INFO9015-1: SAT Solvers in Sudoku Resolution and Generation

†**Guillaume Delporte**[1]
[1] *guillaume.delporte@student.uliege.be (s191981)*

Course given by [1]**Pascal Fontaine**, [1] *pascal.fontaine@uliege.be*

CONTENTS

I. SUDOKU SOLVING

A. Constraint Formulation for Sudoku

The essence of solving a Sudoku puzzle lies in understanding and applying its inherent constraints effectively. These constraints ensure that each number appears exactly once in each row, column, and block. These constraint have to be correctly encoded into Conjunctive Normal Form for the SAT solver to be able to understand them and solve the problem.
These constraints are :

1. Basic Constraints

- Cell Constraint :
  Each cell in the Sudoku grid must contain at least one number. This is fundamental to ensure that every cell is filled.
  - SAT Encoding: The 'newlit' function is used to create a literal for each possible number in each cell. This is done by iterating over each row, column, and possible number, ensuring that at least one number can occupy each cell.

- Column Constraint :
  Each number must appear exactly once in each column.
  - SAT Encoding: The script ensures this by iterating over each column and number, then over each row, creating a literal for each combination. This guarantees that each number appears at least once in each column.

- Row Constraint :
  Each number must appear exactly once in each row.
  - SAT Encoding: Similar to the column constraint, the script iterates over each row and number, then over each column, creating a literal for each combination to ensure that each number appears at least once in each row.

- Block Constraint :
  Each number must appear exactly once in each block.
  - SAT Encoding: The script calculates the starting row and column for each block and iterates over each cell within the block. For each number, it creates a literal, ensuring that each number appears at least once in each block.

2. Extended Constraints

- Uniqueness Constraints :
  Each cell must contain at most one number, and each number must appear at most once in each row, column, and block.
  - SAT Encoding:

  – Cell Uniqueness: The script uses 'newneglit' to create negated literals for pairs of numbers within the same cell, ensuring that a cell cannot contain more than one number.

  – Row and Column Uniqueness: Similarly, for each row (or column), the script creates negated literals for each pair of columns (or rows) for the same number, ensuring each number appears at most once per row or column.

  – Block Uniqueness: The script iterates over each number within a block and creates negated literals for each pair of cells within the block, ensuring each number appears at most once per block.

- Pre-filled Cells :
  Some cells in Sudoku puzzles are pre-filled with numbers. These act as additional constraints.

II. SUDOKU UNIQUENESS CHECKER

To achieve this, I just needed to add constraints that forbids the already found solution. If the solver find's another solution, it was not unique.

III. SUDOKU GENERATION

We were asked to generate Sudoku's with unique solution only, this was made possible by the use of the previous section code. First, we will tackle how to generate sudokus with any number and then how to generate sudoku's with $< size - 1 >$ numbers

A. Classic Sudoku Generation

The Classic Generation method focuses on creating standard Sudoku puzzles. The process involves the following steps:

1. **Initial Setup:** Initialize an empty Sudoku grid of a given size.
2. **Random Solution Generation:** Place a random number in a random cell and use a SAT solver to find a complete solution for the partially filled grid.
3. **Number Removal:** Enter a loop to randomly remove numbers, ensuring after each removal that the puzzle still has a unique solution. If uniqueness is lost, the number is replaced, and a variable named 'health' is decremented. Continue until 'health' reaches zero. The variable health is inspired from Video-Games, where when one makes a mistake, he doesn't lose the game directly but either gets a penalty. This allows the algorithm to still try to improve the solution even if the one he just tried to removed led to the lost of uniqueness.
4. **Result:** The final Sudoku puzzle is uniquely solvable and has been cleared from every number that the algorithm found was possible to remove.

B. $< Size - 1 >$ Sudoku Numbers Generation

The $< size - 1 >$ Numbers Generation method creates puzzles with an additional constraint: the highest possible number (equal to the size of the grid) is not used in the final puzzle. The steps are basically the same as for the classic generation, but it involves a first step where all of the numbers equal to $< size - 1 >$ are removed after the SAT solver has found a complete solution for the partially filled grid. The solution is thus still unique and does not contain the forbidden number anymore.

IV. ADAPTATIONS FOR 16X16 AND 25X25 SUDOKU'S

Now that my solution is able to solve, generate and check for uniqueness of 4x4 and 9x9 sudoku's, let's see how I extended this principle for higher dimensions :

A. Adaptation of SAT Encoding

To achieve this, there was just a need to modify every literal writing function and solution parser to understand higher than 9 digits. This is where i would like to thanks Arthur Louis (One of last year's student) for explaining me that i could unravel the matrix into a 1D array and then simply get a unique identifier per cell of the Sudoku (as seen in Pierre Geurts course). I just needed to ensure that the number (which could go above 9 now), was of consistent length. This was achieve simply by adding zero padding before the number so that it is always of fixed length. This approach is nice because it is scalable and could be simply adapted for Sudoku's of even higher dimension.
Concerning parsing of the SAT Solver anszer, this could be done using the entire division operator with 100 to retrieve the row and column and the modulo to get the value of the cell.

B. Solving Higher Dimension Sudoku's

The algorithm managed to solve the given 16x16 Sudoku's in more or less .45 seconds each and the 25x25 ones in more or less 1.75 seconds each. This seems to be a pretty good result and because it is directly in correlation with uniqueness checking, i am hoping to be in the top 20% of the classroom!

C. Generating Higher Dimension Sudoku's

However, when talking about generation, even if the algorithm was somewhat efficient for 4x4 and 9x9's (given the complexity of the problem), the algorithm took a considerable 2 minutes and 20 seconds to generate a 16x16 Sudoku and 8. minutes to generate a 25x25 Sudoku. It is however possible to generate them.