# SUDOKU GAME USING BACKTRACKING

**A Project Report submitted to**

**JIO INSTITUTE, MAHARASHTRA**

**In partial fulfillment of the requirements for the**

**POST GRADUATE PROGRAMME IN ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**


**Submitted By:**

**GOPI NAIK D**

**SUJOY DEY**

**SAI NIHAL PAMPARA**

**Under the Guidance of:**

**DR. SUDIPTA ROY**

**Jio Institute, Sector-4, Ulwe, Navi Mumbai, Maharashtra – 410206**


## SUDOKU GAME

# Content

# 1. Introduction

Sudoku is a classic logic-based number placement puzzle that has captivated minds across the world for decades. The standard version of Sudoku involves a 9x9 grid divided into nine 3x3 subgrids. The primary goal is to fill each cell in such a way that every row, every column, and each 3x3 subgrid contains all the digits from 1 to 9 without repetition. While the game appears simple at first glance, its combinatorial complexity and constraint satisfaction nature make it an excellent candidate for algorithmic problem solving, particularly using Data Structures and Algorithms (DSA) concepts.

In the realm of computational problem solving, Sudoku provides a rich ground for applying recursive algorithms, constraint checking, and optimization techniques. Solving Sudoku through a brute-force approach is impractical due to its immense solution space. Therefore, more refined strategies like **backtracking combined with depth-first search (DFS)** are employed to systematically and efficiently explore potential number placements, validating constraints dynamically at each recursive step. These techniques make the solution more elegant, robust, and adaptable for various levels of puzzle difficulty.

This project focuses on developing a fully functional and intelligent Sudoku solver using **Python**. The implementation employs **pure DSA principles without relying on any external libraries**, showcasing how foundational programming logic alone can solve complex, real-world problems. The graphical user interface (GUI) is created using **Tkinter**, allowing users to interactively view and solve generated puzzles, reinforcing both conceptual and visual understanding of the solving process.

The backtracking algorithm plays a central role in this project. It attempts to place digits sequentially while continuously checking if the current state of the board adheres to all Sudoku rules. If a conflict arises, the algorithm retracts its last decision (backtracks) and tries the next possible option. This method ensures that all possible combinations are explored only as needed, drastically reducing unnecessary computation. Additionally, DFS is used to ensure that the algorithm dives deep into possible solutions before reconsidering alternative paths—this depth-oriented exploration enhances the efficiency of the recursive process.

Our objective is to explores the entire development lifecycle of the Sudoku solver, from algorithm design and implementation to interface development and results analysis. By the end, readers will gain a comprehensive understanding of how data structures, recursion, and algorithmic thinking converge to solve complex puzzles like Sudoku with precision and efficiency.

## 2. Brute-Force Approach

The brute-force approach is the **simplest** yet most computationally expensive way to solve a Sudoku puzzle. It attempts to **fill all empty cells with every possible number (1–9)** and **checks every combination** until a valid solution is found.

**How Brute-Force Works:**

1. Identify all empty cells.
2. Generate **all possible permutations** of numbers (1–9) for those cells.
3. For each permutation:
    - Fill the board.
    - Check if it satisfies all Sudoku rules (unique in row, column, and 3x3 box).
4. If a valid solution is found → Done. Else → Try the next combination.

**Limitations:**

- Extremely **inefficient** due to the vast number of combinations.
- **Worst-case time complexity** is O(9^81), which is impractical for complex or large boards.
- Not intelligent — it doesn't "learn" or eliminate invalid paths early.

## 3. Backtracking Approach (with DFS)

Backtracking is a **smarter refinement of brute-force**. It still tries numbers 1–9 in empty cells, but with one crucial improvement: it **checks constraints immediately** and **backtracks** as soon as a conflict is detected, avoiding unnecessary computations.

**How Backtracking Works:**

1. Find the next empty cell.
2. Try numbers 1–9 **one by one**.
3. After each placement:
    - **Immediately validate** against row, column, and box constraints.
    - If valid → proceed recursively (DFS) to the next empty cell.
    - If invalid → **undo (backtrack)** and try the next number.
4. Continue until the puzzle is solved or no valid number can be placed (backtrack again).

**Advantages Over Brute-Force:**

- **Much faster** due to early pruning of invalid paths.
- **Only explores valid partial solutions**.
- Naturally implemented using **recursive DFS**.
- Efficient for 9x9 puzzles — the method used in most Sudoku solvers.

**Why DFS for Sudoku?**

- Perfect match for the "fill one cell → go deeper" logic.
- Easy to implement using **recursive functions.**
- Ensures complete solution is explored **before** switching to alternate values.

## 4. Code Implementation

The full implementation is written in Python and includes:

- Solver using backtracking and DFS
- GUI built with tkinter for user interaction

Use of Backtracking and DFS in Code:

- solve_sudoku() is a recursive function that embodies DFS and backtracking.
- It tries numbers from 1 to 9 in every empty cell.
- If a number is valid (checked using is_valid()), it is placed, and the function recurses.

Code Snippet (Key Algorithm Logic):

```python
def is_valid(board, row, col, num):
    if num in board[row]:
        return False
    if num in [board[i][col] for i in range(BOARD_SIZE)]:
        return False
    start_row, start_col = row - row % BOX_SIZE, col - col % BOX_SIZE
    for i in range(BOX_SIZE):
        for j in range(BOX_SIZE):
            if board[start_row + i][start_col + j] == num:
                return False
    return True

def solve_sudoku(board):
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == EMPTY:
                for num in range(1, 10):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = EMPTY  # Backtracking step
                return False
    return True
```

This code uses backtracking and DFS algorithms to try and eliminate invalid paths quickly.

## 5. Python Code Implementation of Sudoku Game

```
import tkinter as tk
from tkinter import messagebox
import random
import time

BOARD_SIZE = 9
BOX_SIZE = 3
EMPTY = 0

# --- Core Sudoku Logic ---
def is_valid(board, row, col, num):
    if num in board[row]:
        return False
    if num in [board[i][col] for i in range(BOARD_SIZE)]:
        return False
    start_row, start_col = row - row % BOX_SIZE, col - col % BOX_SIZE
    for i in range(BOX_SIZE):
        for j in range(BOX_SIZE):
            if board[start_row + i][start_col + j] == num:
                return False
    return True

def solve_sudoku(board):
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == EMPTY:
                for num in range(1, 10):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = EMPTY
                return False
    return True

def fill_box(board, row_start, col_start):
    nums = list(range(1, 10))
    random.shuffle(nums)
    idx = 0
    for i in range(BOX_SIZE):
        for j in range(BOX_SIZE):
            board[row_start + i][col_start + j] = nums[idx]
            idx += 1

def fill_diagonal_boxes(board):
    for i in range(0, BOARD_SIZE, BOX_SIZE):
        fill_box(board, i, i)
```

```
def remove_cells(board, num_holes):
    count = 0
    while count < num_holes:
        row = random.randint(0, 8)
        col = random.randint(0, 8)
        if board[row][col] != EMPTY:
            board[row][col] = EMPTY
            count += 1

def generate_board():
    board = [[EMPTY for _ in range(BOARD_SIZE)] for _ in range(BOARD_SIZE)]
    fill_diagonal_boxes(board)
    solve_sudoku(board)
    solution = [row[:] for row in board]
    remove_cells(board, 40)
    return board, solution

# --- GUI Class ---
class SudokuGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Sudoku Game")
        self.entries = [[None for _ in range(BOARD_SIZE)] for _ in range(BOARD_SIZE)]
        self.board, self.solution = generate_board()

        self.main_frame = tk.Frame(self.root)
        self.main_frame.pack(expand=True, fill='both', padx=10, pady=10)

        self.draw_grid()

        # Buttons Frame
        buttons_frame = tk.Frame(self.root)
        buttons_frame.pack(pady=10)

        self.check_button = tk.Button(buttons_frame, text="Check", font=('Arial', 14), command=self.check_solution)
        self.check_button.pack(side='left', padx=10)

        self.pause_button = tk.Button(buttons_frame, text="Pause", font=('Arial', 14), command=self.pause_game)
        self.pause_button.pack(side='left', padx=10)

        # Timer variables
        self.start_time = time.time()
        self.paused = False
        self.pause_start_time = None
        self.total_paused_time = 0

    def draw_grid(self):
        for i in range(BOX_SIZE):
            self.main_frame.rowconfigure(i, weight=1)
```

```
            for j in range(BOX_SIZE):
                self.main_frame.columnconfigure(j, weight=1)
                box_frame = tk.Frame(self.main_frame, bg="black", borderwidth=2, relief='groove')
                box_frame.grid(row=i, column=j, padx=2, pady=2, sticky='nsew')
                box_frame.rowconfigure(tuple(range(BOX_SIZE)), weight=1)
                box_frame.columnconfigure(tuple(range(BOX_SIZE)), weight=1)

                for m in range(BOX_SIZE):
                    for n in range(BOX_SIZE):
                        global_row = i * BOX_SIZE + m
                        global_col = j * BOX_SIZE + n
                        entry = tk.Entry(box_frame, font=('Arial', 16), justify='center', relief='flat')
                        entry.grid(row=m, column=n, sticky='nsew', padx=1, pady=1)

                        if self.board[global_row][global_col] != EMPTY:
                            entry.insert(0, str(self.board[global_row][global_col]))
                            entry.config(state='disabled', disabledforeground='black')

                        self.entries[global_row][global_col] = entry

    def check_solution(self):
        for i in range(BOARD_SIZE):
            for j in range(BOARD_SIZE):
                val = self.entries[i][j].get()
                if val.isdigit() and int(val) == self.solution[i][j]:
                    continue
                else:
                    messagebox.showerror("Sudoku", "Incorrect solution!")
                    return

        # If correct, calculate elapsed time
        elapsed_time = time.time() - self.start_time - self.total_paused_time
        mins, secs = divmod(int(elapsed_time), 60)
        time_str = f"{mins} minute(s) and {secs} second(s)"

        popup = tk.Toplevel(self.root)
        popup.title("Congratulations!")
        popup.geometry("350x180")
        popup.transient(self.root)
        popup.grab_set()
        popup.resizable(False, False)

        label = tk.Label(popup, text=f"🎉 You solved the puzzle in {time_str}!\nWhat would you like to do next?",
font=('Arial', 12), justify='center')
        label.pack(pady=20)

        btn_frame = tk.Frame(popup)
        btn_frame.pack(pady=10)
```

```
        def start_new_game():
            popup.destroy()
            self.next_level()

        def exit_game():
            self.root.destroy()

        tk.Button(btn_frame, text="New Game", width=12, font=('Arial', 10), command=start_new_game).grid(row=0,
column=0, padx=10)
        tk.Button(btn_frame, text="Exit", width=12, font=('Arial', 10), command=exit_game).grid(row=0, column=1,
padx=10)

    def next_level(self):
        for i in range(BOARD_SIZE):
            for j in range(BOARD_SIZE):
                entry = self.entries[i][j]
                entry.config(state='normal')
                entry.delete(0, tk.END)

        self.board, self.solution = generate_board()

        for i in range(BOARD_SIZE):
            for j in range(BOARD_SIZE):
                if self.board[i][j] != EMPTY:
                    self.entries[i][j].insert(0, str(self.board[i][j]))
                    self.entries[i][j].config(state='disabled', disabledforeground='black')
                else:
                    self.entries[i][j].config(state='normal')

        self.start_time = time.time()
        self.total_paused_time = 0
        self.paused = False
        self.pause_start_time = None

    def pause_game(self):
        if not self.paused:
            self.paused = True
            self.pause_start_time = time.time()
            self.set_entries_state('disabled')
            self.check_button.config(state='disabled')
            self.pause_button.config(text='Resume')
            self.show_pause_popup()
        else:
            self.paused = False
            paused_duration = time.time() - self.pause_start_time
            self.total_paused_time += paused_duration
            self.pause_start_time = None
```

```
            self.set_entries_state('normal')


            for i in range(BOARD_SIZE):
                for j in range(BOARD_SIZE):
                    if self.board[i][j] != EMPTY:
                        self.entries[i][j].config(state='disabled')
            self.check_button.config(state='normal')
            self.pause_button.config(text='Pause')

        def set_entries_state(self, state):
            for i in range(BOARD_SIZE):
                for j in range(BOARD_SIZE):
                    if self.board[i][j] == EMPTY:
                        self.entries[i][j].config(state=state)

        def show_pause_popup(self):
            popup = tk.Toplevel(self.root)
            popup.title("Game Paused")
            popup.geometry("300x150")
            popup.transient(self.root)
            popup.grab_set()

            label = tk.Label(popup, text="Game is paused.\nResume or Exit?", font=('Arial', 12))
            label.pack(pady=20)

            btn_frame = tk.Frame(popup)
            btn_frame.pack(pady=10)

            def resume():
                popup.destroy()
                self.pause_game()

            def exit_game():
                self.root.destroy()

            tk.Button(btn_frame, text="Resume", width=10, command=resume).pack(side='left', padx=10)
            tk.Button(btn_frame, text="Exit", width=10, command=exit_game).pack(side='left', padx=10)

    # --- Run GUI ---
    if __name__ == "__main__":
        root = tk.Tk()
        root.geometry("800x850")
        root.minsize(600, 650)
        root.rowconfigure(0, weight=1)
        root.columnconfigure(0, weight=1)
        SudokuGUI(root)
        root.mainloop()
```

```
def is_valid(board, row, col, num):
    if num in board[row]:
        return False
    if num in [board[i][col] for i in range(BOARD_SIZE)]:
        return False
    start_row, start_col = row - row % BOX_SIZE, col - col % BOX_SIZE
    for i in range(BOX_SIZE):
        for j in range(BOX_SIZE):
            if board[start_row + i][start_col + j] == num:
                return False
    return True

def solve_sudoku(board):
    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == EMPTY:
                for num in range(1, 10):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = EMPTY  # Backtracking step
                return False
    return True
```

## 6. GUI and Game Instances

The implemented GUI enables users to interact with a **randomly generated Sudoku puzzle** in a structured and intuitive manner.

**Displayed Gameplay Screenshots Should Include:**

- The **initial puzzle** as generated.
- A **partially filled puzzle** during user interaction.
- The **completed and correctly solved puzzle**.

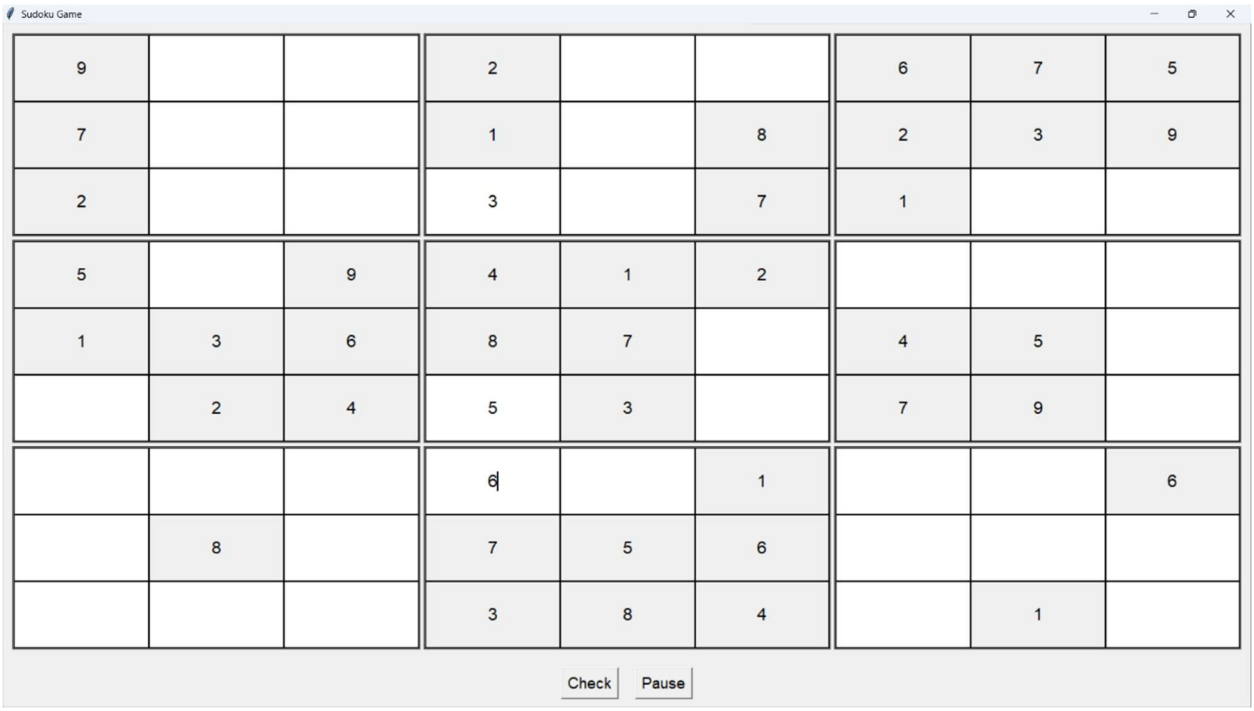
**Functionality and Design Highlights:**

- The puzzle is initialized with randomly filled **diagonal 3x3 boxes**, ensuring a valid base structure.
- A total of **40 cells are cleared** to provide a challenging yet solvable puzzle.
- Users can **input values** in editable cells and click the **"Check"** button to validate their solution at any point.

**Additional Interface Features**

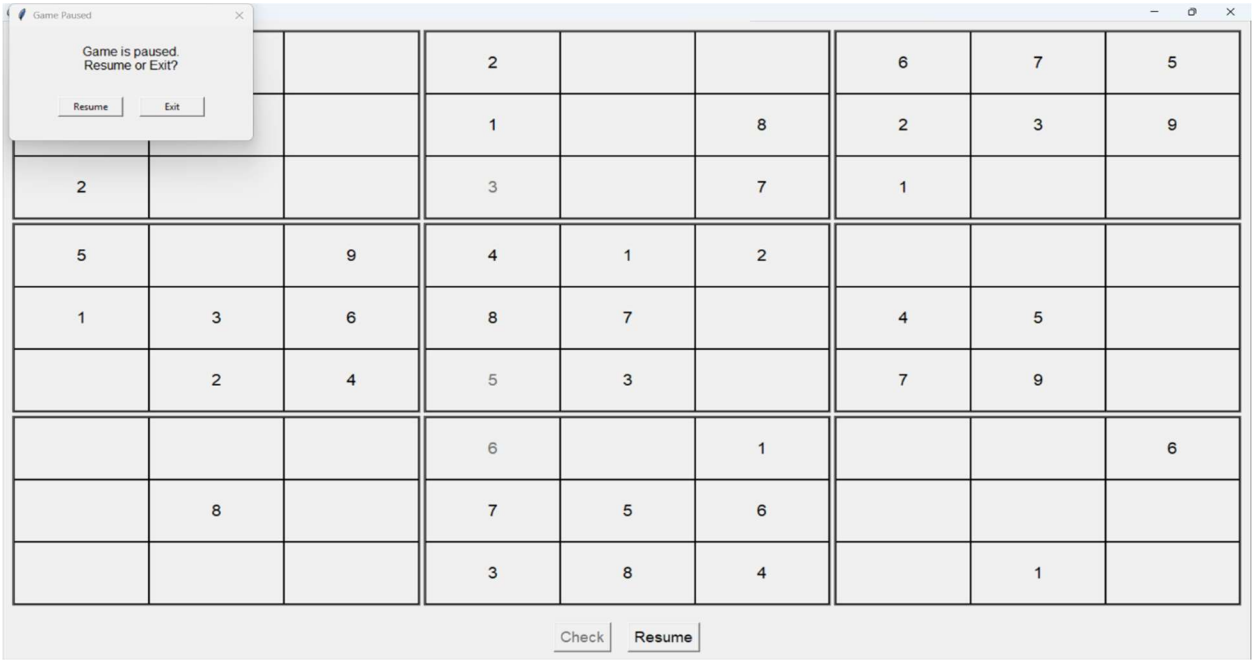
**Gameplay and Puzzle Solving**

- **Users can fill in values in editable cells.**
- **The "Check" button allows players to validate their current inputs against the correct solution.**
- **Partial progress is retained and incorrect cells are flagged using message prompts.**




**Pause Functionality**

- **Clicking the "Pause" button opens a popup window.**
- **When paused, all input fields and controls are temporarily disabled.**

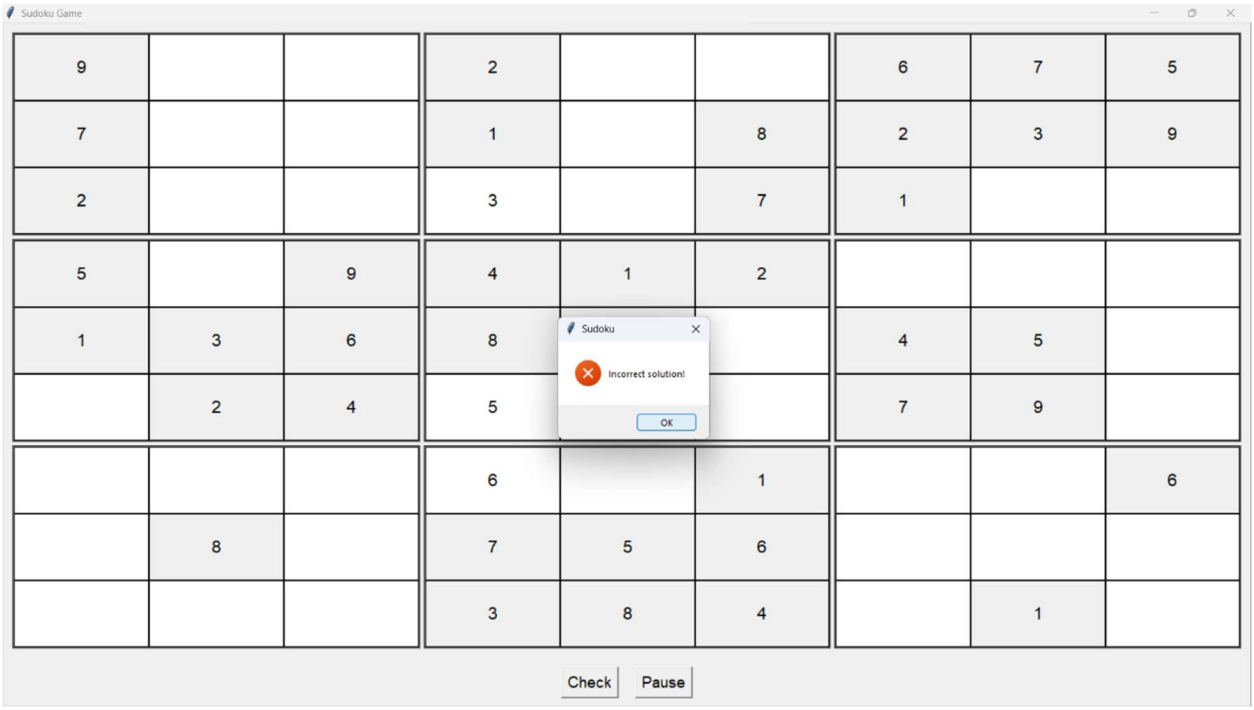


# 7. Results

The game successfully generates, solves, and verifies the user input with the solution for Sudoku puzzles.

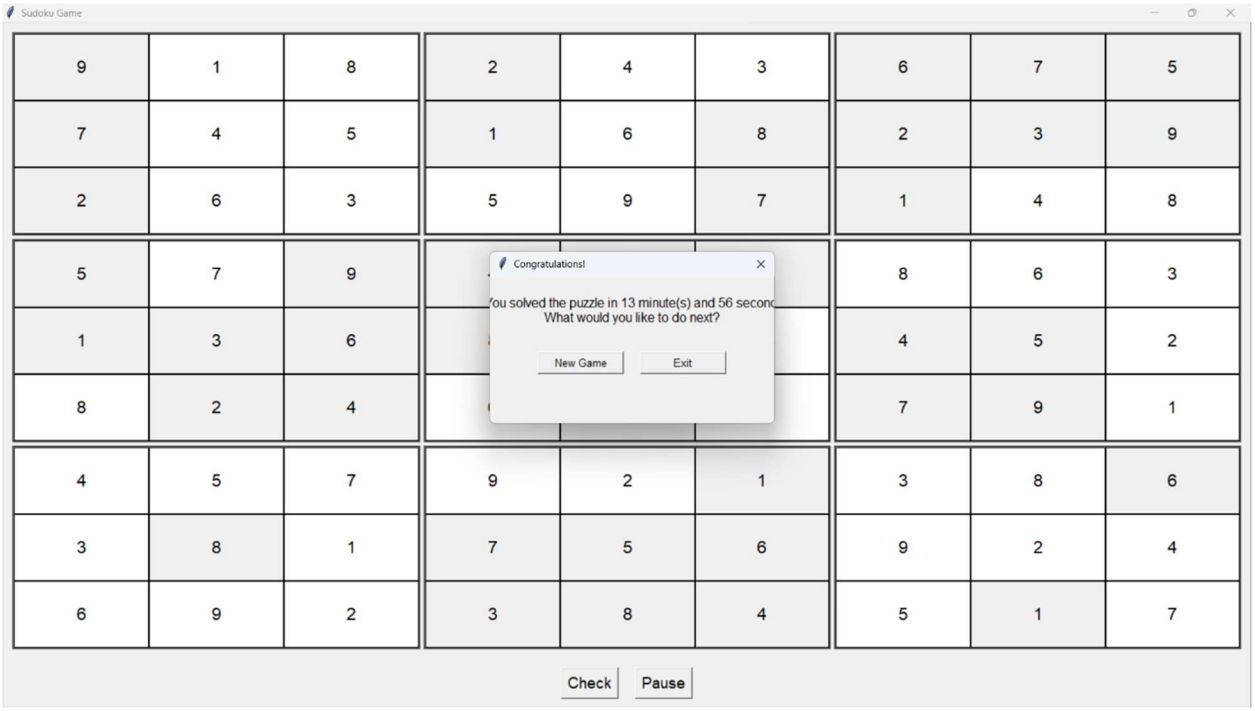
**Incorrect Submission Handling**

- If the solution is incorrect or incomplete, an **error message** is triggered.
- The message distinguishes between:
    - **Unfilled cells**
    - **Invalid or incorrect placements**

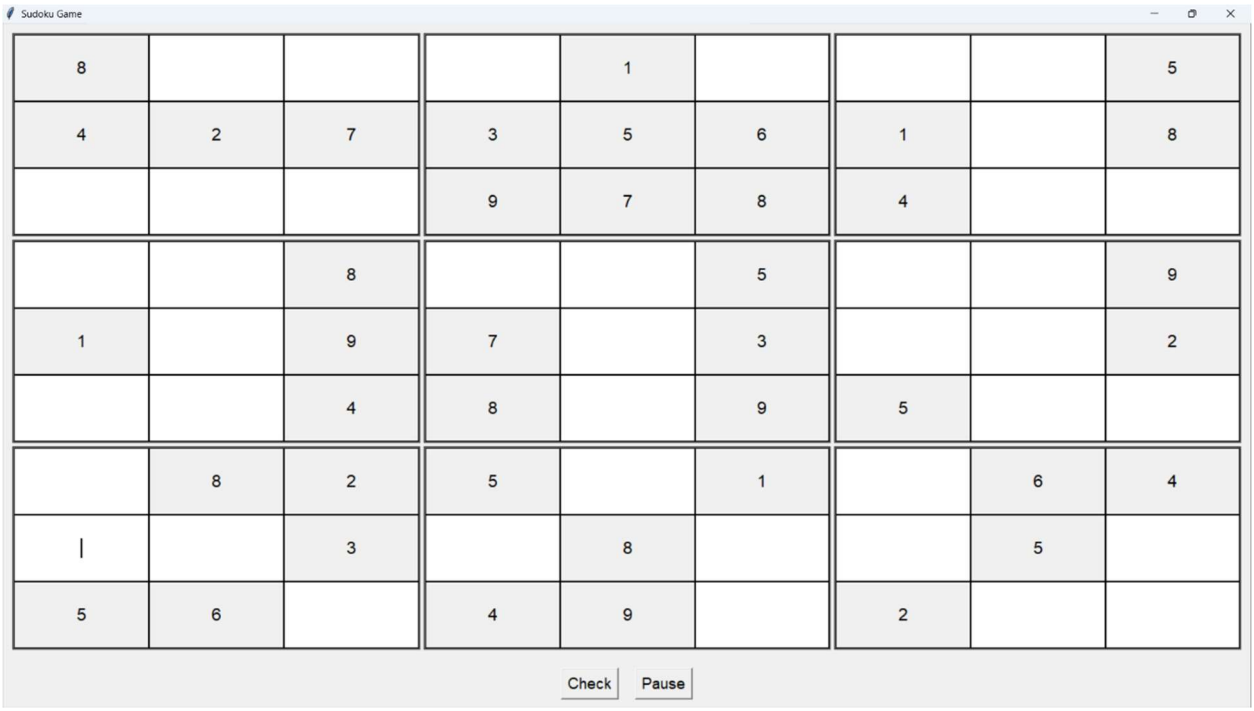


**Successful Completion (Victory Screen)**

- Upon solving the puzzle correctly, a **congratulatory popup** is shown.
- It displays the **time taken** to solve the puzzle.
- Players are given two options:
    - **Start a new game** (with a freshly generated puzzle)
    - **Exit the application**



After a successful completion, if the player chooses to continue, the game automatically resets with a **new randomly generated Sudoku puzzle**, allowing them to play a fresh round seamlessly.




**Time Complexity:**

- Worst-case: O(9^(n)), where n is the number of empty cells.
- Average-case is improved due to pruning via constraint checking.

**Space Complexity:**

- O(1) auxiliary (in-place modification), but call stack depth can reach O(n) due to recursion.

## 8. References


1. Backtracking and DFS Concepts: Narasimha Karumanchi, Data Structures and Algorithms Made Easy (Page no. 61 to Page no. 68)
2. Sudoku Game Theory: https://en.wikipedia.org/wiki/Sudoku
3. Python Official Documentation: https://docs.python.org/3/