**CS202: Mathematics for Computer Science ii**

**ASSIGNMENT-1**

**Divyansh Kankariya(200352), Vaidik Sharma( 201079)**

**Q1**

We started with a 3x3 sudoku since it is supposed to be the most basic and well-known form among the KxK sudoku format. And then we planned to generalise our code over all KxK formats of sudoku.

We learned from the internet what a CSV file is, since it was the input format. We learned that *“a CSV file or a comma-separated values file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.”* And Python provides a ‘csv.reader’ & ‘csv.writer’ functions to read and write through CSV files.

To implement this as a SAT problem, we create k\*k Boolean variables corresponding to each unit cell (k\*k possible values for each unit cell). This creates a total of k^6 Boolean variables for each sudoku.

We expect K^2 numbers as input in each pf the two given sudokus including multiple 0s in place of empty blocks. During Iteration, we assigned all numbers(except 0s) a boolean expression ‘True’.

While adding numbers instead of 0 we had to make sure about four key features of sudoku i.e-

i.→ Every square has to contain a single number.

ii.→ Each KxK box can only contain each number from 1 to K^2 once.

iii.→ Each column can only contain each number from 1 to K^2 once.

Iv -> Each row can only contain each number from 1 to K^2 once.

v.→ We also had to make sure that both sudokus S1 & S2( say), S1[i, j] ≠ S2[i, j], where i is row and j is column.

For achieving i.- We ensured that each cell has at least one number and utmost one number.

For achieving ii.- We ensured that each possible number occurs atleast once and also utmost once in each box

For achieving iii. and iv.- We ensured that each possible number occurs atleast once and also utmost once in each row/column.

For achieving v.- we ensured that the negation of -”both corresponding box have same number” is true

These were implemented by formulating these statements into clauses and adding them one by one to our sat solver (used through the pysat package).

Also, the variables corresponding to numbers already present in the sudokus were directly added as clauses to the main formula so that these numbers remain unchanged in the model solution constructed.

**TEST CASES:**

5 test cases are present in the testcases1 subfolder.

**Q2**

For the second part of the problem, we used a very popular algorithm known as ‘Backtracking’. *“Backtracking is a type of* [*depth-first search*](https://en.wikipedia.org/wiki/Depth-first_search) *algorithm that can be used to find some or all solutions to a problem, and is often used for* [*constraint satisfaction problems*](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem) *such as Sudoku or crossword puzzles, or the* [*eight queens puzzle*](https://en.wikipedia.org/wiki/Eight_queens_puzzle)*.”(* Reference: [*https://en.wikipedia.org/wiki/Backtracking*](https://en.wikipedia.org/wiki/Backtracking)*).* And the best part about Backtracking is that Instead of starting all over again each time it realizes that a solution will not be viable, it takes a step back and then continues in a different direction.

An overview of what we have done in this problem:

i.→We start with an empty grid.

ii.→Then generate a complete solution using backtracking that fills up the grid.

iii.→Then we remove numbers from the grid, while making sure the solution is still unique.

We represent empty grids with a 2D matrix of 0s, where each 0 represents an empty grid similar to the first problem.

It’s a recursive function that goes through each square and tries each number from 1 to K^2 to see if a solution can be built from it.

Now we have a solved KxK sudoku puzzle, and to generate a puzzle with unique solution we cannot randomly remove numbers since that would not lead into a unique solution. To produce a sudoku with unique solution:

i.→We remove a random non-empty square.

ii.→Then solve the new grid with backtracking, but count the solutions and make sure there is only one unique solution.

iii.→If there is only one solution, then continue on and remove another empty square and repeat the process, or if there is more than one solution, put the number back in the grid and either try again removing more squares, or stop and keep the generated puzzle.

The generated puzzle is our solution, we then repeat the above process to generate another puzzle and make sure that they both hold the mutual non-equivalence condition.

**LIMITATIONS**

The code uses Backtracking which consumes a lot of time, as iterating through one of every K^2 block with number choices from 1 to K increases time complexity. Any rational human might try to fill in obvious squares that only have one option and then fill in other squares that have fewer options, while this algorithm systematically goes down the rows and columns to examine the possibilities for each square.

--------------------------------------------------------

*All References used:*

[*https://en.wikipedia.org/wiki/Backtracking*](https://en.wikipedia.org/wiki/Backtracking)

[*https://www.guru99.com/python-csv.html*](https://www.guru99.com/python-csv.html)

*https://lvngd.com/blog/tags/?tag=python*

[*https://pysathq.github.io/docs/html/api/solvers.html*](https://pysathq.github.io/docs/html/api/solvers.html)