ABSTRACT

Sudoku is a popular logic-based combinatorial number-placement puzzle. Solving Sudoku involves placing numbers from 1 to 9 in a 9x9 grid such that each row, each column, and each of the nine 3x3 subgrids contains all of the digits from 1 to 9 without repetition. The backtracking algorithm is well-suited for solving Sudoku puzzles due to its ability to systematically explore potential solutions and backtrack when a solution is found to be invalid.

This Python implementation of a Sudoku solver utilizes the backtracking approach to solve a given Sudoku puzzle. The solver starts by finding an empty cell (represented by 0) and attempts to place numbers from 1 to 9 into that cell while ensuring that the placement satisfies Sudoku rules. If placing a number leads to a valid configuration, the solver recursively attempts to solve the rest of the puzzle. If a valid solution cannot be found with a particular number, the solver backtracks by undoing the placement and trying the next number.

Key functions in the implementation include:

- solve_sudoku(board): The main function that uses backtracking to solve the Sudoku puzzle. It returns True if a solution is found and modifies the input board to contain the solved puzzle.
- find_empty_cell(board): A helper function that identifies the next empty cell in the Sudoku grid.
- is_valid(board, row, col, num): A helper function that checks whether placing num in the cell (row, col) is valid according to Sudoku rules.

The implementation efficiently handles typical 9x9 Sudoku puzzles and outputs the solved puzzle if a solution exists. If no solution exists for the given puzzle configuration (due to constraints or errors), the solver appropriately notifies the user.

This Sudoku solver serves as a practical example of applying the backtracking algorithm to solve constraint satisfaction problems, demonstrating its effectiveness in tackling combinatorial puzzles like Sudoku.

CHAPTER 1:

INTRODUCTION

Sudoku is a well-known logic-based combinatorial number-placement puzzle. It consists of a 9x9 grid divided into nine 3x3 sub grids. The objective is to fill the grid with digits from 1 to 9, ensuring that each row, each column, and each of the nine 3x3 sub grids contains all of the digits exactly once. Originally popularized in Japan in the 1980s, Sudoku has since become a global phenomenon, cherished for its ability to challenge and entertain.

Solving Sudoku puzzles computationally involves employing various algorithms, with the backtracking algorithm being particularly effective due to its ability to systematically explore potential solutions while efficiently handling constraints. Backtracking works by attempting to place a number in an empty cell and recursively solving the remainder of the puzzle. If a placement leads to an invalid configuration (violating Sudoku rules), the algorithm backtracks, undoing the last placement and trying a different number.

In this context, this paper (or implementation) presents a Python program that utilizes the backtracking algorithm to solve Sudoku puzzles. The program systematically navigates through the puzzle, finding empty cells, making tentative placements, and verifying their validity until a solution is found or determined to be impossible. This approach not only demonstrates the application of backtracking in solving constraint satisfaction problems but also illustrates its efficiency in handling the complexities of Sudoku.

The subsequent sections will delve into the details of the implementation, discussing the core functions, the decision-making processes, and providing an analysis of its performance and effectiveness. By the end, readers will have a clear understanding of how backtracking can be leveraged to solve Sudoku puzzles and how this implementation can be adapted or enhanced for broader applications in combinatorial problem-solving.

CHAPTER 2:

PROBLEM STATEMENT

The goal of this project is to develop a Python program that can solve Sudoku puzzles using the backtracking algorithm. Sudoku is a 9x9 grid-based puzzle where the objective is to fill each cell with a number from 1 to 9 such that each row, each column, and each of the nine 3x3 sub grids (also known as boxes) contains all of the digits from 1 to 9 without repetition.

The program should:

1. Take a 9x9 grid as input, where empty cells are represented by the number 0.

2. Implement the backtracking algorithm to systematically attempt to fill each empty cell with a number from 1 to 9, ensuring that the placement is valid according to Sudoku rules.

3. Use recursive exploration to attempt different numbers in each empty cell, backtracking when a number is found to be invalid (i.e., violates Sudoku rules).

4. Output the solved Sudoku grid if a solution exists, or indicates that no solution exists for the given puzzle.

Key functions of the program include:

- solve_sudoku(board): A function that recursively attempts to solve the Sudoku puzzle using backtracking.
- find_empty_cell (board): A helper function that identifies the next empty cell in the Sudoku grid.
- Is_valid (board, row, col, num): A helper function that checks whether placing num in the cell (row, col) is valid according to Sudoku rules.

The program should handle typical 9x9 Sudoku puzzles efficiently, providing solutions for a variety of difficulty levels. It should also include error-checking mechanisms to validate input and notify the user if an invalid Sudoku puzzle configuration is provided.

The objective is to demonstrate proficiency in implementing the backtracking algorithm for constraint satisfaction problems and showcase its application in solving Sudoku puzzles effectively.

Possible approaches:

- Backtracking: This is a common approach that explores possible placements of digits in empty cells while ensuring they follow the Sudoku rules. If a placement leads to a conflict, the algorithm backtracks and tries alternative placements.

CHAPTER 3:

ALGORITHM

Sudoku Solver Using Backtracking

1. Input:

- Board: A 9x9 Sudoku grid represented as a 2D list in Python. Empty cells are represented by 0.

2. Function Definition:

- solve_sudoku(board)
- Base Case: If there are no empty cells left (find_empty_cell (board) returns none), the puzzle is solved, so return True.
- Recursive Case:
- Find the next empty cell (row, col) using find_empty_cell (board).
- Try numbers from 1 to 9:
- If placing num in (row, col) is valid (is_valid(board, row, col, num) returns True):
- Place num in (row, col).
- Recursively attempt to solve the rest of the board (solve_sudoku (board)).
- If the recursive call returns True, the puzzle is solved, so return True.
- If the recursive call returns False, backtrack by undoing the placement (board [row] [col] = 0) and try the next number.
- If no number from 1 to 9 leads to a solution, return False to trigger backtracking at the previous level.

3. Helper Functions:

- find_empty_cell (board)
- Iterate through the board to find the first cell with value 0 (indicating it's empty). Return its (row, col) coordinates.
- If no empty cell is found, return None.
- is_valid (board, row, col, num)
- Check if num can be placed in (row, col) without violating Sudoku rules:
- Check the current row: Ensure num is not already present in the same row (board[row][j] == num for all j).

- Check the current column: Ensure num is not already present in the same column (board[i][col] == num for all i).
- Check the 3x3 sub grid (box): Determine the starting row and column of the sub grid (box_start_row, box_start_col). Ensure num is not already present in the sub grid (board[i][j] == num for all cells (i, j) within the sub grid).

4. Execution:

- Initialize a 9x9 Sudoku board (board).
- Call solve_sudoku (board).
- If solve_sudoku (board) returns True, print or return the solved board.
- If solve_sudoku (board) returns False, indicate that no solution exists for the given puzzle configuration.

CHAPTER: 4

IMPLEMENTATION

```
Def solve_sudoku (board):
    # Base case: If there are no empty cells left, puzzle is solved
    empty_cell = find_empty_cell (board)
    If not empty_cell:
        return True

    # Unpack row and col from empty_cell tuple
    row, col = empty_cell

    # Try numbers 1 through 9
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            # If valid, place the number
            board[row][col] = num

            # Recursively attempt to solve the rest of the board
            if solve_sudoku(board):
                return True

            # If recursion fails, backtrack
            board[row][col] = 0

    # If no number worked, return False (triggering backtracking)
    return False

def find_empty_cell(board):
    # Find the first empty cell (represented by 0)
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return (i, j)
    return None
```

```
def is_valid(board, row, col, num):
    # Check if placing num at board[row][col] is valid
    # Check the row
    if num in board[row]:
        return False

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

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

    return True

# Example usage:
if _name_ == "_main_":
    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]
    ]

    if solve_sudoku(board):
```

```
        print("Sudoku Solved:")
        for row in board:
            print(row)
    else:
        print("No solution exists")
```

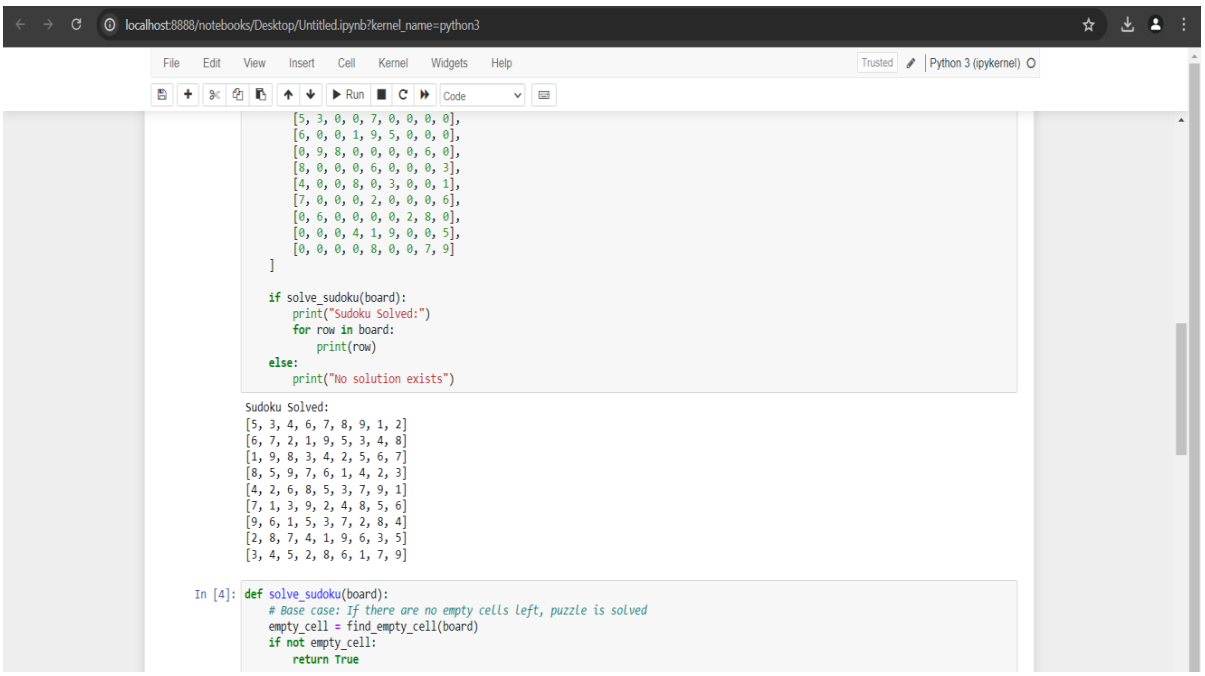
Time Complexity: O(9^(n^2)), where n is the number of empty cells.

Space Complexity: O(n^2).

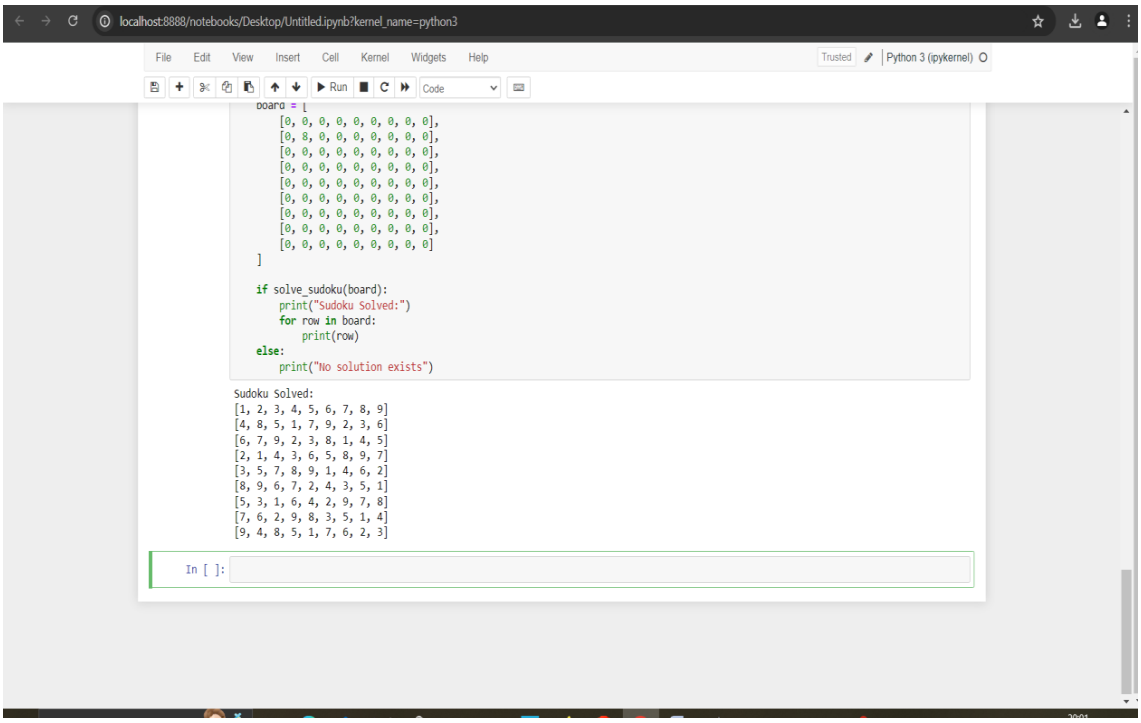
CHAPTER 5:

RESULTS

OUTPUT 1:



OUTPUT 2:



CHAPTER 6:

CONCLUSION

The provided Python program implements a backtracking algorithm to solve a given Sudoku puzzle. Here's a detailed breakdown of how it works:

1. Base Case and Recursion:

- The `solve_sudoku` function is the main function that attempts to solve the Sudoku puzzle.
- It first calls `find_empty_cell` to locate the first empty cell (represented by 0).
- If no empty cell is found, it means the puzzle is solved, and the function returns `True`.
- If an empty cell is found, it tries placing numbers from 1 to 9 in that cell.
- For each number, it checks if placing that number is valid using the `is_valid` function.
- If valid, it places the number and recursively calls `solve_sudoku` to attempt to solve the rest of the puzzle.
- If the recursive call returns `True`, it means the puzzle has been solved, and the function returns `True`.
- If placing the number doesn't lead to a solution, it backtracks by resetting the cell to 0 and tries the next number.
- If no number from 1 to 9 can be placed validly, the function returns `False`, indicating the need to backtrack.

2. Finding Empty Cells:

- The `find_empty_cell` function scans the board for the first empty cell (represented by 0) and returns its coordinates.
- If no empty cell is found, it returns `None`.

3. Validity Check:

- The `is_valid` function checks if placing a specific number at a specific cell is valid.
- It checks the row, column, and the 3x3 box that the cell belongs to, ensuring the number does not already exist in any of these.

4. Example Usage:

- The example usage demonstrates solving a Sudoku board initialized with mostly zeros.

- If a solution exists, the solved board is printed; otherwise, a message indicating no solution exists is displayed.

Conclusion:

This program effectively solves Sudoku puzzles using a systematic backtracking approach. The key to its success lies in the `is_valid` function, which ensures that every move respects Sudoku rules, and in the recursive structure of `solve_sudoku`, which explores potential solutions depth-first. This method guarantees that, if a solution exists, it will be found. If no solution exists, the program will correctly indicate this. This solver can be applied to any valid Sudoku puzzle, making it a versatile tool for Sudoku enthusiasts.