Sudoku Solver Project

03.06.2025

Data structure & Algorithms

1. Detailed project description

The project is a **Sudoku puzzle solver** designed for a Data Structures and Algorithms course. The primary goal of the project is to solve any valid 9x9 Sudoku puzzle using logic-based techniques and optimized data structures.

The application starts by analyzing the puzzle to determine the frequency of each number (1–9) already placed on the board. Using this frequency analysis, it prioritizes solving for the most common numbers first. For each number, it identifies subgrids (3x3 blocks) where that number is missing, and then attempts to place the number only if there is **exactly one**

valid position within that subgrid (a strategy known as **“hidden singles”**). This ensures placements are logically sound.

The system is structured to support further enhancements such as backtracking if the logic-only approach does not complete the solution.

Main Components:

- A 9x9 matrix representing the Sudoku board.
- A solver function using logical deduction based on subgrid constraints.
- A frequency analyzer for number prioritization.
- A placement validator to ensure correctness before inserting any value.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper.

2. Data Structures Used

- **2D Array:**:
  Used to represent the Sudoku grid. Each element in the 9x9 matrix holds a value from 0 to 9, where 0 represents an empty cell.

  Why? It's the most natural and efficient structure for accessing and updating rows, columns, and subgrids in constant time.

- **Counter (from `collections`)**:
  Used to count the frequency of digits (1–9) on the board. The result is sorted to prioritize solving for the most frequent numbers.

  Why? It simplifies counting and ranking digit frequencies efficiently.

- **List of Tuples**:
  Temporarily used to hold all valid positions for a given number within a subgrid. If the list has only one element, the number is placed there.

  Why? Lightweight and easy to traverse for single-placement validation.

3. Use Case or Scenario

Scenario: A student or puzzle enthusiast is faced with a partially completed Sudoku puzzle (from a newspaper, mobile app, or textbook) and wants to verify the solution or get assistance solving it.

value:

- Helps users solve puzzles logically without brute force or trial-and-error.
- Can serve as a teaching tool to demonstrate core Sudoku-solving strategies.
- Acts as a foundation for more advanced AI solvers or puzzle game engines.

4. Main Operations Implemented

- **Search**:
  For every number, the program scans rows, columns, and subgrids to find missing entries and valid positions.
- **Validation**:
  Before placing any number, it checks whether the move satisfies Sudoku rules (no repeats in row, column, or 3x3 subgrid).
- **Insert/Update**:
  Once a safe and logical placement is found, the value is inserted into the grid.
- **Frequency Analysis (Sort + Count)**:
  Counts the frequency of existing digits and sorts them in descending order to prioritize processing.
- **Traversal**:
  Iterates through all cells, rows, columns, and subgrids to locate missing values and evaluate constraints.

5. User Interaction

Currently, the program runs as a **command-line application**. The user provides the Sudoku puzzle as a 2D list (matrix) in the source code or through future input functions.

The program prints the board step-by-step after each confident placement, allowing users to visually follow the solving process. This is especially useful for learning and debugging.

Planned Enhancements:

- Adding command-line input or file reading support.
- Optional GUI for visual interaction using libraries like `tkinter` or `pygame`.

6. Challenges or Uncertainties

- **Correct Placement Mechanism**:
  Placing a number too early or in an incorrect position causes the entire solution to collapse. A major challenge was ensuring that placements are made only when logically certain (not just valid). This was resolved by requiring **exactly one valid position** in a subgrid before placing a number.
- **No Backtracking Yet**:
  The current logic-based solver works well for medium-difficulty puzzles. However, for harder puzzles with fewer clues, it may stall. Implementing a recursive backtracking fallback system is a potential challenge that will be tackled next.
- **Time Complexity**:
  As the puzzle becomes more complex, especially when backtracking is added, managing performance and avoiding unnecessary re-checks will be important.