**Sudoku Solving**

(AI Assignment 3)

**Done by:**

Maram Wael Emara 7523

Ahmed Hany Gabr 7905

Ann Tarek Saad 7853

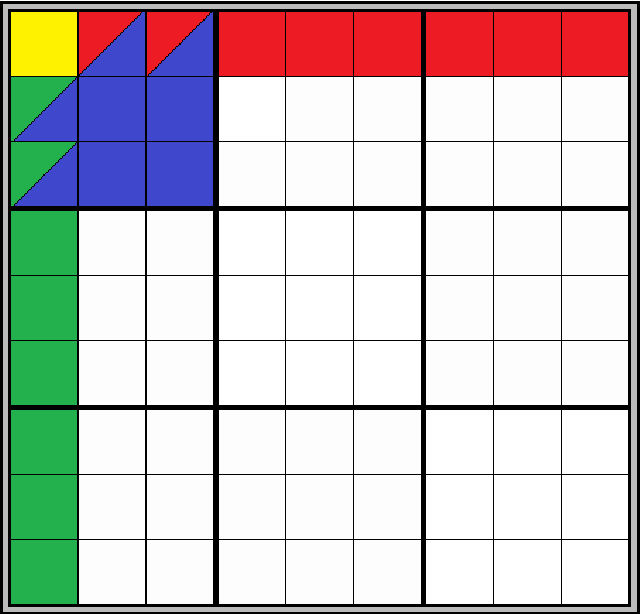
**Problem Definition:**

In Sudoku, the problem can be defined as a constraint satisfaction problem (CSP) where the goal is to fill a 9x9 grid with digits such that each row, each column, and each of the nine 3x3 subgrids (called "boxes") contains all of the digits from 1 to 9 without repetition. The constraints in this problem include:

* Each cell in the grid must contain a digit from 1 to 9 (unary constraint.)
* Each digit can appear only once in each row.
* Each digit can appear only once in each column.
* Each digit can appear only once in each 3x3 subgrid.
* The objective is to find a solution that satisfies all constraints, resulting in a completed Sudoku grid where no conflicts exist between the digits.
* We have a total of 27 constraints each relating 9 variables together (9 rows, 9 cols, 9 grids.)
* Each variable has d = 9
* Total of 81 variables n = 81

**Assumptions:**

We can convert all row, column, and grid constraints into binary constraints between the different variables. There will be a constraint for each variable between all the other 8 variables in its same row, all the other 8 variables in its same column, and finally, all the other 8 variables in its same grid. However, we find that out of the 8 variables in the variable’s grid, there are 4 variables that are on the same row or column. Therefore, when dealing with the grid constraint, we only need to consider the 4 remaining variables that are in the same grid but are on a different row and column. The following figure shows the original variable in yellow, the variables related to it by a row constraint in red, the variables related to it by a column constraint in green, and finally the variables related by a grid constraint in blue. The cells with multiple colours are the ones common between different constraints. Therefore, for each of the 81 variables, there are (8 + 8 + 4) other variables that it is related to, giving us a total of 81 x 20 = 1620 arcs.

.

**Classes and Data Structures:**

1. **Board Class:**

The Board class contains the state of the sudoku board - represented by an 81 character string - along with useful functions for manipulating the state. For example, the class has functions for getting and setting an index in the game state and a function that returns the possible values at an empty position in the board.

1. **Variable:**

Represents a position on the board and its actual value (if it is assigned) or the domain of possible values it could take (in case it is not assigned yet).

1. **Variables:**

A class that contains all the 81 variables in an array and allows efficient and easy retrieval of the variables

1. **Arc**

The Arc class contains pointers for two variable classes, represented by var1 and var2 within the class. Note that the order matters in Arc consistency (swapping var1 and var2 gives a different Arc.) Each Arc is also represented by a unique string id of four numbers: “var1.row,var1.col,var2.row,var2.col”. These identifiers are later used in the AC3 algorithm to check whether an Arc is in the Arc Queue.

1. **Arc Queue:**

A queue containing all the arcs which is used in the AC3 algorithm.

**Algorithms:**

1. **AC-3 (Arc Consistency) Complexity:**

The main goal of this algorithm is to reduce the domains of all variables before backtracking, and in certain cases even solving certain variables in case it reduces their domain to a single value. The following algorithm inserts all the arcs in an arc queue and continuously revises arcs to check that for all arcs A, for all values V1 in variable 1’s domain, there exists at least one value V2 in variable 2’s domain that V1 is consistent with. The algorithm terminates and returns failure instantly if a variable’s domain becomes empty (thus the board would have no solution.) Otherwise, the algorithm continues until the Arc queue is empty.

1. **Backtracking: Complexity**

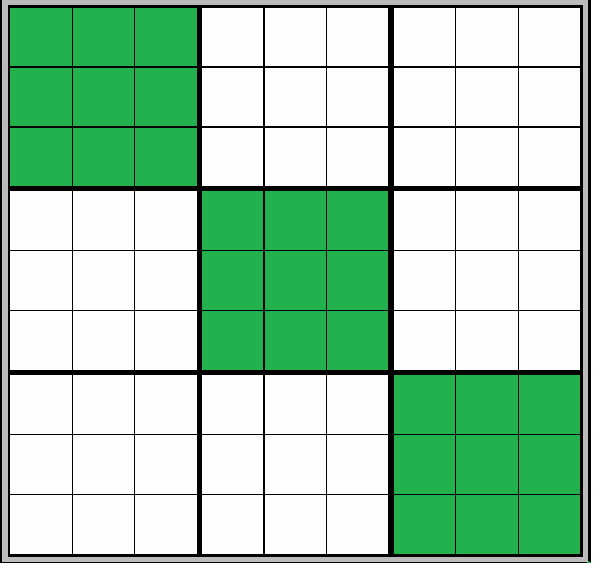
Backtracking Search is the main algorithm that solves the sudoku problem. Backtracking search is an algorithmic technique used to systematically explore all possible solutions to a problem by incrementally building candidates and abandoning those that fail to satisfy the problem constraints, effectively "backtracking" to previous decisions when necessary. It is often employed in constraint satisfaction problems, such as puzzles and combinatorial optimization. Backtracking search shares similarities with depth-first search (DFS), as both algorithms traverse a search space by exploring one branch of the solution tree at a time, recursively diving deeper until a solution is found or all possibilities are exhausted. However, unlike DFS, backtracking search has the ability to backtrack and undo decisions when a dead-end is encountered, allowing it to efficiently prune the search space and avoid unnecessary exploration. The backtracking search algorithm returns a string representing the final solved state of the board if a solution is found. If a solution is not found, however, it simply returns false.

1. **Backtracking Search for Counting Solutions Complexity :**

This is a modified version of the normal backtracking search algorithm which instead of returning a solution upon finding one, increments a counter. This algorithm is vital for board generation since we might want to ensure that a sudoku board has one single solution.

**Board Generation:**

Step 1: Randomly assign the numbers from 1-9 to each of the coloured grids. Notice that the 3 grids are independent from each other, and therefore when filling them with numbers we only need to consider each grid’s constraints separately. This first step speeds up board generation.



Step 2: Use Arc consistency to reduce the domains for the other unassigned variables (uncolored squares)

Step3: Use Backtracking Search to solve the board. After this step we have a fully generated board satisfying all constraints.

Step 4: If we are required to generate a board with a unique solution, move to step 6, otherwise go to step 5

Step 5: Randomly choose k squares on the solved board and make them empty. As K increases the game becomes harder, and there’s a higher chance that we would generate a board with multiple solutions if we don’t actively check that for a unique solution. After the K squares are emptied the function returns the partially full board.

Step 6:

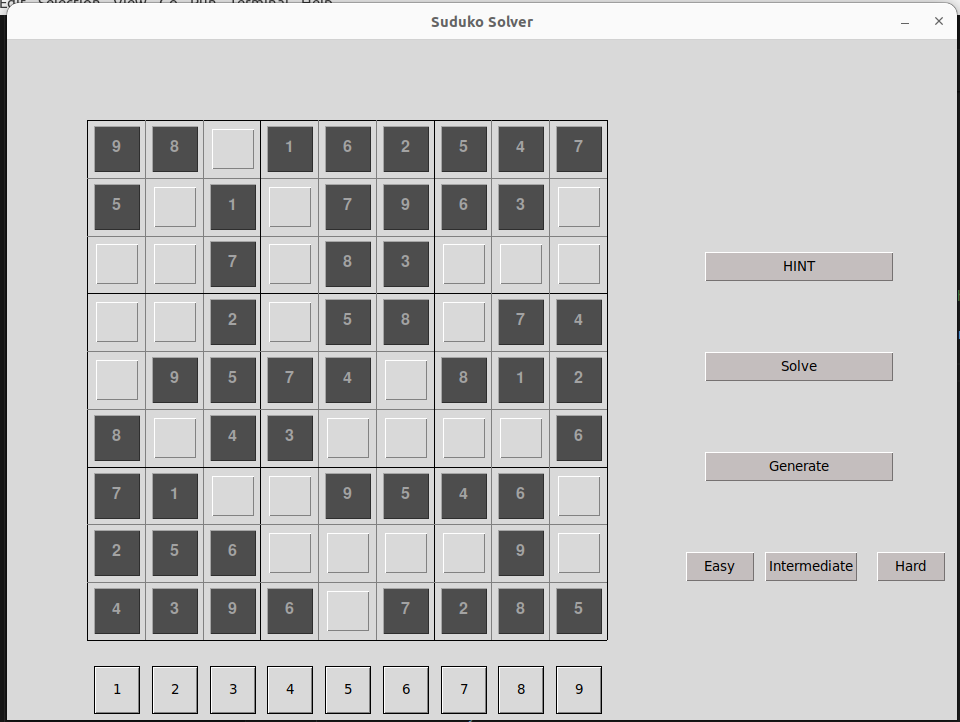
If we want to ensure that there is only one unique solution to the board, we keep setting different squares on the board to be empty as in Step 5. The difference, however, is that after emptying every position, we use the modified Backtracking Search to count the number of solutions to the Sudoku Board. If there is one solution, we move to a new cell to be emptied until the desired number of removed cells is reached. If the modified backtracking search algorithm finds more than one solution, however, the generation algorithm undoes the last action and reassigns to the previous variable its original value. It then attempts to empty another cell until the goal number of empty cells is reached, or there are no possible cells to be emptied without causing the board to have more than one solution.

**Sample Runs:**

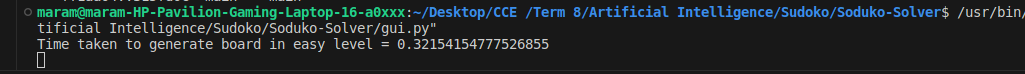
**Mode One : Generated and solved by AI**

1. **Easy Level:**

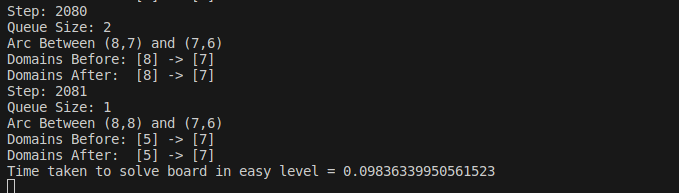
**Board Generated:**

****

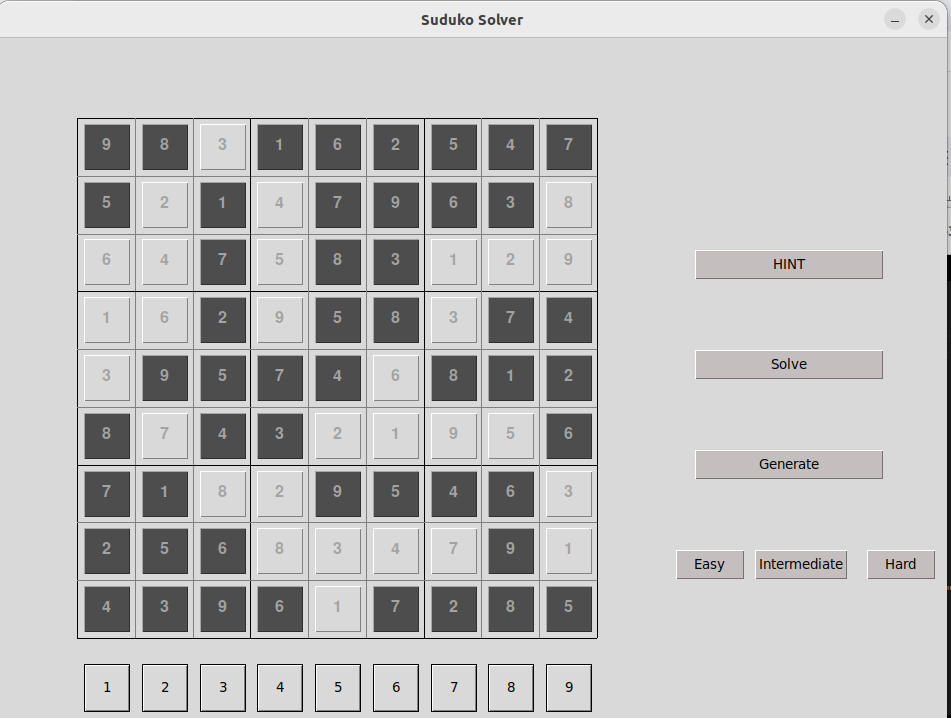
**Time Taken to generate the board :**

****

**Time Taken to solve the board and number of steps taken:**

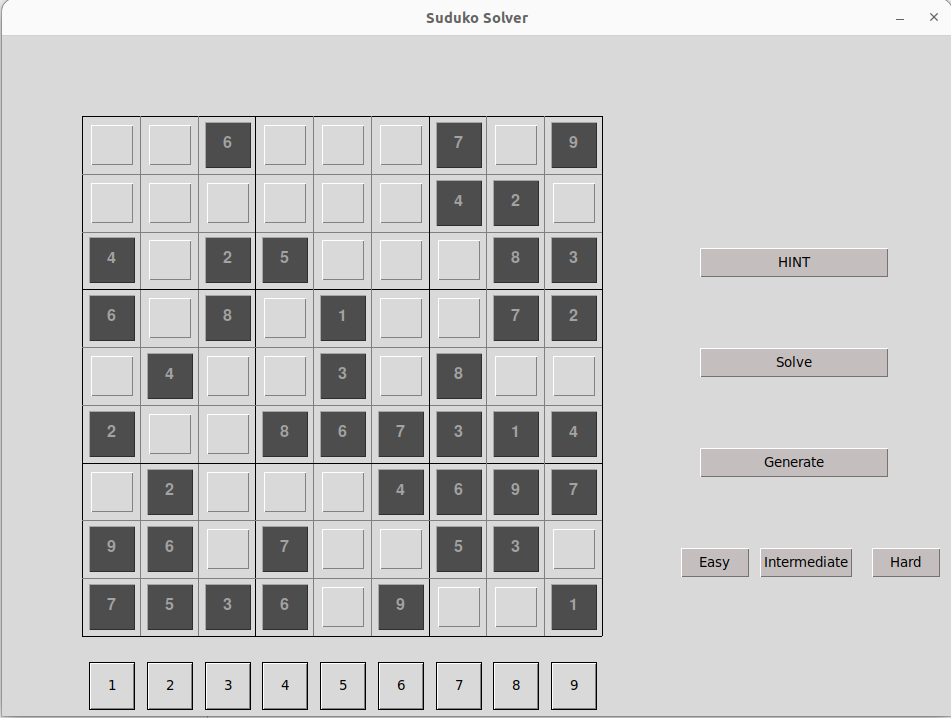
****

**Solution Generated:**

****

**2.Intermediate Level:**

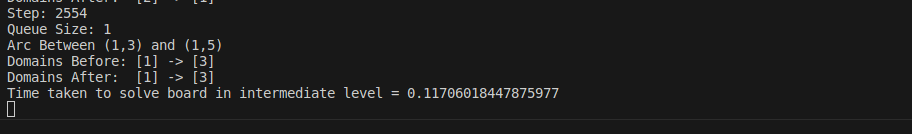
**Board Generated**

****

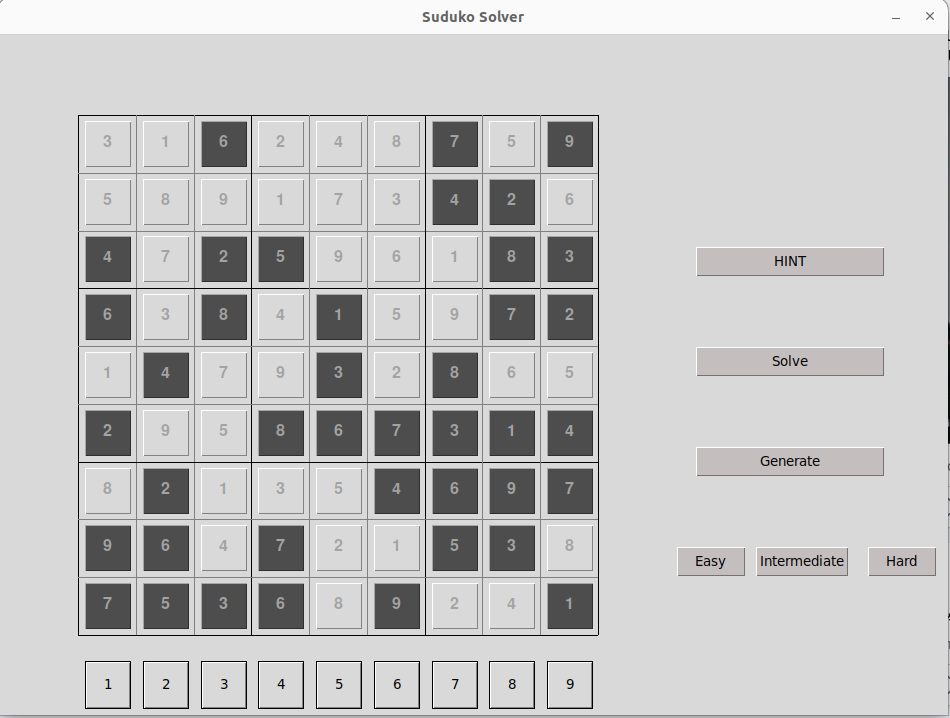
**Time Taken to generate the board :**

****

**Time Taken to generate the solution and number of steps:**

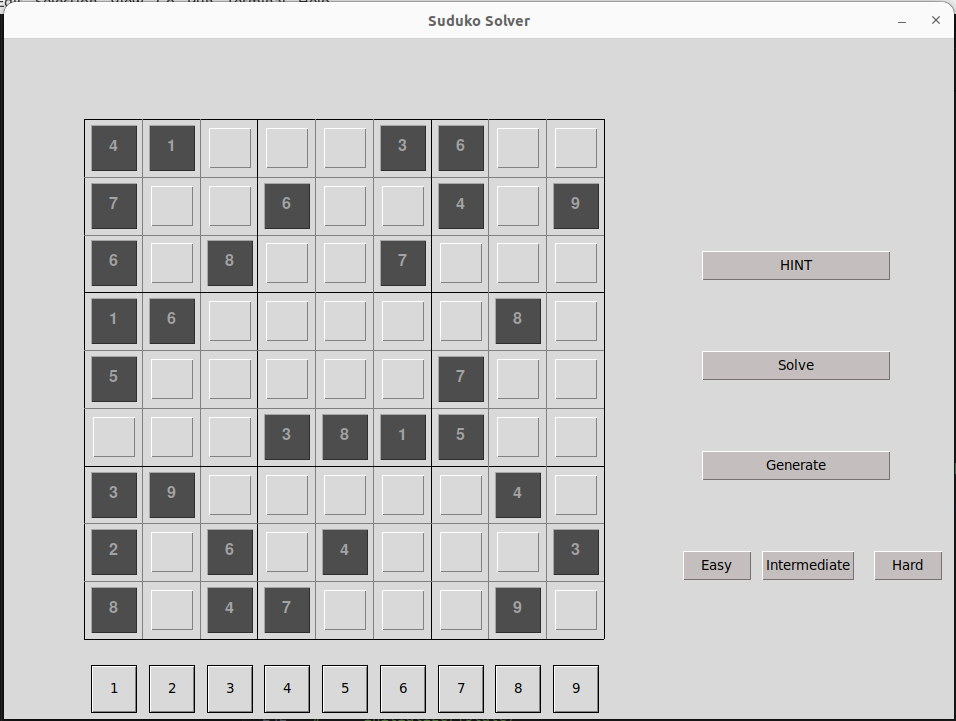
****

**Solution Generated:**

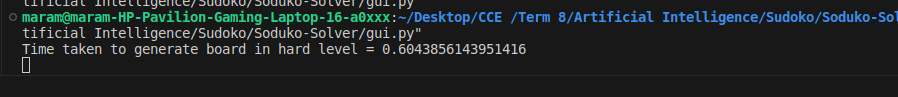
****

**3. Hard Level:**

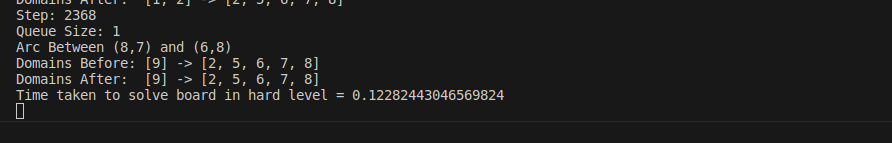
**Board Generated**

****

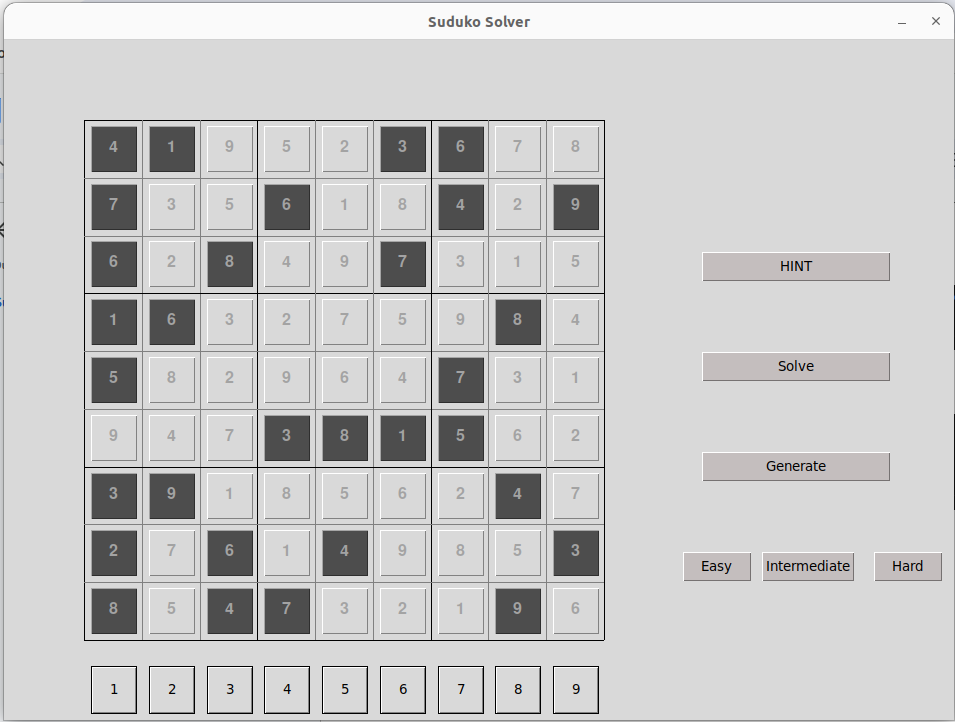
**Time Taken to generate the board**

****

**Time Taken to solve the board and number of steps**

****

**Solution Generated:**

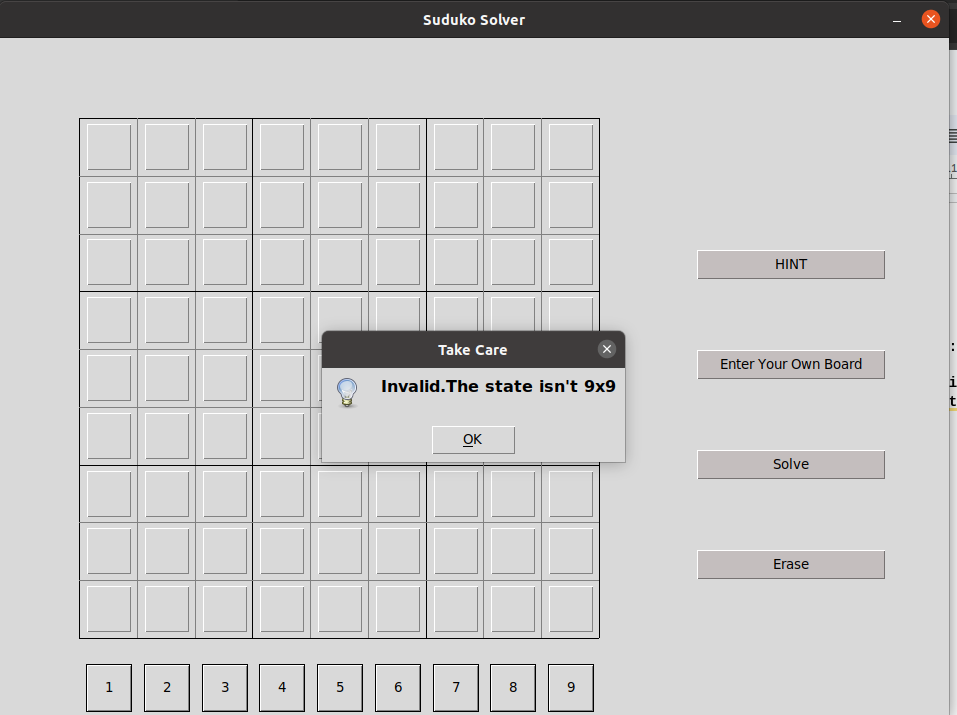
****

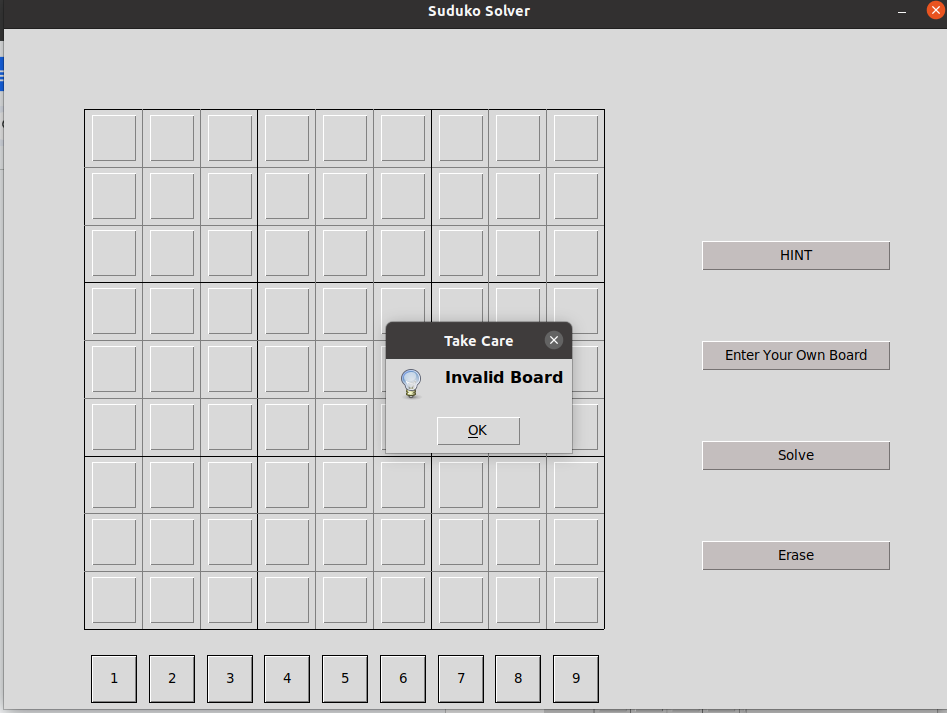
**Mode Two: Input by human and solved by AI**

**2 Features are available here (and in mode 3 too):**

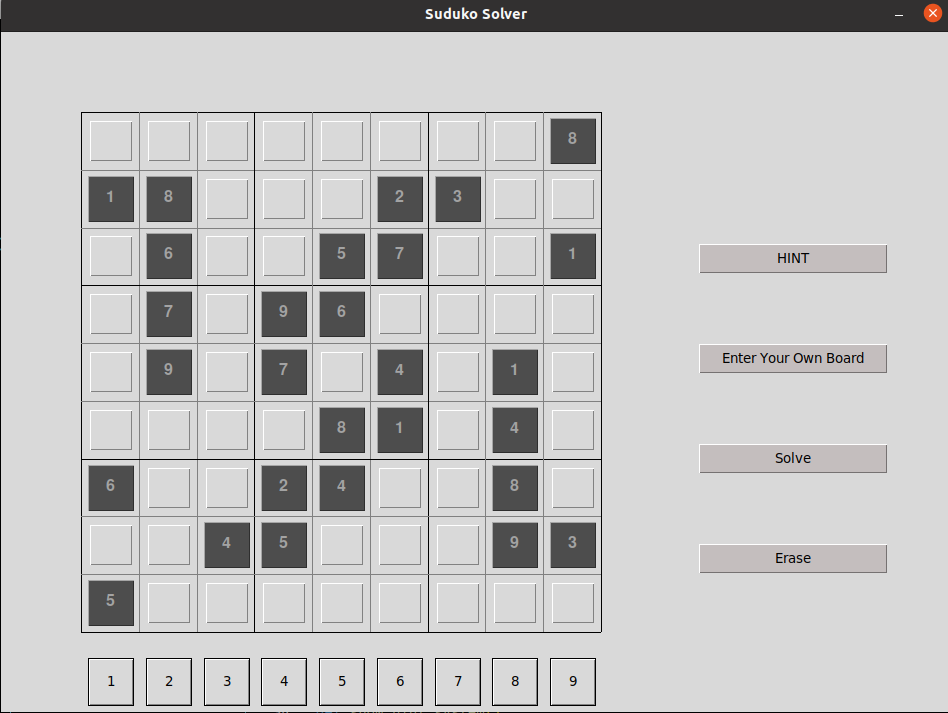
**Feature 1:**

Input the initial state as 81 characters in the simple dialog.

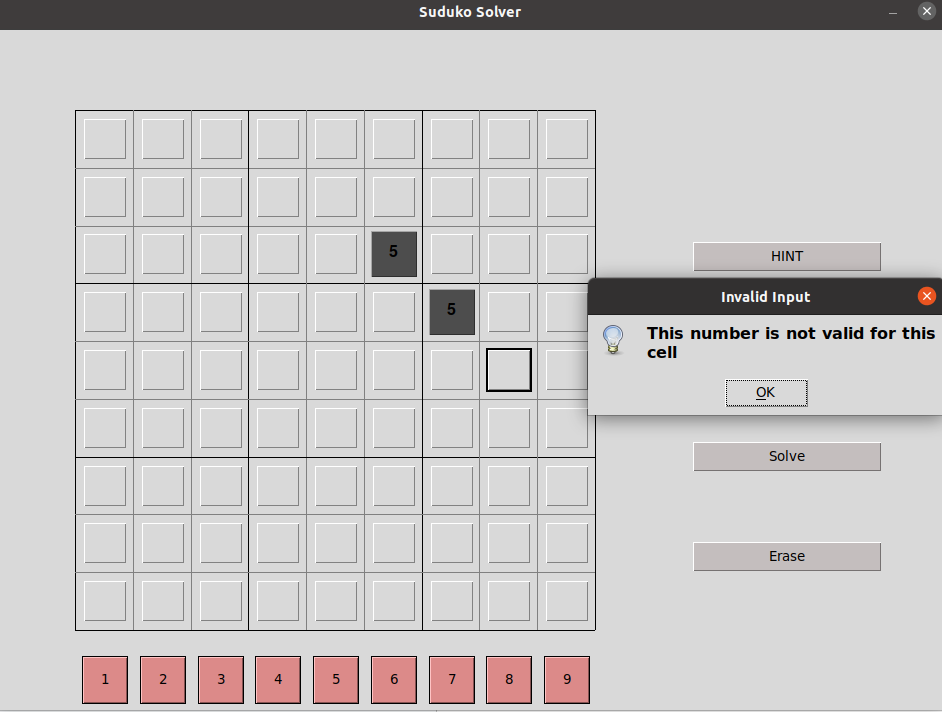
If the state entered is less than 81 characters, then this message will appear indicating why the input is invalid. —-> state = 1234

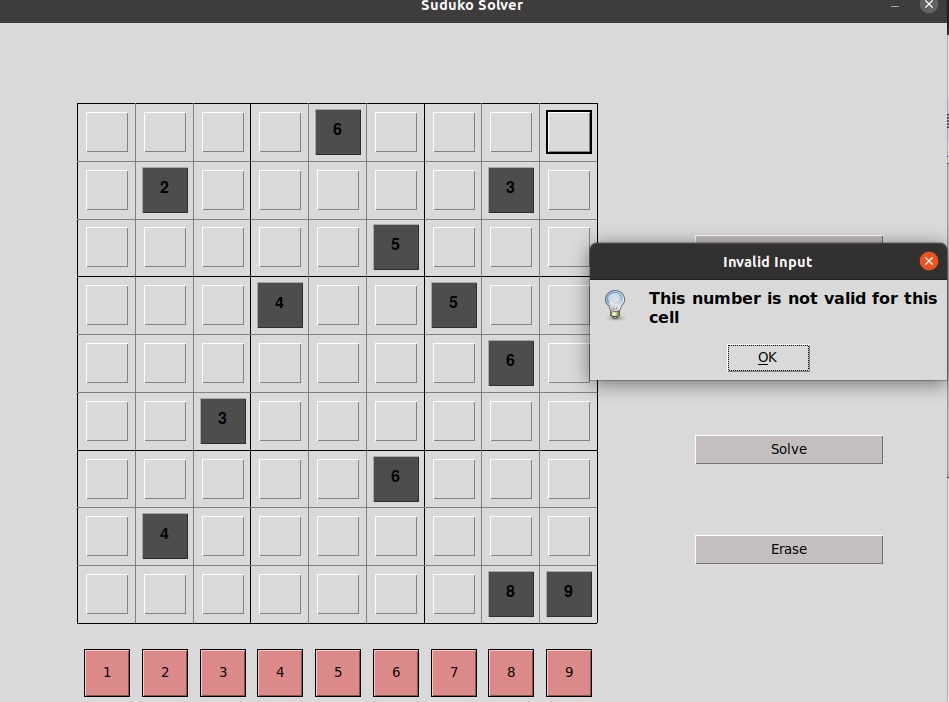
**If the state is invalid: —> state =** 123456789123456789123456789123456789123456789123456789123456789123456789123456789

**If the state is valid —> state entered:** 000000008180002300060057001070960000090704010000081040600240080004500093500000000

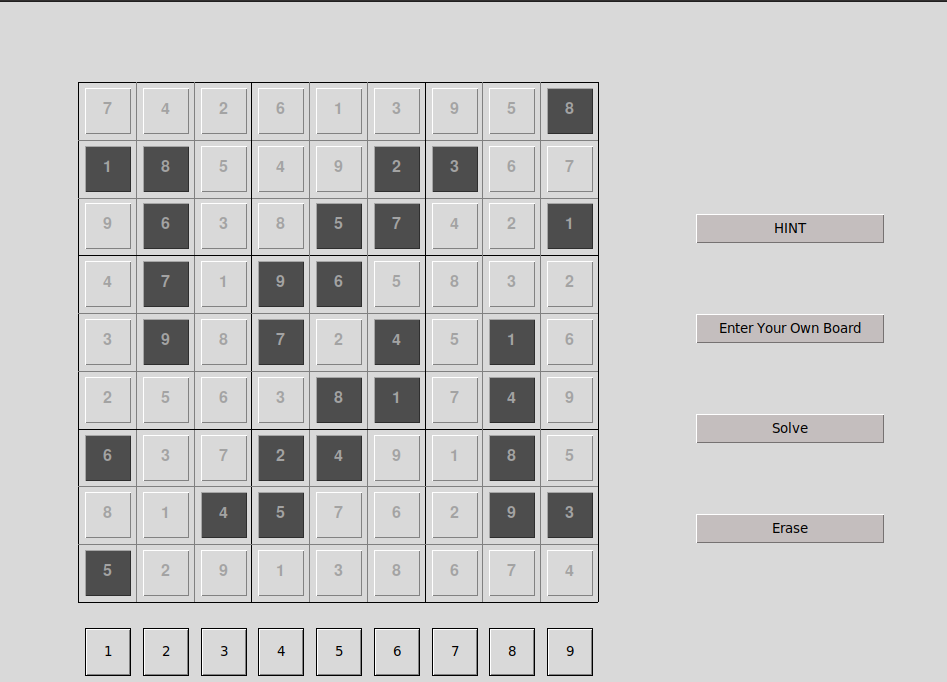
****

**Feature 2:**

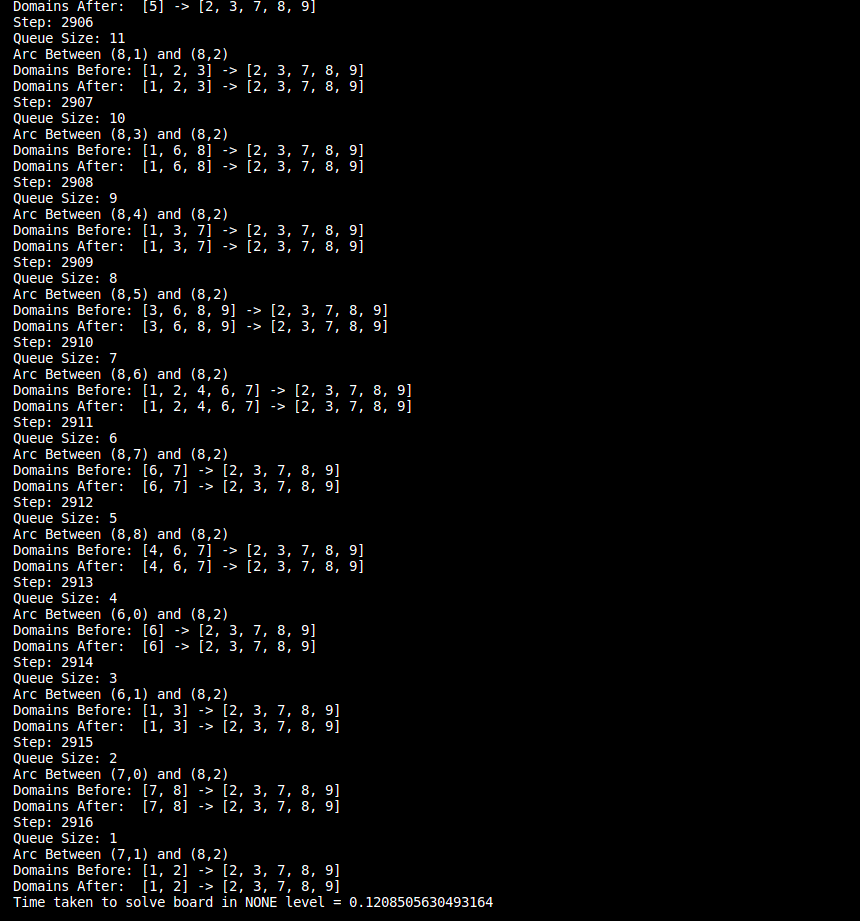
The user can enter his state using buttons. If in any step the board became invalid a message will pop up (In this example a 5 is violating the subgrid which contain a 5).

In this example, a 9 is violating another 9 in the same column.

Whenever a board is entered and is valid, the AI agent can will solve it if it is solvable.

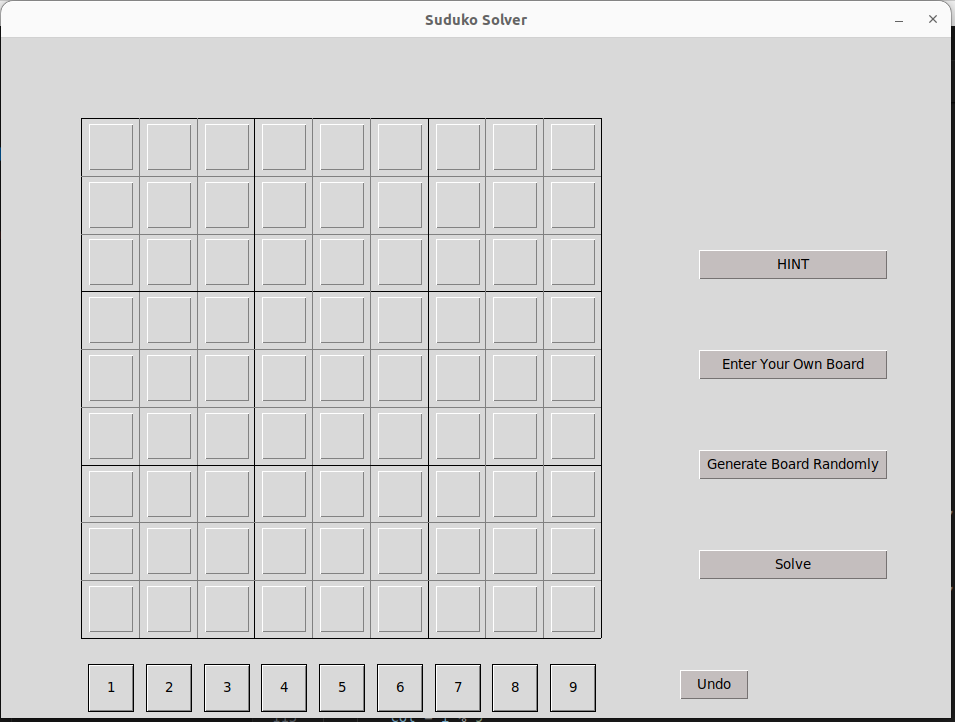


Solution steps, arc consistency, and time

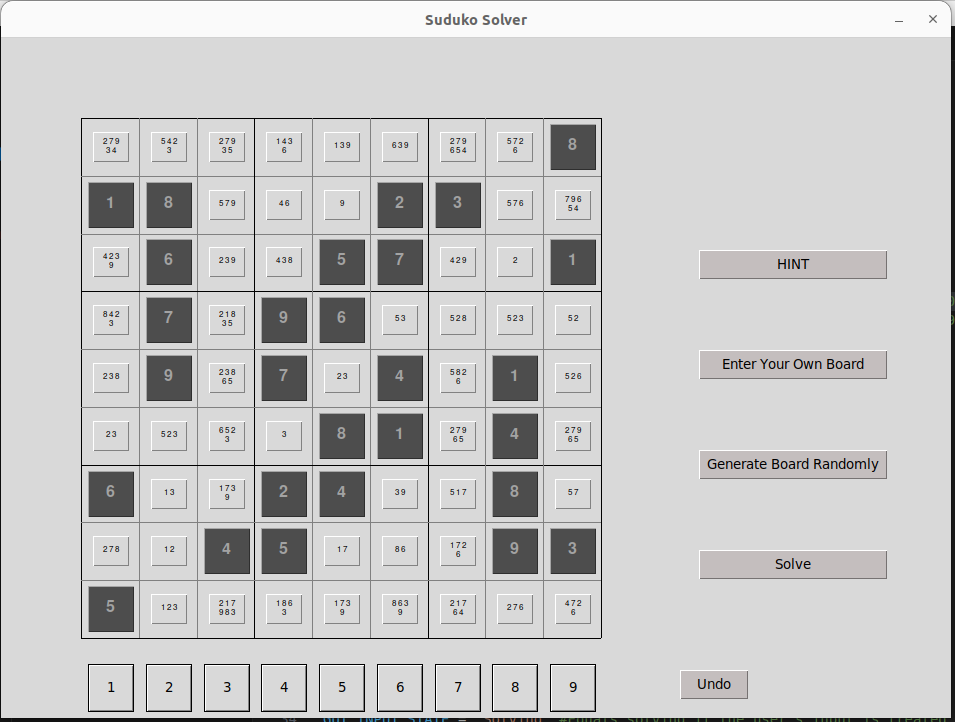


**Mode Three: Interactive Game**

In this mode the user can input a board and then start solving it or he can choose to generate a board with a certain level of difficulty



The user can also show hints to help him solve the puzzle and can undo the last step



If he tries to enter a value that is not possible in a certain cell a pop up message will appear.

In this case we tried entering number 9 in the very first cell.

