Question 1

Approach:

The first thing that was needed to be done was to encode the problem. We considered the “degrees of freedom”(the number of variables needed to find a unique position having a unique value) in the problem -

* which sudoku pair was being solved
* the row of the cell
* the column of the cell
* and the value of the cell.

A very natural encoding was to use a 4-digit number base . Hence, an encoding like , implies that we are considering the:

i) The second sudoku is being solved (Most significant digit)

ii) The third row

iii) The fourth column

iv) The cell has a value of 5

With encoding out of the way, it was time to add the constraints. The constraints in a regular sudoku game are given below:

i) Each cell can occupy **exactly one** value

ii) Each row contains each number between 1 to **exactly once**

iii) Each column contains each number between 1 to **exactly once**

iv) Each sub-block contains each number between 1 to **exactly once**

The extra constraint is:

v) Corresponding cells of the sudoku-pair solution do not have the same value. Equivalently, if we consider a particular cell, and consider the values which the cell holds across the two boards, then each value can appear **at most once**. Note that, **we do not need to ensure that at least one value appears** since that is already ensured by constraint (i).

Implementation:

The first program that we implemented was an independent Sudoku Solver using SAT, which can solve single regular sudoku games (filename: **sudokuSolver.py**). This was implemented using a class. Each SudokuSolver object has its own CNF, which stores the CNF to solve its sudoku. The SudokuSolver has the following functions of interest:

i) extract\_index() and extract\_location() to encode and decode variables

ii) verify\_cell() adds the constraint (i) clauses

iii) verify\_row() adds the constraint (ii) clauses

iv) verify\_col() adds the constraint (iii) clauses

v) verify\_sub\_block() adds the constraint (iv) clauses

**[1) Note that we are not adding constraint (v) clauses at this point of time since we are only solving a single, regular sudoku**

**2) Note that these clauses are independent of the other sudokus]**

vi) encode\_restrictions() adds the assumptions to the SAT solver based on the clues given.

A SudokuSolver object can take an unsolved sudoku board, create the CNF and the assumptions for it, solve it and store the solution.

Since the SudokuSolver class solves single sudoku boards, there was no point solving a sudoku pair only - so we generalised our code to handle sudoku-tuples. The sudoku-tuple solving is done in **Q1.py**. By dividing the number of rows by the number of columns in the CSV, we can get the number of sudoku puzzles in the sudoku-tuple. We initialise the same number of SudokuSolver objects, add the clauses and the assumptions from each sudoku to a global CNF and assumptions list. Finally, we call the **no\_overlaps()** function to add the clauses corresponding to constraint (v).

With the constraints and the assumptions, we can get a model of the CNF, from which we can get the sudoku-tuple solution.

Assumptions:

i) We assumed that **NO CELLS** should overlap between the sudokus. Suppose in the clues that are given, there is an overlapping value in a particular cell location. The solver will output **“No Solutions.”**

ii) The csv filename has to be changed in the source-code itself. It has to be changed in **Q1.py - line number - 22 - csv\_sudoku variable.**

Limitations:

i) The solver is slow for higher order sudokus. k = 6 takes a non-trivial amount of time.

Question 2

Approach:

The solution to this problem is split into the following stages:

1. Generating a **random** sudoku-tuple puzzle of the required dimensions (k)
2. Removing numbers one-by-one from the sudoku-tuple puzzle till we reach a maximal puzzle

Implementation:

Generating a Random Sudoku Puzzle: This part of the algorithm is done in the **random\_generate()** function. We add each number from the set to a random cell in sudoku-tuple. Since each number is distinct, there is no way any of the sudoku-tuple rules are getting violated at any step. The only constraint is that two numbers should not end up on the same cell, which can be dealt with quite easily.

Creating the Maximal Sudoku-Tuple: This part of the algorithm is done in the **create\_maximal()** function. Firstly, we take our sudoku solution and find all the literals that are true, corresponding to the solution. Then, we add those literals to the assumptions list. The order in which we add the literals to the assumptions list should ***ideally be random* (more on this is mentioned in the limitations)**. Now, we remove elements, starting from the top of the list. This is done in the following manner, **(suppose we are currently at the element of the list)**:

1. Take the current list element and negate it. (i.e make it negative)
2. Now make the solver solve the CNF with this new assumption. Since we have negated the element of the list, it essentially implies we are looking for a solution which **DOES NOT contain this particular literal**.
3. If a solution does not exist, then remove the element from the list. Directly removing the element from the list can mess up the indexing of the rest of the elements. To counter this, we use an additional **DUMMY LITERAL**. The integer value by which the DUMMY LITERAL is addressed is **greater than the largest literal present in the sudoku-tuple encoding**. This literal has no connection to the sudoku puzzle. We simply set the element to this **DUMMY LITERAL**, essentially removing the former assumption (and hence clue).
4. If a solution does exist, negate the element again (to make it positive). Removing this clue destroys the uniqueness of the sudoku-tuple.
5. Move to the element.

The steps outlined in the above algorithm produces a maximal sudoku-tuple on termination. We make the key claim that at every step of the algorithm, the sudoku-tuple has exactly one solution. This can be proved by mathematical induction, as follows:

Base Case: We are at the step of the algorithm.

Initially, the sudoku-tuple is completely filled. Obviously it has a unique solution.

Inductive Hypothesis: The sudoku-tuple maintains its uniqueness till the step.

To Show: The sudoku-tuple maintains its uniqueness after the step.

Now, on negating the element of the list, if no solutions exist, it implies that there cannot exist any solution without that literal being true. Hence, if we remove that literal, and solve the sudoku-tuple, it does not matter since that literal **MUST BECOME true while solving the CNF**. And when that literal becomes true (along with the other clues already present), then by the inductive hypothesis, the sudoku-tuple must be unique. Hence, removing the element preserves uniqueness.

If a solution still exists after negating the element, we just revert back to the same list. By the induction hypothesis, the sudoku-tuple remains unique.

We are done by induction.

Assumptions:

i) We assumed that **NO CELLS** should overlap between the sudokus. Suppose in the clues that are given, there is an overlapping value in a particular cell location. The solver will output **“No Solutions.”**

ii) The option to solve a higher number of sudokus must be changed from the source-code itself. It can be accessed from **Q2.py - line number - 9 - NUMBER\_OF\_SUDOKUS variable.**

Limitations:

i) ***Clue removal order*** - The order in which the true literals are added to the assumptions list (after generating the random sudoku tuple), should be random. However, we noticed that if we have a completely random clue removal order, then the sudoku solver takes noticeably more time. On the other hand, if we do not randomize the removal order (i.e the first clue is removed from the first cell, then the second cell and so on…), the sudoku solver takes much much less time to solve. But, the sudokus that we end up obtaining have no clues near the beginning while almost all the cells near the end are filled with clues. So, to make a compromise, a pseudorandom clue-removal order is used, where we remove, alternatively, from the start and the end of the sudoku. However, this concentrates the clues in the middle of the sudoku-tuple. A short analysis regarding time-taken is given:

**The tests are for sudokus of order 5, and they are averaged over 3 attempts.**

| **Clue-Removal Order** | **Average Time Takes** |
| --- | --- |
| Absolutely random | 1200 seconds |
| Fully sequential | 6.8 seconds |
| Pseudorandom | 7.3 seconds |

We have retained all three clue-orders in the source code (**line numbers: 80 - 94; Q2.py**) and if one wishes to try the different clue-orders, they can comment/uncomment appropriately.