Sudoku Solver

Prepared by:
Rishabh_kumar_Singh_202401100300199

Date: 2025-03-11

Introduction

Sudoku is a globally popular number puzzle that relies on logical reasoning rather than mathematical skills. The objective is to fill a 9x9 grid so that each row, each column, and each 3x3 sub-grid contain the numbers from 1 to 9 exactly once. Sudoku puzzles vary in difficulty, with some requiring complex logical deductions to solve.

This project presents a **Sudoku Solver** written in Python that automatically solves Sudoku puzzles using a **backtracking algorithm**. The solver can take an incomplete Sudoku
board as input and produce a complete, valid solution.

The **importance of Sudoku solving** extends beyond entertainment. It plays a role in AI development, combinatorial optimization, and constraint satisfaction problems in computer science. The backtracking approach used in this project is a powerful method for solving constraint-based problems efficiently.

In the next sections, we will discuss the **methodology** behind this solver, explore the algorithm, and demonstrate how it works with an example Sudoku puzzle.

Methodology

Approach Used to Solve the Problem

The **Sudoku Solver** uses a backtracking algorithm, which is a recursive approach to solving
constraint satisfaction problems. The methodology is broken down into the following steps:

1. **Representation of the Sudoku Board:**
   - The board is represented as a 9x9 matrix (list of lists in Python).
   - Empty cells are represented by the number **0**.

2. **Validation of Moves:**
   - A helper function `is_valid(board, row, col, num)` is used to check whether placing a given number in a specific cell is valid.
   - The function ensures that:
     - The number does not already exist in the **same row**.
     - The number does not already exist in the **same column**.
     - The number does not exist in the **3x3 sub-grid**.

3. **Backtracking Algorithm:**
   - The `solve_sudoku(board)` function is called to find an empty cell.
   - It tries placing numbers from **1 to 9** in that cell.
   - If a number is valid, it is placed temporarily, and the function is called recursively to solve the rest of the board.
   - If no valid number can be placed, the function **backtracks** by removing the last placed number and trying the next one.
   - The process continues until the board is completely filled.

4. **Efficiency Considerations:**
   - Backtracking is a brute-force approach, but since Sudoku has a structured set of constraints, it performs efficiently for most standard puzzles.
   - Optimizations like **constraint propagation** and **naked pairs/triples techniques** can enhance performance for more complex puzzles.

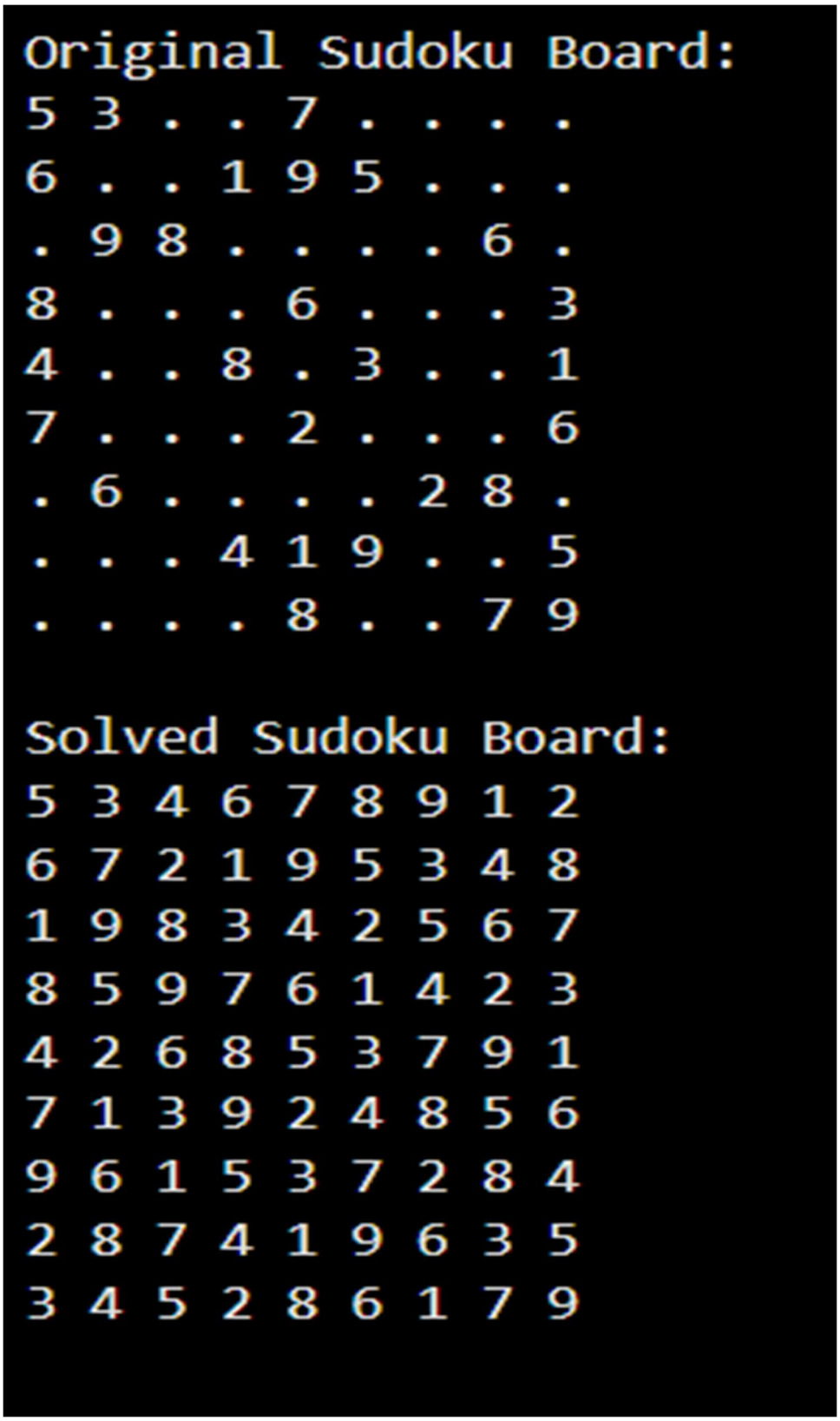
This methodology ensures that any valid Sudoku puzzle can be solved efficiently using Python.

Code

```
# Function to check if placing a number in the given cell is valid according to Sudoku rules
def is_valid(board, row, col, num):
    # Check if the number is already present in the current row or column
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False  # If found, return False

    # Check the 3x3 grid (sub-grid) where the cell is located
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False  # If the number is found in the 3x3 sub-grid, return False

    return True  # If no conflicts found, return True

# Function to solve the Sudoku puzzle using backtracking
def solve_sudoku(board):
    # Iterate through each cell in the 9x9 grid
    for row in range(9):
        for col in range(9):
            # Check if the current cell is empty (0 means empty)
            if board[row][col] == 0:
                # Try numbers from 1 to 9 for the empty cell
                for num in range(1, 10):
                    # If the number is valid for the current cell, place it
                    if is_valid(board, row, col, num):
                        board[row][col] = num

                        # Recursively attempt to solve the rest of the board
                        if solve_sudoku(board):
                            return True  # If the puzzle is solved, return True

                        # If placing num didn't work, reset the cell and try the next number
                        board[row][col] = 0
                return False  # If no number works for this cell, return False (backtrack)
    return True  # If all cells are filled, the puzzle is solved

# Function to print the Sudoku board in a readable format
def print_board(board):
    # Print each row of the board, replacing 0 (empty cells) with '.'
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))

# Example Sudoku Board (0 represents empty cells)
```

```
sudoku_board = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

# Print the original Sudoku board
print("Original Sudoku Board:")
print_board(sudoku_board)

# Attempt to solve the Sudoku puzzle
if solve_sudoku(sudoku_board):
    print("\nSolved Sudoku Board:")
    print_board(sudoku_board)  # If solved, print the solved board
else:
    print("\nNo solution exists!")  # If no solution is found, print an error message
```

Output/Result



References/Credits

1. Sudoku Rules and Problem Definition: https://en.wikipedia.org/wiki/Sudoku
2. Backtracking Algorithm: https://www.geeksforgeeks.org/backtracking-introduction/
3. Python Programming: https://docs.python.org/3/tutorial/index.html
4. Algorithm Efficiency and Optimization:
https://www.khanacademy.org/computing/computer-science/algorithms
5. Sudoku Solving Techniques: https://www.sudokuwiki.org/Sudoku_Techniques

Special Thanks To :

Mayank Sir