Interactive Sudoku Solver

A Game of Logic Using DFS and Backtracking

Statement of the Problem:

The task is to solve a given 9x9 Sudoku puzzle using a backtracking algorithm. A Sudoku puzzle is a partially filled 9x9 grid where the goal is to assign digits (from 1 to 9) to the empty cells in such a way that every row, every column, and every 3x3 subgrid contains each digit exactly once. The solution must ensure that the puzzle remains valid at each step.

Additionally, the program should have two modes:

- **Mode 1:** The program generates a Sudoku puzzle (easy, medium, or hard) and allows the user to solve it.
- **Mode 2:** The user inputs a Sudoku puzzle, and the program checks whether a solution exists.

Explanation of the Problem:

In Sudoku, each row, each column, and each of the nine 3x3 subgrids must contain the numbers 1 to 9 exactly once. The challenge is to fill in the empty cells without violating these constraints.

To solve the puzzle programmatically:

1. We need to fill in the blanks by trying all possible numbers (from 1 to 9) and ensure that each placement follows the Sudoku rules.
2. If placing a number at a particular position violates the rules (making the puzzle invalid), we backtrack and try another number.
3. The backtracking algorithm will systematically explore all possible valid configurations to solve the puzzle.

Steps/Algorithm to Solve the Problem:

The algorithm combines Depth-First Search (DFS) with constraint checking, gradually building up possible solutions. It explores each possible solution path, and whenever a constraint is violated (i.e., a solution doesn't meet the Sudoku

rules), it backtracks and tries a different path. This method efficiently narrows down the search space, eliminating invalid solutions early.

Algorithm:

1. **Check if the grid is safe (Valid Placement)**:
    - Before assigning a number to an empty cell, check if it violates any of the Sudoku constraints (i.e., the number already exists in the row, column, or 3x3 subgrid).
    - If the placement violates a constraint, the solution is discarded (pruned), and we move on to the next possible number.
2. **Depth-First Search (DFS) + Constraints**:
    - The algorithm uses DFS to explore all possible configurations of the grid.
    - DFS explores each number for the current empty cell. If a number is valid (safe according to constraints), it is placed, and the algorithm continues to explore the next empty cell.
    - If a number leads to a valid solution, the search terminates successfully. If it doesn't, the algorithm backtracks by removing the number (undoing the placement) and trying the next number in the range from 1 to 9.
    - This process continues until the entire grid is filled or no solution exists.
3. **Recursive Backtracking Function**:
    - **Base Case**: If no unassigned cell is found, the puzzle is solved, and the function returns true.
    - **Find the first unassigned location**: Identify the first unassigned (empty) cell in the grid.
    - **Try placing numbers 1 to 9**:
        - For each number, check if it violates any constraints using the validity check.
        - If the placement is valid, recursively call the backtracking function for the next empty cell.

        - If the recursive call returns true, the puzzle is solved, and we return true.
        - If the placement doesn't lead to a valid solution, backtrack by removing the number and trying the next one.
    - If none of the numbers from 1 to 9 leads to a valid solution, return false (no solution exists).
4. **Time Complexity**:
    - **O(9^(n*n))**: Since each empty cell can hold 9 possible values, the time complexity is exponential in the number of empty cells.
    - Although the time complexity is high, the use of DFS + constraints helps prune invalid configurations early, improving practical performance compared to a brute-force approach.
5. **Space Complexity**:
    - **O(N*N)**: We need space to store the grid (9x9 matrix).