Sudoku Solver Visualizer

UNDERSTANDING THE BACKTRACKING ALGORITHM

Submitted To: Mr Rahul Singh 64953,

By: Mokshit Jain

Reg no: 12221263

Lovely Professional University

Jalandhar, Punjab, India.

Overview

The Sudoku Solver Visualizer is an interactive tool designed to help users solve Sudoku puzzles while visualising the solving process. It aids in understanding the logical steps involved in solving Sudoku and provides insights into different problem-solving strategies.

Goals

1. Educational Goals:

a. Understand Algorithm Implementation: Gain a deep understanding of the backtracking algorithm and its application in solving Sudoku puzzles.

b. Improve Logical Reasoning: Enhance your logical thinking and problem-solving skills by breaking down complex problems into smaller steps.

c. Learn Visualization Techniques: Develop skills in creating visual representations of algorithms to make them easier to understand.

2. Development Goals:

a. Build a Functional Application: Develop a fully functional Sudoku solver visualizer that can accept user input and solve puzzles efficiently.

b. Implement User-Friendly Interface: Design an intuitive and interactive user interface that allows easy input and visualization of the solving process.

c. Ensure Robust Error Handling: Implement error checking to ensure the puzzle setup is valid and provide meaningful feedback to users.

Specifications

Functional Requirements:

a. Solver Functionality:

i. The application should implement a backtracking algorithm to solve the Sudoku puzzle.

ii. The solver should provide a step-by-step visualization of the solving process.

b. Controls:
    i. Buttons for solving, and speed control the solving process.
    ii. A ‘Load Puzzle’ button to clear the puzzle and load a new one.
    iii. A ‘Change Theme’ button to change the theme of the visualiser.

Understanding the Algorithm

I. Backtracking Basics:

The backtracking algorithm tries to fill in the Sudoku grid one cell at a time.

It places a number in an empty cell and checks if it complies with Sudoku rules.

If a placement leads to a conflict, the algorithm backtracks to the previous cell and tries the next possible number.

II. Step-by-Step Process:

Step 1: Start with the first empty cell.

Step 2: Place the smallest possible number (1-9) that does not violate Sudoku rules.

Step 3: Move to the next empty cell and repeat step 2.

Step 4: If no valid number can be placed, backtrack to the previous cell and try the next number.

Step 5: Continue this process until the grid is completely filled or no solution is found.

Code

```
import javax.swing.*;
import java.awt.*;
import javax.swing.border.MatteBorder;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.HashSet;
import java.util.Random;
```

```
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];
   private volatile boolean stopSolving = false;
   private int delay = 500; // Default delay for slow speed
   private Color[] lightTheme = {Color.BLACK, Color.WHITE, Color.GRAY,
Color.GREEN, Color.RED};
   private Color[] darkTheme = {Color.WHITE, Color.BLACK,
Color.DARK_GRAY, Color.LIGHT_GRAY, Color.RED};
   private boolean isLightTheme = true;



   private int[][][] puzzles = {
           {
                   {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}
           },
           {
                   {0, 0, 0, 0, 0, 0, 0, 1, 2},
                   {0, 0, 0, 0, 0, 3, 0, 0, 0},
                   {0, 0, 1, 9, 0, 0, 4, 6, 0},
```

```
                {7, 0, 0, 1, 0, 0, 9, 0, 0},
                {5, 0, 6, 0, 0, 0, 8, 0, 3},
                {0, 0, 3, 0, 0, 8, 0, 0, 1},
                {0, 1, 2, 0, 0, 9, 5, 0, 0},
                {0, 0, 0, 7, 0, 0, 0, 0, 0},
                {6, 8, 0, 0, 0, 0, 0, 0, 0}
        },
        {
                {8, 0, 0, 0, 0, 0, 0, 0, 0},
                {0, 0, 3, 6, 0, 0, 0, 0, 0},
                {0, 7, 0, 0, 9, 0, 2, 0, 0},
                {0, 5, 0, 0, 0, 7, 0, 0, 0},
                {0, 0, 0, 0, 4, 5, 7, 0, 0},
                {0, 0, 0, 1, 0, 0, 0, 3, 0},
                {0, 0, 1, 0, 0, 0, 0, 6, 8},
                {0, 0, 8, 5, 0, 0, 0, 1, 0},
                {0, 9, 0, 0, 0, 0, 4, 0, 0}
        }
        // Add more puzzles here
};

public SudokuSolverGUI() {
    setTitle("Sudoku Solver");
    setSize(600, 600);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new BorderLayout());

    JPanel gridPanel = new JPanel();
    gridPanel.setLayout(new GridLayout(3, 3));
    gridPanel.setBorder(new MatteBorder(2, 2, 2, 2, Color.BLACK));

    for (int i = 0; i < 3; i++) {
```

```
            for (int j = 0; j < 3; j++) {
                JPanel subGrid = new JPanel();
                subGrid.setLayout(new GridLayout(3, 3));
                subGrid.setBorder(new MatteBorder(1, 1, 1, 1,
Color.GRAY));

                for (int row = 0; row < 3; row++) {
                    for (int col = 0; col < 3; col++) {
                        cells[i * 3 + row][j * 3 + col] = new
JTextField();
                        cells[i * 3 + row][j * 3 +
col].setHorizontalAlignment(JTextField.CENTER);
                        cells[i * 3 + row][j * 3 + col].setFont(new
Font("Arial", Font.BOLD, 20));
                        subGrid.add(cells[i * 3 + row][j * 3 + col]);
                    }
                }

                gridPanel.add(subGrid);
            }
        }

        add(gridPanel, BorderLayout.CENTER);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1, 4));

        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) {
                stopSolving = false; // Reset the stop flag
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        solvePuzzle();
                    }
                }).start();
            }
        });
        buttonPanel.add(solveButton);

        JButton stopButton = new JButton("Stop");
        stopButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                stopSolving = true;
            }
        });
        buttonPanel.add(stopButton);

        JComboBox<String> speedSelector = new JComboBox<>(new
String[]{"Slow", "Fast", "Superfast"});
        speedSelector.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
```

```
                String selectedSpeed = (String)
speedSelector.getSelectedItem();
                switch (selectedSpeed) {
                    case "Slow":
                        delay = 500;
                        break;
                    case "Fast":
                        delay = 100;
                        break;
                    case "Superfast":
                        delay = 10;
                        break;
                }
            }
        });
        buttonPanel.add(speedSelector);
        add(buttonPanel, BorderLayout.SOUTH);

        JButton changeThemeButton = new JButton("Change Theme");
        changeThemeButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                changeTheme();
            }
        });
        buttonPanel.add(changeThemeButton);

        initializeSets();
        loadPuzzle(); // Load a random puzzle on start
    }

    private void initializeSets() {
```

```
        for (int i = 0; i < SIZE; i++) {
            rows[i] = new HashSet<>();
            cols[i] = new HashSet<>();
            subgrids[i] = new HashSet<>();
        }
    }

    private void loadPuzzle() {
        clearBoard(); // Clear any previous data

        Random rand = new Random();
        int[][] puzzle = puzzles[rand.nextInt(puzzles.length)];

        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);
                    cells[row][col].setForeground(Color.BLACK); // Fixed
puzzle cells in black
                    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);
                    cells[row][col].setForeground(Color.RED); // New
puzzle cells in red
                }
            }
```

```
        }
    }

    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,
"Sudoku Not Solvable!", "Error", JOptionPane.ERROR_MESSAGE);
                }
            });
        }
    }

    private void changeTheme() {
        isLightTheme = !isLightTheme;
        Color[] theme = isLightTheme ? lightTheme : darkTheme;

        // Change the colors of the cells
        for (int row = 0; row < SIZE; row++) {
            for (int col = 0; col < SIZE; col++) {
                if (cells[row][col].isEditable()) {
```

```
                    cells[row][col].setForeground(theme[4]); // New
puzzle cells
                } else {
                    cells[row][col].setForeground(theme[0]); // Fixed
puzzle cells
                }
                cells[row][col].setBackground(theme[1]); // Background
color
            }
        }
    }

    private void clearBoard() {
        for (int row = 0; row < SIZE; row++) {
            for (int col = 0; col < SIZE; col++) {
                board[row][col] = 0;
                cells[row][col].setText("");
                cells[row][col].setEditable(true);
                cells[row][col].setForeground(Color.RED);
                rows[row].clear();
                cols[col].clear();
                subgrids[(row / 3) * 3 + col / 3].clear();
            }
        }
    }

    private boolean isValid(int row, int col, int num) {
        return !rows[row].contains(num) && !cols[col].contains(num) &&
!subgrids[(row / 3) * 3 + col / 3].contains(num);
    }

    private boolean solve() {
        if (stopSolving) {
```

```
            return false;
        }

        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, isLightTheme); // Valid numbers
in green
                delay(delay); // 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, isLightTheme); // Reset cell color
to red for new entries
                delay(delay); // 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, boolean isValid) {
        Color[] theme = isLightTheme ? lightTheme : darkTheme;
        Color color = isValid ? theme[3] : theme[4];
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                cells[row][col].setText(num == 0 ? "" :
String.valueOf(num));
                cells[row][col].setForeground(color);
            }
        });
    }

    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);
            }
        });
    }
}
```

Code Overview

The SudokuSolverGUI class includes the following components:

GUI Setup: Uses Swing components to create the Sudoku grid and control buttons.

Puzzle Initialization: Randomly selects and loads a puzzle from a predefined set.

Solving Mechanism: Implements a backtracking algorithm to solve the Sudoku puzzle with visual updates.

Control Buttons: Provides functionalities to load a new puzzle, start/stop solving, change solving speed, and switch themes.

Key Components

1. GUI Setup:

Grid Layout:

The main Sudoku grid is divided into 3x3 subgrids, each containing 3x3 cells.

Each cell is a JTextField for user input and display.

Control Panel:

Buttons for loading a puzzle, solving, stopping the solver, and changing the theme.

A JComboBox for selecting the solving speed.

2. Puzzle Initialization:

Predefined Puzzles:

The class contains an array of predefined puzzles.

A random puzzle is selected and loaded into the grid upon initialization or when the "Load Puzzle" button is clicked.

3. Solving Mechanism:

Backtracking Algorithm:

The solve() method implements the backtracking algorithm.

The algorithm fills the grid step-by-step, checking the validity of each number placement.

Includes a delay mechanism to visualise each step.

Final Thoughts

We hope you find the Sudoku Solver Visualizer a valuable tool for improving your Sudoku-solving skills. Your feedback is important to us, so please share your thoughts and suggestions for further enhancements.