Sudoku Solver

Possibility Position Score (PPS) Approach

1. Introduction

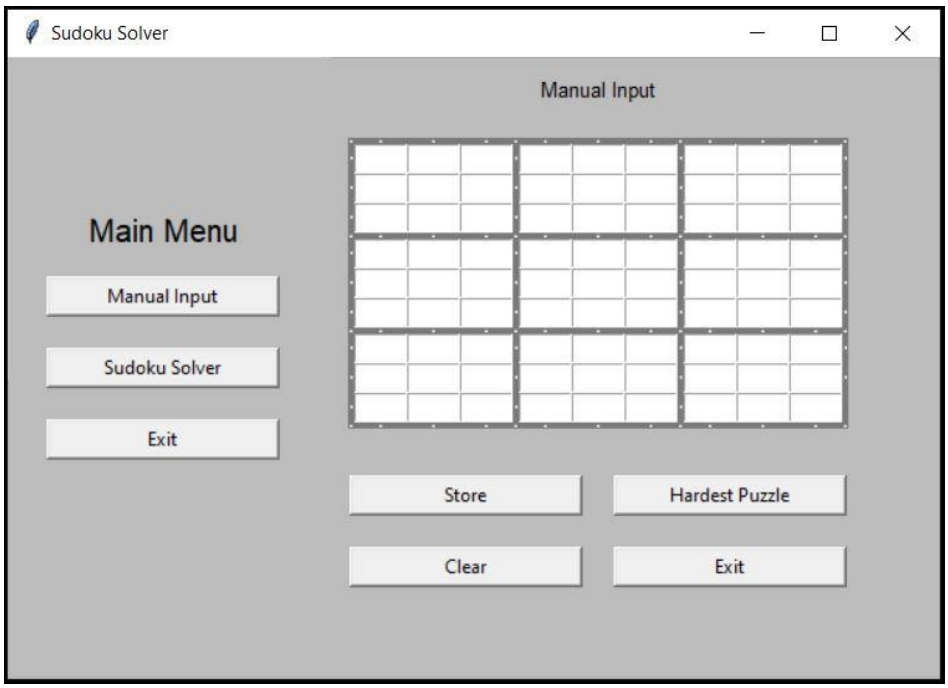
Despite multiple strategies existing for solving sudokus, a player may need to make smart guesses to move forward.

With an algorithmic program, we can brute force a solution by checking combinations of all possibilities, but this can become computationally expensive. Sudoku is an NP or Non-Polynomial type of problem, meaning with an increase in open positions in the sudoku puzzle, the computational time increases exponentially.

In this Python-based solver program, we attempt the puzzle using two approaches: conventional and modified brute force.

The program has been designed in a modular fashion, allowing for easy customization for other formats.

For the time being, a player needs to manually input the sudoku puzzle into the program through a Tkinter GUI.



2. Methodology

2.1. Puzzle setup

We begin by storing the sudoku puzzle in a 9x9 Numpy array, referred to as the *base* array. The first and second index of the array represents the row and column of the sudoku puzzle.

Using the *base* array, we create two additional arrays: *layers* and *box_value*.

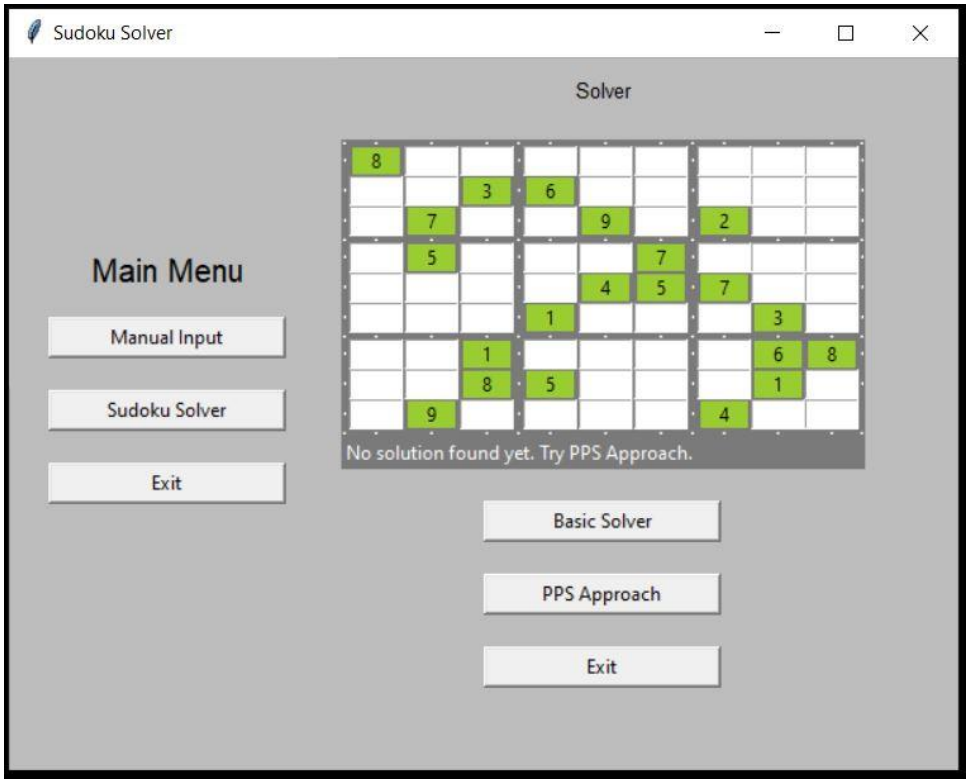
Layers is a 9x9x9 Numpy array, where the first index represents the digit, while the other two represent the row and column numbers. Each 9x9 sub-set of the *layers* array provides a possibility map for the digit in the sudoku puzzle.

Each element in a digit's layer can assume one of the following values,

- 0, indicates a possible location for the digit (open status)
- 1, indicates the digit is present in that location (present status)
- 2, indicates that location is unavailable for the digit (blocked status)

Box_value is a 9x9x2 Numpy array that stores the formation value of a digit's open positions in each box (3x3 as per conventional definitions) of a sudoku puzzle; the two values stored are the count of open positions in the box and the value of the formation or arrangement of the open positions.

To search and place missing digits in the sudoku, we employ a simple logic: if a row, column or 3x3 box (conventional definition) in a digit's layer has only one open position, then that digit is placed in that location or if any location or row-column combination has only one open position in the *layers* array, the digit corresponding to that open position is placed in that location.



2.2. Conventional Approach – Basic Solver

Here we utilize the typical approach employed by most sudoku players for solving any puzzle.

The conventional approach algorithm has the following steps,

1. We update the *layers* array based on the digit's position in the *base* array
    - The corresponding location of a digit is updated to present status in its layer and the same location in other digit *layers* to blocked status
    - The corresponding row, column and box (3x3 as per conventional definitions) of the digit's location is updated to blocked status in its layer
2. Based on the open positions in each digit's layer, we analyze complementary boxes for limiting possibility placements
    - A complementary box (3x3 as per conventional definition) is vertically or horizontally in the same row or column as the box under consideration; For box A, box 1 and box 2 are row complements, and box 3 and box 4 are column complements
    - Open positions of a row or column are blocked in a box (3x3 as per conventional definitions), if all open positions of a digit are concentrated in that row or column in the complementary boxes
    - This prevents the possibility of placing a digit in locations that would result in complementary boxes having no open positions for that digit

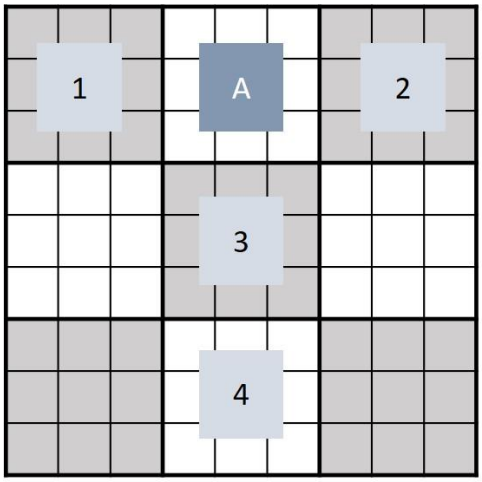


Fig. 1

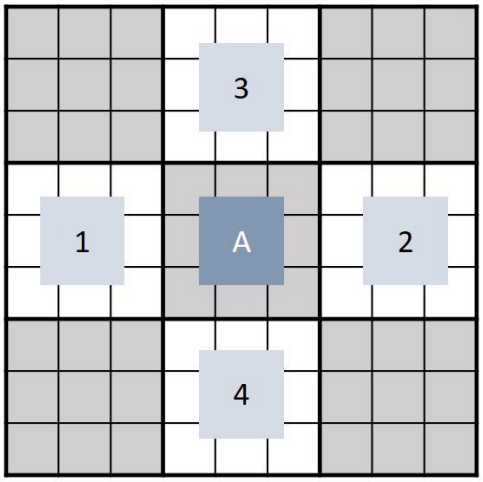
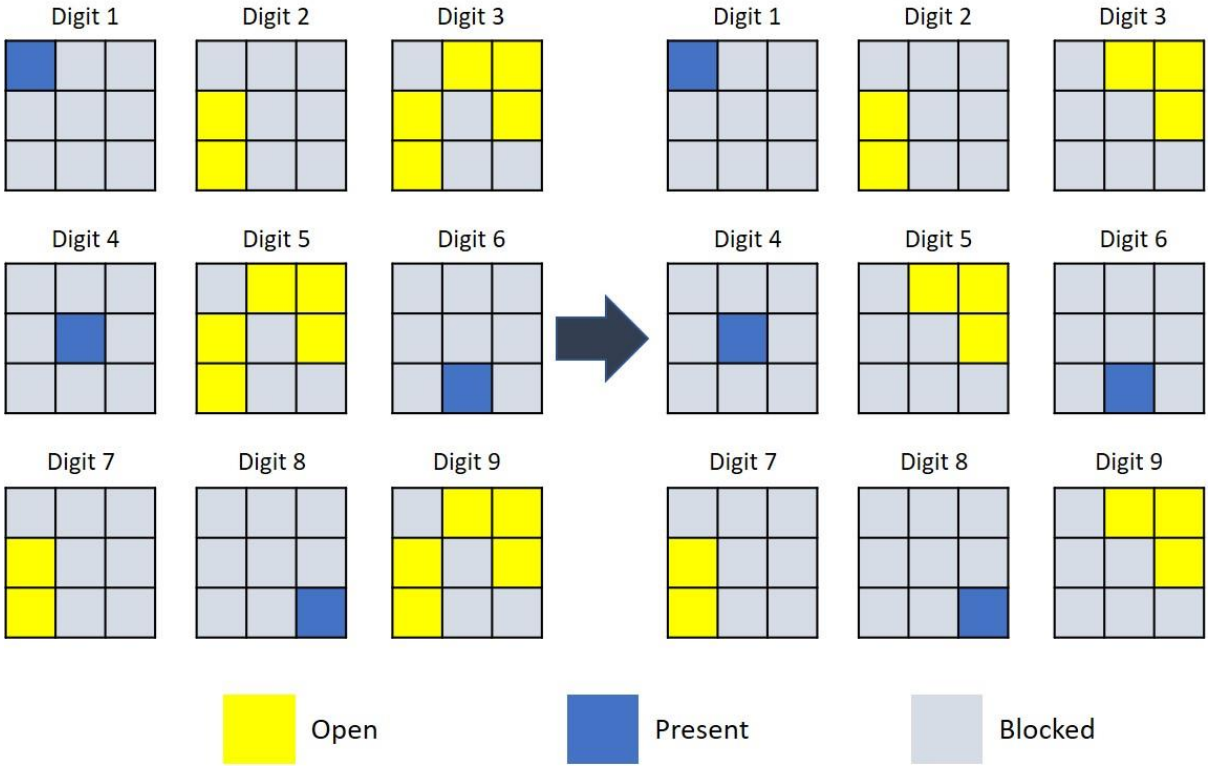


Fig. 2

3. If open positions for a set of digits are in the same locations within a box (3x3 as per conventional definitions) and the count of positions is equal to the count of digits in the set, any open positions for other digits (from outside the set) in these locations would are blocked
    - For example, if 2 digits have only 2 open positions each in the same positions, any other digit that has an open position in any of these locations would be updated to the blocked status
    - This prevents the possibility of placing other digits (from outside the set) in locations required for the set of digits under consideration, thereby allowing the box to contain all digits from the set



4. Finally, digits are placed in rows, columns, or boxes (3x3 as per conventional definitions) with only 1 open position (as mentioned at the end of the previous section)

Using the above algorithm, the solver updates the *layers* array, checks for any row, column or box with only one open position in a digit's layer or only open position at a given location, and updates it with the corresponding digit. The algorithm continues to run as long as the number of open positions in the *layers* array changes.

The puzzle is considered to be solved when the number of open positions in the *layers* array becomes zero; this should correspond to all locations in the *base* array having a digit present in them.

Based on the puzzle's complexity, a situation may arise that the further runs of the above-mentioned algorithm don't result in changes to the count of open positions while the count of open positions is greater than 0.

In this such a scenario we proceed with the modified brute force approach.

2.3. Modified Brute Force Approach – Possibility Position Score

To search for a set of possibilities that solve the sudoku puzzle, we generate a 9x9 Possibility Position Score (PPS) array for all open and present status positions in the *layers* array generated at the end of the conventional approach run. This array allows us to efficiently compare and combine possibilities and avoid any contradictions or missing digits in our final solution.

The solution search space, consisting of these PPS arrays, is segmented digit-box wise.

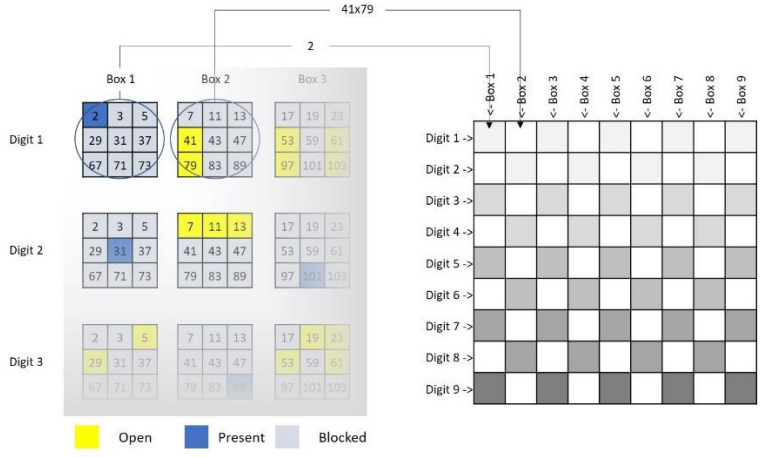
Constructing the PPS array for a possibility

1. Digit possibility is implemented/placed in the *base* array and the conventional approach algorithm is run
2. This results in updates to the *layers* array, and may reveal additional digits; It must be noted that in some instances, a digit possibility leads to zero open positions in the *layers* array, but with the *base* array still containing empty locations; such possibilities are treated as contradictory and are dropped from the solution search space
3. Each position/location or row-column combination of the *base* array is assigned a prime number; this represents the score of that location

2	3	5	7	11	13	17	19	23
29	31	37	41	43	47	53	59	61
67	71	73	79	83	89	97	101	103
107	109	113	127	131	137	139	149	151
157	163	167	173	179	181	191	193	197
199	211	223	227	229	233	239	241	251
257	263	269	271	277	281	283	293	307
311	313	317	331	337	347	349	353	359
367	373	379	383	389	397	401	409	419

Position Score

4. For each possibility, the algorithm computes the product of position scores for open and present status positions in each box (3x3 as per conventional definition) for all digits, resulting in a 9x9 array as stated above



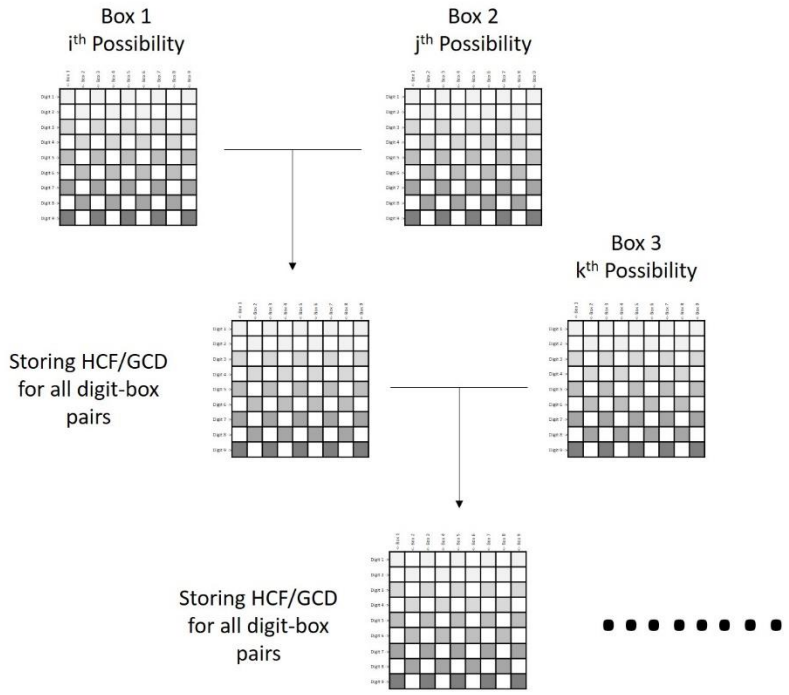
Possibility Position Score
(9x9 Array)

For a box (3x3 as per conventional definitions) with a present status position for a digit, the PPS array contains only a single prime number, which corresponds to the row-column location of that digit. The algorithm attempts to generate a PPS array with unique single prime numbers in all locations; this PPS array would be the solution for the sudoku.

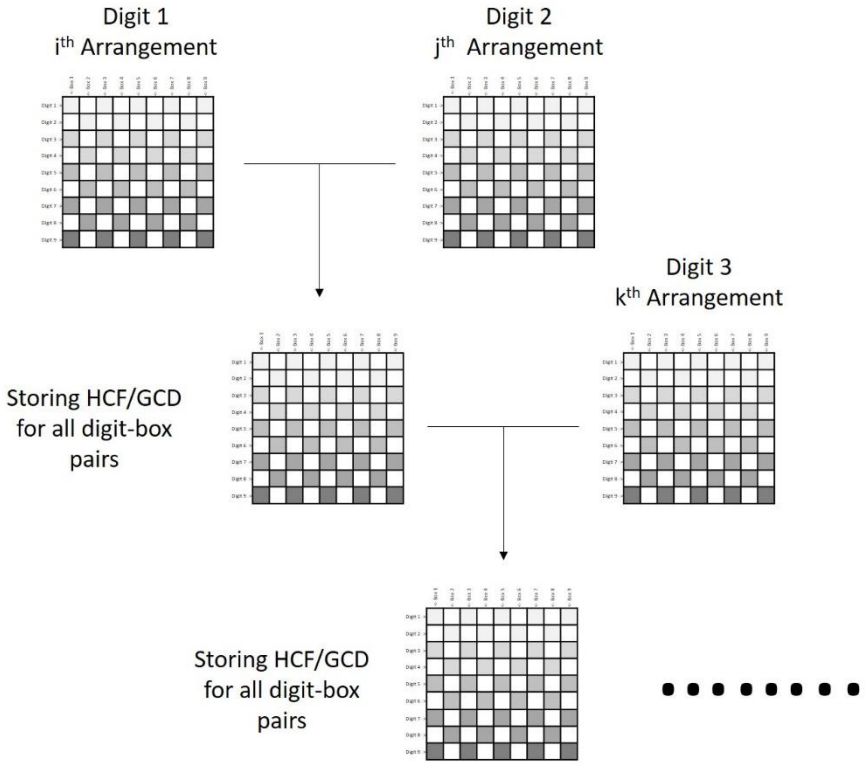
The algorithm generates the solution in two discrete steps: Building digit wise sub solutions, and then combining digit wise sub solutions into the final solution.

Building digit wise sub-solutions

- A digit sub-solution is a PPS array with only single unique prime numbers in the row corresponding to that digit
- This is analogous to generating all possible arrangements of a digit in the sudoku puzzle without causing any contradictions (row-column-box conflicts) for any digit (including itself)
- As the solution search space is segmented box wise (3x3 as per conventional definitions), the algorithm builds the digit sub solutions by sequentially picking possibilities from each box; possibilities within a box are sorted based on the count of digits they have revealed
- Possibilities are added to a sub-solution if they are compatible with each other: Compatibility is defined as having a highest common factor (HCF) or greatest common denominator (GCD) greater than 1 for all digit-box pairs of the PPS arrays; this implies that the *layers* array for the two selected possibilities agree with each other on the potential (open positions) or final (present positions) placement of digits throughout the *layers* array
- After adding a possibility to the sub-solution, the algorithm stores the HCF or GCD for all digit-box pairs in the sub-solution's PPS array



The next step involves combining these digit wise sub-solutions using the same compatibility and combination logic as mentioned above to arrive at the final solution for the puzzle. Before sequentially picking arrangements for all digit, the digits are sorted in ascending order of count of possible arrangements.



2.4. Details of the code

1. Libraries
    - Numpy
    - Tkinter

2. Global variables
    - base - 9x9 numpy array, dtype = int
    - layers - 9x9x9 numpy array, dtype = int
    - box_value - 9x9x2 numpy array, dtype = float
    - entries - array to store user inputs

3. List of all functions
    - Main Menu
        - manual_input(base) - to receive inputs from user
        - solver(base, layers, box_value) - to create main view for solver
        - exit_all() - to exit the solver program
        - show_grid(base, base_og, frame) - to display grid used in the solver

    - Manual Input
        - clear_input_space(frame) - to remove all users entries
        - exit_manual_input(frame) - to exit manual input frame
        - store_value(base, frame) - to store user inputs
        - input_checker(user_inputs) - to check if there are any contradictions in user inputs
        - hardest_puzzle() - to setup predefined sudoku

    - Solver
        - run_solver(base, layers, box_value, frame_main) - to run solver based on conventional approach
        - PPSRun(base, layers, box_value, frame_main) - to run solver based on PPS approach
        - clear_solver_view(frame) - to exit solver frame

    - Basic Solver
        - layerCounter(layers) - to count possibilites at a position
        - blankCounter(layers) - to count all possibilities/open spaces in the layers
        - solver_layer_updater(base, layers) - to update layer based on digit present in base table
        - solver_row_column_check(layers) - to update each layer for blocked cells
        - solver_box_check(layers) - to update each box in a layer for blocked cells
        - solver_comp_box(layers) - to check complementary boxes for limiting possibility placements
        - comp_box_row_sum(p,q,digit,row) - to check possibility placements in rows of complementary boxes
        - comp_box_col_sum(p,q,digit,col) - to check possibility placements in columns of complementary boxes
        - matches_updates(layers,box_value) - to prefinalize of possibility formations

- matches(layers, box_value) - to match and eliminate possibilities based on possibility formations
- naked_single(base, layers) - to identify and fill digits where possible

- PPS Solver
  - PPS_gen(layers) - to digit wise possibility map
    - Structure -> [digit 1, [box 1, [[digit,x1,y1],...[digit,xn,yn]]],..,[box 9, [[digit,x1,y1],...[digit,xn,yn]]]]
  - PPS_build(PPS_set, base, layers, box_value) - to generate layer score for each possibility in the digit layer
    - Updated structure -> [digit 1, [box 1, [[[digit,x1,y1], NS_count, box_score_set],...[[digit,xn,yn], NS_count, box_score_set]]],..,[box 9..]]
  - run_solver_PPS(base, layers, box_value) - to generate list of revealed digits for each possibility
  - naked_single_PPS(base, layers) - to identify and fill digits where possible; also returns list of revealed digits and their positions
  - cleanArray(nsSet) - to check list of revealed digits for duplicates, and remove the same
  - isContradiction(base, layers) - to check if a possibility has resulted in a contradiction
  - box_score_set_gen(layers) - to score each digit layer based on actual and possible positions
  - posSort(posSet) - to sort possibility position space based on count of digits revealed by a possibility
  - build_full_sol_set(PPS_set) - to generate list of potential solutions for all digits
  - build_sol_set(PPS_sub_set, index) - to generate list of potential solutions for a digit
  - no_contradiction(sol1, sol2) - to identify contradictions between two possible solutions for a digit; ensures there is overlap between solutions
  - sol_combine(sol1, sol2) - to combine digit solutions assuming no contradiction
  - PPS_sol_set_sorter(PPS_sol_set) - to sort PPS for all digits in ascending order
  - build_sol_set_final(PPS_sol_set) - to generate list of potential solutions
  - display_solution(PPS_sol_set, sol_set) - to compile solution and return the same

3. Results & Discussion

While the conventional approach can successfully solve most sudoku puzzles, for expert-level sudokus the modified brute force or Possibility Position Score (PPS) approach works well.

The program solved the sudoku puzzle created by Dr Arto Inkala in 2010 within ~32 seconds, and the sudoku puzzle created by him in 2006 within ~12 seconds.

Furthermore, the PPS approach has no inbuilt limit on the number of solutions it has to generate. While a valid sudoku puzzle has only one solution, the algorithm can be used to check the validity of a sudoku puzzle.

Potential next steps:

- Reducing the size of the code by matching array structures; this will lead to increase in function code reusability
- Adding more options for user input like image recognition
- Adding a timer on the GUI to display actual time taken to solve a sudoku
- Tweaks to code to handle sudoku's with non-standard box definitions
- Converting the program into a sudoku puzzle generator