Sudoku Solver Visualizer Project Report

Introduction

The Sudoku Solver Visualizer is a Java-based application that visually demonstrates the process of solving a Sudoku puzzle using a backtracking algorithm. The application provides an interactive interface for users to input Sudoku puzzles and observe the solving process step-by-step, highlighting cells as they are filled and backtracked.

Project Components

1. SudokuSolver Class

The `SudokuSolver` class is responsible for implementing the backtracking algorithm to solve the Sudoku puzzle. It uses a recursive method to try different numbers in each cell and backtracks when a number cannot lead to a solution.

Key Methods:

- **`solveSudoku(int[][] board)`**: This method initiates the solving process by calling the `solve` method starting from the top-left cell of the board.
- **`solve(int[][] board, int row, int col)`**: This recursive method attempts to solve the board starting from the given row and column. It updates the visualizer with the current state of the board.
- **`isValid(int[][] board, int row, int col, int num)`**: This method checks if placing a given number in a specified cell is valid according to Sudoku rules.

2. SudokuVisualizer Class

The `SudokuVisualizer` class provides the graphical user interface (GUI) for the application. It uses Java Swing components to create a grid of text fields where users can input the initial Sudoku puzzle and a button to start the solving process.

Key Components:

- **`JTextField[][] cells`**: A 9x9 grid of text fields for user input and displaying the solution.
- **`JPanel gridPanel`**: A panel that holds the grid of text fields.
- **`JButton solveButton`**: A button that initiates the solving process when clicked.
- **`updateCell(int row, int col, int value, boolean isBacktrack)`**: This method updates the visual representation of a cell during the solving process, changing its background color to indicate forward moves and backtracks.

3. Main Class

The `Main` class contains the `main` method, which serves as the entry point of the application. It creates an instance of `SudokuVisualizer` and makes it visible.

Functionality

User Input

Users can input their Sudoku puzzles into the grid by typing numbers into the text fields. Each text field accepts single-digit numbers from 1 to 9.

Solving Process

When the user clicks the "Solve" button, the application reads the numbers from the text fields into a 2D array representing the Sudoku board. A new thread is started to solve the puzzle using the `SudokuSolver` class. This ensures the GUI remains responsive during the solving process.

Visualization

As the `SudokuSolver` class attempts to solve the puzzle, it calls the `updateCell` method of the `SudokuVisualizer` to update the text fields in the GUI. The cells are colored green for forward moves and red for backtracks, providing a clear visual indication of the solving process.

Code Overview

SudokuSolver.java

```
public class SudokuSolver {
    private static final int SIZE = 9;
    private SudokuVisualizer visualizer;

    public SudokuSolver(SudokuVisualizer visualizer) {
        this.visualizer = visualizer;
    }

    public boolean solveSudoku(int[][] board) {
        return solve(board, 0, 0);
    }

    private boolean solve(int[][] board, int row, int col) {
        if (row == SIZE) {
            return true;
```

```
    }

    if (col == SIZE) {
        return solve(board, row + 1, 0);
    }

    if (board[row][col] != 0) {
        return solve(board, row, col + 1);
    }

    for (int num = 1; num <= SIZE; num++) {
        if (isValid(board, row, col, num)) {
            board[row][col] = num;
            visualizer.updateCell(row, col, num, false);
            if (solve(board, row, col + 1)) {
                return true;
            }
            board[row][col] = 0;
            visualizer.updateCell(row, col, 0, true);
        }
    }

    return false;
}

private boolean isValid(int[][] board, int row, int col, int num) {
    for (int i = 0; i < SIZE; i++) {
        if (board[row][i] == num || board[i][col] == num ||
            board[row - row % 3 + i / 3][col - col % 3 + i % 3] == num) {
            return false;
        }
```

```
        }
        return true;
    }
}
```

SudokuVisualizer.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class SudokuVisualizer extends JFrame {
    private static final int SIZE = 9;
    private JTextField[][] cells;
    private SudokuSolver solver;
    private JPanel gridPanel;

    public SudokuVisualizer() {
        solver = new SudokuSolver(this);
        cells = new JTextField[SIZE][SIZE];
        setTitle("Sudoku Solver");
        setSize(600, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());
        setResizable(false);

        gridPanel = new JPanel(new GridLayout(SIZE, SIZE));
        gridPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        gridPanel.setBackground(Color.DARK_GRAY);
        for (int row = 0; row < SIZE; row++) {
            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));
        cells[row][col].setBorder(BorderFactory.createLineBorder(Color.BLACK));
        cells[row][col].setBackground(Color.WHITE);
        gridPanel.add(cells[row][col]);
    }
}
add(gridPanel, BorderLayout.CENTER);

JPanel bottomPanel = new JPanel();
bottomPanel.setLayout(new BorderLayout());
bottomPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
bottomPanel.setBackground(Color.LIGHT_GRAY);

JButton solveButton = new JButton("Solve");
solveButton.setFont(new Font("Arial", Font.BOLD, 18));
solveButton.setBackground(new Color(0, 123, 255));
solveButton.setForeground(Color.WHITE);
solveButton.setFocusPainted(false);
solveButton.setBorder(BorderFactory.createEmptyBorder(10, 20, 10, 20));
solveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        int[][] board = new int[SIZE][SIZE];
        for (int row = 0; row < SIZE; row++) {
            for (int col = 0; col < SIZE; col++) {
                String text = cells[row][col].getText();
                if (!text.isEmpty()) {
                    board[row][col] = Integer.parseInt(text);
                }
```

```
            }
        }
        new Thread(() -> solver.solveSudoku(board)).start();
      }
    });
    bottomPanel.add(solveButton, BorderLayout.CENTER);

    add(bottomPanel, BorderLayout.SOUTH);
  }

  public void updateCell(int row, int col, int value, boolean isBacktrack) {
    try {
      Thread.sleep(50); // delay for visualization
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
        cells[row][col].setText(value == 0 ? "" : String.valueOf(value));
        cells[row][col].setBackground(isBacktrack ? new Color(255, 102, 102) : new Color(102, 255,
178));
      }
    });
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() {
```

```
            new SudokuVisualizer().setVisible(true);
        }
    });
  }
}
```

Main.java

```
public class Main {
  public static void main(String[] args) {
    SudokuVisualizer sudokuVisualizer = new SudokuVisualizer();
    sudokuVisualizer.setVisible(true);
  }
```

}

Conclusion

The Sudoku Solver Visualizer project successfully demonstrates the backtracking algorithm used to solve Sudoku puzzles. The visual feedback provided during the solving process enhances understanding and makes the application both educational and engaging. Future improvements could include additional features like saving/loading puzzles, hint generation, and solving puzzles of different sizes.