eta_8 \wedge \eta_9$$

Having assigned the columns (ψ), the rows (ϕ), and mini boxes (η) to symbols, I can now enforce the rules presented in the introduction. To verify that, for example, that column ψ_1 holds, I would need to verify that every member in ψ_{1i} where $i \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is a mapping to a member referencing the cell in the current column and has the range. The uniqueness or distinctness should also be checked, as duplicates are a direct violation of the rules. In this case, ψ_1 holds if and only if

$$\psi_1 = \psi_{11} \wedge \psi_{12} \wedge \psi_{13} \wedge \psi_{14} \wedge \psi_{15} \wedge \psi_{16} \wedge \psi_{17} \wedge \psi_{18} \wedge \psi_{19}$$

Likewise, ψ holds if and only if $\psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5 \wedge \psi_6 \wedge \psi_7 \wedge \psi_8 \wedge \psi_9$ holds. The same holds true for ϕ (all the rows) and for η (all the mini boxes).

Following this logic, if the conjunction $\psi \wedge \phi \wedge \eta$ holds, then all the rules hold and the user has solved the Sudoku puzzle.

While I have previously completed a Sudoku solver using dynamic programing, this time I plan on using a satisfiability module theory (SMT) solver, specifically the Z3 SMT solver. This will allow me to apply the knowledge learned from class and to learn to program in the Z3 environment. The rules in Z3 are illustrated in Figure 5.

```
# Constraints every cells to the number 1 - {size}, where size is the size of the Sudoku as in sizeXsize
constraint_1 = [ And( L[row][col] >= 1, L[row][col] <= size)  for col in range(size) for row in range(size) ]

# Constraint on columns, so that each column has to be distinct
constraint_2 = []
for col in range(size):
    constraint_2.append( Distinct( [L[row][col] for row in range(size)] ) )

# Constraint on rows, so that each row has to be distinct
constraint_3 = []
for row in range(size):
    constraint_3.append( Distinct( [L[row][col] for col in range(size)] ) )

# Constraint on mini grids, so that each mini grid has to be distinct
constraint_4 = [ Distinct( get_mini_grid(L, y, x, m_size) )  for y in range(m_size) for x in range(m_size) ]
```

Figure 5 - Z3 constraints in Python that reinforce the Sudoku rules.

4 Proposed Evaluation

Originally, I had proposed two different methods to evaluate my SMT solution.

The first method would be to buy Sudoku puzzle books at a bookstore. The advantage of this method is that the Sudoku puzzles in these books have already been verified by experts, therefore the solutions can be trusted. The disadvantage is that I would need to cover the book(s) out of pocket.

The second method is to look for websites that have Sudoku puzzles, but that also provide solutions. The advantage of this method is that it will be free. The disadvantage is that some of these Sudoku puzzles and the solved solutions that are found online come from enthusiasts and the solutions cannot always be trusted.

Therefore, there are problems with the proposed evaluations, but there is an even bigger problem: the problem with the proposed evaluations is that Sudoku puzzles may contain multiple solutions. Sometimes books and/or internet enthusiasts do not include all the possible solutions because sometimes there are just too many of them.

A Sudoku book might include a puzzle such as the one pictured in Figure 6a, but with more open spaces. The puzzle in Figure 6a is almost complete, but the Sudoku book might include one or two solutions to this problem. However, as shown by Figure 6b to Figure 6e, this Sudoku puzzle can have four possible and viable solutions. If we backtracked to an earlier time, where there are more open spaces, then there is the possibility that the same Sudoku puzzle can have many more solutions. Therefore, I asked myself: is there a implementation that can verify any possible solution that a player can come up with?

The answer is yes: there is one. Since I know the rules, I can write functions to read a Sudoku puzzle that has been solved, enforcing the rules, checking that all the conjunctions

hold, and returning that status.

a)

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

b)

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

c)

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

d)

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

e)

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

Figure 6 Featuring the same Sudoku puzzle, where a) the original has eight empty spaces. The top-left mini box has two possible solutions: 1 and 4. b) and c) show what would happen if 1 if placed on the top-left mini box in the left open space, and the numbers on the top-right mini box are exchanged. d) and e) show what would happen if 4 was placed on the top-left mini box in the left open space, and the number on the top-right mini box are exchanged.

```python
def sudoku_distinct(puzzle, which='row'):
    if which not in ['col', 'row']:
        raise Exception(f'Invalid value for parameter which: "{which}"')
    grid = puzzle if isinstance(puzzle, np.ndarray) else np.array(puzzle)
    size = grid.shape[1]
    grid = grid if which == 'row' else grid.T
    distinct_row = np.arange(1, size + 1)
    condition_1 = np.logical_and(grid >= 1, grid <= size).all()
    condition_3 = np.all(np.sort(grid, axis=1, kind=None, order=None) == distinct_row)
    return np.all([condition_1, condition_3])

def sudoku_minigrids(puzzle):
    grid = puzzle if isinstance(puzzle, np.ndarray) else np.array(puzzle)
    size = grid.shape[0]
    m_size = int(size**(0.5))
    minigrids = []
    for i in range(m_size):
        for j in range(m_size):
            minigrids.append(grid[(i*m_size):(i*m_size + m_size), (j*m_size):(j*m_size + m_size)].reshape((-1,size)))
    return np.all([sudoku_distinct(_) for _ in minigrids])
```

Figure 7 - Python code for enforcing the Sudoku rules.

Since my solution is based on the Z3 Python module, I can use NumPy, which is vectorized and has logical operators that work over vectors. My solution employs its own verification as shown in Figure 7. There are two main functions that I created: **`sudoku_distinct`** and **`sudoku_minigrids`**. The function **`sudoku_distinct`** takes two parameters, `puzzle` and `which`. The parameter `puzzle` takes a solution in the form of a 2-D array. If the array is

not a NumPy array, it converts it into a NumPy array to manipulate the data efficiently. The parameter, `which`, has two choices, `row` or `col`, which either checks for the rules based on rows or columns, respectively. The line-by-line explanations are described in Table 1.

Line	Explanation
2-3	Reinforces that only `row` for rows, or `col` for columns are used.
4, 13	Either takes a NumPy array or converts a list into an NumPy array.
5	Takes the size of the grid (the latter works for a 1-D vector).
6	If checking for rows, use the grid as-is; otherwise transpose the grid.
7	Create a `distinct_row` or vector (no duplicates) in a sequence from 1 to `size`.
8	Check if all the numbers in the grid are between 1 and `size`.
9	Sorts the rows, then checks that each row/vector matches with `distinct_row`.
10	Return the result of the junction of lines 8 and 9.
14	Takes the size of the grid.
15	Calculate the size of a mini box in `m_size`.
16	Create an empty list for the mini grids in `minigrids`.
17-18	Iterate over all possible mini grids via a double for-loop.
19	Using `m_size`, extract each mini grid from the grid, and re-shape it into a 1-D vector, and add it to the `minigrids` array.
20	Iterate over the `minigrids` , on each instance calling `distinct_row` to verify that the current 1-D vector (the current mini grid) is valid. Return all these as an array, which is passed into the NumPy.all function, which checks a junction and returns the result.

Table 1 The verification code that checks if a solved puzzle is following all the rules.

The verification solution works wonderfully to verify my Z3 solution. This is better, as it can verify any solution and it's not limited by what a book or an enthusiast might provide.

Ultimately, I extended my Z3 solution to include any Sudoku puzzle size. A proper Sudoku puzzle size is a square grid, where each side is a square number, starting with 3, therefore: $3^2 \times 3^2 = 9 \times 9$, $4^2 \times 4^2 = 16 \times 16$, $5^2 \times 5^2 = 25 \times 25$, etc., are proper grid sizes.

How does my Z3 Sudoku solution fare against others? I recorded the execution times for my solution as well as three other solutions: 1) a Naive solution, 2) a Back Tracking solution, and 3) a Bit Masks solution, the latter which is an optimization on the back tracking solution, all created by Geeks for Geeks [2]. Figure 8 show the execution times for all the solutions in milliseconds.

The Naive, Back Tracking, and Bit Masks solved the 9x9 puzzle at approximately the same time, 30±1 ms, while my Z3 solution took additional time, approximately 8x, at 244ms. While this was a bit disappointing, all the 9x9 solutions solved the puzzle in less than a second. However, the tables turned when I tested all the solutions with a 16x16 puzzle. As shown in Figure 8, my Z3 solution solved the puzzle the fastest at 26109ms (26.109 seconds). The other three solutions solve the same 16x16 puzzle as follows: the Bit Masks

at 35324ms (35.324 seconds), the Naive at 35890ms (35.890 seconds), and the Back Tracking at 56982ms (56.982 seconds).

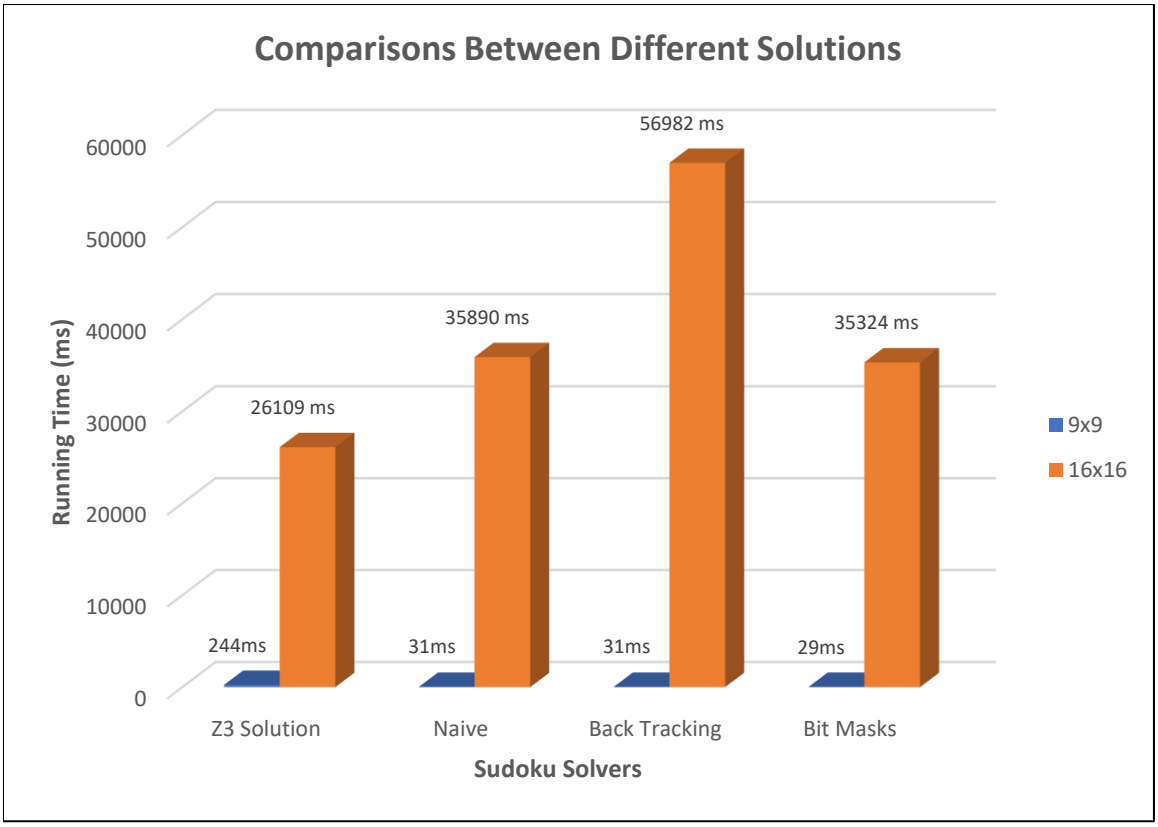

Figure 8 Execution times by several Sudoku solvers with two grid sizes: 9x9 and 16x16.

My Z3 solution outperformed the rest of the solutions at the larger puzzle size (16x16). What can I deduce from these results? The SMT might have taken a bit longer while setting up the logic in the background, hence the longer time it took in solving the smaller puzzle. However, given a larger problem, such as the larger puzzle, the logic setup pays off as shown in Figure 8.

Therefore, we can conclude that the Z3 solver is a very efficient SMT solver. To the naked eye solving a 9x9 sudoku seems to take less than a second for all solvers and thus one cannot see a notable difference between algorithms. The difference becomes clears when larger sudoku puzzle sizes are used (see Figure 8), placing the Z3 solver in the first place.

References

[1] "Wikipedia | Sudoku," 8 March 2022. [Online]. Available: https://en.wikipedia.org/wiki/Sudoku. [Accessed 4 May 2023].

[2] "Algorithm to Solve Sudoku | Sukdoku Solver," 22 March 2023. [Online]. Available: https://www.geeksforgeeks.org/sudoku-backtracking-7/. [Accessed 4 May 2023].

[3] "Wikpedia | Crossword," 23 April 2022. [Online]. Available: https://en.wikipedia.org/wiki/Crossword. [Accessed 4 May 2023].