Sudoku Solver – Detailed Project Overview

The **Sudoku Solver** is a terminal-based application designed to allow users to input, solve, and manage Sudoku puzzles interactively. It leverages a combination of fundamental and advanced Data Structures and Algorithms (DSA) to deliver a comprehensive and efficient user experience. The program is modular, with clearly defined components to handle puzzle input, solving logic, performance analysis, persistent storage, and user interaction.

Main Components

1. Main Menu

The main menu acts as the central navigation hub of the application. It runs in a continuous loop until the user chooses to exit. It provides the following options:

- **Input Puzzle** – Allows manual entry or loading from a file.
- **Display Puzzle** – Displays the current puzzle in a grid format.
- **Solve Puzzle** – Automatically solves the current puzzle using an optimized algorithm.
- **Performance Report** – Shows the time and resource usage for solving the puzzle.
- **Manage Puzzles** – Enables saving, loading, editing, and deleting puzzle entries.
- **Display Guide** – Provides user instructions for utilizing all features.
- **Exit** – Terminates the application safely.

---

2. Puzzle Input

The `inputPuzzle()` function facilitates puzzle entry through two methods:

- **Manual Input**: Users can enter the values for each cell in a 9x9 grid manually. Difficulty settings may be provided to control how much of the puzzle is pre-filled.

- **File Input**: Users can load a Sudoku puzzle from a predefined file. The file contains puzzle data in a specific format readable by the program.

The input is stored in a 2D vector (9x9) representing the Sudoku grid.

---

3. Puzzle Solver

The `solveSudoku()` function is the core of the solving engine and uses the **backtracking algorithm**. It operates recursively to attempt to fill each empty cell (denoted by 0) with a valid number from 1 to 9. If a valid configuration cannot be achieved, the function backtracks and tries a different value.

To enhance the solving process, the solver includes the **Minimum Remaining Values (MRV) heuristic**, implemented in `optimizeSolver()`:

- **MRV Heuristic**: Prioritizes cells with the fewest legal values, reducing unnecessary recursion and improving performance.

- Constraints for rows, columns, and 3x3 boxes are enforced using efficient set-based lookups.

---

4. Performance Report

The `performanceReport()` function measures and displays the time taken to solve the current puzzle. It uses the **`high_resolution_clock`** from C++'s `<chrono>` library for precise time tracking. The report includes:

- **Execution Time**: Duration in milliseconds.

- **Time Complexity Estimate**: Based on theoretical analysis of the backtracking approach.

- **Space Complexity Estimate**: Memory used during execution, based on grid size and auxiliary data structures.

This module helps in understanding the efficiency of the solving algorithm under different puzzle complexities.

---

5. Puzzle Management

The `managePuzzles()` function provides comprehensive database-like functionality:

- **Save Puzzle**: Stores the current puzzle (solved or unsolved) to a file.
- **Load Puzzle**: Retrieves a saved puzzle from storage.
- **Update Puzzle**: Modifies an existing puzzle in the storage file.
- **Delete Puzzle**: Removes a specific puzzle from the database.

Puzzles are encapsulated in a `PuzzleRecord` structure containing:

- A 9x9 grid (vector of vectors).
- Timestamp of creation or last update.
- Solved status (boolean flag).

File I/O operations handle persistent storage, using readable and structured formats (e.g., plain text or CSV).

---

6. User Guide

The `displayGuide()` function offers users clear instructions on how to operate the application. It includes:

- Step-by-step guidance on entering and solving puzzles.
- Descriptions of menu options and their purposes.
- Tips for understanding solver performance output.
- Notes on how puzzles are stored and retrieved.

This guide ensures that users of any technical level can navigate the application effectively.

DSA Concepts Used

1. Vectors

Vectors are dynamic arrays from the C++ STL used to store the Sudoku grid. They allow for:

- Efficient indexing and resizing.
- A clean representation of the 9x9 matrix.
- Easy iteration during solving and display processes.

cpp
CopyEdit
```
std::vector<std::vector<int>> sudokuGrid(9, std::vector<int>(9, 0));
```

---

2. Unordered Maps and Sets

These structures are employed to track and enforce Sudoku rules:

- **Unordered Sets** hold the digits already used in each row, column, and 3x3 box.
- **Unordered Maps** associate cell positions with constraints.

These allow constant average-time complexity for operations such as checking if a number is valid in a given position.

---

3. Backtracking Algorithm

A fundamental algorithm for solving constraint satisfaction problems:

- Begins at the first empty cell.
- Tries placing digits 1 through 9, checking constraints.

- Recursively solves the next cell.
- If no number fits, backtracks to try different values in previous cells.

This recursive process continues until a solution is found or all possibilities are exhausted.

---

4. Minimum Remaining Values (MRV) Heuristic

A performance optimization for backtracking:

- Identifies the most constrained cell (i.e., the one with the fewest possible valid entries).
- Reduces branching in the recursion tree.
- Results in faster puzzle-solving by limiting unproductive search paths.

---

5. File I/O

Handles saving and loading puzzle data:

- Uses standard file streams (`ifstream`, `ofstream`) for reading and writing.
- Data is structured in a consistent format for easy parsing.
- Enables persistent puzzle management across application sessions.

Example pseudocode:

cpp
CopyEdit

```
std::ofstream file("puzzle.txt");
for (auto& row : sudokuGrid) {
    for (int val : row) file << val << " ";
    file << "\n";
}
```

---

6. Multithreading

Enhances the responsiveness of the application:

- The `displayLoadingAnimation()` function uses a separate thread to show an animated loading screen.
- While the main thread processes heavy tasks like puzzle solving or file I/O, the UI remains active.
- Improves user experience by avoiding freezes or lag.

Conclusion

This terminal-based Sudoku Solver exemplifies the practical use of Data Structures and Algorithms in application development. It combines:

- Dynamic memory structures (vectors),
- Efficient constraint tracking (unordered sets/maps),
- Powerful recursive algorithms (backtracking with MRV),
- Persistent data handling (file I/O),
- And responsive user interaction (multithreading).

Together, these components form a robust system that is both educational and functional, making it an excellent project for learning and applying core programming and algorithmic concepts.