**CSD311: Artificial Intelligence****Project Report**

**SUDOKU**

**Prof. Snehasis Mukherjee**

Group Members –

Aditya Doneria(21101100030)

Aditya Mukerji(2110110032)

INTRODUCTION:

Sudoku Solver, an engaging computational challenge, delves into the art of logic and deduction. Set in the structured 9x9 grid world of numbers, the program tackles the timeless puzzle of filling empty cells with digits from 1 to 9. Players assume the role of a solver, navigating constraints imposed by rows, columns, and 3x3 subgrids. Through dynamic backtracking and strategic elimination, the solver explores countless possibilities, balancing precision and efficiency. Each placement carries the weight of its consequences, turning Sudoku Solver into not just a test of algorithmic prowess but a captivating journey into problem-solving and mathematical elegance.

PROBLEM ANALYSIS:

The Sudoku solver code demonstrates a solid implementation of backtracking combined with constraint optimization, but there are areas for improvement. The primary challenge lies in further enhancing the efficiency of the backtracking process. While sorting empty cells by the number of possible candidates reduces unnecessary branching, more advanced techniques, such as constraint propagation or implementing the Dancing Links algorithm, could significantly speed up the search. Additionally, the code's memory management could be optimized by reducing redundant set operations or leveraging bitwise operations for tracking constraints. Exploring parallel processing for independent subgrid solving or applying machine learning techniques to predict high-probability values could also improve performance. The problem analysis centers on balancing algorithmic sophistication, execution speed, and maintainability to create an even more efficient and intelligent Sudoku solver.

SOLVER IDEATION:

To enhance the Sudoku solver in our code, we considered implementing more advanced techniques for constraint satisfaction and search optimization. One approach is to integrate the **Dancing Links (DLX) algorithm**, which represents constraints as a sparse matrix and uses an efficient backtracking mechanism to solve exact cover problems like Sudoku. This method significantly reduces memory usage and speeds up solving.

We also explored constraint propagation techniques, such as the forward-checking algorithm combined with arc consistency (AC-3). These methods help prune the search space early by eliminating invalid possibilities before deeper recursion.

Another option is to employ machine learning models, where a neural network, trained on a large dataset of solved puzzles, predicts high-probability candidates for empty cells, guiding the backtracking process.

Finally, we considered parallelizing the solving process by dividing the board into independent sections when feasible, leveraging multi-threading or GPU-based computation for faster results. The focus is on balancing algorithm complexity, real-time efficiency, and scalability to create an advanced and adaptable Sudoku solver.

DATA STRUCTURES USED:

The primary data structures used in the Sudoku solver code are **sets** and **lists**, tailored to manage constraints and the board state efficiently.

* **2D List**: The 9x9 Sudoku board is represented as a 2D list, where each element corresponds to a cell on the grid. Empty cells are denoted by 0, while filled cells hold their respective numerical values.
* **Sets**: Arrays of sets are employed to track possible numbers for each row, column, and 3x3 subgrid. These sets allow for efficient addition, removal, and intersection operations, critical for enforcing Sudoku rules and determining valid candidates for empty cells.
* **List of Tuples**: Empty cells are stored as a list of (row, column) tuples, enabling sorted traversal based on the number of possible candidates for each cell.

GOAL ORDERING:

The primary goal of the Sudoku solver code is to provide an efficient and accurate solution to any valid 9x9 Sudoku puzzle. The initial goal involves parsing the puzzle, identifying empty cells, and determining possible candidates for each cell based on Sudoku rules.

The main solving loop focuses on filling these empty cells through a backtracking algorithm that evaluates candidate numbers while adhering to the constraints of rows, columns, and subgrids. Sub-goals within the solving process include:

1. Sorting empty cells by the number of potential candidates to minimize branching.
2. Placing a valid number in a cell and updating the associated constraint sets.
3. Reverting changes (backtracking) if no valid moves remain.

The overarching objective is to create an optimized and user-friendly Sudoku solver that efficiently handles puzzles of varying difficulty, ensuring correctness and scalability.

DEADLOCK CONDITIONS USED:

The deadlock conditions in the Sudoku solver code arise when the backtracking algorithm encounters a state where no valid number can be placed in an empty cell without violating Sudoku rules. This occurs when:

1. The **possible numbers set** for a cell becomes empty due to constraints from the row, column, or subgrid, making further progress impossible.
2. A number placement causes cascading constraint violations, leaving no valid moves for subsequent cells.

The solver detects these conditions dynamically during the backtracking process. If a deadlock is encountered, the algorithm reverts (backtracks) to the previous cell, removes the conflicting number, and tries the next possible candidate. If all candidates for a cell fail, the algorithm continues backtracking until a valid solution path is found or determines that no solution exists. These deadlock conditions ensure the algorithm exhaustively explores all possibilities, providing a definitive resolution to the puzzle.

ALGORITHMS:

The key algorithm used in the Sudoku solver code is a **constraint-driven backtracking algorithm**. This algorithm systematically fills the empty cells on the Sudoku board while ensuring that the rules of Sudoku are upheld—each number from 1 to 9 must appear exactly once in every row, column, and 3x3 subgrid.

The algorithm begins by identifying all empty cells on the board and sorting them based on the number of potential candidates for each cell. This **dynamic variable ordering** heuristic minimizes branching by prioritizing cells with the fewest possibilities, thus reducing the number of recursive calls needed to reach a solution.

During the recursive solving process, the algorithm:

1. Selects an empty cell.
2. Iterates over its possible candidates, derived from the intersection of allowed numbers in its row, column, and subgrid.
3. Places a candidate number in the cell and updates the constraint sets for rows, columns, and subgrids.
4. Proceeds to the next cell recursively. If it encounters a state where no valid number can be placed (a deadlock), it backtracks to the previous cell, removes the number, and tries the next candidate.

The efficiency of the algorithm is further enhanced by **constraint propagation**. Each number placement dynamically updates the constraint sets, allowing the algorithm to eliminate invalid candidates for subsequent cells early in the search.

CODE FLOW:

1. **Initialization:**
   * The board is defined as a 9x9 grid represented by a 2D list, where empty cells are marked as 0.
   * Constraint sets (rows, cols, grids) are initialized for each row, column, and 3x3 subgrid, holding the possible numbers (1–9) for that segment.
   * The empty\_cells list is populated with the coordinates of all empty cells on the board.
   * The possible candidates for each cell are calculated using the get\_possible\_numbers function, and the empty\_cells list is sorted based on the number of candidates for each cell.
2. **Constraint Calculation:**
   * The get\_possible\_numbers function computes valid candidates for a given cell by intersecting the constraint sets for its row, column, and subgrid.
   * The get\_grid\_index function maps a cell's coordinates to its respective 3x3 subgrid for efficient indexing.
3. **Printing the Board:**
   * The print\_board function displays the current state of the board, replacing empty cells with dots (.) for better readability.
4. **Backtracking Solver:**
   * The solve\_sudoku\_optimized function employs a recursive backtracking algorithm to solve the puzzle.
   * It starts with the first empty cell in the empty\_cells list and attempts to place a valid number from its candidate set.
   * After placing a number, the constraint sets are updated to reflect the placement.
   * The algorithm then recursively attempts to solve the next empty cell.
   * If a deadlock (no valid moves) is encountered, the algorithm backtracks, removing the last placed number, restoring the constraints, and trying the next candidate.
5. **Deadlock Handling:**
   * During recursion, if no valid candidates remain for a cell, the algorithm backtracks to the previous cell, ensuring exhaustive exploration of possible configurations.
6. **Solution Completion:**
   * If the backtracking successfully fills all empty cells, the function exits and returns the solved board.
   * If no solution exists, the function indicates failure.
7. **Performance Metrics:**
   * The program tracks the total number of iterations performed during backtracking to evaluate efficiency.
   * Execution time is measured to provide insight into the solver's performance.
8. **Main Execution:**
   * The board is initialized and printed in its unsolved state.
   * The solve\_sudoku\_optimized function is called to solve the puzzle.
   * Upon completion, the solved board is printed along with performance metrics such as the time taken and iteration count.

OPTIMIZATION OF MEMORY USE:

To optimize memory usage in the Sudoku solver code, consider replacing the list of sets used for rows, columns, and subgrids with more compact representations, such as bitmasks. Bitwise operations can efficiently track and manipulate possible candidates for each row, column, and subgrid while significantly reducing memory overhead.

Additionally, minimize the storage of intermediate states by directly updating and restoring the board and constraint structures during backtracking, avoiding the need for additional data structures. The empty\_cells list could also be optimized by dynamically maintaining it as a priority queue, reducing the overhead of sorting it repeatedly.

Where possible, avoid redundant recalculations by caching results of get\_possible\_numbers for frequently accessed cells. Conduct memory profiling to identify bottlenecks and verify that these optimizations do not compromise the clarity or functionality of the code.

**References:**

* *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
* Focus: Comprehensive guide on algorithms, including backtracking and optimization techniques.
* *Constraint Processing* by Rina Dechter.
* Focus: Detailed exploration of constraint satisfaction problems, relevant for Sudoku solvers.
* Python Official Documentation: *Built-in Types — Lists and Sets*.
* <https://docs.python.org/3/library/stdtypes.html>