CS202 - Assignment 1

Group members:
Arnav Gupta (200186)
Hitesh Anand (200449)

Q1

We are given k, the dimension of the input sudoku, and 2 partially filled sudokus in the form of a single .csv file. Now, we wish to find a solution for both of them such that the individual sudoku constraints hold, and additionally no 2 corresponding positions of the sudokus have the same numbers.

Approach

We have used the PySAT toolkit to solve this problem.

- We have taken k^6 boolean variables for each sudoku (i.e. $2 \times k^6$ variables for both sudokus) to encode the problem. This is because the sudoku problem can be encoded using the 4 variables that define any cell - the value of k, its row, its column, and the value that it contains.
- This gives $k^2 \times k^2 \times k^2 = k^6$ possible literals. More information on the encoding is given in the "implementation" section.
- Next, we have added the usual constraints of the sudoku and the additional constraint of $S1[i,\ j] \neq S2[i,\ j]$, where $S1$ and $S2$ are the sudokus and $[i, j]$ is the position of the cell.
- Since the sudoku is partially filled, we have also added the constraints over all the filled cells.
- Next, we ran the SAT solver and the solution, if found, was converted to a suitable matrix form and displayed. If the solution does not exist for given input by the user(for ex, when S1[i,j] = S2[i,j] for some values of i,j), then the output displays that "This pair of sudokus can't be solved!".
- In addition to the output displayed, the output is also being stored in "solvedsudoku.csv" for better visualisation.

Implementation

The constraints of the problem are listed below:

1. Each cell contains exactly one number.
2. Each row contains all the numbers in $[1, k^2]$ exactly once.
3. Each column contains all the numbers in $[1, k^2]$ exactly once.
4. Each $k \times k$ block contains all the numbers in $[1, k^2]$ exactly once.
5. $S1[i,\ j] \neq S2[i,\ j]$, where $S1$ and $S2$ are the sudokus and $[i, j]$ spans over all cells.
6. For each cell in the input which doesn't contain 0, we have to add the encoding of the cell as a constraint.

All these constraints must be satisfied at once, so they would be given in the CNF to the SAT solver.

As specified earlier, we have taken k^6 variables for each sudoku: in $[1, k^6]$ for the first sudoku and in [k^6+1, 2k^6] for the second sudoku.

To model the situation wherein the y^{th} sudoku (in 0 indexing), the cell in row i and column j has the number n, we encode it as $x = y \times k^6 + i \times k^4 + j \times k^2 + n$.
To perform the reverse process, i.e. getting the 4 inputs from the literal, we apply the following algorithm:

1. Take the integral part of x/k^6. This gives the sudoku to which the literal corresponds to. (0 for first sudoku and 1 for second sudoku)
2. Now, perform $x := x\%k^6$(% denotes remainder) and,
    a. If x is divisible by k^4, then the value of row is given by row = (x/k^4)-1 since we are using the cell indices in 0-based indexing.
    For example, if x = 32 and k = 2, then the value of row = 1(and not 2) since we are using the cell indices in the range [0, k^2-1] and the entries are in the range [1, k^2].
    b. Else, row = x / k^4.
3. Now,
    a. If value of x after removing the contribution of row in the encoding(= row*k^4) is a multiple of k^2, then the value of column will be given by col = (x-row*k^4)/k^2-1.
    For ex, if x-row*k^4 = 16 and k = 2, then the value of col = 3(and not 4) since the rest of the contribution will come from the number placed in the cell(4 in this case) and col=4 is not possible in 0-based indexing. This is because we are using the cell indices in the range [0, k^2-1] and the entries are in the range [1, k^2].
    b. Else, col = (x % k^4)/k^2.
4. Now that we have the values of row and col used in the encoding, we can remove their individual contributions from the encoding value to get the value of the cell entry.

Now, we use a SAT solver. Since the solver accepts constraints in the form of CNF(Conjunctive Normal Form), we add the clauses mentioned at the start of the section.

As an illustration, the implementation of the constraint “Each row contains all the numbers in $[1, k^2]$ exactly once” is explained below for one sudoku :

1. We iterate over all the rows in the outermost loop. Since we have added the clause which assures that each cell contains one number, it is sufficient to check that each number is contained in each row.
2. In the second loop, we iterate over the values from 1 to k^2. We create a new clause for this value.
3. Now in the third loop, we iterate over all the columns in the row and add the encoding of the current literal to the clause. This guarantees that there must be **at least** one cell in the row that contains that value. Since we already assured that each cell contains 1 value, The clause guarantees there is exactly 1 of each value in each row.

The pseudocode for the above process is written below (for the first sudoku):

```
for row in [1,k**2]:
  for val in [1,k**2]:
    declare_new_clause(clause)
    for col in [1,k**2]:
      clause.append(encode(1,row,col,val))
    Solver.add_clause(clause)
```

The pseudocode to ensure exactly one value in a sudoku cell is also given below:

```
for row in [1,k**2]:
  for col in [1,k**2]:
    declare_new_clause(clause)
    for val in [1,k**2]:
      clause.append(encode(1, row, col, val))//atleast 1 value in each cell
    Solver.add_clause(clause)
    declare_new_clause(clause)
    for val1 in [1,k**2]:
      for val2 in [1,k**2]:
        clause.append(-encode(1,row,col,val1),-encode(1,row,col,val2))
    Solver.add_clause(clause)   //less than 2 values in the cell
```

Similarly, all the sudoku constraints have been added.
The additional condition of $S1[i, j] \neq S2[i, j]$ is implemented as follows :

```
for row in [1,k**2]:
  for col in [1,k**2]:
    for val in [1,k**2]:
      Solver.add_clause([-encode(1,row,col,val),-encode(2,row,col,val)])
```

This assures that at each position of both sudokus, we can’t have the same values.

Assumptions

We have assumed that the sudoku is filled in the .csv files as $k^2 \times k^2$ rows and columns.
The path provided to the .csv file should be legitimate, otherwise the program will give an error.
Also, only one number is present in each cell, and the only integers present in the cells lie in the range $[0, k^2]$. A 0 in a cell means that the cell is empty.
If the above conditions are not satisfied in the input, the program may give haywire results.

Limitations

Since SAT solving is an NP-hard problem, and the number of literals in the sudoku grows very fast (k^4(positions)$\times$ k^2(values for each position), the number of interpretations grows exponentially resulting in higher execution time for relatively higher values of k.

Q2

We are given k, the dimension of the input sudoku. Now, we wish to generate 2 partially-filled sudoku puzzles such that they are maximal i.e. they have a unique solution, and also on removing any one value from the sudokus, the uniqueness of the solution is not retained.

Approach

We have used the PySAT toolkit to solve this problem. The encoding used is the same as that used in Q1. We have adopted the following approach towards this problem:

We defined a function **generatePairLinear(k, out1, out2)** which performs the following functions:

- We first generate two empty matrices of dimensions(k^2 x k^2).
- We generate one random cell index(say (rr,rc)) and a random number(num) in [1, k^2]. Then, we put num at the cell located at (rr, rc) in the sudoku.
- We do the above step for the second sudoku as well while ensuring that it does not generate the same rr/rc/num as the first sudoku.
- We then remove random entries from both the sudokus one by one. The number of entries removed is determined by the variable l which runs from k to $2k^4$ - 2..
- In each iteration, we remove a random cell entry from one of the sudokus in an alternate manner. Suppose for l = l1, there were total n empty cells, and the previous entry was removed from the first sudoku, then for l = l1 + 1, a random cell entry will be removed from the second sudoku leading to n+1 empty entries in total.
- In each iteration, we solve the sudokus generated after removing another entry, then we perform the same process(removing entries and solving the pair) while adding another constraint that this sudoku cannot be similar to the previous sudoku generated.
- If we get a solution other than the one generated the first time after removing entries, then it means that we have got multiple solutions, and we break out of the loop.
- Else, we continue this process as long as we keep getting unique solutions. This is done to ensure that we get a maximal solution.

Implementation

We have infused randomness in the sudoku pairs generated at 2 stages :

1. The generation of the sudoku solutions. This is randomized by setting a value in some cell of the first $k \times k$ block.
2. Now that the "solution" of the sudoku pairs has been fixed, we randomly pick the cells from the sudokus which are to be removed. The number of these cells is varied until we get to a point where removing any more values will give multiple solutions. Then, we finish our program and store the sudoku puzzles in a csv file named "puzzlepair.csv".

Additionally, we have used the 'random' library in python

Assumptions

The only assumption is that the input k is an integer.

Limitations

Since SAT solving is an NP-hard problem, and the number of literals in the sudoku grows very fast (k^4(positions)$\times$ k^2(values for each position), the number of interpretations grows exponentially resulting in higher execution time for relatively higher values of k.