DS2020 Lab 4

Sudoku-solver-using-Pycosat

Roll No: 112301030

Running the code:

To run the program,

1. Move the input file into the same directory as that of the code.
2. Change the argument of `open()` in **line 67** of **solver.py** to change the input file from **p.txt**.
3. Execute `python3 solver.py`.
4. The output will be written into **output.txt**.

Approach to solving the puzzle

The code of the solver is distributed across 2 files - **sudoku.py** and **solver.py**. **sudoku.py** contains a class description of the Sudoku class with attributes `puzzle` (A **2×2** grid representation of the puzzle with strings) and `variable_constraint` (Initial setup constraint - which is unique for each puzzle)

solver.py contains the main code which constructs the other 5 constraints, initialises the Sudoku class and calls the `solve()` function in **pycosat**.

Since `pycosat.solve()` accepts a 2D integer list as argument the following convention is used to specify whether a number is present in a cell or not:

If a cell in row number *r* and column number *c* contains a number *x* then we represent it using the value **100×*r* + 10×*c* + *x***

If a cell does not contain *x* we represent it using the negative of the value.

Constraint 1 - Each cell in puzzle contains at least one value

For enforcing constraint 1 we loop through all the cells and for every number from 1 to 9 we add the condition that the cell contains the number. **constraint1** will be a 2D list where each list corresponds to a cell.

```
constraint1 = list()
for row in range(1, 10):
    for column in range(1, 10):
        for number in range(1, 10):
            constructor.append(100*row+10*column+number)
        constraint1.append(constructor)
        constructor = []
```

Constraint 2 - Each cell in the puzzle contains at most one value

For constraint 2 we loop through each cell and then for each cell we select a value to assign to the cell and inorder to enforce the atmost one value constraint add negative statements for the corresponding cell and every value for the cell other than the selected value.

Here the disjunctions belonging to the CNF statement corresponds to all combinations where the selected value is the only value assigned to a particular cell.

```
constraint2 = list()
for row in range(1, 10):
    for column in range(1, 10):
        for selected_number in range(1, 10):
            for number in range(1, 10):
                if number == selected_number:
                    constructor.append(100*row+10*column+number)
                else:
                    constructor.append(-(100*row+10*column+number))
            constraint2.append(constructor)
            constructor = []
```

Constraint 3 - Each row in the puzzle should contain all the values

For constraint 3, we loop through each row and inside it loop through each number in the interval [1, 9] and for each cell in the row insert literals which indicate that the value is assigned to the cell.

Here the disjunctions belonging to the CNF satisfies in cases where the particular value exists in each cell in the row

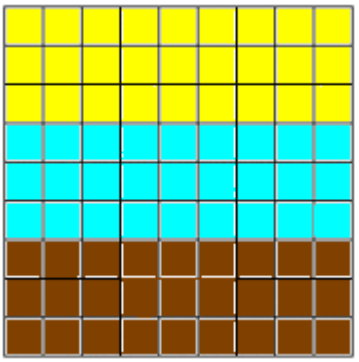
```
constraint3 = list()
for row in range(1, 10):
    for number in range(1, 10):
        for column in range(1, 10):
            constructor.append(100*row+10*column+number)
        constraint3.append(constructor)
        constructor = []
```

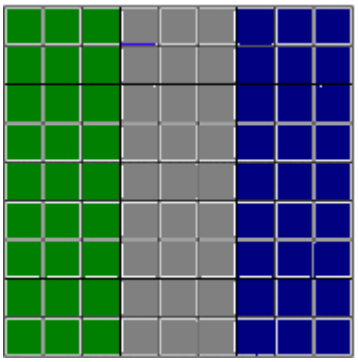
Constraint 4 - Each column in the puzzle should contain all the values

Similar to constraint 3, we just switch the rows and columns in the code to iterate along each column.

```
constraint4 = list()
for column in range(1, 10):
    for number in range(1, 10):
        for row in range(1, 10):
            constructor.append(100*row+10*column+number)
        constraint4.append(constructor)
        constructor = []
```

Constraint 5 - Each smaller block should contain all the values





The rows and columns of the whole grid are divided into groups of three. Taking the intersection of a row group and column group gives us a **3×3** grid. We have to ensure each grid contains all the values from 1 to 9.

For enforcing this we iterate through each row group and column group to make the grids. Inside the grid, we loop through each number from 1 to 9 and for each cell in the grid, we make a disjunction containing literals which correspond to each cell having the current number in the loop. However this is not enough, we need to add statements to ensure that values are not repeated in a grid. For doing this we use the **De Morgan's Law NOT**(P **AND** Q) = **NOT** (P) **AND NOT** (Q) ie, if the condition **111** satiesfies then **112** should not satisfy Therefore we apply this law over all pairs inside the `constructor` list.

```
constraint5 = list()
for row_group in range(0, 3):
    for column_group in range(0, 3):
        for number in range(1, 10):
            for row in range(1 + 3*row_group, 4 + 3*row_group):
                for column in range(1 + 3*column_group, 4 + 3*column_gro
                    constructor += [100*row+10*column+number]
            constraint5.append(constructor)

            for i in range(len(constructor)):
                    for j in range(i + 1, len(constructor)):
                        constraint5.append([-constructor[i], -constructo

            constructor = []
```

Constraint 6 - The initial setup (values for some of the cells)

The initial constraints are the only variable constraints for solving **Sudoku**, so we store all the other constraints in a list to improve computational efficiency. To set up the initial constraints we iterate through each cell in the puzzle and if it has a value we add the value corresponding to the cell having the particular value.

Since constraint 6 is different for each puzzle we implement it as a method belonging to the Sudoku class.

```
def constraint6(self):
        constraint6 = list()
        for row in range(9):
            for column in range(9):
                if self.puzzle[row][column] != '.':
                    constraint6.append([100*(row+1)+10*(column+1)+int(se
        return constraint6
```