Sudoku Solver

Report

Shubham Singh

12210791

9SK02

Submitted to – Rahul Sir

11 July 2024

Prepared for Summer PEP - 2024

Table Of Contents

Abstract

This report details the development of a Sudoku Solver GUI in Java, featuring functionalities such as loading and solving puzzles, pausing/resuming the solving process, clearing the board, and adjusting solving speed via a slider. The GUI also emphasizes 3x3 subgrids with thick black borders. Implemented using Java Swing components, the Sudoku board comprises a 9x9 grid of JTextField elements. Key functions include loading puzzles, solving via backtracking, validating moves, and real-time GUI updates. The solving algorithm employs a depth-first search strategy with backtracking, ensuring efficient puzzle resolution. System requirements include Java Development Kit (JDK) version 8 or later and knowledge of Java Swing and multi-threading. The project demonstrates effective use of Java Swing for creating an interactive and visually appealing application, enhancing user experience with clear visual feedback during the solving process.

Sudoku Solver GUI Report

Introduction

This report documents the development of a Sudoku Solver GUI in Java, which includes features such as loading a puzzle, solving it with visual highlights for the solving process, pausing/resuming the solving process, clearing the board, and adjusting the speed of the solving process using a slider. Additionally, the GUI visually emphasizes the 3x3 subgrids with black borders, similar to traditional Sudoku grids.

Features

- **Load Puzzle**: This feature allows users to load a predefined Sudoku puzzle into the GUI.
- **Solve**: Initiates the solving process of the loaded Sudoku puzzle.
- **Stop**: Pauses the solving process, allowing users to halt and later resume the solution.
- **Resume**: Resumes the solving process after it has been paused.
- **Clear**: Clears the Sudoku board, allowing users to input a new puzzle.
- **Speed Control**: A slider to adjust the speed of the solving process for better visualization.
- **Highlight Subgrids**: Each 3x3 subgrid is highlighted with a thick black border to mimic traditional Sudoku grids.

Implementation Details

The GUI is implemented using Java Swing components. The main features of the implementation are:

- **JTextField Cells**: A 9x9 grid of `JTextField` components is used to represent the Sudoku board.
- **ActionListeners**: Various buttons have associated `ActionListeners` to handle load, solve, stop, resume, and clear actions.
- **Solving Algorithm**: The solving algorithm uses backtracking and updates the GUI in real-time to show the solving process.
- **Speed Slider**: A `JSlider` component is added to control the speed of the solving visualization.
- **Custom Borders**: Each `JTextField` cell is given a custom `Border` to highlight the 3x3 subgrids with thicker black lines on the appropriate edges.

Functions and Their Use Cases

`loadPuzzle()`

- **Purpose**: Loads a predefined Sudoku puzzle into the GUI.
- **Use Case**: When the user wants to load a default puzzle to solve or visualize the solving process.
- **Requirements**: A predefined 2D array representing the Sudoku puzzle.

`solvePuzzle()`

- **Purpose**: Initiates the solving process of the loaded Sudoku puzzle in a separate thread.
- **Use Case**: When the user wants to solve the loaded puzzle and see the solving steps visually.
- **Requirements**: A correctly loaded puzzle with empty cells represented by 0.

`stopSolving()`

- **Purpose**: Pauses the solving process.
- **Use Case**: When the user wants to temporarily halt the solving process to observe or analyze the current state of the board.
- **Requirements**: An ongoing solving process.

`resumeSolving()`

- **Purpose**: Resumes the solving process after it has been paused.
- **Use Case**: When the user wants to continue solving the puzzle after pausing.
- **Requirements**: A previously paused solving process.

`clearBoard()`

- **Purpose**: Clears the Sudoku board and resets the GUI.
- **Use Case**: When the user wants to start fresh with a new puzzle or reset the current board.
- **Requirements**: None, can be used anytime.

`isValid(int row, int col, int num)`

- **Purpose**: Checks if a number can be placed in a specific cell without violating Sudoku rules.
- **Use Case**: During the solving process to validate potential moves.
- **Requirements**: Current state of the board, row, column, and number to be validated.

`solve()`

- **Purpose**: Uses backtracking to solve the Sudoku puzzle.
- **Use Case**: Called by `solvePuzzle()` to perform the actual solving process.
- **Requirements**: A loaded puzzle and appropriate methods to validate moves and update the GUI.

`findEmptyCell()`

- **Purpose**: Finds the next empty cell in the board.
- **Use Case**: During the solving process to identify the next cell to fill.
- **Requirements**: Current state of the board.

`updateGUI(int row, int col, int num)`

- **Purpose**: Updates the GUI to reflect changes in the board.
- **Use Case**: During the solving process to visually show the current state of the board.
- **Requirements**: Row, column, and number to be updated in the GUI.

`highlightCell(int row, int col, Color color)`

- **Purpose**: Highlights a cell with a specified color.
- **Use Case**: To visually emphasize cells during the solving process, such as showing cells being filled or backtracked.
- **Requirements**: Row, column, and color for highlighting.

`delay(int milliseconds)`

- **Purpose**: Introduces a delay to control the speed of the solving visualization.
- **Use Case**: To slow down the solving process for better visualization.
- **Requirements**: Time in milliseconds to delay.

Code Snippets

Here are some key code snippets demonstrating the implementation:

Grid Initialization with Custom Borders

```
java
Copy code
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].setBackground(lightBlue);
        Border border = BorderFactory.createMatteBorder(
            row % 3 == 0 ? 2 : 1,
            col % 3 == 0 ? 2 : 1,
            row % 3 == 2 ? 2 : 1,
            col % 3 == 2 ? 2 : 1,
            Color.BLACK
        );
        cells[row][col].setBorder(border);
        gridPanel.add(cells[row][col]);
    }
}
```

Speed Control with JSlider

```
java
Copy code
JSlider speedSlider = new JSlider(JSlider.HORIZONTAL, 0, 200, 50);
speedSlider.setMajorTickSpacing(50);
speedSlider.setMinorTickSpacing(10);
speedSlider.setPaintTicks(true);
speedSlider.setPaintLabels(true);
controlPanel.add(speedSlider, BorderLayout.CENTER);
```

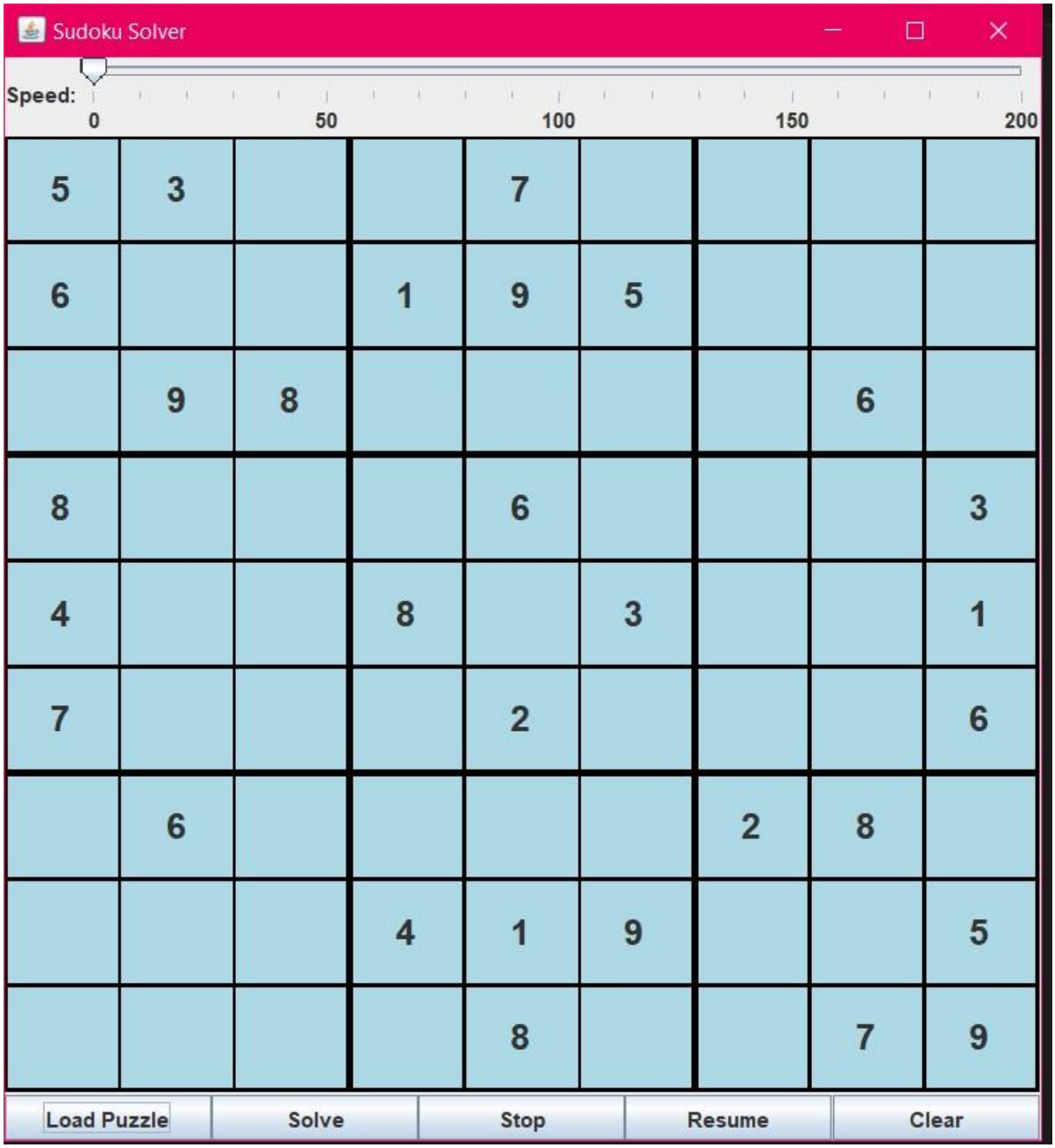
```
delay = speedSlider.getValue();
speedSlider.addChangeListener(e -> delay = speedSlider.getValue());
```

Solving with Visualization

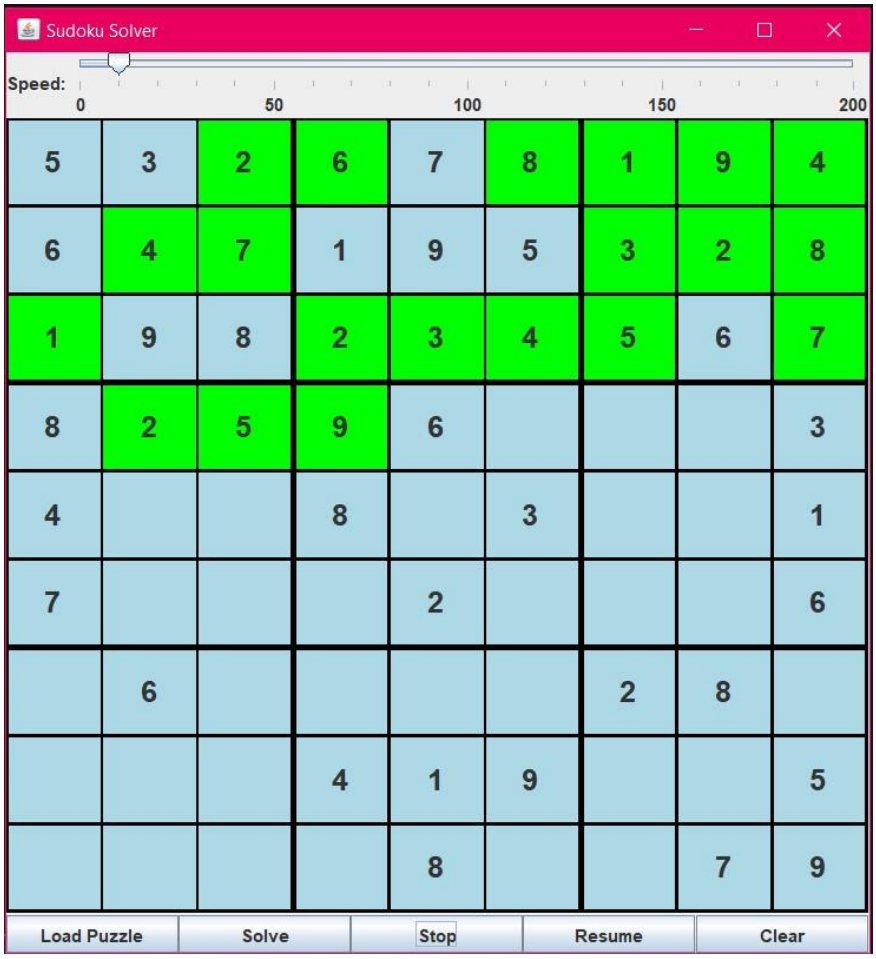
```
java
Copy code
for (int num = 1; num <= SIZE; num++) {
    if (solvingStopped.get()) {
        return false;
    }
    if (solvingPaused.get()) {
        synchronized (solvingThread) {
            try {
                solvingThread.wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                return false;
            }
        }
    }
    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);
        highlightCell(row, col, Color.GREEN);
        updateGUI(row, col, num);
        delay(delay);
        if (solve()) {
            return true;
        }
        board[row][col] = 0;
        rows[row].remove(num);
        cols[col].remove(num);
        subgrids[(row / 3) * 3 + col / 3].remove(num);
        highlightCell(row, col, Color.RED);
        updateGUI(row, col, 0);
        delay(delay);
        highlightCell(row, col, lightBlue);
    }
}
```

Images

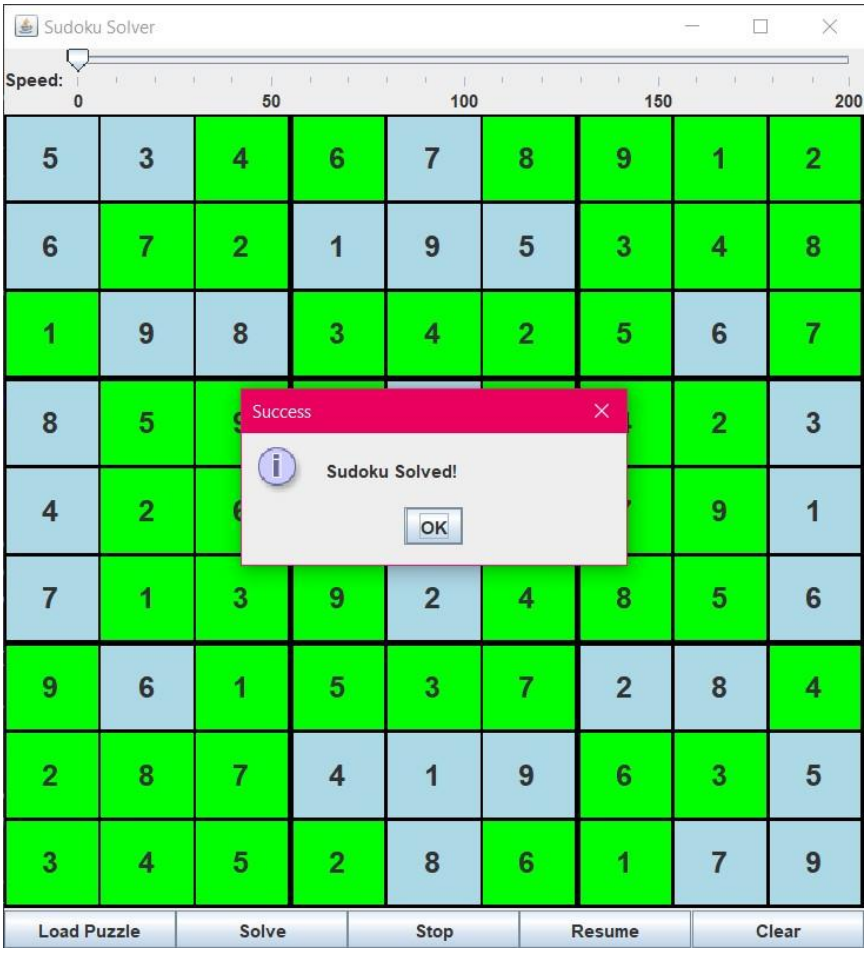
Load Sudoku



Solving Sudoku



Sudoku Solved



Backtracking Algorithm for Sudoku Solving

Overview

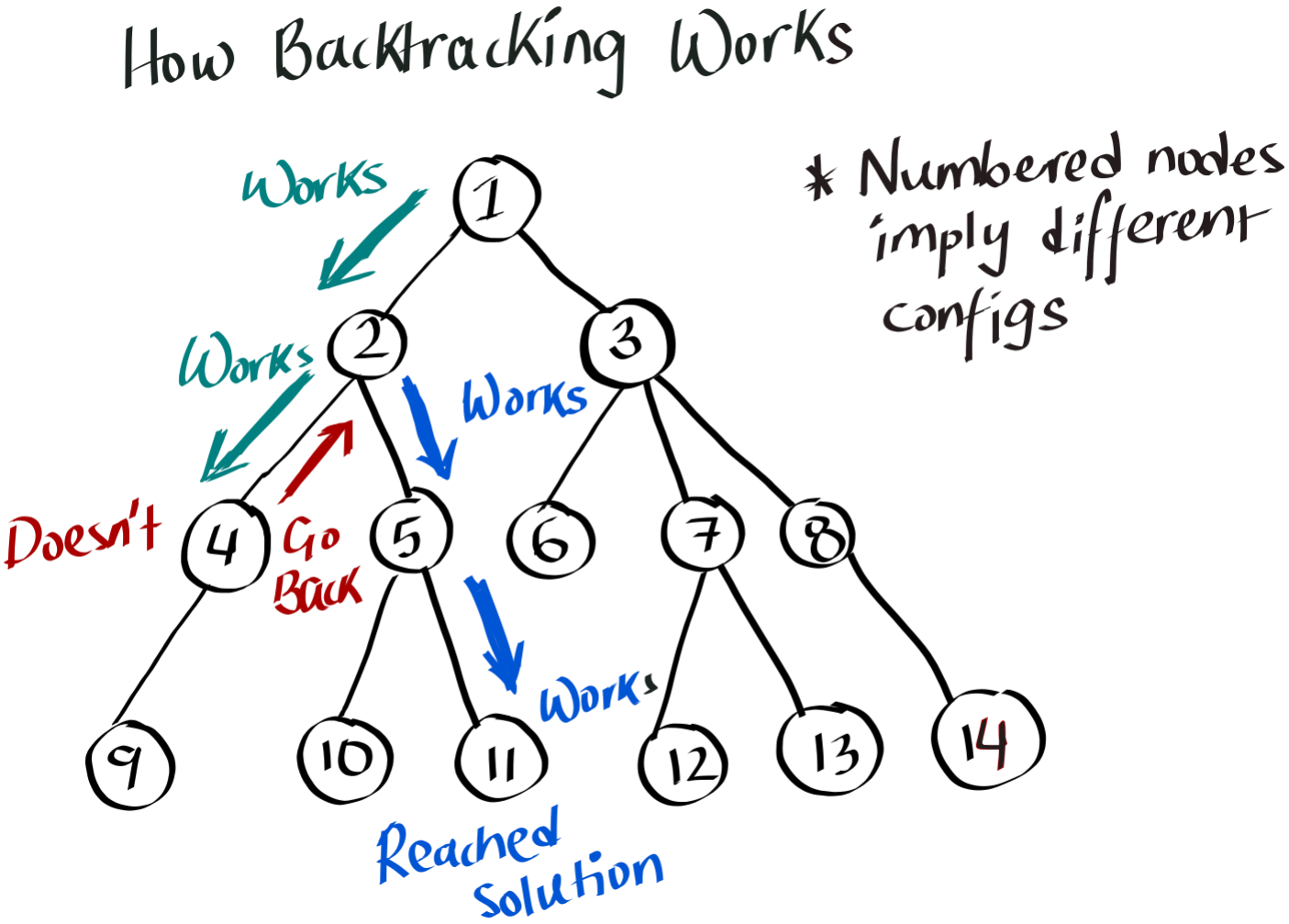
The Sudoku puzzle solving algorithm employs a recursive depth-first search strategy combined with backtracking. This approach systematically tries possible numbers for each empty cell until the puzzle is solved or determines that a solution does not exist.

Steps Involved

1. **Finding an Empty Cell**
    - The algorithm begins by searching for an empty cell (a cell with the value 0) on the Sudoku board. This is typically done from left to right and top to bottom.
2. **Validating Possible Numbers**
    - For each empty cell found, the algorithm tries each number from 1 to 9 to determine if it can be placed in the current cell without violating Sudoku rules.
3. **Validation Rules**
    - Before placing a number in a cell, the algorithm checks three conditions:
        - **Row Constraint**: Ensures the number is not already present in the current row.
        - **Column Constraint**: Ensures the number is not already present in the current column.
        - **Subgrid Constraint**: Ensures the number is not already present in the 3x3 subgrid that contains the current cell.
4. **Recursive Backtracking**
    - If a number is found that satisfies all three conditions, it is placed in the cell, and the algorithm recursively attempts to solve the rest of the puzzle.
    - If placing a number leads to a dead end (i.e., no valid numbers can be placed in subsequent cells), the algorithm backtracks to the previous cell and tries the next possible number.
5. **Completion**
    - The puzzle is considered solved when the algorithm successfully fills all cells with numbers that satisfy Sudoku rules.

Implementation Considerations

- **Efficiency**: Although Sudoku puzzles have a large search space (9^81 possible board configurations), efficient pruning of the search tree using backtracking significantly reduces the number of configurations checked.
- **Optimizations**: Techniques like constraint propagation (such as Naked Singles, Hidden Singles) can be used to further reduce the search space and speed up the solving process.
- **Parallelization**: For more complex puzzles or real-time solving visualizations, parallel or concurrent approaches can be employed to speed up the solving process.



Requirements

System Requirements

- **Operating System**: Windows, macOS, or Linux
- **Processor**: 1 GHz or faster
- **RAM**: 512 MB or more
- **Storage**: 100 MB of free space

Software Requirements

- **Java Development Kit (JDK)**: Version 8 or later
- **Integrated Development Environment (IDE)**: Recommended IntelliJ IDEA, Eclipse, or NetBeans
- **Java Swing Library**: Included with JDK

Other Requirements

- **Sudoku Puzzle Input**: A predefined 2D array representing the Sudoku puzzle
- **Java Swing Components Knowledge**: Basic understanding of Java Swing for GUI development
- **Thread Handling**: Understanding of multi-threading to manage the solving process

Conclusion

The Sudoku Solver GUI project demonstrates the effective use of Java Swing components to create an interactive and visually appealing application. By incorporating features such as custom borders to highlight subgrids and a speed control slider, the application enhances the user experience and provides clear visual feedback during the solving process.

References

1. **Java Swing Documentation**: Comprehensive guide on Java's Swing library for building graphical user interfaces.
    - Oracle. (n.d.). *Java Swing Documentation*. Retrieved from https://docs.oracle.com/javase/tutorial/uiswing/
2. **Concurrency in Swing**: Information on handling concurrency in Swing applications to keep the UI responsive.
    - Oracle. (n.d.). *Concurrency in Swing*. Retrieved from https://docs.oracle.com/javase/tutorial/uiswing/concurrency/
3. **Sudoku Solving Algorithms**: Explanation of various algorithms used for solving Sudoku puzzles, including backtracking.
    - Wikipedia. (n.d.). *Sudoku Solving Algorithms*. Retrieved from https://en.wikipedia.org/wiki/Sudoku_solving_algorithms
4. **Java HashSet Class**: Official documentation on Java's `HashSet` class, used for storing unique elements.
    - Oracle. (n.d.). *Java HashSet Class*. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html
5. **AtomicBoolean in Java**: Guide on using `AtomicBoolean` for thread-safe boolean operations.
    - Oracle. (n.d.). *AtomicBoolean (Java Platform SE 8)*. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicBoolean.html