Cross-Sum Sudoku SMT Solver

By: Trevor McDonald and Timothy Porter

CS 2800 – Logic and Computation

April 25, 2021

Abstract:

The goal of this paper is to automate the process of creating a SAT encoding for any given game of Cross-Sum Sudoku. Cross-Sum Sudoku is a combination of the games of Sudoku and Kakuro. To do this we need a process for creating propositional statements that represent the constraints of one such puzzle. This must be applicable for all variations of Cross-Sum Sudoku boards. To complete this goal of automatic encoding we used a program called Z3 for python to SAT-encode the propositional logic. We were successful in creating an encoding for these puzzles using this SMT solver; the solver can solve any Cross-Sum Sudoku puzzle given enough time, with our longest test taking around 10 seconds.

Introduction:

Cross-Sum Sudoku is a game that combines the ruleset of Sudoku and Cross-Sum Kakuro. What this means is that numbers must fit into a 9x9 grid such that each 3x3 section contains the numbers 1-9 exactly once, each row contains numbers 1-9 exactly once, and each column must contain the numbers 1-9 exactly once. Also, each row and column of each smaller 3x3 grid has an assigned sum that the three numbers which make up that column or row must add up to. Due to the nature of this variation, there are no numbers placed in the 9x9 grid at the start as with normal Sudoku; the only clues provided are the sums of each row and column.

On the page below is an example of a puzzle, initially unsolved with only the clues provided. Green cells represent the sums for each vertical 3-cell column, while orange cells represent the sums for each horizontal 3-cell row.

	19	12	14		19	9	17		17	14	14
15				9				21			
12				19				14			
18				17				10			
	6	17	22		13	15	17		16	16	13
13				19				13			
18				11				16			
14				15				16			
	20	16	9		13	21	11		12	15	18
16				14				15			
10				17				18			
19				14				12			

Figure 1: Empty Cross-Sum Sudoku problem

Below is the solved version of the above puzzle. Note how the inserted numbers both satisfy the given sums and obey the one-digit-per-row, -column, and -box rule of traditional Sudoku.

	19	12	14		19	9	17		17	14	14
15	6	2	7	9	5	3	1	21	9	8	4
12	8	1	3	19	6	4	9	14	2	5	7
18	5	9	4	17	8	2	7	10	6	1	3
	6	17	22		13	15	17		16	16	13
13	1	7	5	19	4	9	6	13	8	3	2
18	3	6	9	11	2	1	8	16	7	4	5
14	2	4	8	15	7	5	3	16	1	9	6
	20	16	9		13	21	11		12	15	18
16	7	3	6	14	1	8	5	15	4	2	9
10	4	5	1	17	9	6	2	18	3	7	8
19	9	8	2	14	3	7	4	12	5	6	1

Figure 2: Solution for the Cross-Sum Sudoku problem in Fig. 1

The lack of any starting digits means that initially, each grid space has the potential to be any of the digits 1-9 which forces the player to use different logical reasoning than regular sudoku, and often requiring more backtracking.

Approach:

When attempting to find a solution to satisfy several constraints, it makes sense to first look toward using a SAT solver, given that the Boolean operation would be too complicated to be computed by hand. However, in this instance of Cross-Sum Sudoku the arithmetic nature of some constraints makes this process more complicated. In addition to encoding the constraints into Boolean expressions, the arithmetic operations would need to be rewritten into propositional statements. This problem can be avoided using a more modern SMT solver; an SMT solver is equipped to reason around arithmetic expressions and numeric equality, making it ideal for use with arithmetic constraints such as the sums on a Cross-Sum Sudoku board.

We researched different SAT and SMT solvers that have been applied to standard Sudoku, as parts of these solutions were applicable to the Cross-Sum Sudoku problem. The goal was then narrowed to learning the syntax of a specific SMT solver in a particular language due to the variation between different solvers.

For encoding this problem, we chose Microsoft's open-source Z3 solver[1], specifically the Python implementation of the solver. The use of an SMT solver changes this from a problem of Boolean constraints to one of natural number constraints. For the purposes of this problem, each of the grid spaces on the 9x9 board (81 total) is a variable passed into the SMT solver, initially constrained from a range of 1-9 and then further constrained by the rules of the game.

The SMT solver works by attempting solutions for the given problem and testing for their validity. Each variable is adjusted in turn, one possibility being chosen to occupy that variable temporarily. If during this process the assignment of some variable causes the violation of one clause or constraint, the solver will walk back this assignment and attempt a different solution. This continues until all variables have been assigned without violating the validity of any clause, at which point a valid solution will have been created in the current assignment of the variables.

Methodology:

We began tackling this problem first by doing research on Cross-Sum Sudoku, which had little documentation as it is an obscure Sudoku variant. From the little information we did find, we reproduced a sample board to understand the game better as described previously[2]. From here we used the knowledge gained from researching standard Sudoku solvers and applied it to the Z3 SMT solver in python[3].

[1] Bjørner and Moura, "The Inner Magic behind the Z3 Theorem Prover."
[2] Yang, *Cryptic Kakuro and Cross Sums Sudoku.*
[3] Pony, "Z3 API in Python."

Encoding Cross-Sum Sudoku

To produce a Cross-Sum Sudoku solver using the Z3 SMT solver, we modeled the board as a two-dimensional array within Python, where each cell had a distinct name. Standard Sudoku boards are a 9x9 grid and can contain the integers 1-9, so we modeled this as a 9x9 array of integer variables as defined by the Z3 syntax. Each of these Integer cells were then labeled as x with its corresponding y and x positions on the board to produce the matrix as shown below. Defining the board in this form as Integers allows for the Z3 solver to solve for these integer values.

```
X = [ [ Int("x_%s_%s" % (i, j)) for j in range(9)
      for i in range(9) ]
```

x_0_0	x_0_1	x_0_2	x_0_3	x_0_4	x_0_5	x_0_6	x_0_7	x_0_8
x_1_0	x_1_1	x_1_2	x_1_3	x_1_4	x_1_5	x_1_6	x_1_7	x_1_8
x_2_0	x_2_1	x_2_2	x_2_3	x_2_4	x_2_5	x_2_6	x_2_7	x_2_8
x_3_0	x_3_1	x_3_2	x_3_3	x_3_4	x_3_5	x_3_6	x_3_7	x_3_8
x_4_0	x_4_1	x_4_2	x_4_3	x_4_4	x_4_5	x_4_6	x_4_7	x_4_8
x_5_0	x_5_1	x_5_2	x_5_3	x_5_4	x_5_5	x_5_6	x_5_7	x_5_8
x_6_0	x_6_1	x_6_2	x_6_3	x_6_4	x_6_5	x_6_6	x_6_7	x_6_8
x_7_0	x_7_1	x_7_2	x_7_3	x_7_4	x_7_5	x_7_6	x_7_7	x_7_8
x_8_0	x_8_1	x_8_2	x_8_3	x_8_4	x_8_5	x_8_6	x_8_7	x_8_8

Figure 3: Cell names of the encoded Sudoku board

Now that we encoded the sudoku board itself, we had to encode the Cross-Sums of each of the 3x3 grids, as seen in Fig. 1, into python as well. We represented these with the actual numerical values directly in two separate arrays. The first array is for the Horizontal rows of sums that represents the 3-cell vertical sums. This array is 9 numbers across and there are 3 rows, so we represented it using a 9x3 array as seen in the example below for the puzzle shown in Fig. 1. In this scenario the value at Horizontal[0][1] represents 12 = x_0_1 + x_1_1 + x_2_1.

```
Horizontal = [[19, 12, 14, 19, 9, 17, 17, 14, 14],
              [6, 17, 22, 13, 15, 17, 16, 16, 13],
              [20, 16, 9, 13, 21, 11, 12, 15, 18]]
```

Similarly, we did this for the vertical Cross-Sums, representing the numerical values in a 3x9 array as shown below. These are the Vertical rows of sums that represent the 3-cell horizontal sums. This makes the value at Vertical[1][1] in the example below represent 19 = x_1_3 + x_1_4 + x_1_5.

```
Vertical =  [[15, 9, 21],
             [12, 19, 14],
             [18, 17, 10],
             [13, 19, 13],
             [18, 11, 16],
             [14, 15, 16],
             [16, 14, 15],
             [10, 17, 18],
             [19, 14, 12]]
```

Encoding Constraints

Now that we encoded the Cross-Sum Sudoku board into the Z3 format, we must add the constraints as defined by the rules for Cross-Sum Sudoku. These constraints limit the possible values at each square, allowing the Z3 SMT solver to find a solution or return that there is no valid solution. The first constraint is that every cell in the array must be an integer value between 1 and 9, as seen below.

```
cellValue = [ And(1 <= X[i][j], X[i][j] <= 9)
              for i in range(9) for j in range(9) ]
```

The next two constraints are that each row and column have distinct values, meaning none of the values in a row or column are the same. We wrote these rules as follows, applying the built-in Z3 function Distinct over every row and column in the array using Python array parsing methods.

```
rows = [ Distinct(X[i]) for i in range(9) ]

cols = [ Distinct([ X[i][j] for i in range(9) ])
         for j in range(9) ]
```

A similar constraint must also be applied to the 9 different 3x3 grids in the Cross-Sum Sudoku array. To do this we applied the same Z3 Distinct function to each of the sub-grids individually, rather than simple columns or rows, as seen below.

```
square = [ Distinct([ X[3*i0 + i][3*j0 + j]
                      for i in range(3) for j in range(3) ])
                      for i0 in range(3) for j0 in range(3) ]
```

Now that we programmed the base rules for regular Sudoku, we must add the extra rules for this Sudoku variant. Specifically, the sums of each 3-cell row and column must equal the given Cross-Sum. We did this using the == comparator to set the Cross-Sum equal to the specific three integer values added together for every given Cross-Sum. Z3 SMT is powerful in this way over a SAT-solver in that it directly compares Integers rather than Boolean expressions alone. We did this in two separate parts, one for the Horizontal Cross-Sums and one for the Vertical Cross-Sums, both of which use a more complicated way of parsing the arrays. For example, in the Horizontal Cross-Sums, each number must be set equal to the 3 values below it on the board. This means when the next row of Cross-Sums is computed the cell rows used on the board increases by more than one. We accomplished this for both Cross-Sum arrays in comparison to the Sudoku board as follows.

```
horizSum = [ Horizontal[i][j] == X[i+(2*i)][j] + X[i+1+(2*i)][j] + X[i+2+(2*i)][j]
             for i in range(3) for j in range (9)]


vertSum = [ Vertical[i][j] == X[i][j+(2*j)] + X[i][j+1+(2*j)] + X[i][j+2+(2*j)]
            for i in range(9) for j in range (3) ]
```

SMT Solver

Now that we encoded all the constraints in the Z3 form, we add them to the solver. Then we check if the given board is satisfiable based on the given Cross-Sum values. If it is satisfiable then the 9x9 solution board is modeled, evaluated for each cell, and then printed. The model function of the Z3 SMT solver works by returning a list of all the previously defined integer variables (i.e., x_1_3) equal to an integer, such that the entire solution satisfies the constraints inputted. Then the evaluate function is run on every x integer returning an array of the solved values at each position. Finally, the array is printed in the Sudoku grid format using Z3's pp command for formatting.

Calculating Cross-Sums

As it is tedious to manually produce a valid Cross-Sum Sudoku puzzle, we made a program that takes in a Filled Sudoku puzzle as a 2-dimensional array and prints the horizontal and vertical Cross-Sum arrays for the given board. We did this using the simple arithmetic and array parsing seen below.

```
HorizontalSums = [ [ FilledSudoku[i+(2*i)][j]
                     + FilledSudoku[i+1+(2*i)][j]
                     + FilledSudoku[i+2+(2*i)][j]
                     for j in range (9)]
                     for i in range(3) ]


VerticalSums = [ [ FilledSudoku[i][j+(2*j)]
                   + FilledSudoku[i][j+1+(2*j)]
                   + FilledSudoku[i][j+2+(2*j)]
                   for j in range (3) ]
                   for i in range(9) ]
```

This made the testing process more efficient as we could input a Sudoku solution to this program to produce the Cross-Sums Sudoku puzzle. Then we could input this puzzle into the solver to see if the original solution matched the solution solved for. This includes one caveat that some Cross-Sum Sudoku puzzles have multiple solutions.

Metrics:

Our metric for success was to use Z3 SMT to produce a working Cross-Sum Sudoku solver. This entails that it solves any given Cross-Sum Sudoku puzzle, producing a filled-in Sudoku board of values if a solution exists and if there is no solution to the given puzzle the program returns that it is unsolvable. Specifically, we used the board in Fig. 2 as a test by inputting the Cross-Sums calculated from the solved board, into the CrossSumSudoku program and comparing the output to the original solution. Running

Example 1 in the Examples module produced the following Cross-Sum values for the example shown in Fig. 1 after using the FormCrossSum module to calculate these Cross-Sums.

```
Horizontal Cross Sums
[[19, 12, 14, 19, 9, 17, 17, 14, 14],
 [6, 17, 22, 13, 15, 17, 16, 16, 13],
 [20, 16, 9, 13, 21, 11, 12, 15, 18]]

Vertical Cross Sums
[[15, 9, 21],
 [12, 19, 14],
 [18, 17, 10],
 [13, 19, 13],
 [18, 11, 16],
 [14, 15, 16],
 [16, 14, 15],
 [10, 17, 18],
 [19, 14, 12]]
```

Figure 4: Cross-Sum values produced from FormCrossSum program for the Sudoku board in Fig. 2

Fig. 4 shows that the FormCrossSum program works in calculating the Cross-Sum arrays as when we inputted the solved board from Fig. 2, the arrays returned matched the expected Cross-Sums. We then inputted these Cross-Sum arrays into the CrossSumSudoku function. This produced the following solution for this Cross-Sum Sudoku puzzle.

```
Cross Sum Sudoku solution found
[[6, 2, 7, 5, 3, 1, 9, 4, 8],
 [8, 1, 3, 6, 4, 9, 2, 7, 5],
 [5, 9, 4, 8, 2, 7, 6, 3, 1],
 [2, 6, 5, 7, 9, 3, 8, 1, 4],
 [3, 7, 8, 4, 1, 6, 5, 9, 2],
 [1, 4, 9, 2, 5, 8, 3, 6, 7],
 [7, 3, 6, 1, 8, 5, 4, 2, 9],
 [4, 5, 1, 9, 6, 2, 7, 8, 3],
 [9, 8, 2, 3, 7, 4, 1, 5, 6]]

Solving time: 8.54 seconds
```

Figure 5: Solved Cross-Sum Sudoku puzzle for the Cross-Sums in Fig. _

Fig. 5 is an exact match to the expected solution for the Cross-Sum Sudoku puzzle shown in Fig. 2, proving that the Z3 SMT Cross-Sum Sudoku encoding solves puzzles correctly based on the given Cross-Sum values.

```
Board2  =  ((1,2,6,4,3,7,9,5,8),
            (8,9,5,6,2,1,4,7,3),
            (3,7,4,9,8,5,1,2,6),
            (4,5,7,1,9,3,8,6,2),
            (9,8,3,2,4,6,5,1,7),
            (6,1,2,5,7,8,3,9,4),
            (2,6,9,3,1,4,7,8,5),
            (5,4,8,7,6,9,2,3,1),
            (7,3,1,8,5,2,6,4,9))
```

```
Cross Sum Sudoku solution found
[[1, 2, 6, 4, 3, 7, 5, 9, 8],
 [8, 9, 5, 6, 2, 1, 7, 4, 3],
 [3, 7, 4, 9, 8, 5, 2, 1, 6],
 [4, 5, 7, 1, 9, 3, 6, 8, 2],
 [9, 8, 3, 2, 4, 6, 1, 5, 7],
 [6, 1, 2, 5, 7, 8, 9, 3, 4],
 [2, 6, 9, 3, 1, 4, 8, 7, 5],
 [5, 4, 8, 7, 6, 9, 3, 2, 1],
 [7, 3, 1, 8, 5, 2, 4, 6, 9]]
```

Figure 6: Example 3 board given (Left) vs solution based on calculated Cross-Sums (right)

Example 3 in the Examples module produced an interesting result shown in Fig. 6 as the board that the Cross-Sums were calculated for, was different than the solution the solver came up with. This is

because some Cross-Sum puzzles have multiple solutions and the solver happened to come up with a different solution than the given one. But both solutions when analyzed are equally valid for the specific Cross-Sums.

It is possible to add conjunction statement conditions that not every cell value is equal to its corresponding cell in a given board. This guarantees that the solution cannot have every value equal to this other completed Sudoku board even if some values are the same. Then by using the first solution produced by the solver as this completed board, the solver will produce a different result.

Our program was successful in solving Cross-Sum Sudoku puzzles in under 10 seconds for each of the four tests run in the Examples module, meaning the application of this SMT solver was both efficient and effective. One intriguing result from testing this program is that there may be multiple valid solutions for a given set of Cross-Sums, leaving room for possible future work in finding how commonly this occurs.

Summary:

Overall, this encoding of Cross-Sum Sudoku into the Z3 SMT solver was successful in solving any given puzzle, returning a valid solution or that there is no solution. This project stands as an example of how to SAT-encode a puzzle for the Python version of the Z3 SMT Solver in general, but also expounds upon and analyzes the details around this Sudoku variant that has little documentation.

This project has opened the opportunity for deeper analysis of Cross-Sum Sudoku and other questions surrounding it. Now that there is a base solver encoding for these puzzles, research may be done around questions like the pervasiveness of these puzzles that have multiple solutions. Also, what this does to the number of solvable Cross-Sum Sudoku problems in relation to regular Sudoku problems, and the patterns that cause a specific set of Cross-Sums to have multiple solutions. Finally, now that this Cross-Sum Sudoku solver exists it would be relatively easy to implement a program that uses a loop and disequality constraints being added to the solver for each solution already found, to produce a list of all the valid solutions to a given set of Cross-Sums.

Source Code:

The code for this project can be found at,
https://github.com/Trevor4811/Cross-Sums-Sudoku-Z3-SMT-Solver

References:

[1] Bjørner, Nikolaj, and Leonardo Moura. “The Inner Magic behind the Z3 Theorem Prover.” *Microsoft Research Blog* (blog), October 16, 2019. https://www.microsoft.com/en-us/research/blog/the-inner-magic-behind-the-z3-theorem-prover/.

[2] Pony, Eric. “Z3 API in Python.” Z3Py Guide. Accessed April 10, 2021. https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

[3] Yang, Xin-She. *Cryptic Kakuro and Cross Sums Sudoku*. Exposure Publishing, 2006.