CS202 Assignment-1 Report

Assumptions-

For the part, we have assumed that Python-sat solves correctly for the clauses which we give as input.

In 1st part, we have assumed that input k is given in accordance with the dimensions of sudoku in csv file.

Implementation -

Sudoku-pair-solver (using PYSAT)

Firstly, our program takes k i.e. sudoku parameter as input. Its value is stored in a variable named a. Variable n stores the value of overall width/height of one sudoku (= a*a).

Function defined on top - val(num,row,col,dig) :

This takes 4 inputs, num (denoting whether the element belongs to sudoku 1 or sudoku 2), row (denoting row number of the cell), col (denoting column number of the cell), dig (denoting the digit).
We're having n*n*n elements in each sudoku. So, this function assigns a number to each combination of (num,row,col,dig) with first n*n*n values indicating information about first sudoku and next n*n*n about the second one.

We then read a csv file, which consists of the sudoku pair, one placed below the other and store the values in it in the form of a list of lists named **df**. Then we take an empty list with name **x,** which will be used to add the clauses (list of clauses).

As we know SAT solver stores propositions in terms of numbers. So our sudoku will have propositions starting from 1 and going till (k^6)*2 for 2 sudokus.

Since our sudokus may be non-empty, initially we use two for (one for each sudoku) loops for adding the elements corresponding to non-zero values to x. They are added to ensure that these don't change in the final sudoku solution.

There was another condition, that the corresponding cells (with the same row and column) can't hold the same digit. So, the next for loop is used to add the given cells in x using the val function, and since we need negation of them, we multiply the value with -1 (negative values imply **not**), a convention which holds further in this code as well.

Now we traverse each cell and ensure there is only one digit assigned to it. This was done by satisfying the following two cases together - at most one **and** at least one assigned (at most also uses nested loop), then adding the corresponding clauses in x.

Next for loop helps in implementing further properties of a sudoku i.e. each row and column has only one occurrence of every digit, and every smaller square of dimension k*k has all numbers unique and between 1 and k inclusive. The clauses are added in x.

In the end, we add the lists present in x to a SAT solving object named sol, which evaluates the formula. If it is unsatisfiable, then we return NONE, else we print the sudoku pair. For doing so, we extract the valuation of each clause (get the model) by using library function get_model() and name it ans.

'ans' is a list consisting of the interpretation of the formula. Hence, we check the true literal, i.e. which digit is assigned to a cell, and print that value. The same is done for every cell in both sudokus.

Sudoku-pair generator(using Pysat)

K is given as an input to the program.
Then we will generate an empty pair of sudoku to get started.
We need to give 3 qualities to our sudoku pair- Randomness, Unique soln and maximal number of holes.

As we know SAT solver stores propositions in terms of numbers. So our sudoku will have propositions starting from 1 and going till (k^6)*2 for 2 sudokus.

We will make a list of all these propositions and name them ‘availables’.
So at first we will randomly select a proposition and make it true (say 1 which means I filled the first box of sudoku with digit 1). Then some of the propositions cannot be true (like the same row cannot have digit 1). So we will remove all those propositions from the list and again select a proposition randomly and make it true .Keeping in mind that sudoku should be random we will repeat these steps for k^2 times.

Then we will solve this sudoku using part 1.
We will get a solution now. We want to hold only this solution and also maximize the holes.

Now we will make a list of all the filled propositions from the solved sudoku and name it ‘filled’.
Then we will shuffle the filled list randomly so that for the same solved puzzle we can get a different output puzzle every time.

Then we will randomly take any proposition from this list and check if there exists any other solution after removing that proposition(removing a digit from the sudoku).
To check this, we will append the negation of solved sudoku as a clause so that we cannot get that very solution again. So if we get Satisfiable then the sudoku after removing this proposition has multiple solutions otherwise unique.

So if we empty a box and still get a unique solution then we can empty that box otherwise refill that box with the removed digit.

We will repeat these steps for all the boxes and then we will be left with some propositions in the list ‘filled’ which is our desired output.

Now we will transform this to an easily readable format.

Limitations

In part 2, For K>4, the program is not being able to run on my local system as the terminal gets killed.