KIET GROUP OF INSTITUTIONS

Introduction to AI

MSE-1

Name – Srish Jana

Univ. Roll No. – 202401100400189

Branch Sec. – CSE-AIML-C

Date – 10/03/2025

<u>Introduction</u>-

This Python-based Sudoku Solver efficiently solves 9x9 Sudoku puzzles using the **Backtracking Algorithm**. It systematically fills empty cells while ensuring that each row, column, and 3x3 subgrid adheres to Sudoku rules. If a conflict arises, the algorithm backtracks to find an alternative solution. The program is designed to handle standard Sudoku grids and provides a quick and reliable solution if the puzzle is solvable. Ideal for Sudoku enthusiasts and developers, this solver demonstrates the power of recursive problem-solving. Simply input your unsolved grid, and the program will return the completed puzzle in an easy-to-read format.

Methodology of the Sudoku Solver Program

- The **Sudoku Solver** uses a systematic approach based on the **Backtracking Algorithm** to efficiently find a valid solution for a given 9x9 Sudoku puzzle. Below is a step-by-step breakdown of the methodology:
- **1. Input Representation**
  - The Sudoku puzzle is represented as a **9x9 grid** (a list of lists in Python).
  - Empty cells are denoted by 0, indicating they need to be filled.
- **2. Finding an Empty Cell**
  - The function find_empty_location(grid) iterates through the grid to find the first empty cell (i.e., a cell containing 0).
  - If no empty cells are found, the puzzle is considered solved.
- **3. Checking Validity of a Number Placement**
  - The function is_valid_placement(grid, row, col, num) checks if placing a number (1-9) in a given cell is valid by ensuring:
    - **Row Constraint:** The number is not already present in the same row.

    - **Column Constraint:** The number is not already present in the same column.
    - **3x3 Subgrid Constraint:** The number is not already present in the corresponding 3x3 subgrid.
- **4. Recursive Backtracking Algorithm**
  - The function solve_sudoku(grid) implements **backtracking**:
    1. Find an empty cell.
    2. Try placing numbers 1-9 and check if the placement is valid.
    3. If valid, recursively attempt to solve the rest of the puzzle.
    4. If the placement leads to an unsolvable state, **backtrack** by resetting the cell to 0 and trying the next possible number.
    5. Repeat until the entire grid is solved.
- **5. Output the Solved Grid**
  - If the puzzle is solvable, the completed grid is displayed in a readable format.
  - If no solution exists, a message is displayed indicating that the puzzle is unsolvable.
- This **methodology ensures** that the solver efficiently explores possible solutions while maintaining the constraints of Sudoku.

Code –

```
def find_empty_location(grid):
    """Finds an empty location in the Sudoku grid.

    Args:
        grid: The Sudoku grid represented as a 9x9 list of lists.

      Returns:
          A tuple (row, col) representing the coordinates of an empty location,
          or None if no empty location is found.
    """
    for row in range(9):
        for col in range(9):
            if grid[row][col] == 0:       # 0 represents an empty cell
                return row, col
    return None

def is_valid_placement(grid, row, col, num):
    """Checks if placing the given number at the specified location is valid.

    Args:
        grid: The Sudoku grid.
        row: The row index.
        col: The column index.
        num: The number to place.

    Returns:
        True if the placement is valid, False otherwise.
    """
    # Check row
    for x in range(9):
        if grid[row][x] == num:
            return False

    # Check column
    for x in range(9):
```

```
        if grid[x][col] == num:
            return False

    # Check 3x3 subgrid
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if grid[i + start_row][j + start_col] == num:
                return False
    return True

def solve_sudoku(grid):
    """Solves the Sudoku puzzle using backtracking.

    Args:
        grid: The Sudoku grid to solve.

    Returns:
        True if the puzzle is solvable, False otherwise.  The grid is modified in
place.
    """
    empty_location = find_empty_location(grid)
    if not empty_location:
        return True               # No empty locations left, puzzle solved

    row, col = empty_location

    for num in range(1, 10):
        if is_valid_placement(grid, row, col, num):
            grid[row][col] = num   # Place the number

            if solve_sudoku(grid): # Recursively try to solve the rest of the grid
                return True       # If successful, the whole grid is solved

            grid[row][col] = 0  # Backtrack: if the placement leads to no solution,
reset the cell
```

```
    return False  # No valid number can be placed here, backtrack from previous steps.


# Example usage (replace with your own grid):
grid = [
    [0, 0, 0, 6, 0, 0, 0, 0, 3],
    [0, 6, 0, 0, 3, 0, 0, 8, 0],
    [0, 0, 9, 0, 0, 0, 5, 0, 0],
    [5, 0, 0, 0, 8, 0, 0, 0, 6],
    [0, 0, 3, 0, 0, 0, 7, 0, 0],
    [8, 0, 0, 0, 4, 0, 0, 0, 1],
    [0, 0, 6, 0, 0, 0, 9, 0, 0],
    [0, 3, 0, 0, 6, 0, 0, 1, 0],
    [9, 0, 0, 0, 0, 4, 0, 0, 0]
]

if solve_sudoku(grid):
    for row in grid:
        print(row)

else:
print("No solution exists.")
```

Outputs:

Example – 1

Problem:

```
grid = [
    [0, 0, 0, 6, 0, 0, 0, 0, 3],
    [0, 6, 0, 0, 3, 0, 0, 8, 0],
    [0, 0, 9, 0, 0, 0, 5, 0, 0],
    [5, 0, 0, 0, 8, 0, 0, 0, 6],
    [0, 0, 3, 0, 0, 0, 7, 0, 0],
    [8, 0, 0, 0, 4, 0, 0, 0, 1],
    [0, 0, 6, 0, 0, 0, 9, 0, 0],
    [0, 3, 0, 0, 6, 0, 0, 1, 0],
    [9, 0, 0, 0, 0, 4, 0, 0, 0]
]
```

```
[1, 2, 5, 6, 9, 8, 4, 7, 3]
[4, 6, 7, 2, 3, 5, 1, 8, 9]
[3, 8, 9, 4, 1, 7, 5, 6, 2]
[5, 4, 1, 7, 8, 3, 2, 9, 6]
[6, 9, 3, 1, 5, 2, 7, 4, 8]
[8, 7, 2, 9, 4, 6, 3, 5, 1]
[7, 5, 6, 8, 2, 1, 9, 3, 4]
[2, 3, 4, 5, 6, 9, 8, 1, 7]
[9, 1, 8, 3, 7, 4, 6, 2, 5]
```

Example – 2

Problem:

grid = [
    [0, 0, 0, 0, 0, 0, 0, 1, 2],
    [0, 0, 0, 0, 3, 5, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 6, 0, 0],
    [0, 1, 0, 5, 0, 0, 0, 0, 0],
    [4, 0, 0, 0, 6, 0, 0, 0, 3],
    [0, 0, 0, 0, 0, 2, 0, 8, 0],
    [0, 0, 7, 0, 0, 0, 4, 0, 0],
    [0, 0, 0, 7, 2, 0, 0, 0, 0],
    [3, 2, 0, 0, 0, 0, 0, 0, 0]
]

```
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[2, 6, 9, 1, 3, 5, 7, 4, 8]
[8, 7, 1, 2, 9, 4, 6, 3, 5]
[9, 1, 3, 5, 8, 7, 2, 6, 4]
[4, 8, 2, 9, 6, 1, 5, 7, 3]
[7, 5, 6, 3, 4, 2, 1, 8, 9]
[1, 9, 7, 8, 5, 3, 4, 2, 6]
[6, 4, 8, 7, 2, 9, 3, 5, 1]
[3, 2, 5, 4, 1, 6, 8, 9, 7]
```