SAT Encoding for Sudoku Problem:

SAT problem is represented by propositional variables :
#representing each possible value in a cell with unique number for sudoku1
v1(i,j,d) : (pow(p,4))*i + p*p*j +d

#representing each possible value in a cell with unique number for sudoku2
v2(i,j,d) : ((pow(p,4))*i + p*p*j +d+ ((pow(p,6)) + pow(p,4) + p*p))

Variables in ith row, jth column and value d which can be assigned as truth value. SAT clauses are written in CNF form. Clauses are appended as list which can be sent to SAT solver to check its satisfiability.

Clauses in Sudoku Pair are the following constraints :

Constraint	Formula
Number Constraint	$\wedge_{i=1}^{p^2}(\wedge_{j=1}^{p^2}(\vee_{d=1}^{p^2}v(i,j,d)))$
Row Constraint	$\wedge_{i=1}^{p^2}(\wedge_{j=1}^{p^2}(\vee_{d=1}^{p^2}v(i,j,d)))$
Column Constraint	$\wedge_{i=1}^{p^2}(\wedge_{j=1}^{p^2}(\vee_{d=1}^{p^2}v(i,j,d)))$
Duplicate Constraint	$\wedge_{i=1}^{p^2}(\wedge_{j=1}^{p^2}(\vee_{d=1}^{p^2}(\wedge_{d1=d}^{p^2}v(i,j,d) \rightarrow \neg v(i,j,d1)))$
Sub-grid Constraint	$\wedge_{i=1}^{p^2}(\vee_{j=1}^{p^2}(\vee_{d=1}^{p^2}v(i,j,d)))$
Pair Constraint	$\wedge_{i=1}^{p^2}(\wedge_{j=1}^{p^2}(\vee_{d=1}^{p^2}(\wedge_{d=1}^{p^2}v(i,j,d) \rightarrow \neg v2(i,j,d)))$

SUDOKU PAIR SOLVER :

Implementation :
The program “pair_solver.py” takes input from a csv file, “input.csv” , which contains two sudokus (both of them having p dimensions) , sudoku1 and sudoku2, and stores them in the form of 2-dimensional array, grid1 and grid2 respectively. Then it calls function solve() which takes grid1 and grid2 as its parameters. In this function, another function sudoku_clauses is called which returns all the clauses in the form of a list of lists. The sudoku_clauses() function appends all the constraints, i.e, number constraint (each cell contains at least one number in the range(1,p*p)), (column constraint (no two cells in the same column have same value), row

constraint (no two cells in the same row have same value) , grid constraint (no two cell in the same subgrid (p*p) have the same value), duplicate constraint (one cell cannot contain duplicate values) and corresponding constraint (corresponding cells in the two sudokus cannot have same value) in the form of clauses in a list and returns it. In the solve() function, clauses are appended with all the clauses related to the already known digits in both the sudokus. Then the cnf encoding is written in the “tmp.cnf” file. Then, “pair_solver.py” invokes minisat to take input from “tmp.cnf” and write its output in “tmp.sat”. Finally, the output from “tmp.sat” is converted to a readable sudoku format and writes it in the file “output.csv”.

Assumption :

- k (input as p) is same as k, the dimension of sudokus in the input, “input.csv” file
- Input sudokus are already satisfying the constraint of sudoku pair, i.e, corresponding cells in the two sudokus have distinct values
- k>1

Limitations :

- K must be greater than 1
- It might take a few minutes for very large k

SUDOKU PAIR GENERATOR

The code takes ‘p’ as input which is the dimension of the sudoku. The main function is ‘puzzle’, which generates the sudoku pair satisfying conditions as given in the question. In this function, we have arrays :

- inp : array containing the values in sudokus which has to be checked for creating holes
- filled : array with the values of each filled cell in sudokus after repeatedly creating holes as well as satisfying unique solution

In both arrays, we append all the values we get after calling pysat solver with initial sudoku conditions.
Now, we go into the while loop. In this loop, each time we start with two arrays , clauses1 and clauses2 with initial sudoku clauses and a random element from inp array, till the array is empty. We remove that selected element stored in variable ‘a’ , is removed from both arrays , inp and filled. Now, in both arrays , clauses1 and clauses2, we append all the filled elements.
Now, SAT-solver is called for the holes present. The solution which we get upon filling the holes is appended in clauses2 with negation in the form a list. Again SAT-solver is called, if it gives a solution, that means it has multi solution, i.e, not minimal case. Thus, we append it back into the filled array. If it has not solution, it implies that it has unique solution. Thus, the loop continues. We reinitialize the solver.
This function returns the filled array, which can be used to print our sudokus.

Limitations:

With increase in value of “p” i.e., size of sudokus to be generated, the time taken by the program increases. Likewise, for p=2 time = 1sec, p=3 time =8 to 10 sec, p=4 time =5 min and so on
Also, number of holes in puzzle vary with the rando case generated and hence large variation can be seen. Likewise, for p=3, number of zeros vary between 110 to 120

Assumption:
K(input as p) value is taken optimum to get puzzle in short time constraint