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

AYA RAGAA 7793
ALIA ESSAM 7810
JAIDAA MOHAMED 7360

SUDOKU AS CSP

Code:

```
from tkinter import messagebox

class SudokuSolver:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.size = 9  # Size of the Sudoku grid
        self.steps = []  # Store steps made during solving
        self.domains = {}  # Store domains of each cell

    def solve(self):
        if not self.is_valid():  # Validate the initial puzzle
            print("Invalid Sudoku puzzle.")
            return False, 0

         self.initializeDomains()

        if self.backtrack_solve():
            return self.puzzle, self.steps # Return solved puzzle, steps, and
domains
        else:
            print("No solution exists.")
            return False, 1
```

➔ **This function solves the sudoku puzzle using backtracking. First, it validates the board and then calls the backtrack_solve() to solve the puzzle. If there's a solution, it returns the steps and domain, else, it returns a flag and false. If the flag is 0, then the sudoku puzzle is invalid, but if the flag is 1, then there is no solution to the board.**

```
    def is_valid(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.puzzle[i][j] != 0:  # Check non-empty cells
                    num = self.puzzle[i][j]
                    self.puzzle[i][j] = 0  # Temporarily remove the number
                    if not self.is_safe(i, j, num):  # Check if it's safe
                        self.puzzle[i][j] = num  # Restore the number
                        return False
                    self.puzzle[i][j] = num  # Restore the number
        return True
```

```python
    def is_safe(self, row, col, num):
        return (
            self.is_valid_row(row, num)
            and self.is_valid_col(col, num)
            and self.is_valid_box(row - row % 3, col - col % 3, num)
        )

    def is_valid_row(self, row, num):
        return num not in self.puzzle[row]

    def is_valid_col(self, col, num):
        return num not in [self.puzzle[row][col] for row in range(self.size)]

    def is_valid_box(self, start_row, start_col, num):
        for i in range(3):
            for j in range(3):
                if self.puzzle[i + start_row][j + start_col] == num:
                    return False
        return True
```

- **The previous functions checks the validation of each cell. It iterates through each cell, temporarily removes the number, checks if it's safe to place the number in that cell, and then restores the number. If any violation of Sudoku rules is found, it returns False; otherwise, it returns True.**

- **Is_safe() checks if it safe to place num in the specified row, column, and grid by checking if num is used in any of the cells of the row, column, and grid. We achieve this by the 3 helper functions is_valid_row(), is_valid_column(), and is_valid_box().**

```python
def initializeDomains(self):
        for i in range(self.size):
            for j in range(self.size):
                domain = self.get_domain(i,j)
                self.domains[(i,j)] = domain

    def backtrack_solve(self):

        empty_cell = self.find_empty_cell()
        if not empty_cell:  # If there are no empty cells, puzzle is solved
            return True  # Return True
```

```
row, col = empty_cell

domain = self.domains[(row, col)].copy()
for num in domain:  # Try placing numbers 1 through 9
    dom = self.get_domain(row,col)
    self.puzzle[row][col] = num
    self.steps.append((row, col, num, dom))  # Record the step

    # Update domains after placing a number
    self.revise(row, col, num)

    no_empty_dom_flag ,rowe, cole = self.arc_consistency()
    if no_empty_dom_flag:
        if self.backtrack_solve():
            return True
    # If no solution found with the current number, backtrack
    self.steps.append((row, col, 0, []))  # Record the backtracking step
    self.puzzle[row][col] = 0  # Backtrack by resetting the cell value

    # Restore the original domain of the cell
    self.domains[(row, col)] = dom

    # Update domains of affected cells
    self.get_domains(row, col)
return False  # If no solution found from this point, return False
```

➔ **This is the core method for solving the puzzle using backtracking. It first finds an empty cell, and if no empty cell is found, it returns True indicating the puzzle is solved. Otherwise, it tries placing numbers in the empty cell and recursively calls itself. If a solution is found, it returns True; otherwise, it backtracks and tries a different number.**

```
def arc_consistency(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.puzzle[i][j] == 0:  # Consider only empty
cells
                    domain = self.get_domain(i, j)
                    if len(domain) == 0:  # If domain is empty,
inconsistency found
                        return False, i,j
        return True, 0, 0
```

```
    def revise(self, row, col, num):
        # Update domains after placing a number
        for i in range(self.size):
            # Update row domains
            self.domains[(row, i)].discard(num)
            # Update column domains
            self.domains[(i, col)].discard(num)

        # Determine the top-left cell of the 3x3 square
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)

        # Update 3x3 square domains
        for i in range(3):
            for j in range(3):
                self.domains[(start_row + i, start_col +
    j)].discard(num)
```

➔ **Arc consistency checks if there's an empty set in a cell's domain after adding a number to the puzzle and revise updates the row, column, and 3x3 subgrid domains of the updated cell.**

```
    def get_domain(self, row, col):

        domain = set(range(1, 10))  # Initialize domain with all numbers from 1
to 9
        for i in range(self.size):
            domain.discard(self.puzzle[row][i])  # Remove numbers in the same row
            domain.discard(self.puzzle[i][col])  # Remove numbers in the same
column
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)  # Top-left cell of
the box
        for i in range(3):
            for j in range(3):
                domain.discard(self.puzzle[start_row + i][start_col + j])  #
Remove numbers in the same box
        return domain

    def get_domains(self,row,col):
        for i in range(self.size):
            self.domains[(i,col)] = self.get_domain(i,col)

        for j in range(self.size):
            self.domains[(row,j)] = self.get_domain(row,j)
```

```
        # Determine the top-left cell of the 3x3 square
        start_row, start_col = 3 * (row // 3), 3 * (col // 3)

        # Update 3x3 square domains
        for i in range(3):
            for j in range(3):
                    self.domains[(start_row + i, start_col + j)]=
self.get_domain(start_row + i, start_col + j)
```

- **get_domain() method returns the domain (possible values) for the cell at the specified row and column. It considers numbers not present in the same row, column, or 3x3 box.**
- **get_domains() method takes row and column(cell's position) and iterates over the cell. It updates the cell's domains that are in same row, column, and 3x3 subgrid and it's used in backtracking.**

```
    def find_empty_cell(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.puzzle[i][j] == 0:
                    return i, j
        return None
```

- **This method searches for an empty cell (cell with value 0) in the puzzle. If found, it returns the row and column indices of the empty cell; otherwise, it returns None.**

So, the class of sudoku solver is the main class for solving the board.

```
import tkinter as tk
import copy

class SudokuGUI:
    def __init__(self, master, puzzle):
        self.master = master
        self.puzzle = puzzle
        self.initial = copy.deepcopy(puzzle)
        self.size = 9

        self.canvas = tk.Canvas(self.master, width=360, height=360)
        self.canvas.pack()

        self.draw_grid()
        self.draw_puzzle()
```

```
        self.solve_button = tk.Button(self.master, text="Solve",
command=self.solve)
        self.solve_button.pack()

    def draw_grid(self):
        for i in range(10):
            width = 2 if i % 3 == 0 else 1
            self.canvas.create_line(i * 40, 0, i * 40, 360, width=width)
            self.canvas.create_line(0, i * 40, 360, i * 40, width=width)

    def draw_puzzle(self):
        for i in range(self.size):
            for j in range(self.size):
                if self.puzzle[i][j] != 0:
                    x = j * 40 + 20
                    y = i * 40 + 20
                    self.canvas.create_text(x, y, text=str(self.puzzle[i][j]),
font=('Arial', 16, 'bold'))
```

➔ **The previous functions are all used to draw the GUI.**

```
    def solve(self):
        solver = SudokuSolver(self.puzzle)
        solution, steps = solver.solve()
        if solution:
            self.puzzle = solution
            for row in solution:
                print(row)
            self.clear_canvas()
            self.draw_grid()
            self.draw_puzzle()
            self.master.after(1000, lambda: self.display_solution(solution,
steps))
        elif  not steps:
            messagebox.showerror("Warning", "Invalid Sudoku Puzzle")
            self.master.withdraw()  # Hide the current GUI
            self.return_to_mode_selection()
        else:
            messagebox.showerror("Warning", "No Solution Exists !")
            self.master.withdraw()  # Hide the current GUI
            self.return_to_mode_selection()
```

➔ **This function creates a SudokuSolver object to initialize the board and update the GUI after each update. It then solves the board. If there's a**

solution, then it will be displayed. Else, if the steps returned is 0 which means there's no steps, then it means that the sudoku is invalid. If there's no solution and 0 steps, then there's no solution for the board.

```
    def clear_canvas(self):
        self.canvas.delete(tk.ALL)


    def display_solution(self, solution, steps):
        self.master.withdraw()  # Hide the current GUI
        solution_gui = SolutionGUI(self.initial, solution, steps)
        solution_gui.show()

    def return_to_mode_selection(self):
        mode_selection_gui = ModeSelectionGUI()
        mode_selection_gui.root.mainloop()
```

➔ **The method then hides the solution GUI and displays the GUI with eah step update. Then, it returns to the window of the mode's selection.**

```
class SolutionGUI:
    def __init__(self, initial_puzzle, solved_puzzle, steps):
        self.initial_puzzle = initial_puzzle
        self.solved_puzzle = solved_puzzle
        self.steps = steps
        self.current_puzzle = [row[:] for row in initial_puzzle]  # Copy of the
initial puzzle
        self.size = 9

        self.root = tk.Tk()
        self.root.title("Solution Steps")

        self.canvas = tk.Canvas(self.root, width=360, height=360)
        self.canvas.pack()

        self.draw_grid()
        self.display_steps()

    def draw_grid(self):
        for i in range(10):
            width = 2 if i % 3 == 0 else 1
            self.canvas.create_line(i * 40, 0, i * 40, 360, width=width)
            self.canvas.create_line(0, i * 40, 360, i * 40, width=width)
```

```
    def get_domain(self, row, col):
        # Get the domain of the cell (row, col)
        if self.current_puzzle[row][col] != 0:
            domain = (self.current_puzzle[row][col])
        else:
            domain = set(range(1, 10))
            for i in range(self.size):
                domain.discard(self.current_puzzle[row][i])  # Remove numbers in
the same row
                domain.discard(self.current_puzzle[i][col])  # Remove numbers in
the same column
            start_row, start_col = 3 * (row // 3), 3 * (col // 3)  # Top-left
cell of the box
            for i in range(3):
                for j in range(3):
                    domain.discard(self.current_puzzle[start_row + i][start_col +
j])  # Remove numbers in the same box
        return domain



    def display_steps(self):
        for step_index, step in enumerate(self.steps):
            row, col, num, dom = step
            # Remove any existing text in the cell
            text_id = f"step_cell_{row}_{col}"
            self.canvas.delete(text_id)
            domain = []
            for i in range(9):
                domain.append(self.get_domain(row,i))
            if num != 0:
                self.clear_canvas()
                self.draw_grid()
                self.draw_puzzle(self.current_puzzle)  # Display the current
state of the puzzle
                x = col * 40 + 20
                y = row * 40 + 20
                self.canvas.create_text(x, y, text=str(num), font=('Arial', 16,
'bold'), fill='red', tags=text_id)
                self.current_puzzle[row][col] = num  # Update the current puzzle
state

                # Display the domain of the cell below the grid
                domain_text = ", ".join(str(num) for num in dom)
```

```
                self.canvas.create_text(x, y + 20, text=domain_text,
font=('Arial', 10), fill='blue')

                self.root.update()

                print("Domain of this row")
                cell = 1

                for x in domain:
                    print(f"Cell ({row} , {cell}) =", end=" ")
                    print(x)
                    cell += 1

                print("----------------------")
                print()
                self.root.after(5000)
            else:
                self.current_puzzle[row][col] = num  # Update the current puzzle
state


        self.root.after(1000, self.return_to_mode_selection)



    def draw_puzzle(self, puzzle):
        for i in range(self.size):
            for j in range(self.size):
                if puzzle[i][j] != 0:
                    x = j * 40 + 20
                    y = i * 40 + 20
                    self.canvas.create_text(x, y, text=str(puzzle[i][j]),
font=('Arial', 16, 'bold'))

    def show(self):
        self.root.mainloop()

    def clear_canvas(self):
        self.canvas.delete(tk.ALL)

    def return_to_mode_selection(self):
        self.root.destroy()  # Close the current GUI
        mode_selection_gui = ModeSelectionGUI()
        mode_selection_gui.root.mainloop()
```

```python
import random
```

- **get_domain() returns the possible values of each cell's domain. Then, it iterates through the solution steps, updates the puzzle state on the canvas, displays the current step's domain, and prints the domain for each cell in the row. It schedules itself to run after a delay to simulate step-by-step display. All other functions are used for the GUI display of the steps of backtracking and domain update.**

```python
def generate_valid_puzzle(flag):
    # Start with an empty puzzle
    puzzle = [[0] * 9 for _ in range(9)]

    # Shuffle numbers 1 through 9 randomly
    numbers = list(range(1, 10))
    random.shuffle(numbers)

    # Fill the puzzle using a Sudoku solver with shuffled numbers
    for i in range(9):
        for j in range(9):
            puzzle[i][j] = numbers[(3*(i%3) + i//3 + j) % 9]

    # Fill the puzzle using a Sudoku solver
    solver = SudokuSolver(puzzle)
    solver.backtrack_solve()

    # Randomly clear cells in the puzzle
    num_cells_to_clear = random.randint(50, 70)
    cells_to_clear = random.sample(range(81), num_cells_to_clear)

    if flag == 1:
        num_cells_to_clear = 25

    if flag == 2:
        num_cells_to_clear = 45

    if flag == 3:
        num_cells_to_clear = 65

    for cell_index in cells_to_clear:
        row = cell_index // 9
        col = cell_index % 9
        puzzle[row][col] = 0
```

```
    return puzzle
```

- **This function generates a valid Sudoku puzzle by first creating a filled Sudoku grid with a random arrangement of numbers, then solving the grid using a Sudoku solver. After that, it randomly clears cells to create the puzzle. The number of cells to clear depends on the difficulty level specified by the `flag` argument.**

- **The `SudokuGUI` class handles the user interface for the Sudoku puzzle, allowing the user to interact with the puzzle and solve it. The `SolutionGUI` class displays the solution steps in a separate window, simulating the solving process step by step. The `generate_valid_puzzle` function generates valid Sudoku puzzles with varying levels of difficulty. Together, these components provide a complete Sudoku solving and puzzle generation system with a graphical user interface.**

```
class ModeSelectionGUI:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("Mode Selection")

        self.mode_label = tk.Label(self.root, text="Select a mode:")
        self.mode_label.pack()

        self.mode1_button = tk.Button(self.root, text="Mode 1",
command=self.mode1)
        self.mode1_button.pack()

        self.mode2_button = tk.Button(self.root, text="Mode 2",
command=self.mode2)
        self.mode2_button.pack()

        self.mode3_button = tk.Button(self.root, text="Mode 3",
command=self.mode3)
        self.mode3_button.pack()

    def mode1(self):
        self.root.destroy()  # Close the current GUI
        main(mode=1)  # Start the main function with mode 1

    def mode2(self):
        self.root.destroy()  # Close the current GUI
```

```
        main(mode=2)  # Start the main function with mode 2

    def mode3(self):
        self.root.destroy()  # Close the current GUI
        main(mode=3)  # Start the main function with mode 3
```

➔ **This class is responsible for the GUI of mode's selection.**

```
class SudokuInputGUI:
    def __init__(self, master, puzzle):
        self.master = master
        self.puzzle = puzzle
        self.size = 9

        # Calculate the total width and height of the canvas based on the grid
size
        canvas_width = canvas_height = 360
        cell_size = canvas_width // self.size

        self.canvas = tk.Canvas(self.master, width=canvas_width,
height=canvas_height)
        self.canvas.pack()

        self.draw_grid(cell_size)

        self.save_button = tk.Button(self.master, text="Save",
command=self.save_puzzle)
        self.save_button.pack()

    def draw_grid(self, cell_size):
        for i in range(self.size + 1):
            width = 2 if i % 3 == 0 else 1
            self.canvas.create_line(i * cell_size, 0, i * cell_size, self.size *
cell_size, width=width)
            self.canvas.create_line(0, i * cell_size, self.size * cell_size, i *
cell_size, width=width)

    def save_puzzle(self):
        # Save the puzzle from user input
        for i in range(self.size):
            for j in range(self.size):
                entry = self.entries[i][j].get()
                if entry.isdigit() and 1 <= int(entry) <= 9:
                    self.puzzle[i][j] = int(entry)
```

```
        # Close the input GUI
        self.master.destroy()

    def show(self):
        cell_size = 360 // self.size
        self.entries = [[None for _ in range(self.size)] for _ in
range(self.size)]
        for i in range(self.size):
            for j in range(self.size):
                x = j * cell_size + cell_size // 2
                y = i * cell_size + cell_size // 2
                self.entries[i][j] = tk.Entry(self.master, width=3,
font=('Arial', 16, 'bold'))
                self.entries[i][j].place(x=x, y=y, anchor="center")

        self.master.mainloop()
```

➔ **This class is responsible for taking the input board from the user. Once the board is entered into the GUI, the board is saved and validated. If the board is valid then the GUI of the saved board is closed, and the puzzle is being sent to the SudokuSolver to start solving it exactly like mode 1.**

```
class UserInteractiveSudokuGUI:
    def __init__(self, master, initial_puzzle):
        self.master = master
        self.initial_puzzle = initial_puzzle
        self.current_puzzle = [row[:] for row in initial_puzzle]  # Copy of the
initial puzzle
        self.size = 9
        self.entries = [[None for _ in range(self.size)] for _ in
range(self.size)]  # Store references to the entry widgets

        self.canvas = tk.Canvas(self.master, width=360, height=360)
        self.canvas.pack()

        self.draw_grid()
        self.draw_puzzle()

        self.enter_button = tk.Button(self.master, text="Enter",
command=self.check_solvable)
        self.enter_button.pack()

    def draw_grid(self):
        for i in range(10):
```

```
            width = 2 if i % 3 == 0 else 1
            self.canvas.create_line(i * 40, 0, i * 40, 360, width=width)
            self.canvas.create_line(0, i * 40, 360, i * 40, width=width)

    def draw_puzzle(self):
        #self.entries = [[None for _ in range(self.size)] for _ in
range(self.size)]
        for i in range(self.size):
            for j in range(self.size):
                x = j * 40 + 20
                y = i * 40 + 20
                if self.current_puzzle[i][j] == 0:
                    # Create an entry for user input
                    entry = tk.Entry(self.master, width=3, font=('Arial', 16,
'bold'))
                    entry.place(x=x, y=y, anchor="center")
                    setattr(self, f"entry_{i}_{j}", entry)  # Store the entry in
an attribute with a specific name
                    self.entries[i][j] = entry  # Also store the entry in the
list
                else:
                    # Display the initial value
                    self.canvas.create_text(x, y,
text=str(self.current_puzzle[i][j]), font=('Arial', 16, 'bold'))


    def check_solvable(self):
        previous_puzzle = copy.deepcopy(self.current_puzzle)
        solver = SudokuSolver(self.current_puzzle)
        # Update the current puzzle with user input
        for i in range(self.size):
            for j in range(self.size):
                entry = getattr(self, f"entry_{i}_{j}", None)
                if entry:
                    if entry.get():
                        value = entry.get()
                        if value.isdigit() and 1 <= int(value) <= 9:
                            self.current_puzzle[i][j] = int(value)
                        else:
                            # If any entry is not a valid digit, show an error
message
                            messagebox.showerror("Error", "Invalid input!")
                            return

        current = copy.deepcopy(self.current_puzzle)
```

```
    # Check if the current puzzle is valid
    if not solver.is_valid():
        messagebox.showerror("Error", "Invalid Sudoku puzzle!")
        self.current_puzzle = copy.deepcopy(previous_puzzle)
        return

    # Try solving the puzzle
    solution, steps = solver.solve()
    if solution:
        if not any(0 in row for row in current):
            messagebox.showinfo("Congratulations", "Great Job!")
            self.return_to_mode_selection()

        #messagebox.showinfo("Solvable", "Puzzle is solvable!")
        self.current_puzzle = copy.deepcopy(current)
        self.reset_gui()
    else:
        messagebox.showerror("Unsolvable", "Puzzle is unsolvable!")
        self.current_puzzle = copy.deepcopy(previous_puzzle)
```

➔ **First, we keep a copy of the current puzzle and store it in previous_puzzle. We create an object of Sudoku solver to check each time if the move we made is valid and makes the board solvable or not. Iterates over each cell of the puzzle grid. For each cell, it retrieves the corresponding entry widget (if it exists) where the user might have entered a value. If the entry widget contains a value, it validates whether the value is a digit between 1 and 9. If it's valid, the value is updated in the current_puzzle attribute. If the value is not valid (not a digit or not within the range), it displays an error message and returns from the function.**

```
def reset_gui(self):

        # Update the existing entry widgets with the new puzzle values
    for i in range(self.size):
        for j in range(self.size):
            entry = getattr(self, f"entry_{i}_{j}", None)
            if entry:
                if self.current_puzzle[i][j] == 0:
                    entry.config(state="normal")  # Enable the entry widget
                else:
                    entry.delete(0, "end")  # Clear the entry widget
```

```
                        entry.insert(0, str(self.current_puzzle[i][j]))  # Insert
the new value
                        entry.config(state="readonly")  # Disable the entry
widget for initial values

        # Clear the entries list
        self.entries = [[None for _ in range(self.size)] for _ in
range(self.size)]

    def return_to_mode_selection(self):
        self.master.destroy()  # Close the current GUI
        mode_selection_gui = ModeSelectionGUI()
        mode_selection_gui.root.mainloop()




def mode3_handler(initial_puzzle):
    root = tk.Tk()
    root.title("User Interactive Sudoku")

    user_sudoku_gui = UserInteractiveSudokuGUI(root, initial_puzzle)
    root.mainloop()

import tkinter.simpledialog as simpledialog

# Inside the function where you print "Mode 3 selected"
def select_difficulty():
    root = tk.Tk()
    root.withdraw()  # Hide the root window

    # Prompt the user to select the difficulty level
    difficulty = simpledialog.askstring("Difficulty Selection", "Choose
difficulty level (easy, medium, hard):")

    # Check the selected difficulty level
    if difficulty.lower() == "easy":
        print("Easy mode selected")
        return 1
        # Handle easy mode
    elif difficulty.lower() == "medium":
        print("Medium mode selected")
        return 2
        # Handle medium mode
    elif difficulty.lower() == "hard":
```

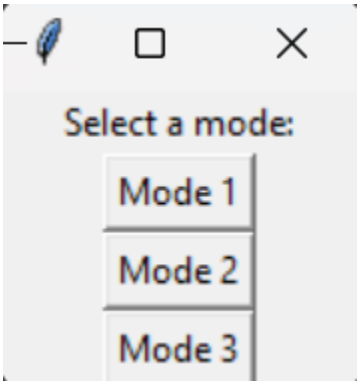
```
        print("Hard mode selected")
        return 3
        # Handle hard mode
    else:
        print("Invalid difficulty level selected")
```

➔ **This part is for the GUI and logic for choosing the difficulty level.**

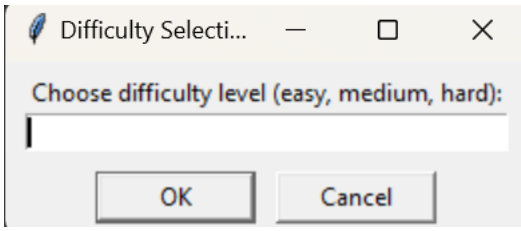
```
def main(mode):
    if mode == 1:

        print("Mode 1 selected")
        #puzzle = generate_valid_puzzle(0)
        mode = select_difficulty()
        puzzle = generate_valid_puzzle(mode)
        root = tk.Tk()
        root.title("Sudoku Solver")
        SudokuGUI(root, puzzle)
        root.mainloop()
    elif mode == 2:
        print("Mode 2 selected")
                # Create an empty puzzle
        puzzle = [[0] * 9 for _ in range(9)]

        # Open GUI for the user to input initial state
        root = tk.Tk()
        root.title("Sudoku Initial State Input")

        sudoku_input_gui = SudokuInputGUI(root, puzzle)
        sudoku_input_gui.show()
        root = tk.Tk()
        root.title("Sudoku Solver")
        SudokuGUI(root, puzzle)
        root.mainloop()

    elif mode == 3:

        print("Mode 3 selected")
        mode = select_difficulty()
        puzzle = generate_valid_puzzle(mode)
        mode3_handler(puzzle)

if __name__ == "__main__":
    mode_selection_gui = ModeSelectionGUI()
    mode_selection_gui.root.mainloop()
```

Sample Runs:

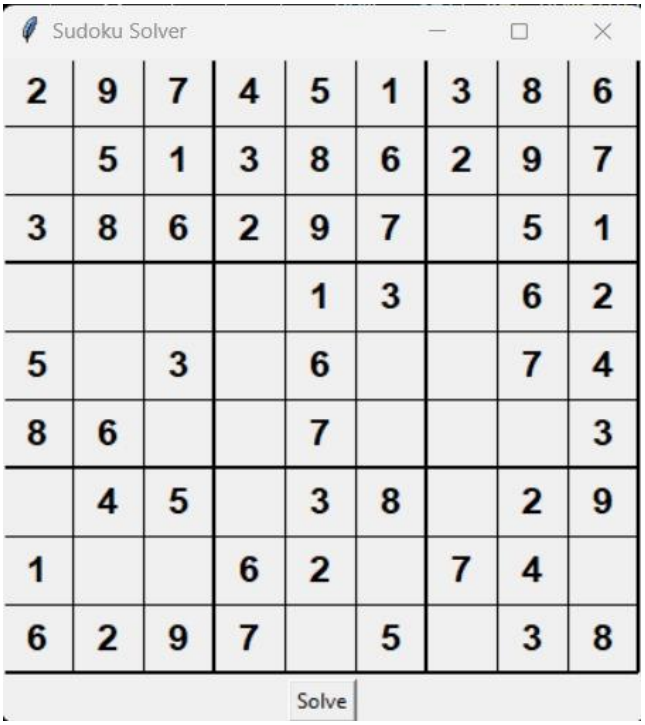
a) Choosing the Mode:



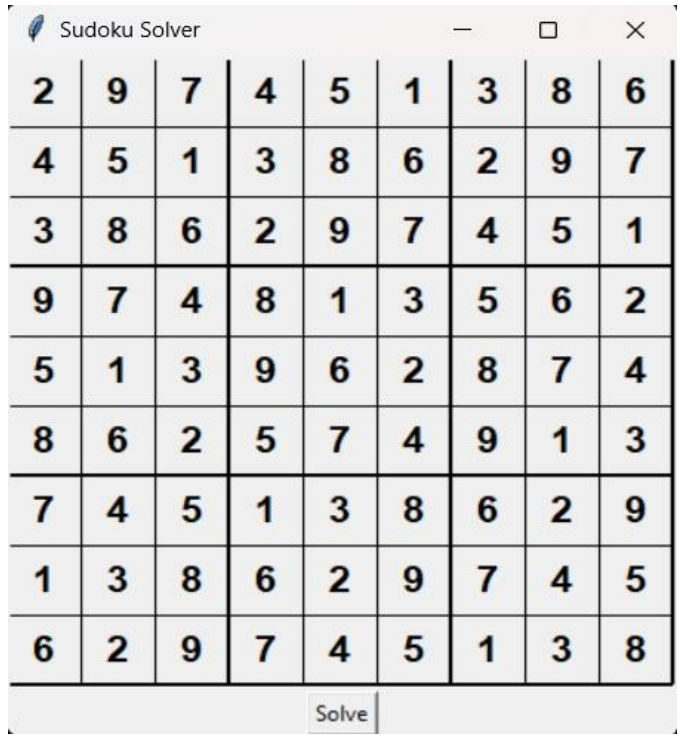
i) Mode 1: Randomly Generated Board
Select the Difficulty:



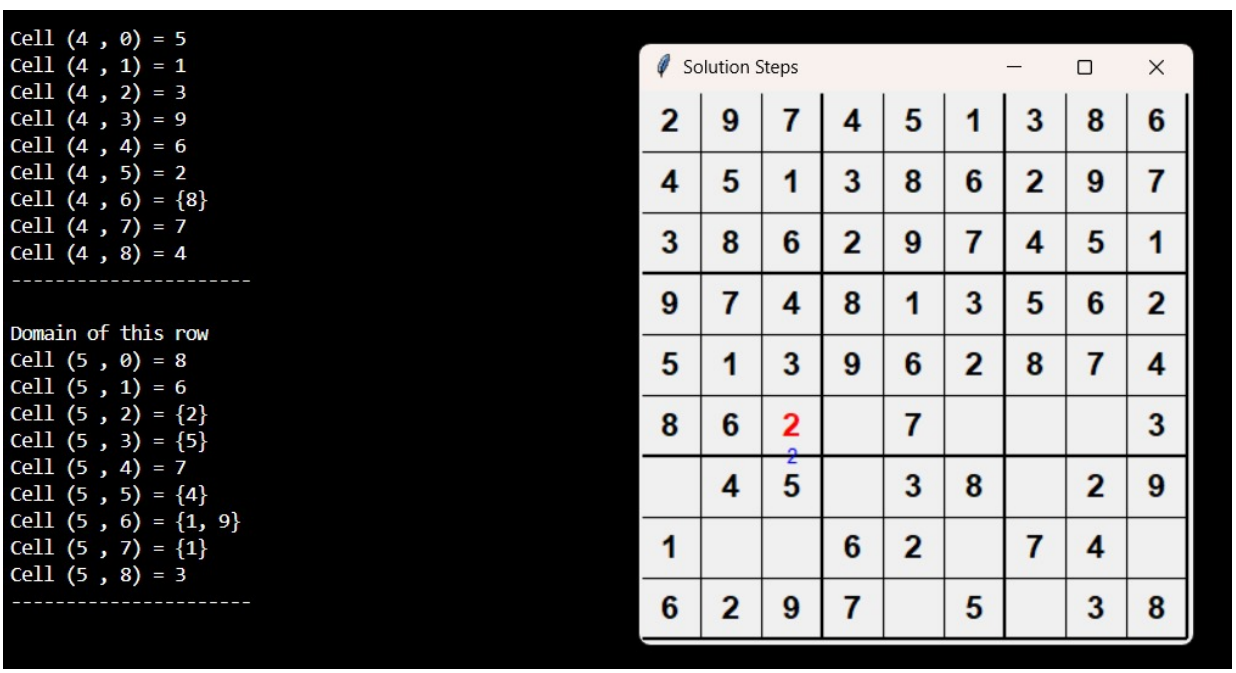
a) Easy:



Initial Board

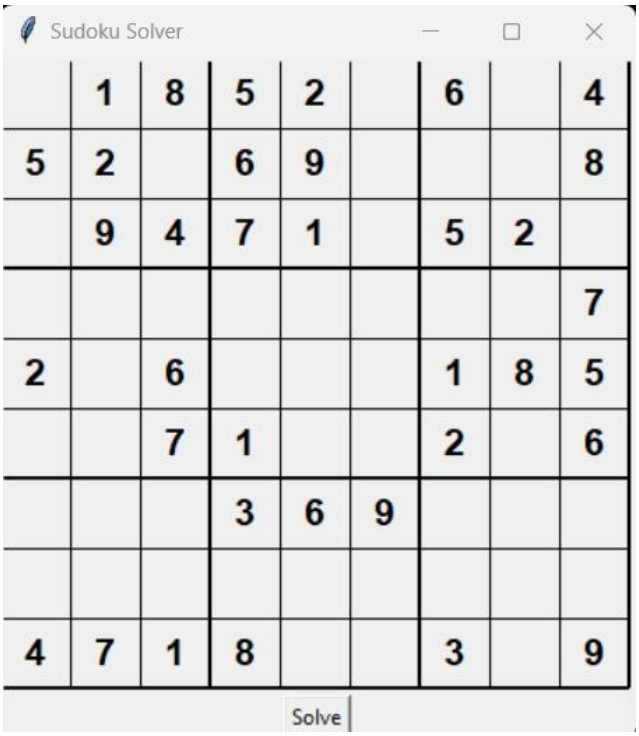


The Final Solution



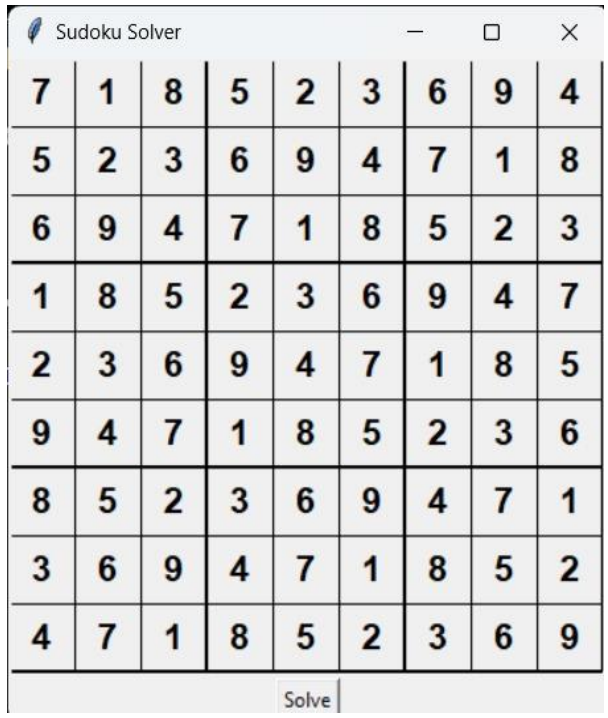
Time Taken: 0.999689 msec

b) Medium:



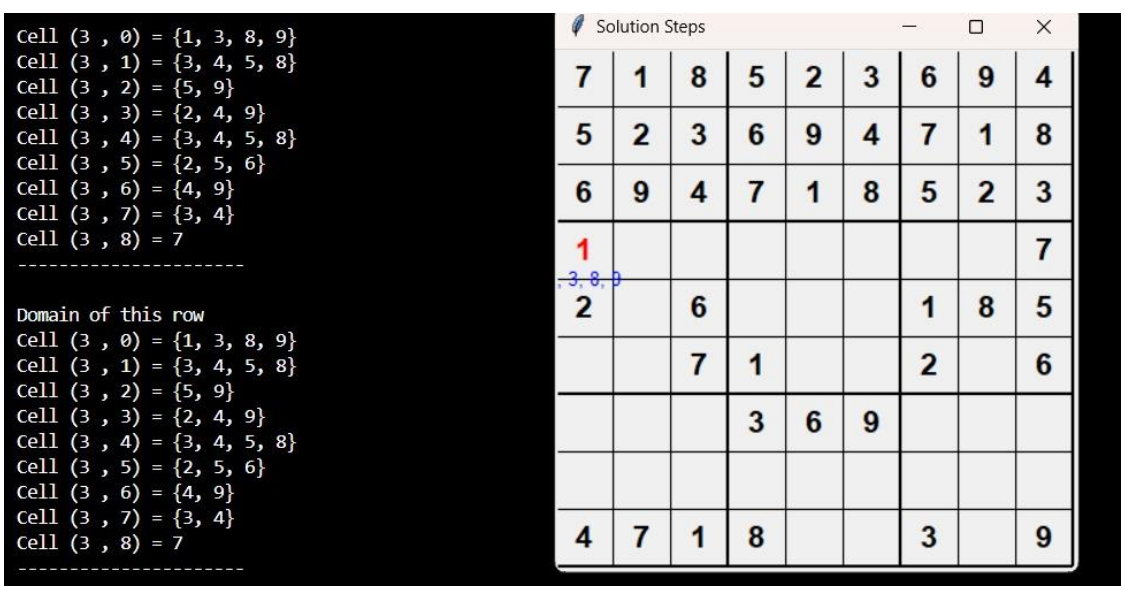
	1	8	5	2		6		4
5	2		6	9				8
	9	4	7	1		5	2	
								7
2		6				1	8	5
		7	1			2		6
			3	6	9			
4	7	1	8			3		9

Initial Board



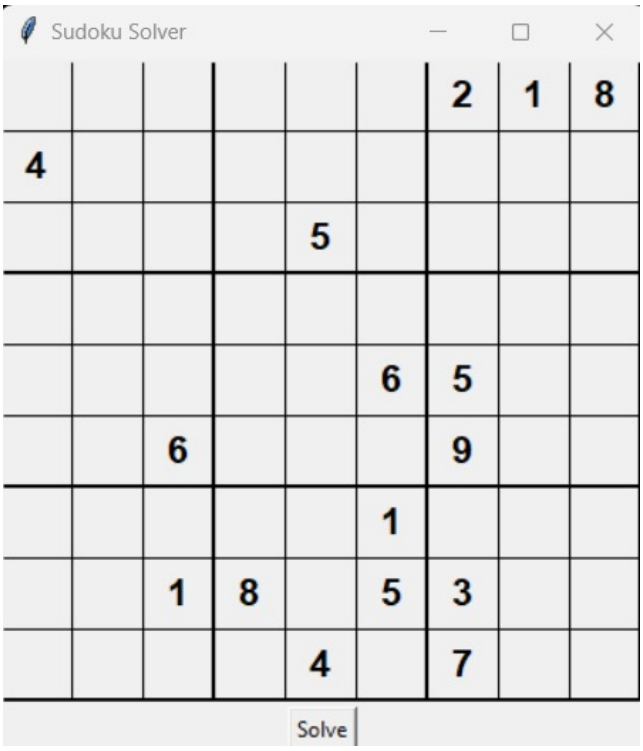
7	1	8	5	2	3	6	9	4
5	2	3	6	9	4	7	1	8
6	9	4	7	1	8	5	2	3
1	8	5	2	3	6	9	4	7
2	3	6	9	4	7	1	8	5
9	4	7	1	8	5	2	3	6
8	5	2	3	6	9	4	7	1
3	6	9	4	7	1	8	5	2
4	7	1	8	5	2	3	6	9

Final Board



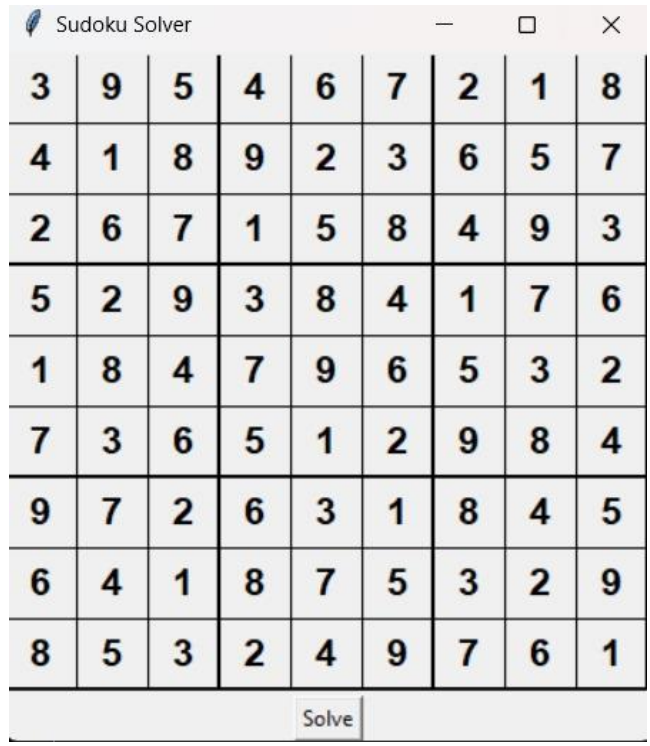
Time Taken: 5.466461 msec

c) Hard:



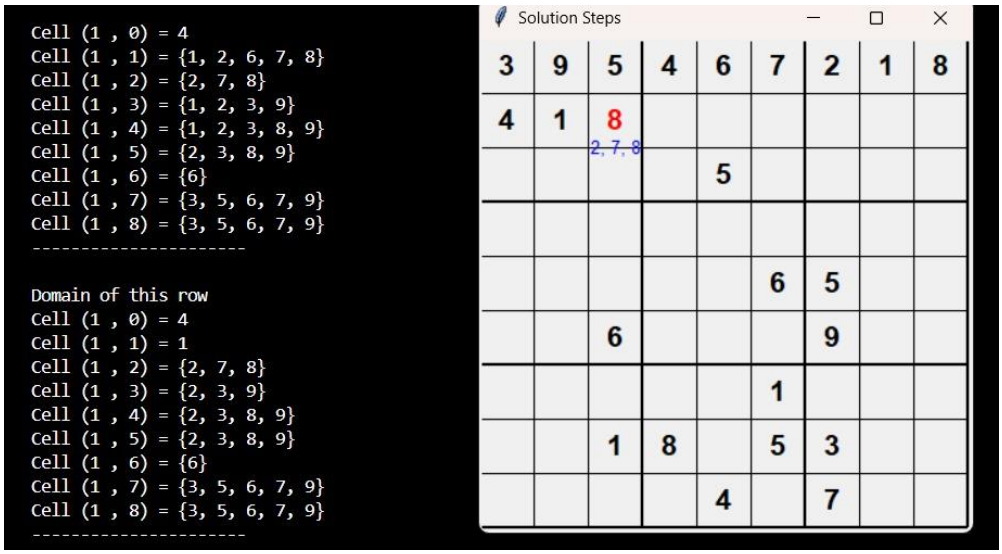
						2	1	8
4								
				5				
					6	5		
		6				9		
					1			
		1	8		5	3		
				4		7		

Initial Board



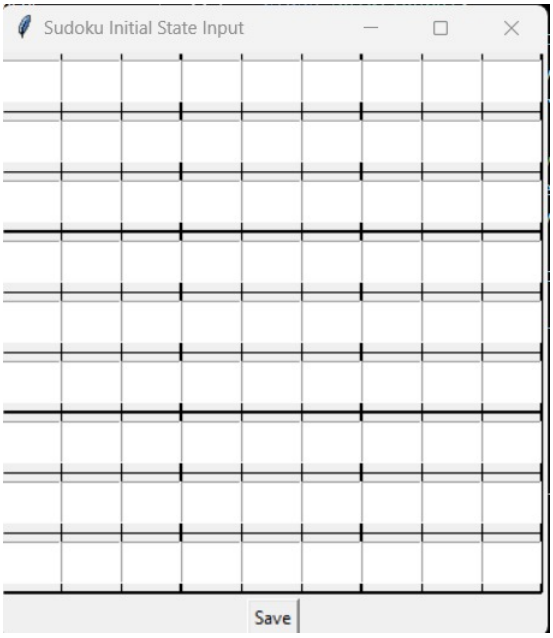
3	9	5	4	6	7	2	1	8
4	1	8	9	2	3	6	5	7
2	6	7	1	5	8	4	9	3
5	2	9	3	8	4	1	7	6
1	8	4	7	9	6	5	3	2
7	3	6	5	1	2	9	8	4
9	7	2	6	3	1	8	4	5
6	4	1	8	7	5	3	2	9
8	5	3	2	4	9	7	6	1

Final Board



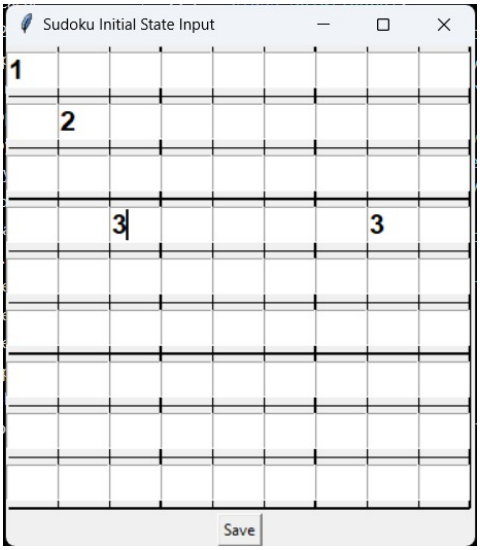
Time Taken: 69.773197

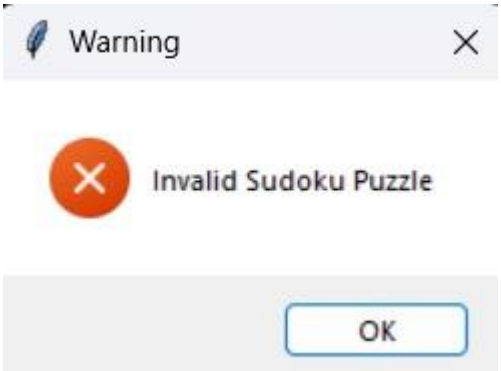
ii) Mode 2: User's Input Board



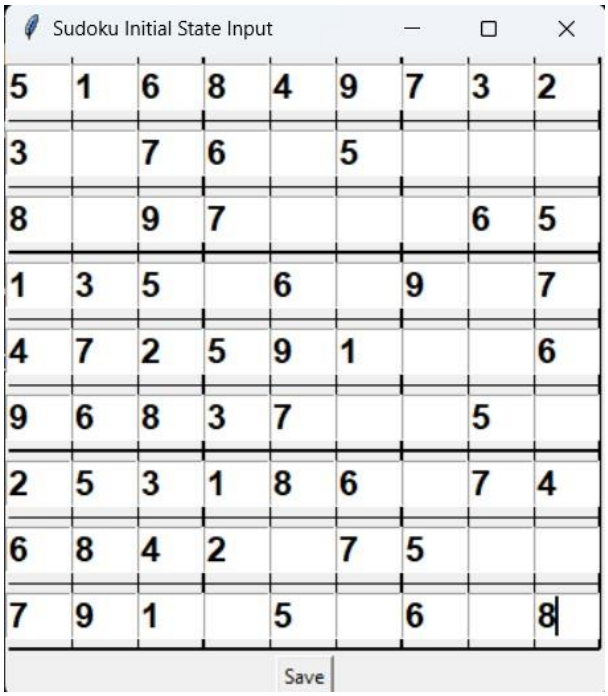
Empty Board to Fill

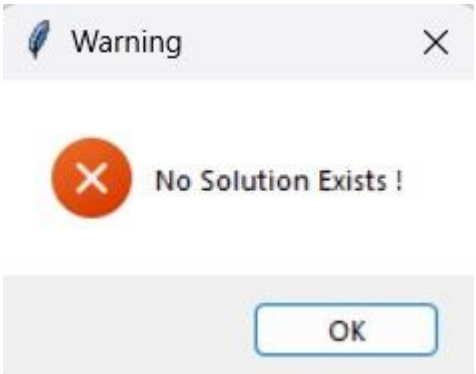
a) State 1: Invalid Input Board



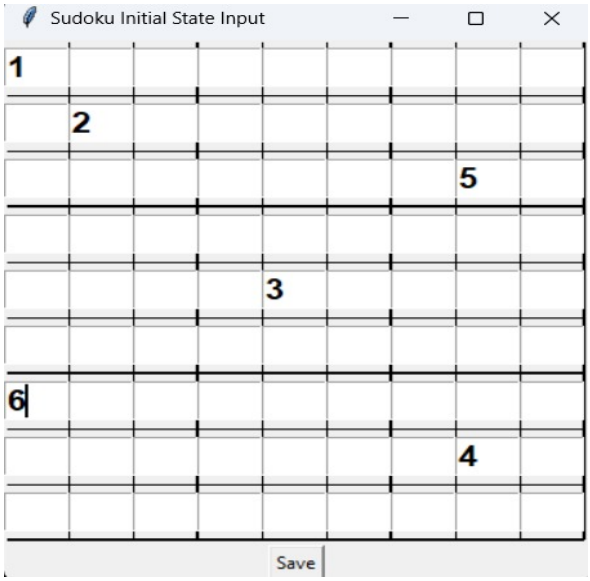


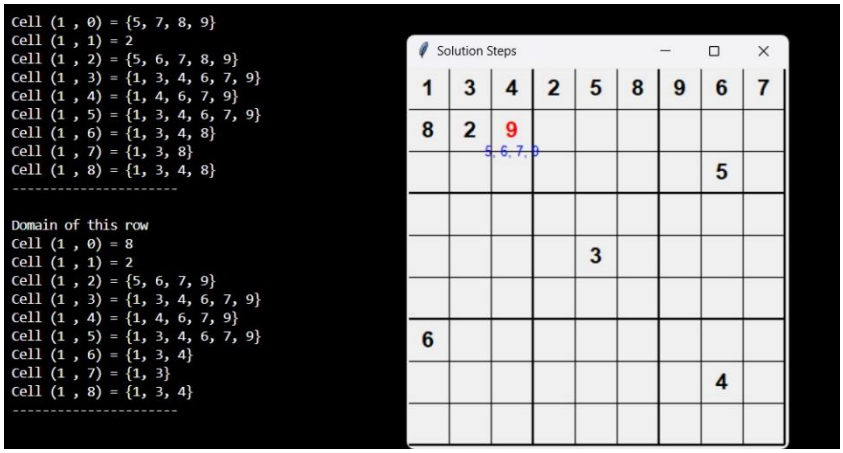
b) State 2: No Solution





c) State 3: Solvable Board



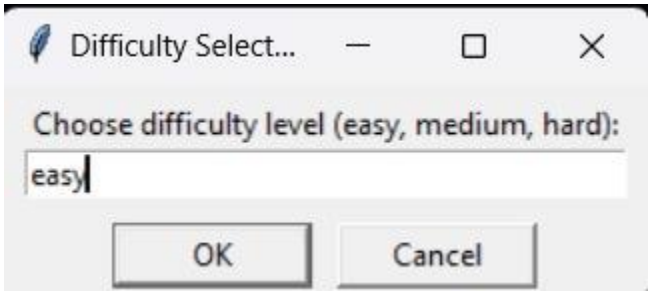


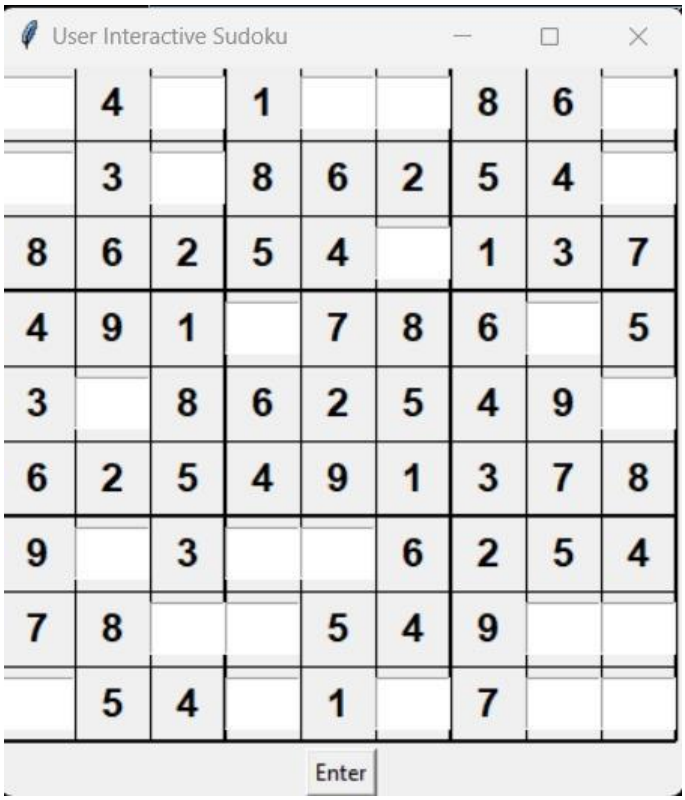
Time Taken: 76.752663 msec

iii) Mode 3: User Interactive

Has 3 difficulty levels: Easy, Medium, Hard

Example: Easy

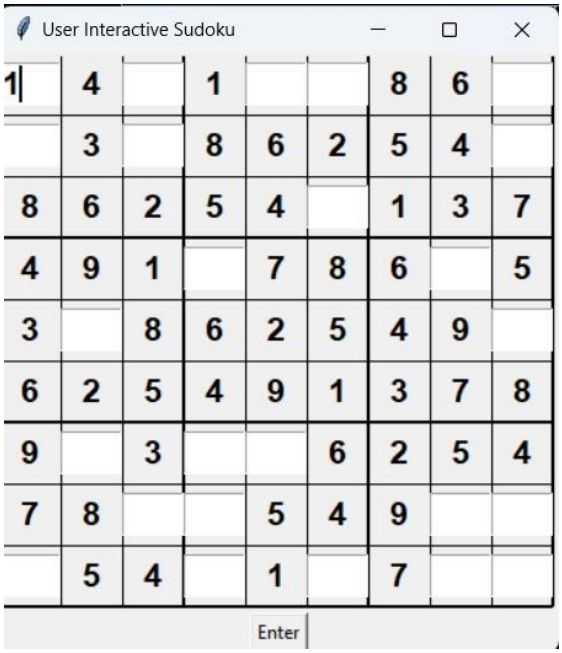


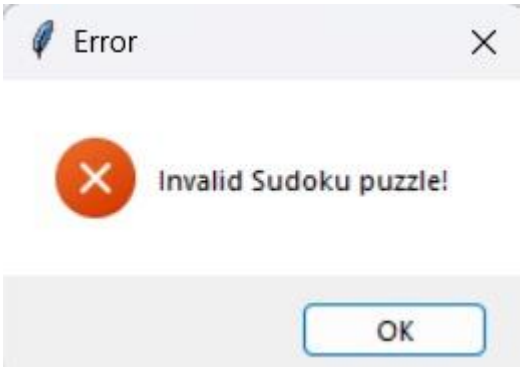


	4		1			8	6	
	3		8	6	2	5	4	
8	6	2	5	4		1	3	7
4	9	1		7	8	6		5
3		8	6	2	5	4	9	
6	2	5	4	9	1	3	7	8
9		3			6	2	5	4
7	8			5	4	9		
	5	4		1		7		

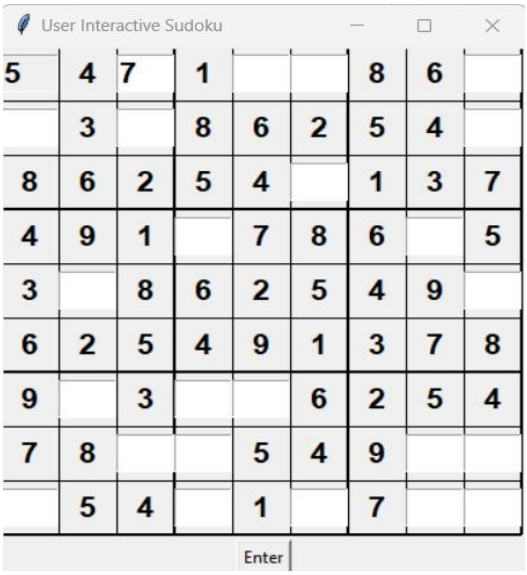
Initial Board

a) Invalid Move:

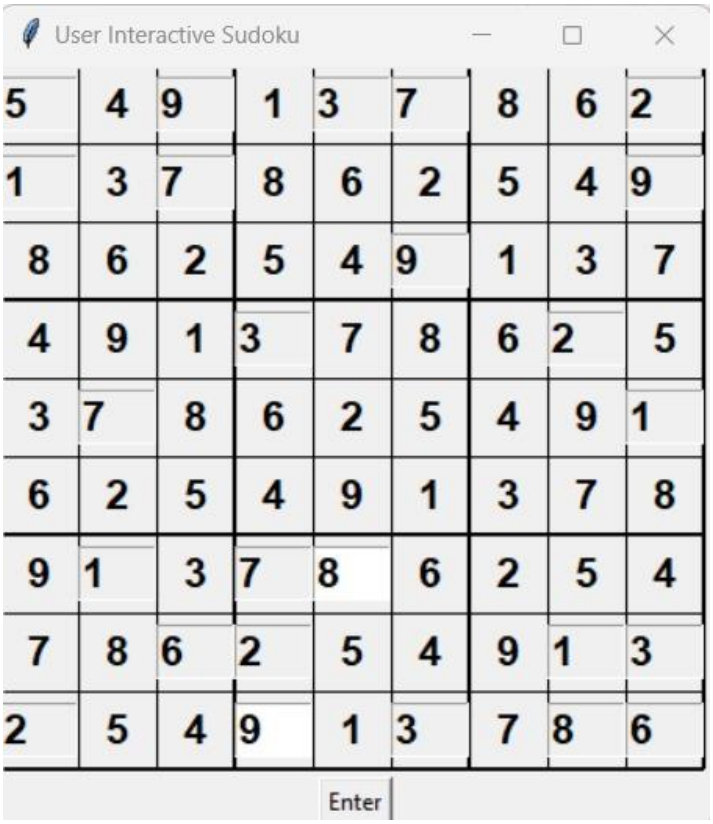


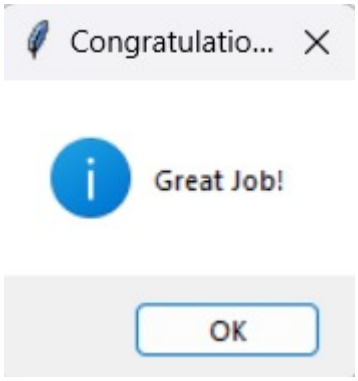


b) Unsolvable Puzzle:



c) Correctly Solved:





Data Structures:

➔ **In SudokuSolver Class:**

1) **Puzzle: 2D array**
2) **Steps: List of Sets**
   Each set has row, column, number, and domain
3) **Domains: Dictionary where row and columns are they keys and the value is each cell's domain**

➔ **In SudokuGUI Class:**

1) **Initial Puzzle ,Puzzle: 2D array**
2) **Steps: List of Sets**

   Each set has row, column, number, and domain

➔ **SolutionGUI Class:**

1) **Puzzle, Initial Puzzle , Current: 2D array**
2) **Steps: List of Sets**

   Each set has row, column, number, and domain

➔ **SudokuInputGUI Class:**

1) **Current Puzzle, Initial Puzzle: 2D array**

➔ **UserInteractiveGUI Class:**

1) **Initial Puzzle, Current Puzzle, Previous Puzzle: 2D array**
2) **Entries: List of lists (reference to the input widgets**
3) **)**