SUDOKU SOLVER

Problem Statement : Sudoku Solver

Name : Mahi

Roll no. : 202401100300147

Branch : CSEAI-B

1. Introduction

Sudoku is a logic-based puzzle involving a 9x9 grid where some cells are pre-filled with digits from 1 to 9, and others are left empty. The goal is to fill in the empty cells such that each row, column, and 3x3 subgrid contains all digits from 1 to 9 without repetition.

The problem can be efficiently solved using **backtracking**, a type of depth-first search (DFS) algorithm. This report discusses a Python-based implementation of a Sudoku solver using backtracking.

---

2. Problem Statement

Given a partially filled 9x9 Sudoku grid, where 0 represents an empty cell, the task is to fill in the grid following these conditions:

- Each row contains numbers from 1 to 9, without repetition.
- Each column contains numbers from 1 to 9, without repetition.
- Each 3x3 subgrid contains numbers from 1 to 9, without repetition.

The goal is to solve the puzzle, filling the empty cells with appropriate digits, if a solution exists.

---

3. Methodology

The backtracking approach is chosen for solving the Sudoku puzzle. Backtracking is an algorithm for finding all (or some) solutions to a problem by incrementally building candidates and abandoning partial candidates as soon as it is determined they cannot lead to a valid solution.

3.1 Steps in the Backtracking Approach:

1. **Find an Empty Cell**: The algorithm searches for an empty cell (denoted by 0).
2. **Try Possible Numbers**: For each empty cell, try placing numbers from 1 to 9 in the cell.
3. **Check Validity**: Before placing a number in the empty cell, the algorithm ensures that it does not violate Sudoku rules:
    - The number must not already exist in the current row.
    - The number must not already exist in the current column.
    - The number must not already exist in the current 3x3 subgrid.
4. **Recursive Backtracking**: If a valid number is placed, the algorithm recursively tries to fill in the next empty cell. If a number cannot be placed in the current empty cell, it backtracks to the previous cell and tries the next number.
5. **Repeat**: The process continues until the entire grid is filled or until it is determined that no valid solution exists.

---

4. Implementation Details

The Sudoku solver in Python uses the backtracking algorithm. The following functions are implemented:

1. **is_safe(board, row, col, num)**: Checks if placing the number num in the cell (row, col) is valid.
2. **solve_sudoku(board)**: Recursively tries to solve the Sudoku puzzle using backtracking.
3. **print_board(board)**: Prints the current state of the Sudoku board.

4. **get_user_input()**: Takes the Sudoku puzzle as input from the user.

CODE:

```
# Function to check if a number can be placed at a given position
def is_safe(board, row, col, num):
    # Check the row
    for i in range(9):
        if board[row][i] == num:
            return False

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

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

    return True

# Function to solve the Sudoku using backtracking
```

```
def solve_sudoku(board):
    # Find the next empty space (denoted by 0)
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                # Try digits 1-9 for this position
                for num in range(1, 10):
                    if is_safe(board, row, col, num):
                        board[row][col] = num  # Place the number

                        # Recursively try to solve the next positions
                        if solve_sudoku(board):
                            return True

                        # If placing num doesn't lead to a solution, backtrack
                        board[row][col] = 0

                return False  # No valid number can be placed here

    return True  # All cells are filled, puzzle is solved

# Function to print the Sudoku board
def print_board(board):
    for row in board:
        print(" ".join(str(num) for num in row))
```

```
# Function to take input from the user for the Sudoku puzzle
def get_user_input():
    board = []
    print("Enter the Sudoku puzzle row by row. Use 0 for empty cells.")

    for i in range(9):
        while True:
            try:
                row = list(map(int, input(f"Enter row {i + 1} (space-separated values): 
").split()))
                if len(row) != 9:
                    raise ValueError("Each row must have exactly 9 numbers.")
                board.append(row)
                break
            except ValueError as e:
                print(f"Invalid input. Please try again: {e}")
    return board

# Main code to execute
if __name__ == "__main__":
    # Get user input for the Sudoku puzzle
    board = get_user_input()

    # Solve the Sudoku puzzle
```

```
if solve_sudoku(board):
    print("Solved Sudoku:")
    print_board(board)
else:
    print("No solution exists")
```

5. Explanation of the Code

1. **is_safe(board, row, col, num)**: This function ensures that the number num can be placed in the cell (row, col) without violating Sudoku constraints. It checks the row, column, and 3x3 subgrid for any repetition of the number.
2. **solve_sudoku(board)**: This is the main backtracking function that iteratively tries to fill empty cells (0s). It uses recursion to fill the grid, and if an invalid configuration is reached, it backtracks to the previous state and tries a different number.
3. **print_board(board)**: This function prints the Sudoku grid in a readable format with each row on a new line.
4. **get_user_input()**: Prompts the user to input the Sudoku puzzle row by row. The user must input space-separated numbers for each row. Empty cells are represented by 0.

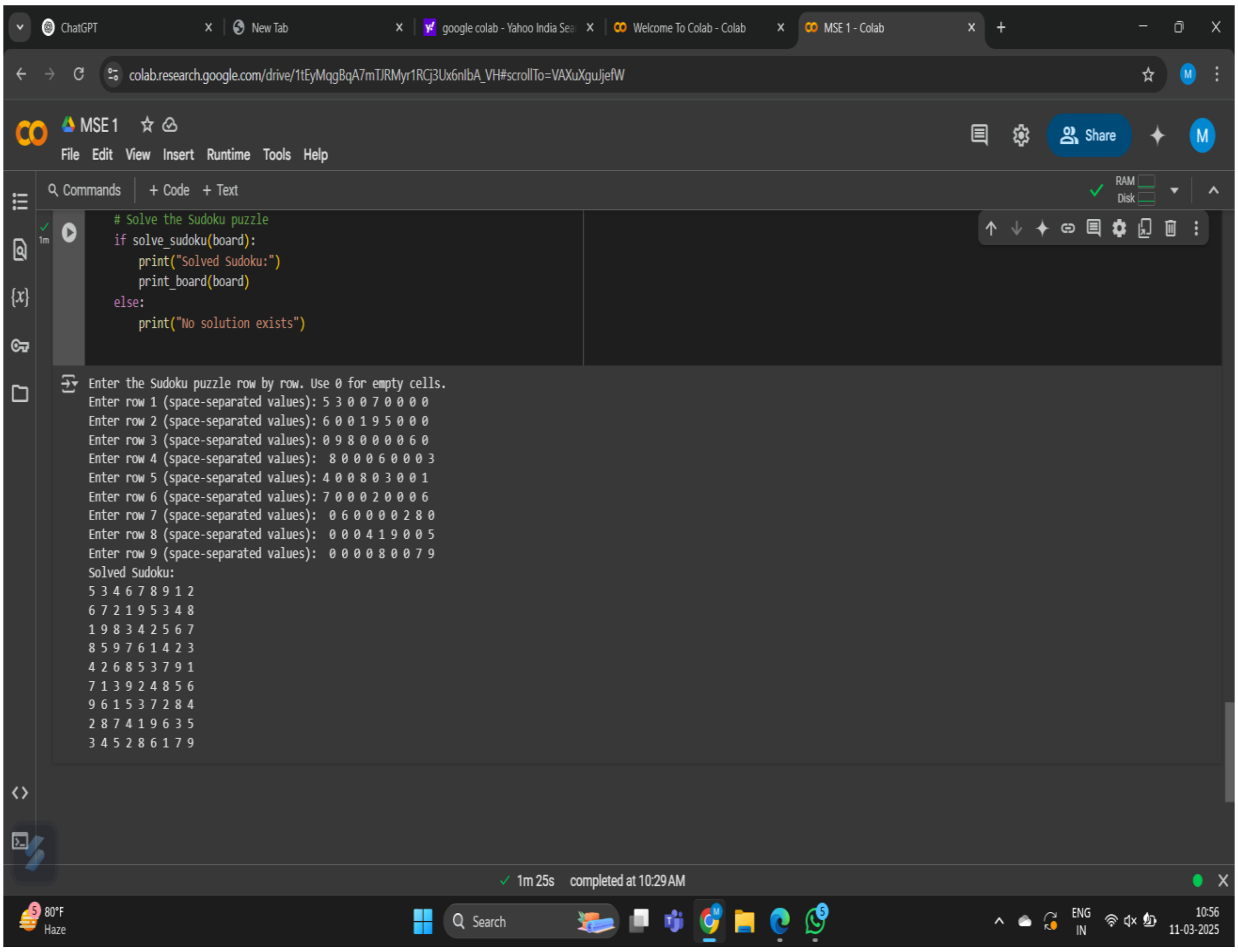
---

6. Performance Evaluation

Backtracking is a brute-force search algorithm that explores every possible combination. The time complexity for backtracking in the worst case is **O(9^n)**, where n is the number of empty cells. However, due to the constraints of Sudoku (i.e., rows, columns, and subgrids), the algorithm usually terminates faster than this theoretical worst case.

Optimizations:

1. **Minimum Remaining Values (MRV)**: When selecting the next empty cell, choosing the one with the least number of valid values reduces the search space.
2. **Forward Checking**: Eliminating values from other cells as soon as a number is placed can help to speed up the solving process.
3. **Constraint Propagation**: Deductions can be made early to eliminate invalid possibilities, reducing the search space.


MSE 1
File Edit View Insert Runtime Tools Help
Commands
+ Code + Text
Share
1m 25s completed at 10:29 AM

```
# Solve the Sudoku puzzle
if solve_sudoku(board):
    print("Solved Sudoku:")
    print_board(board)
else:
    print("No solution exists")
```

```
Enter the Sudoku puzzle row by row. Use 0 for empty cells.
Enter row 1 (space-separated values): 5 3 0 0 7 0 0 0 0
Enter row 2 (space-separated values): 6 0 0 1 9 5 0 0 0
Enter row 3 (space-separated values): 0 9 8 0 0 0 0 6 0
Enter row 4 (space-separated values):  8 0 0 0 6 0 0 0 3
Enter row 5 (space-separated values): 4 0 0 8 0 3 0 0 1
Enter row 6 (space-separated values): 7 0 0 0 2 0 0 0 6
Enter row 7 (space-separated values):  0 6 0 0 0 0 2 8 0
Enter row 8 (space-separated values):  0 0 0 4 1 9 0 0 5
Enter row 9 (space-separated values):  0 0 0 0 8 0 0 7 9
Solved Sudoku:
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
```