Name- Srashti Gupta

Branch- CSE AI

Section- D

AKTU Roll no.- 202401100300251

Problem – Sudoku Solver

Introduction-

Sudoku is a widely popular logic-based number placement puzzle that consists of a 9×9 grid, further divided into nine 3×3 subgrids. The objective is to fill the grid with numbers 1 to 9 in such a way that each row, column, and 3×3 subgrid contains all digits exactly once, without repetition. While some numbers are pre-filled, the challenge lies in determining the missing values while adhering to the puzzle's constraints.

Solving a Sudoku puzzle manually requires logical deduction and pattern recognition. However, for more complex puzzles, computational approaches such as backtracking algorithms provide an efficient solution. This report explores the implementation of a Sudoku solver using Python, employing a recursive backtracking method to systematically fill in missing numbers. The algorithm tries different possibilities, verifies their validity, and backtracks when necessary to arrive at a correct solution.

The purpose of this report is to outline the problem statement, algorithm selection, methodology, implementation details, and performance analysis of the Sudoku solver. Additionally, it includes intermediate stages of the solving process to provide insight into how the algorithm progresses towards a solution.

By automating Sudoku solving, this project demonstrates the power of algorithmic problem-solving and highlights the efficiency of backtracking in constraint-based puzzles.

Methodology-

Understanding the Problem

Sudoku is a **9×9 grid**, further divided into **nine 3×3 subgrids**. The objective is to fill the grid so that:

- Each **row** contains the numbers **1 to 9**, without repetition.
- Each **column** contains the numbers **1 to 9**, without repetition.
- Each **3×3 subgrid** contains the numbers **1 to 9**, without repetition.

Some numbers are **pre-filled**, while others are left **empty** (represented as `0` in our implementation). The challenge is to fill in the missing numbers while following these rules.

---

2. Choosing an Algorithm

We use a **backtracking algorithm**, which is a recursive approach that:

1. Finds an empty cell in the Sudoku grid.
2. Tries placing a number (1 to 9) in that cell.
3. Checks whether the number follows Sudoku rules.
4. If valid, moves to the next empty cell and continues.
5. If stuck, it **backtracks** (removes the last placed number) and tries a different one.
6. Repeats this process until the board is solved.

Backtracking ensures that all possible solutions are explored until a valid one is found.

---

3. Steps to Solve Sudoku

Step 1: Read the Sudoku Board

- Represent the Sudoku grid as a **9×9 matrix** (a list of lists in Python).
- Empty cells are marked as `0`.

Step 2: Find an Empty Cell

- Scan the grid row by row to locate the first empty cell (`0`).
- If no empty cells remain, the puzzle is solved.

Step 3: Try Possible Numbers

For each empty cell, attempt numbers **1 to 9**, ensuring 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 already exist** in the **same 3×3 subgrid**.

Step 4: Recursive Backtracking

- If a number is valid, **place it in the cell** and move to the next empty spot.
- Continue filling numbers recursively.
- If a contradiction arises (no valid number can be placed), **remove the last placed number (backtrack)** and try the next possibility.
- Repeat until the board is completely filled.

Step 5: Save and Display Intermediate Stages

- Store snapshots of the board at different stages to track progress.
- Print:
  - **Initial Board** (before solving begins).
  - **Intermediate Step** (somewhere midway through solving).
  - **Final Solved Board** (fully completed solution).

Step 6: Solution Completion

- Once all empty cells are filled correctly, display the solved Sudoku board.

---

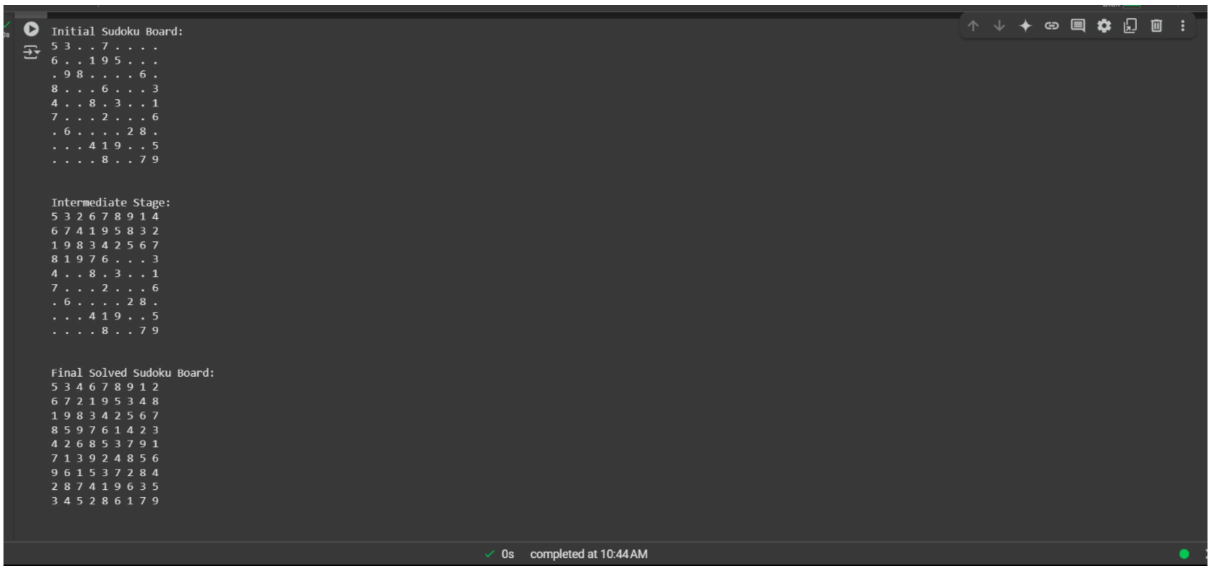
4. Complexity Analysis

- In the worst case (an empty grid), the time complexity can reach **O(9^8)**, as each empty cell has up to **9 possible choices**.
- However, optimizations like **constraint checking** and **early pruning** significantly reduce execution time in most cases.

Code-

```
import copy


def print_sudoku(board):
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))
    print("\n")


def find_empty_cell(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i, j
    return None


def is_valid(board, num, pos):
    row, col = pos


    # Check row
    if num in board[row]:
        return False


    # Check column
    if num in [board[i][col] for i in range(9)]:
        return False

    # Check 3x3 box
    box_x, box_y = col // 3, row // 3
    for i in range(box_y * 3, box_y * 3 + 3):
        for j in range(box_x * 3, box_x * 3 + 3):
            if board[i][j] == num:
                return False

    return True

def solve_sudoku(board, steps):
    empty_cell = find_empty_cell(board)
    if not empty_cell:
        steps.append(copy.deepcopy(board))
        return True

    row, col = empty_cell
    for num in range(1, 10):
        if is_valid(board, num, (row, col)):
            board[row][col] = num
            steps.append(copy.deepcopy(board))

            if solve_sudoku(board, steps):
                return True
            board[row][col] = 0
            steps.append(copy.deepcopy(board))

    return False

def main():
    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("Initial Sudoku Board:")
    print_sudoku(sudoku_board)

    steps = []
    solve_sudoku(sudoku_board, steps)

    print("Intermediate Stage:")
    print_sudoku(steps[len(steps)//2])


    print("Final Solved Sudoku Board:")
    print_sudoku(sudoku_board)


if __name__ == "__main__":
    main()
```

Output-



```
Initial Sudoku Board:
5 3 . . 7 . . . .
6 . . 1 9 5 . . .
. 9 8 . . . . 6 .
8 . . . 6 . . . 3
4 . . 8 . 3 . . 1
7 . . . 2 . . . 6
. 6 . . . . 2 8 .
. . . 4 1 9 . . 5
. . . . 8 . . 7 9


Intermediate Stage:
5 3 2 6 7 8 9 1 4
6 7 4 1 9 5 8 3 2
1 9 8 3 4 2 5 6 7
8 1 9 7 6 . . . 3
4 . . 8 . 3 . . 1
7 . . . 2 . . . 6
. 6 . . . . 2 8 .
. . . 4 1 9 . . 5
. . . . 8 . . 7 9


Final Solved Sudoku Board:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```