Assignment-1

Solving and Generating Sudoku Pair using SAT

Group-59

Suket Raj (201013), Sweta Kumari (201036)

31/01/2022

CS202A

Problem Statement

In this question, you have to write a k-sudoku puzzle pair solver and generator by encoding the problem to propositional logic and solving it via a SAT solver https://pysathq.github.io/.

Given a sudoku puzzle pair S1, S2 (both of dimension k) as input, your job is to write a program to fill the empty cells of both sudokus such that it satisfies the following constraints,
Individual sudoku properties should hold. For each cell S1[i, j] ≠S2[i, j], where i is row and j is column.

Input: Parameter k, single CSV file containing two sudokus. The first k*k rows are for the first sudoku and the rest are for the second sudoku. Each row has k*k cells. Each cell contains a number from 1 to k*k. Cell with 0 specifies an empty cell.
output: If the sudoku puzzle pair doesn't have any solution, you should return None otherwise return the filled sudoku pair.

In the second part, you have to write a k-sudoku puzzle pair generator. The puzzle pair must be maximal (have the largest number of holes possible) and must have a unique solution.
Input: Parameter **k**
output: CSV file containing two sudokus in the format mentioned in Q1.

Answer:

- **Input:**
  - k, Two k x k sudokus

Encoding:-

- **Variables:**
  - $Sudoku_{axyp}$: The number $(p+1)$ is located at y^{th} column and x^{th} row of $(a+1)^{th}$ sudoku.
    - $0 \leq a \leq 1$
    - $0 \leq x < k^2$
    - $0 \leq y < k^2$
    - $0 \leq p < k^2$

- **Constraints:**
  Assuming $n = k^2$
  - **Each cell c i.e. $\forall sudoku_{axy}$ in both sudokus have exactly one number.**

$$\forall_c(sudoku_{axy0} \vee sudoku_{axy1} \vee sudoku_{axy2} ..... \vee sudoku_{axy(n-1)})$$
and
$$\forall_c(\neg sudoku_{axyi} \vee \neg sudoku_{axyj})$$

  - **Each row r in both sudokus have each number p exactly once.**

$$\forall_r(sudoku_{ax0p} \vee sudoku_{ax1p} \vee sudoku_{ax2p} ..... \vee sudoku_{ax(n-1)p})$$
and
$$\forall_r(\neg sudoku_{axip} \vee \neg sudoku_{axjp})$$

  - **Each column col in both sudokus have each number p exactly once.**

$$\forall_{col}(sudoku_{a0yp} \vee sudoku_{a1yp} \vee sudoku_{a2yp} ..... \vee sudoku_{a(n-1)yp})$$
and
$$\forall_{col}(\neg sudoku_{aiyp} \vee \neg sudoku_{ajyp})$$

  - **Each box box in both sudokus have each number p exactly once.**

$$\forall_{box}(sudoku_{aijp} \vee ..... \vee sudoku_{a(i+k-1)(j+k-1)p})$$
and
$$\forall_{box}(\neg sudoku_{ai_1j_1p} \vee \neg sudoku_{ai_2j_2p})$$

  - **For each row x and column y and number p the number filled in both sudokus should be different.**

$$\forall_{x,y}(\neg sudoku_{0xyp} \vee \neg sudoku_{1xyp})$$

- **Hashing:**
  Assuming $n = k^2$
  - pySAT SAT solver takes positive or negative integers as arguments. Positive integer represent that the particular value is True and vice-versa. for which hashing from indices ($axyp$) to positive integers is needed to be done.
  - Hashing needs to be done Bijective and such that extra variables do not appear.
  - We propose polynomial hashing:

$$\boxed{\text{hashFunction} = an^3 + xn^2 + yn + p + 1}$$
$$0 \leq a \leq 1$$
$$0 \leq x < n$$
$$0 \leq y < n$$
$$0 \leq p < n$$

- Clearly the least value of hash is 1 at (0000) and highest value is $2n^3$ at $(2(n-1)(n-1)(n-1))$ as the function is increasing for all a, x, y, p.
- Now as the sizes of domain and co-domain are same, we propose inverse-hash to show that the function is bijective
- **Let** $hash = h$
  Note* $\%$ is modulo operator
  * $a = \lfloor \frac{h}{n^3} \rfloor$
  * $x = \lfloor \frac{h\%n^3}{n^2} \rfloor$
  * $y = \lfloor \frac{(h\%n^3)\%n^2}{n} \rfloor$
  * $p = ((h\%n^3)\%n^2)\%n$
- **Hence, the hash function is bijective.**

- All pre-filled cells sudoku should be taken as assumptions(not to be changed again) while solving sudoku by the SAT Solver.

Approach for part-(i)

- After getting the clauses from the constraints, and using assumption from the pre-filled values. We can use enum_models provided with the SAT solver to get a **satisfying** list of hash values for each variable.
- For every cell and value, we can again get hash to the value and check if the satisfying value is positive or negative (True or False)

Approach for part-(ii)

- First, we have to get a randomly filled sudoku pair
- Iterate through the elements of the sudokus in a randomised manner and remove the element if removing it still gives unique sudoku.
- Finally we get a partially filled local optimal sudoku in which no other value can be removed such that it gives unique solution.

Proof of local optimality in part-(ii)

- First we prepare a list of hashes of each cell in both sudokus.
- Then we randomly shuffle the list.
- We iterate the sudokus by un-hashing the elements of the list.
- If we can remove the element and still get unique solution, we remove it. Otherwise:

  - If we can't remove the element now, after removing some any elements, the scope of removing current element cannot increase, so we cannot remove it later.

- Hence, we achieve a local optima where no other value can be removed such that it gives unique solution.

Getting a randomly filled sudoku for part-(ii)

- We start with a empty sudoku
- We randomly pick a row, a shift and a randomly shuffled list of all the numbers from 1 to k^2.
- Fill the picked row of first sudoku with the shuffled list and fill the same row in the second sudoku with the cyclic shift of the randomly selected shift of the shuffled list.
- Hereby, a randomly seeded partially filled sudoku is ready.
- Use the method in part-(i) of the question to get the randomly fully filled sudoku.

Repetition of logic in various clauses:

- To make the runtime of the program less, and making it easy for the SAT Solver to rule out the wrong combinations, redundant logic is used. For example, after checking that every cell has exactly one value, even if we don't check for atmost one case for rows or columns, the logic is correct. But practically, the SAT solver will take more time to solve it.

Limitations:

- The module on which the randomization of the program depends, is *random* module of python. It uses a **pseudo-random generator**. So the numbers generated are not truly random numbers. So the claim of randomly generated sudokus is not entirely true or proven.
- The runtime analysis of the code could not be proven.

Requirements:

- *python* or *python3*
- pySAT **https://pysathq.github.io/**
- Modules used : {time, random, csv, math, argparse, pysat}