

**Department of Computer Engineering**

**Course: DAA**

**Course Code: BTECCE21501**

**Mini-Project - Report**

**PHASE - II**

**Guidance By - Prof. Trupti**

**Topic: Algorithmic Solutions for Solving Sudoku Puzzles**

**By**

|  |  |  |
| --- | --- | --- |
| **Roll No** | **SRN** | **Name of Student** |
| 38 | 202200930 | Shyamal S. Patil |
| 39 | 202200931 | Shaikh M. Asad |
| 33 | 202101389 | Tanmay Gawali |

**1. Introduction**

* **1.1 Brief Introduction to Sudoku and Its Rules**
* Sudoku is a classic puzzle that involves filling a 9x9 grid with digits from 1 to 9.
* Each row, column, and the nine 3x3 subgrids, known as "regions," must contain all the digits from 1 to 9 without repetition.
* Sudoku puzzles are popular for their simple rules but challenging gameplay, making them an interesting problem for algorithmic solutions.
* **1.2 Purpose and Scope of the Project**
* The purpose of this project is to develop and implement efficient algorithmic solutions for solving Sudoku puzzles.
* The scope encompasses creating software that can solve Sudoku puzzles of varying difficulty levels.
* This project aims to contribute to the field of artificial intelligence, constraint satisfaction, and puzzle-solving algorithms.
* **1.3 Significance of Developing Efficient Sudoku-Solving Algorithms**
* Developing efficient Sudoku-solving algorithms has practical applications in puzzle games, education, and AI.
* Such algorithms can be used to automatically generate Sudoku puzzles, assist players in solving them, and even design new puzzle variations.
* Efficient Sudoku solvers are relevant in AI research, as they demonstrate the power of constraint satisfaction and backtracking techniques in problem-solving.
* **1.4 Objectives of the Project**
* The primary objectives of this project include:
* Designing and implementing a robust Sudoku-solving algorithm.
* Evaluating the performance and accuracy of the algorithm on a variety of Sudoku puzzles.
* Comparing the developed algorithm with existing Sudoku solvers to assess its efficiency and accuracy.
* Providing insights into the algorithm's strengths, weaknesses, and potential areas for improvement.
* A secondary objective is to make the Sudoku solver accessible through user-friendly software or APIs, allowing enthusiasts to experience efficient Sudoku solving.

**2. Problem Statement**

**2.1 Sudoku as a Computational Problem**

* Sudoku can be framed as a computational problem involving a 9x9 grid where the objective is to fill each cell with a digit from 1 to 9.
* The problem constraints are straightforward: no digit can be repeated in any row, column, or 3x3 region.
* While humans may intuitively solve Sudoku puzzles, solving them algorithmically presents unique challenges, as they require a systematic approach that ensures a valid and complete solution.

**2.2 Key Challenges in Solving Sudoku Puzzles Algorithmically**

* Constraint Satisfaction: The primary challenge is enforcing the constraints effectively to ensure that the solution adheres to Sudoku's rules. This involves designing algorithms to verify that no digit repeats in rows, columns, or regions.
* Efficient Search: Finding the solution requires search techniques, as the number of possibilities is immense. Developing efficient search methods to navigate this solution space is a significant challenge.
* Complexity Analysis: Sudoku puzzles vary in difficulty, and algorithmic solutions need to adapt to puzzles of different complexities. Classifying and solving puzzles appropriately is a challenge.
* Optimization: Optimizing the solving process to minimize computational resources and time is crucial for practical applications.

**3. Literature Review**

**3.1 Overview of Existing Algorithms and Techniques**

* Existing algorithms for solving Sudoku puzzles range from simple techniques like brute force to advanced methods such as constraint propagation, backtracking, and advanced heuristics.
* Brute force involves trying all possible combinations, which can be time-consuming for complex puzzles.
* Constraint propagation methods like the Naked Single, Hidden Single, and Naked Pair are more efficient but may not work for challenging puzzles.
* Advanced techniques such as Backtracking and Constraint Propagation (like the Dancing Links algorithm) are capable of solving even the most challenging puzzles.

**3.2 Strengths and Weaknesses**

* Brute force algorithms are simple but inefficient for hard puzzles.
* Constraint propagation methods are efficient but may not handle the hardest puzzles.
* Advanced techniques like backtracking offer efficient solutions for most puzzles.
* Strengths include adaptability, accuracy, and generalizability to other constraint satisfaction problems.
* Weaknesses include increased computational complexity for hard puzzles and the need for specialized data structures.

**3.3 Relevant Research in the Field**

* Researchers have explored variations of Sudoku, like irregular Sudoku and killer Sudoku.
* Research in AI has resulted in hybrid algorithms that combine heuristics and constraint propagation.
* Techniques like symmetry reduction have been applied to optimize solving.
* Research has also been focused on automatically generating puzzles of various difficulties.

**4. Methodology**

**4.1 Algorithm Development**

* The project aims to develop a Sudoku-solving algorithm based on a combination of constraint propagation and backtracking techniques.
* The algorithm will utilize advanced data structures to efficiently represent the Sudoku grid and constraints.

**4.2 Step-by-Step Breakdown**

* The algorithm will start with constraint propagation methods to reduce the search space by identifying and filling cells where the values are uniquely determined.
* If necessary, backtracking will be employed to explore the solution space while maintaining feasibility.

**4.3 Data Structures and Techniques**

* Data structures will include matrices to represent the Sudoku grid and constraint propagation data structures like sets and arrays.
* Techniques such as heuristics for selecting cells to fill and constraint propagation checks for validity will be implemented.

**5. Algorithm**

**Algorithm: SolveSudoku**

**Input:**

* A 9x9 Sudoku puzzle grid, where 0 represents an empty cell.

**Output:**

* A solved Sudoku puzzle or a message indicating that no solution exists.

**Procedure:**

1. *Find an empty cell:*
   * Search the grid to find the first empty cell (a cell with the value 0).
   * If no empty cell is found, the puzzle is solved; return true.
2. *Try numbers from 1 to 9:*
   * For num in the range 1 to 9:
     1. If num is a valid move for the empty cell:
        + Place num in the empty cell.
        + Recursively call the SolveSudoku function.
3. *Backtrack:*
   * If the recursive call in step 2 fails to find a solution, backtrack by setting the value of the empty cell back to 0 and returning false.
4. *Return result:*
   * If any recursive call in step 2 returns true, the Sudoku puzzle is solved; return true.
   * If no valid number can be placed in the empty cell, return false to trigger backtracking.

Here is a more detailed breakdown of the key functions used within the algorithm:

Function: FindEmptyCell

* Searches the Sudoku grid for the first empty cell (cell with a value of 0).
* Returns the row and column indices of the empty cell, or null if no empty cell is found.

Function: IsValidMove

* Checks if it's valid to place a given number in a specific cell.
* Validates that the number is not already present in the same row, column, or 3x3 box.
* Returns true if the move is valid, false otherwise.

Function: SolveSudoku

* The main recursive function that attempts to solve the Sudoku puzzle.
* It first calls FindEmptyCell to find an empty cell.
* If an empty cell is found, it iterates through numbers 1 to 9 and attempts to place each number.
* For each valid placement attempt, it recursively calls itself.
* If the puzzle is solved, it returns true.
* If no valid placement leads to a solution, it backtracks and returns false.

This algorithm effectively explores the solution space of the Sudoku puzzle, trying different numbers in each empty cell and backtracking when it reaches an invalid state. When a solution is found, it returns true, and the Sudoku puzzle is solved. If no solution exists, it returns false.

**5. Implementation of code**

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public class SudokuSolver {

    private static final int N = 9;

    public static void main(String[] args) {

        int[][] puzzle = readSudokuFromFile("sudoku.txt");

        if (puzzle == null) {

            System.out.println("Failed to read the puzzle.");

            return;

        }

        if (solveSudoku(puzzle)) {

            System.out.println("Solved Sudoku:");

            printSudoku(puzzle);

        } else {

            System.out.println("No solution found.");

        }

    }

    private static int[][] readSudokuFromFile(String filename) {

        int[][] puzzle = new int[N][N];

        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {

            String line;

            int row = 0;

            while ((line = reader.readLine()) != null && row < N) {

                String[] values = line.split(" ");

                for (int col = 0; col < N; col++) {

                    puzzle[row][col] = Integer.parseInt(values[col]);

                }

                row++;

            }

        } catch (IOException e) {

            e.printStackTrace();

            return null;

        }

        return puzzle;

    }

    private static void printSudoku(int[][] grid) {

        for (int row = 0; row < N; row++) {

            for (int col = 0; col < N; col++) {

                System.out.print(grid[row][col] + " ");

            }

            System.out.println();

        }

    }

    private static boolean solveSudoku(int[][] grid) {

        int[] emptyCell = findEmptyCell(grid);

        if (emptyCell == null) {

            return true; // All cells are filled

        }

        int row = emptyCell[0];

        int col = emptyCell[1];

        for (int num = 1; num <= 9; num++) {

            if (isSafe(grid, row, col, num)) {

                grid[row][col] = num;

                if (solveSudoku(grid)) {

                    return true;

                }

                grid[row][col] = 0; // Backtrack

            }

        }

        return false; // No valid number can be placed

    }

    private static int[] findEmptyCell(int[][] grid) {

        for (int row = 0; row < N; row++) {

            for (int col = 0; col < N; col++) {

                if (grid[row][col] == 0) {

                    return new int[] { row, col };

                }

            }

        }

        return null;

    }

    private static boolean isSafe(int[][] grid, int row, int col, int num) {

        return !usedInRow(grid, row, num) && !usedInColumn(grid, col, num) && !usedInBox(grid, row - row % 3, col - col % 3, num);

    }

    private static boolean usedInRow(int[][] grid, int row, int num) {

        for (int col = 0; col < N; col++) {

            if (grid[row][col] == num) {

                return true;

            }

        }

        return false;

    }

    private static boolean usedInColumn(int[][] grid, int col, int num) {

        for (int row = 0; row < N; row++) {

            if (grid[row][col] == num) {

                return true;

            }

        }

        return false;

    }

    private static boolean usedInBox(int[][] grid, int boxStartRow, int boxStartCol, int num) {

        for (int row = 0; row < 3; row++) {

            for (int col = 0; col < 3; col++) {

                if (grid[row + boxStartRow][col + boxStartCol] == num) {

                    return true;

                }

            }

        }

        return false;

    }

}