Sudoku Solver

Possibility Position Score (PPS) Approach

1. Introduction

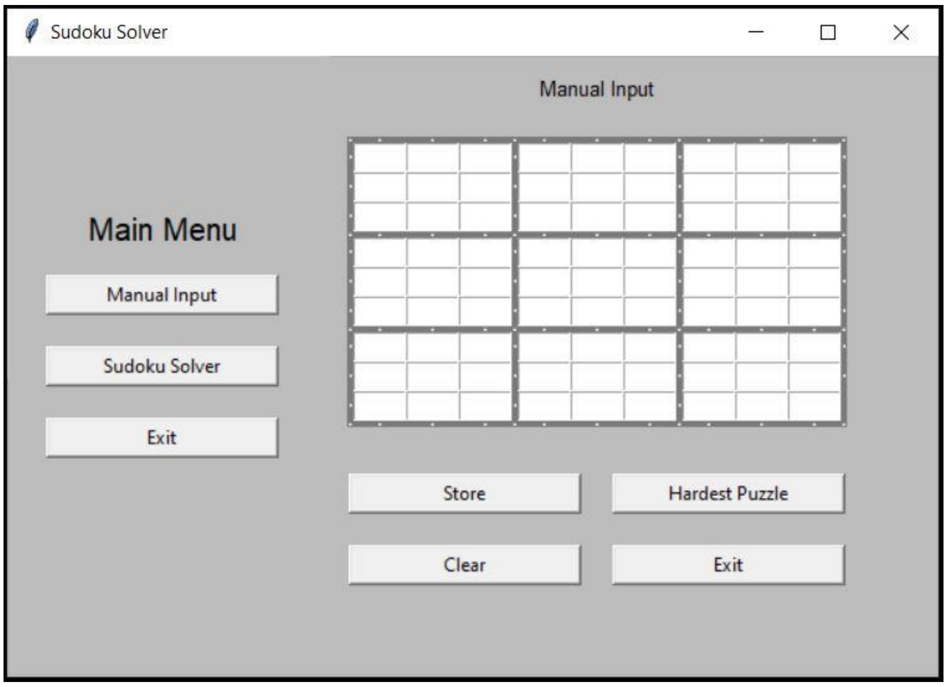
While multiple strategies exist for solving a sudoku, beyond a point a player is required to make smart guesses to move forward.

With an algorithmic program, we can brute force a solution by checking combinations of all possibilities, but this can become computationally expense. This is due to the fact that a sudoku is an NP or Non-Polynomial type of problem, meaning with 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. Both approaches are described in details in the Methodology section.

The program has been designed in a modular fashion, allowing for easy customization to different formats of the game.

For the time being, a player needs to manually input the sudoku puzzle into the program through a Tkinter GUI. An image recognition-based input procedure may be considered in the future.



2. Methodology

2.1. Puzzle setup

The sudoku is stored in a 9x9 Numpy array which we shall refer to as the *base*. The first index is used to determine the row in the sudoku puzzle, while the second index is used to determine the column.

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

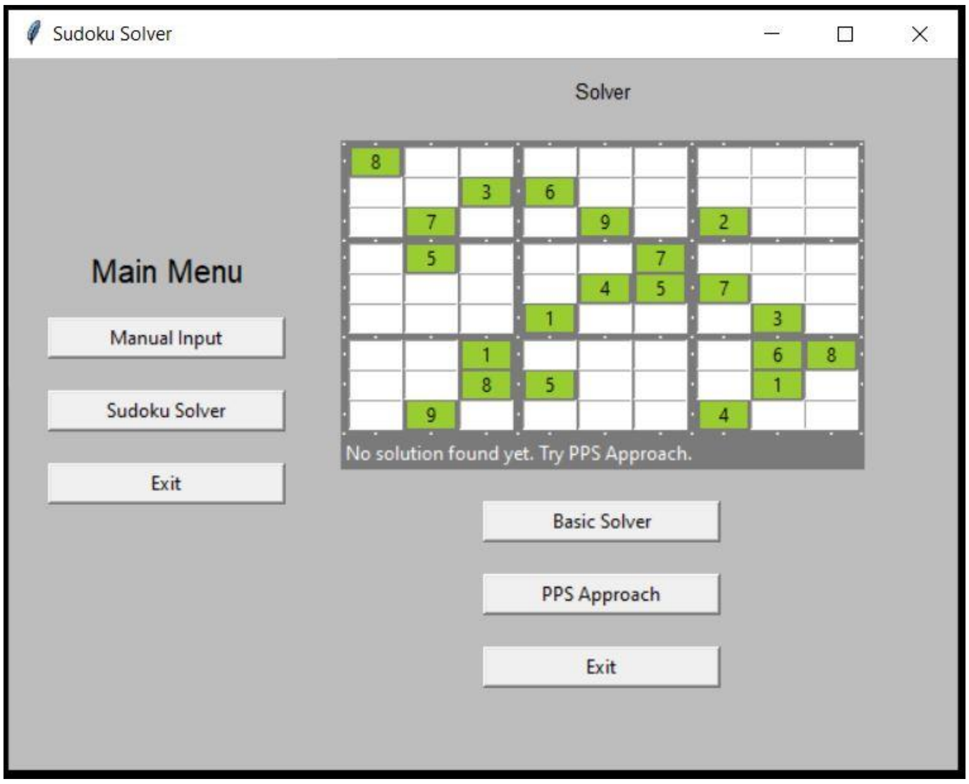
'*layers'* is a 9x9x9 Numpy array, representing the digit wise view of the sudoku puzzle. The first index is used to determine the digit, and the second and third are used for row and column identifiers.

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

- 0 indicates that the digit can be placed in the location a.k.a. *open* status
- 1 indicates that the digit is present in the location a.k.a. *present* status
- 2 indicates that the location is blocked for the particular digit a.k.a. *blocked* status

'*box_value*' is a 9x9x2 Numpy array which is used to store the formation value of a digit's open positions in each box of sudoku puzzle. The first index represents the digit, the second index represents the box number. The final index stores two values, count of open positions in the box and value of the formation. This aspect of the solver shall be explained in greater detail in the next section.

In order to search and identify missing digits in the sudoku, we primarily make use of the '*layers*' array. In any digit layer, if a row, column or 3x3 box (conventional definition) has only one *open* position, i.e., only position with value 0, then the digit is placed in that location; the position in the digit layer is updated to 1 or *present*, and the corresponding position in the *base* is updated with the digit. Similarly, if any location in the *base* has only one *open* position in the *layers*, the digit corresponding to that *open* position is updated in the *base*, and the *open* position is updated to 1 or *present* status.



8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

No solution found yet. Try PPS Approach.

2.2. Conventional Approach – Basic Solver

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

Based on the digits present in the *base*, the *layers* and *box_value* are updated.

1. Each digit layer is updated based on the digit's position in the *base*
    a. Corresponding position in the digit's layer is updated with 1 or *present* status, and the same position in other digit layers is updated with 2 or *blocked* status
    b. The corresponding row, column and box of the digit's position in the digit layer is updated with 2 or *blocked* status
2. Based on *open* positions in a digit's layer, complimentary boxes are checked for limiting possibility placements
    a. Complimentary boxes (3x3 as per conventional definition) are ones that are vertically or horizontally placed in the same row or column
    b. If all *open* positions for a digit are concentrated in a row or column, this results in all *open* positions of that particular row or column in complimentary boxes to become *blocked*
3. If *open* positions of a set of digits is present in the same locations within a box (3x3 as per conventional definitions) and count of locations is equal to count of digits in question, all *open* positions for any other digit outside the set in these locations would become blocked
    a. Assuming within a box, 3 digits have 3 *open* positions each, all of them are in the same 3 locations, any other digit that has an *open* position in any of these 3 locations would be updated to the *blocked* status

Using the above-mentioned algorithms, the solver updates the *layers*, 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* changes.

The puzzle is solved when the number of *open* positions in the *layers* becomes zero.

Based on the puzzle's complexity, a situation may rise that the further runs of the above-mentioned algorithm don't result in changes to count of open positions and count of open position 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 Space

We generate a search space using the final state of the *base*, *layers*, and *box_value* arrays at the end of conventional approach run. The search space consists is a multi-dimensional array consisting of the following elements,

- Possible digit placement (based on *open* status in the *layers*)
- Count of other digits revealed by the above-mentioned digit placement
- Set of possibility position scores (explained below)

Possibility Position Score

Each location in the puzzle or *base* is assigned a prime number; this represents the *score* of the position.

Using a digit's layer from *layers*, the algorithm computes the product of all *open* position scores or *present* position score for each box (3x3 as per conventional definition) and generates a 1x9 array of possibility position score for the digit. This is combined with other digits to form a 9x9 array of possibility position score for a possible digit placement.

The search space is generated using the following algorithm,

1. List of all *open* and *present* positions in the *layers* is compiled, digit wise and box wise (3x3 as per conventional definition)
2. Each *open* position is updated in the *base* (one at a time) and the conventional approach is run on this new *base*; this potentially reveals more digits in the *base*, and results in the *layers* being updated. It must be noted that in some instances, an open position leads to the *layers* being filled up, i.e., zero *open* positions, but with the *base* still containing empty locations; such possibilities are dropped from the search space
3. The updated *layers* is used to generate the Possibility Position Score for the *open* position

Once the search space is generated, the algorithm begins to generate sub-solutions for each digit. As the search space for a digit is organized box wise (3x3 as per conventional definitions), the algorithm selects possibilities from each box and combines them to form a sub-solution. It uses the possibility position score for each selected possibility to check its compatibility with other possibilities selected.

Here compatibility is defined as the highest common factor (HCF) or greatest common denominator (GCD) being greater than 1 for any combination of digit and box in the possibility position scores for a pair of possibilities selected; this implies that the *layers* for the two selected possibilities agree with each other on potential placement of digits through the *layers*. HCF or GCD of 1 for any combination of digit and box in the possibility position scores for a pair of possibilities would mean that when these possibilities are combined, the algorithm would not be able to place that digit in that box, hence leading to an incomplete final solution.

When two possibilities are combined, the resultant possibility position score would contain the HCF or GCD for all combinations of digits and boxes.

Sub-solutions for each digit would be a collection of 9x9 arrays of possibility position scores formed by combination of *open* positions of that particular digit from the *layers*.

These digit sub-solutions are then combined with each other using the same compatibility and combination logic as mentioned above to arrive at the final solution for the puzzle.

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 complimentary boxes for limiting possibility placements
        - comp_box_row_sum(p,q,digit,row) - to check possibility placements in rows of complimentary boxes
        - comp_box_col_sum(p,q,digit,col) - to check possibility placements in columns of complimentary boxes
        - matches_updates(layers,box_value) - to prefinalize of possibility formations

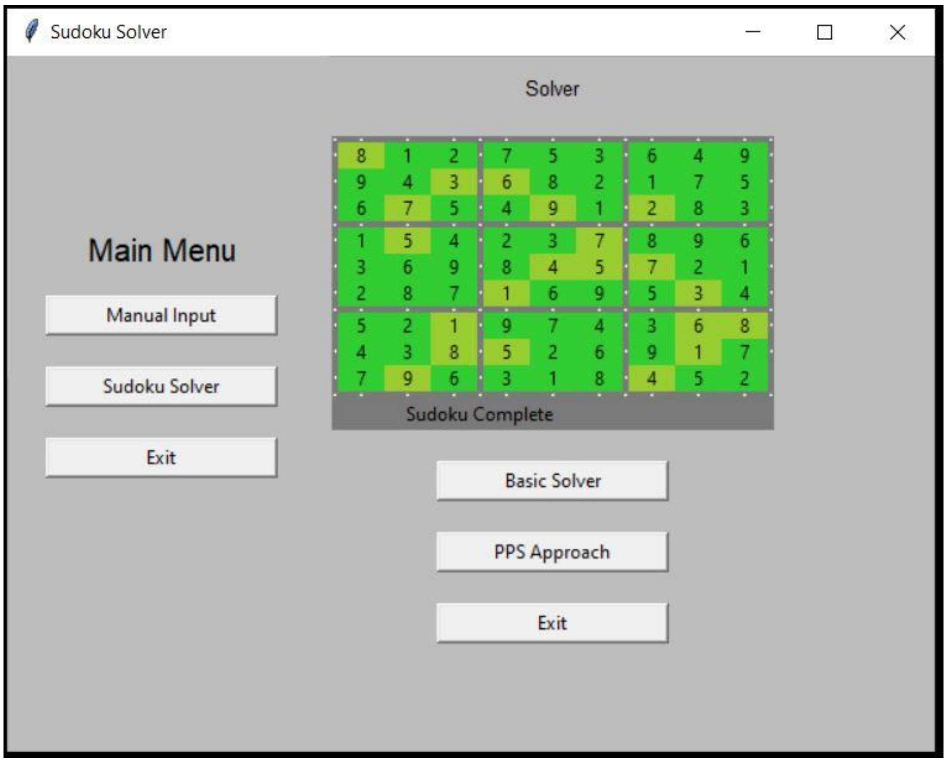
    - matches(layers, box_value) - to match and eliminate possibilities based on possibility formations
    - naked_single(base, layers) - to identify and fill digits where possible

- 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; new 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

Results & Discussion

While the conventional approach is able to successfully solve most sudoku puzzle, for expert level sudokus the modified brute force or Possibility Position Score 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.



8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

Sudoku Complete

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