### 

### Project Title: Sudoku Solver with Visualization

**Name:** Logeshwaran  
**Registration Number:** 12210238  
  
**Project Introduction :**

This project focuses on the development of a Sudoku solver that incorporates a graphical user interface (GUI) for visualization. The solver is implemented in Java using the Swing framework, which provides a platform for users to input Sudoku puzzles and observe the solving process in real-time. The algorithm used for solving the Sudoku puzzle is based on backtracking, a recursive method that explores all possible configurations to find a solution.

The primary objective of this project is to provide a clear and interactive way to understand how Sudoku puzzles are solved algorithmically. By combining a visual interface with a robust solving algorithm, the project aims to create an educational tool that can be used by both beginners and advanced users interested in the mechanics of Sudoku solving.

**Sudoku Solver with Visualization**

**Abstract**

This project involves developing a Sudoku solver with a visual interface using Java Swing. The solver employs a backtracking algorithm to find the solution to a given Sudoku puzzle within a specified time limit. The visual interface allows users to input their Sudoku puzzles and watch the algorithm solve them step-by-step.

Sudoku is a popular logic-based number puzzle that consists of a 9x9 grid, divided into nine 3x3 subgrids. The objective is to fill the grid with numbers from 1 to 9, such that each row, column, and subgrid contains each number exactly once. This project implements a Sudoku solver using Java, providing a visual representation of the solving process through a graphical user interface (GUI).

**Objectives**

1. Implement a Sudoku solver using the backtracking algorithm.
2. Create a visual interface using Java Swing to input Sudoku puzzles and display the solving process.
3. Ensure the solver completes the puzzle within a specified time limit (2 minutes).

**System Design**

**Components**

1. **Visualizer Interface**
   * Defines the visualizeBoard method for updating the visual representation of the board.
2. **SudokuVisualizer Class**
   * Extends JFrame and implements Visualizer.
   * Contains the Sudoku board as a 2D array of JTextField.
   * Provides a button to trigger the solving process.
   * Updates the board visually as the solver progresses.
3. **SudokuSolver Class**
   * Contains the backtracking algorithm to solve the Sudoku puzzle.
   * Communicates with the SudokuVisualizer to update the visual representation.
   * Ensures the solving process respects the time limit.

**SudokuSolver Algorithm**

* **Backtracking Approach:**
  + Iterate through each cell in the Sudoku grid.
  + If a cell is empty, try placing numbers 1 through 9.
  + For each number, check if it is safe to place it in the cell (i.e., it does not violate Sudoku rules).
  + If a number is safe, place it and proceed to the next cell.
  + If placing the number leads to a solution, return true.
  + If no number leads to a solution, backtrack by resetting the cell and returning false.

**Implementation**

**Visualizer Interface**

interface Visualizer {

void visualizeBoard(int[][] newBoard);

}

**SudokuVisualizer Class**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class SudokuVisualizer extends JFrame implements Visualizer {

private static final int SIZE = 9;

private static final int GRID\_SIZE = 50;

private static final int TOTAL\_TIME\_MS = 2 \* 60 \* 1000; // 2 minutes

private int[][] board;

private JPanel panel;

private JButton solveButton;

private JTextField[][] fields;

public SudokuVisualizer(int[][] board) {

this.board = board;

this.fields = new JTextField[SIZE][SIZE];

panel = new JPanel(new GridLayout(SIZE, SIZE));

setUpBoard();

solveButton = new JButton("Solve");

solveButton.setBackground(Color.GRAY);

solveButton.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

new Thread(() -> {

SudokuSolver solver = new SudokuSolver(SudokuVisualizer.this, TOTAL\_TIME\_MS);

solver.solveSudoku(board);

}).start();

}

});

this.setLayout(new BorderLayout());

this.add(panel, BorderLayout.CENTER);

this.add(solveButton, BorderLayout.SOUTH);

this.setSize(SIZE \* GRID\_SIZE, SIZE \* GRID\_SIZE + 50);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setVisible(true);

}

private void setUpBoard() {

panel.removeAll();

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

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

JTextField field = new JTextField();

field.setHorizontalAlignment(JTextField.CENTER);

field.setFont(new Font("Arial", Font.BOLD, 20));

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

field.setText(String.valueOf(board[row][col]));

field.setEditable(false);

field.setBackground(Color.LIGHT\_GRAY);

} else {

field.setEditable(true);

field.setBackground(Color.WHITE);

}

fields[row][col] = field;

panel.add(field);

}

}

panel.revalidate();

panel.repaint();

}

@Override

public void visualizeBoard(int[][] newBoard) {

SwingUtilities.invokeLater(() -> {

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

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

JTextField field = fields[row][col];

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

field.setText(String.valueOf(newBoard[row][col]));

field.setBackground(Color.GRAY);

} else {

field.setText("");

field.setBackground(Color.WHITE);

}

}

}

});

}

public static void main(String[] args) {

int[][] board = {

{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}

};

new SudokuVisualizer(board);

}

}

**SudokuSolver Class**

class SudokuSolver {

private Visualizer visualizer;

private long startTime;

private long endTime;

private long duration;

private static final int MINIMUM\_DELAY = 10;

public SudokuSolver(Visualizer visualizer, long durationMillis) {

this.visualizer = visualizer;

this.duration = durationMillis;

}

public void solveSudoku(int[][] board) {

startTime = System.currentTimeMillis();

endTime = startTime + duration;

solve(board);

}

private boolean solve(int[][] board) {

for (int row = 0; row < 9; row++) {

for (int col = 0; col < 9; col++) {

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

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

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

board[row][col] = num;

visualizer.visualizeBoard(board);

try {

long currentTime = System.currentTimeMillis();

long timeRemaining = endTime - currentTime;

long delay = 5;

Thread.sleep(delay);

} catch (InterruptedException e) {

e.printStackTrace();

}

if (solve(board)) {

return true;

} else {

board[row][col] = 0; // backtrack

visualizer.visualizeBoard(board);

}

}

}

return false;

}

}

}

return true;

}

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

for (int d = 0; d < 9; d++) {

if (board[row][d] == num || board[d][col] == num) {

return false;

}

}

int sqrt = (int) Math.sqrt(9);

int boxRowStart = row - row % sqrt;

int boxColStart = col - col % sqrt;

for (int r = boxRowStart; r < boxRowStart + sqrt; r++) {

for (int d = boxColStart; d < boxColStart + sqrt; d++) {

if (board[r][d] == num) {

return false;

}

}

}

return true;

}

}

**Usage**

1. **Running the Application:** Execute the main method in the SudokuVisualizer class to launch the application.
2. **Input Sudoku Puzzle:** The application window displays a 9x9 grid. Users can input their Sudoku puzzles directly into the grid.
3. **Solving the Puzzle:** Click the "Solve" button to start the solving process. The solver will fill in the grid, updating the visual representation in real-time.
4. **Visualization:** The solver's progress is displayed on the grid, with each step visually updated.

**Time Complexity Analysis**

**Backtracking Algorithm**

The backtracking algorithm for solving Sudoku has a worst-case time complexity of O(9^(n\*n)), where n is the size of the grid (n=9 for standard Sudoku). This is because, in the worst case, each empty cell can be filled with any of the 9 digits, leading to an exponential number of possible board configurations.

**Optimizations**

1. **Early Termination:** The algorithm stops as soon as a valid solution is found.
2. **Constraint Propagation:** By checking the safety of placing a number before making a move, the algorithm reduces the number of invalid configurations explored.
3. **Time Limit:** The solver is constrained by a maximum solving time of 2 minutes. If the puzzle is not solved within this limit, the process stops.

## Conclusion

This project demonstrates a practical implementation of a Sudoku solver with a visual interface. It combines algorithmic problem-solving with GUI development, providing an interactive experience for users. The backtracking algorithm, along with visualization, makes it easier to understand the process of solving a Sudoku puzzle.