SUDOKU SOLVER

Name :- Hiten Mehra

Branch :- CSE(AI)

Section :- B

University Roll No. :- 2024011100300126

Report on Sudoku Solver

1. Introduction

Sudoku is a popular logic-based combinatorial number-placement puzzle. The objective is to fill a 9x9 grid with digits from 1 to 9, so that each row, each column, and each of the nine 3x3 subgrids (also called "boxes") contain all of the digits from 1 to 9 without repetition.

A **Sudoku Solver** is a computer algorithm that is designed to automatically solve Sudoku puzzles, either by determining a solution for a given puzzle or by providing a mechanism to check if a puzzle is solvable.

2. Problem Definition

In Sudoku, the puzzle starts with a partially filled grid, and the solver's task is to determine the correct values for the remaining empty cells while respecting the constraints of the puzzle:

- Each number 1 through 9 must appear exactly once in each row.
- Each number 1 through 9 must appear exactly once in each column.
- Each number 1 through 9 must appear exactly once in each of the nine 3x3 subgrids.

3. Types of Sudoku Solvers

There are several different algorithms to solve Sudoku puzzles. These can broadly be categorized as:

3.1 Backtracking Algorithm

Backtracking is a type of depth-first search algorithm where the solver tries to place numbers in empty cells one by one and backtracks if it encounters a conflict. This method is often referred to as a "trial and error" approach.

Steps of the Backtracking Algorithm:

1. Start from the first empty cell in the grid.
2. Try placing numbers from 1 to 9 in the cell.
3. For each number, check whether placing it violates Sudoku's constraints (row, column, and subgrid uniqueness).
4. If a number fits, move to the next empty cell and repeat the process.
5. If the number doesn't fit, backtrack and try the next possible number in the previous cell.
6. Repeat until the entire grid is filled.

Advantages of Backtracking:

- Simple and easy to implement.
- Guaranteed to find a solution (if one exists) through exhaustive search.

Disadvantages of Backtracking:

- Computationally expensive for large grids or more difficult puzzles (though it's effective for 9x9 grids due to their relatively small size).
- May take a significant amount of time for very complex puzzles.

3.2 Constraint Propagation (e.g., Dancing Links)

Constraint propagation algorithms, like Dancing Links, reduce the search space by propagating constraints as the algorithm progresses. The algorithm keeps track of which numbers are possible for each cell based on the current state of the grid. By eliminating impossible choices early, these methods speed up the solving process.

Advantages:

- Can solve puzzles faster than backtracking for certain cases.
- More efficient in terms of reducing the search space early on.

Disadvantages:

- More complex to implement than backtracking.

- Computational overhead in maintaining constraint sets.

3.3 Exact Cover

The Exact Cover approach solves the Sudoku puzzle by converting it into a matrix problem, where the goal is to cover all the constraints of the Sudoku board without duplication.

One such algorithm that utilizes Exact Cover is **Knuth's Algorithm X**, which efficiently handles these types of problems through a backtracking approach that optimizes constraint satisfaction.

Advantages:

- Extremely efficient for solving standard Sudoku puzzles.
- Can be adapted for use in other constraint satisfaction problems.

Disadvantages:

- Complex to understand and implement.
- Requires significant memory and preprocessing.

4. Solving a Sudoku Puzzle Using Backtracking Algorithm (Pseudocode)

Here’s a simplified pseudocode for solving a Sudoku puzzle using the backtracking algorithm:

```
plaintext

Copy

function solveSudoku(board):
    if no empty cell is found:
        return True  // Puzzle solved
    find the next empty cell (row, col)
    for num from 1 to 9:
        if the number can be placed in the cell (row, col) without violating the constraints:
            place num in the cell
            if solveSudoku(board):
                return True  // Recursive call
            remove num from the cell (backtrack)
    return False  // No valid number found, need to backtrack
```

5. Complexity Analysis

Time Complexity:

The time complexity of a backtracking Sudoku solver can be quite high due to the exhaustive search:

- The worst-case scenario occurs when every cell is empty, and the algorithm has to try every possible number for every cell.
- The time complexity can be expressed as **O(9^(N^2))**, where N is the size of the grid (usually 9 for standard Sudoku).

However, this worst-case scenario is very rare, and for most puzzles, backtracking will solve the puzzle quickly due to the constraints that eliminate many possibilities early on.

Space Complexity:

The space complexity of the backtracking approach is **O(N^2)**, where N is the grid size. This is because the algorithm stores the state of the entire grid while trying out different solutions.

6. Optimizations and Improvements

6.1 Heuristics:

- **Most Constrained Variable:** This heuristic involves choosing the next empty cell that has the fewest possible valid numbers. This minimizes the number of choices at each step, speeding up the solving process.

- **Least Constraining Value:**
- This heuristic involves trying numbers that leave the maximum number of valid choices for other cells. This improves the solver's efficiency by reducing the likelihood of encountering dead-ends.

6.2 Parallelization:

For extremely large Sudoku puzzles or when multiple Sudoku puzzles need to be solved simultaneously, parallelization of the backtracking algorithm can speed up the process by distributing the workload across multiple processors.

6.3 Advanced Algorithms (e.g., Dancing Links, Knuth's Algorithm X):

These algorithms can solve puzzles faster than traditional backtracking methods by reducing the search space and using advanced techniques to propagate constraints and handle exact covers.

7. Applications of Sudoku Solvers

- **Puzzle Solving:**
- The primary use of Sudoku solvers is to automate the solving of Sudoku puzzles in newspapers, apps, and websites.

- **Constraint Satisfaction Problems (CSPs):** Techniques used in Sudoku solvers can be applied to other CSPs in areas such as scheduling, planning, and logic puzzles.
- **AI & Game Theory:** Sudoku solvers are used as examples in AI research to demonstrate problem-solving algorithms and search techniques.

8. Conclusion

Sudoku solvers play a crucial role in automating the process of solving these popular puzzles. While basic backtracking remains a fundamental approach for solving most puzzles, advanced methods like constraint propagation and exact cover algorithms offer more efficient solutions for larger or more complex puzzles. The techniques used in Sudoku solvers also have broader applications in AI and other fields that deal with constraint satisfaction problems.

By understanding and implementing these solvers, we gain insight not only into the mechanics of the puzzle but also into how algorithms can be applied to real-world problems requiring efficient problem-solving techniques.

```python
def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False

    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True

def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == 0:
                for num in range(1, 10):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = 0
                return False
    return True

def print_board(board):
    for row in board:
        print(" ".join(str(num) if num != 0 else '.' for num in row))
```

```python
def get_user_input():
    board = []
    print("Enter Sudoku puzzle row by row (use 0 for empty cells):")
    for _ in range(9):
        row = list(map(int, input().split()))
        board.append(row)
    return board

if __name__ == "__main__":
    sudoku_board = get_user_input()
    print("\nSudoku Puzzle:")
    print_board(sudoku_board)

    if solve_sudoku(sudoku_board):
        print("\nSolved Sudoku:")
        print_board(sudoku_board)
    else:
        print("\nNo solution exists.")
```