**Sudoku Solver**

**Introduction**

*The Sudoku solver game is a puzzle where players ﬁll a 9x9 grid with numbers from 1 to*

*9, ensuring each row, column, and 3x3 sub grid contains all numbers without repetition.*

*Players input numbers, validate their choices, and a solver algorithm ﬁnds the solution*

*using backtracking. Hints may be provided, and players can request assistance. Once*

*the grid is ﬁlled, the game displays the solution. Sudoku offers a challenging experience that requires logical thinking and problem-solving skills.*

#### Purpose

The purpose of this project is to provide an interactive platform where users can load Sudoku puzzles, attempt to solve them manually, or use an automated solver to find the solution.

**Implementation Details**

**GUI Layout and Components:**

* The GUI is implemented using Java Swing components.
* The main window (JFrame) consists of a grid of JTextField elements where users can input numbers for the Sudoku puzzle.
* There are three main buttons:
  + **Load Puzzle:** Loads a predefined Sudoku puzzle into the grid.
  + **Solve:** Initiates the solving process using a backtracking algorithm.
  + **Clear:** Resets the grid for a new puzzle.

**Data Structures Used:**

* **JTextField Array (cells):** Represents each cell in the Sudoku grid where numbers are entered or displayed.
* **board Array (int[][]):** Stores the current state of the Sudoku board.
* **rows, cols, subgrids (HashSet Arrays):** Used for quick lookup to ensure that each number appears only once per row, column, and 3x3 subgrid.

**Functionality:**

* **Load Puzzle:**
  + Loads a predefined Sudoku puzzle into the grid from a hardcoded 2D array.
  + Initializes necessary data structures (rows, cols, subgrids) and populates them with initial values from the puzzle.
  + Non-zero values are displayed in the respective cells and made uneditable.
* **Solve:**
  + Initiates a solving process using a recursive backtracking algorithm (solve() method).
  + Uses isValid() method to check if a number can be placed in a particular cell based on Sudoku rules.
  + Updates the GUI after each step to show progress visually using updateGUI() and delay() methods.
* **Clear:**
  + Resets the Sudoku grid and clears all data structures (rows, cols, subgrids).

**CODE:**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.util.HashSet;

public class SudokuSolverGUI extends JFrame {

    private static final int SIZE = 9;

    private JTextField[][] cells = new JTextField[SIZE][SIZE];

    private int[][] board = new int[SIZE][SIZE];

    private HashSet<Integer>[] rows = new HashSet[SIZE];

    private HashSet<Integer>[] cols = new HashSet[SIZE];

    private HashSet<Integer>[] subgrids = new HashSet[SIZE];

    public SudokuSolverGUI() {

        setTitle("Sudoku Solver");

        setSize(600, 600);

        setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

        setLayout(new BorderLayout());

        JPanel gridPanel = new JPanel();

        gridPanel.setLayout(new GridLayout(SIZE, SIZE));

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

            rows[row] = new HashSet<>();

            cols[row] = new HashSet<>();

            subgrids[row] = new HashSet<>();

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

                cells[row][col] = new JTextField();

                cells[row][col].setHorizontalAlignment(JTextField.CENTER);

                cells[row][col].setFont(new Font("Arial", Font.BOLD, 20));

                gridPanel.add(cells[row][col]);

            }

        }

        add(gridPanel, BorderLayout.CENTER);

        JPanel buttonPanel = new JPanel();

        buttonPanel.setLayout(new GridLayout(1, 3));

        JButton loadButton = new JButton("Load Puzzle");

        loadButton.addActionListener(new ActionListener() {

            @Override

            public void actionPerformed(ActionEvent e) {

                loadPuzzle();

            }

        });

        buttonPanel.add(loadButton);

        JButton solveButton = new JButton("Solve");

        solveButton.addActionListener(new ActionListener() {

            @Override

            public void actionPerformed(ActionEvent e) {

                new Thread(new Runnable() {

                    @Override

                    public void run() {

                        solvePuzzle();

                    }

                }).start();

            }

        });

        buttonPanel.add(solveButton);

        JButton clearButton = new JButton("Clear");

        clearButton.addActionListener(new ActionListener() {

            @Override

            public void actionPerformed(ActionEvent e) {

                clearBoard();

            }

        });

        buttonPanel.add(clearButton);

        add(buttonPanel, BorderLayout.SOUTH);

    }

private void loadPuzzle() {

        int[][] puzzle = {

            {5, 3, 0, 0, 7, 0, 0, 0, 0},

            {6, 0, 0, 1, 9, 5, 0, 0, 0},

            {0, 9, 8, 0, 0, 0, 0, 6, 0},

            {8, 0, 0, 0, 6, 0, 0, 0, 3},

            {4, 0, 0, 8, 0, 3, 0, 0, 1},

            {7, 0, 0, 0, 2, 0, 0, 0, 6},

            {0, 6, 0, 0, 0, 0, 2, 8, 0},

            {0, 0, 0, 4, 1, 9, 0, 0, 5},

            {0, 0, 0, 0, 8, 0, 0, 7, 9}

        };

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

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

                board[row][col] = puzzle[row][col];

                if (puzzle[row][col] != 0) {

                    cells[row][col].setText(String.valueOf(puzzle[row][col]));

                    cells[row][col].setEditable(false);

                    rows[row].add(puzzle[row][col]);

                    cols[col].add(puzzle[row][col]);

                    subgrids[(row / 3) \* 3 + col / 3].add(puzzle[row][col]);

                } else {

                    cells[row][col].setText("");

                    cells[row][col].setEditable(true);

                }

            }

        }

    }

 private void solvePuzzle() {

        if (solve()) {

            SwingUtilities.invokeLater(new Runnable() {

                @Override

                public void run() {

                    JOptionPane.showMessageDialog(SudokuSolverGUI.this, "Sudoku Solved!", "Success", JOptionPane.INFORMATION\_MESSAGE);

                }

            });

        } else {

            SwingUtilities.invokeLater(new Runnable() {

                @Override

                public void run() {

                    JOptionPane.showMessageDialog(SudokuSolverGUI.this, "No solution exists for the given Sudoku board.", "Error", JOptionPane.ERROR\_MESSAGE);

                }

            });

        }

    }

 private void clearBoard() {

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

            rows[row].clear();

            cols[row].clear();

            subgrids[row].clear();

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

                cells[row][col].setText("");

                cells[row][col].setEditable(true);

                board[row][col] = 0;

            }

        }

    }

private boolean isValid(int row, int col, int num) {

        if (rows[row].contains(num) || cols[col].contains(num) || subgrids[(row / 3) \* 3 + col / 3].contains(num)) {

            return false;

        }

        return true;

    }

    private boolean solve() {

        int[] empty = findEmptyCell();

        if (empty == null) {

            return true;

        }

        int row = empty[0];

        int col = empty[1];

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

            if (isValid(row, col, num)) {

                board[row][col] = num;

                rows[row].add(num);

                cols[col].add(num);

                subgrids[(row / 3) \* 3 + col / 3].add(num);

                updateGUI(row, col, num);

                delay(50); // Delay to visualize steps

                if (solve()) {

                    return true;

                }

                board[row][col] = 0;

                rows[row].remove(num);

                cols[col].remove(num);

                subgrids[(row / 3) \* 3 + col / 3].remove(num);

                updateGUI(row, col, 0);

                delay(50); // Delay to visualize steps

            }

        }

        return false;

    }

    private int[] findEmptyCell() {

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

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

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

                    return new int[]{row, col};

                }

            }

        }

        return null;

    }

    private void updateGUI(int row, int col, int num) {

        SwingUtilities.invokeLater(new Runnable() {

            @Override

            public void run() {

                cells[row][col].setText(num == 0 ? "" : String.valueOf(num));

            }

        });

    }

    private void delay(int milliseconds) {

        try {

            Thread.sleep(milliseconds);

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

    }

    public static void main(String[] args) {

        SwingUtilities.invokeLater(new Runnable() {

            @Override

            public void run() {

                SudokuSolverGUI solver = new SudokuSolverGUI();

                solver.setVisible(true);

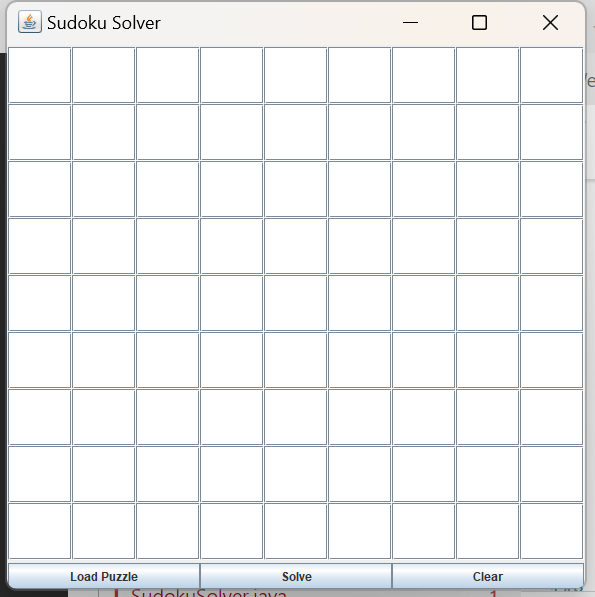
            }

        });

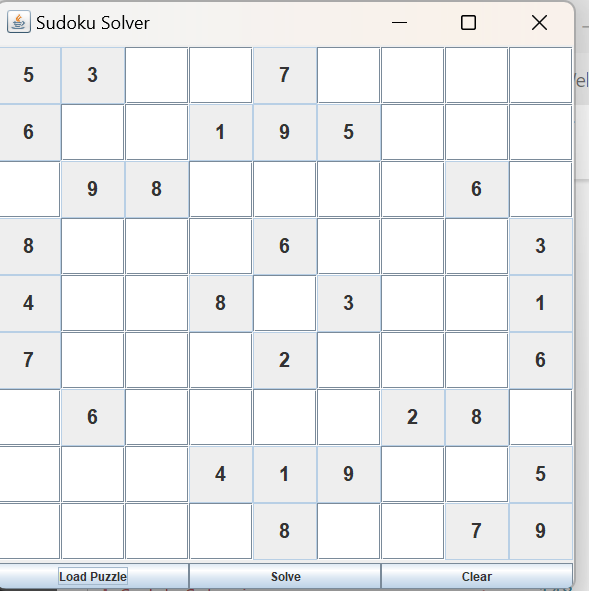
    }

}

**Interface**



When you run your code



After clicking LOAD PUZZLE button

**Conclusion**

This Sudoku Solver GUI project successfully combines Java Swing for the graphical interface with a backtracking algorithm to solve Sudoku puzzles. It allows users to interactively load, solve, and clear puzzles while visualizing the solving process step-by-step. The use of multithreading (Thread) and SwingUtilities.invokeLater() ensures smooth execution of the solving algorithm without freezing the GUI.