Six Weeks Summer Training

Data Structures and Algorithms

A training report

Submitted in partial fulfilment of the requirements for the award of degree of

B.Tech. (Computer Science and Engineering)

Submitted to

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA, PUNJAB

From 06/06/24 to 21/07/24

SUBMITTED BY

Name of student: Yelugotla Sanket

Registration Number: 12223854

[illegible]

Signature of the student

Student Declaration

To whom so ever it may concern

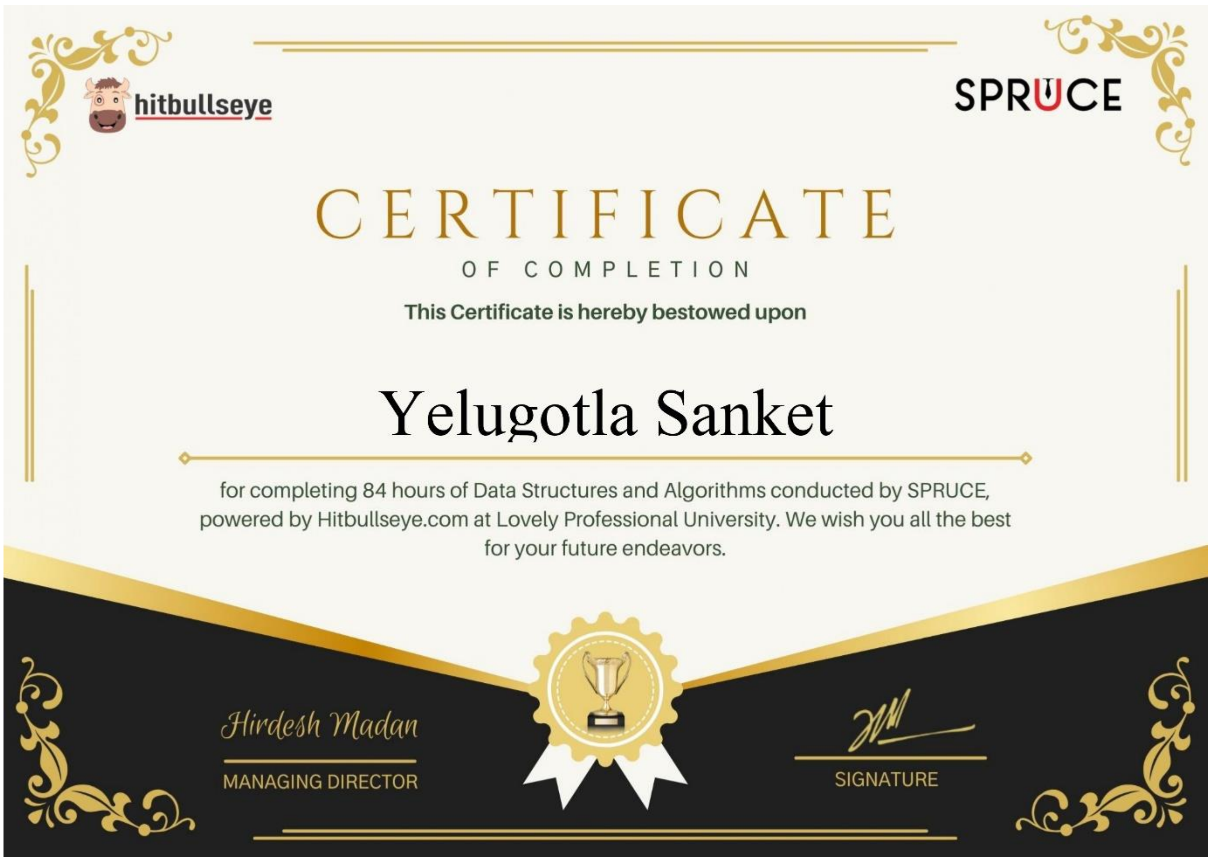
I, **Yelugotla Sanket**, **12223854**, hereby declare that the work done by me on "**Data Structures and Algorithms**" from **06, 2024** to **07, 2024** is a record of original work for the partial fulfilment of the requirements for the award of the degree, B.Tech. (Computer Science and Engineering)

Yelugotla Sanket (12223854)

[illegible]

Signature of the student

Dated:30/08/2024



hitbullseye

SPRUCE

CERTIFICATE

OF COMPLETION

This Certificate is hereby bestowed upon

Yelugotla Sanket

for completing 84 hours of Data Structures and Algorithms conducted by SPRUCE, powered by Hitbullseye.com at Lovely Professional University. We wish you all the best for your future endeavors.

Hirdesh Madan

MANAGING DIRECTOR

[illegible]

SIGNATURE

Chapter-1 Introduction

Problem solving is the most essential skill needed for a software developer, and learning this skill can empower each individual with immense knowledge. These skills can further be used in solving real-world problems, allowing developers to create innovative solutions that address complex challenges. By mastering problem-solving, developers not only enhance their technical capabilities but also improve their ability to think critically, adapt to new situations, and drive technological advancements that can make a significant impact on society.

Data structures and competitive programming (CP) form the backbone of efficient problem solving in software development. Data structures are specialized formats for organizing, managing, and storing data to allow efficient access and modification. From arrays and linked lists to more complex structures such as trees and graphs, they are the foundation for designing algorithms that solve complex computational problems. Competitive programming, on the other hand, is a sport in which programmers tackle coding challenges within tight deadlines, requiring both a solid knowledge of algorithms and data structures and the ability to implement them quickly and accurately. Together, mastering data structures and excelling at competitive programming will enhance analytical thinking, foster a deep understanding of computer science fundamentals, and prepare developers to solve real-world problems with optimized solutions.

I have Learned Various Topics Like:

- Vectors
- Linked List
- Stack
- Queue
- Trees
- Dynamic Programming
- Graphs
- Algorithms

Chapter-2 Topics Learnt

1. Vectors

Vectors are dynamic, linear data structures that store elements of the same type and automatically adjust their size as elements are added or removed. Unlike static arrays, which have a fixed size, vectors in languages like C++ can grow or shrink dynamically. This adaptability makes vectors a powerful tool for managing collections of data when the size is unknown or changes over time. Vectors maintain the order of insertion and provide fast random access to elements using indexing, making them efficient for retrieval operations. Internally, vectors allocate additional memory when resizing to minimize the frequency of reallocations, ensuring good performance when growing the vector incrementally. Despite the overhead associated with dynamic resizing, vectors strike a good balance between performance and flexibility. They are particularly useful for use cases involving frequent additions or deletions, such as managing collections of user input, dynamic datasets, or implementing data structures like stacks and queues.

2. Linked List

A linked list is a linear data structure consisting of nodes, where each node contains a data element and a pointer to the next node in the sequence. Unlike arrays or vectors, linked lists do not use contiguous memory locations to store elements. Instead, elements are scattered in memory and connected via pointers. This structure allows for efficient insertions and deletions, especially at the beginning or end of the list, without the need to shift elements like in arrays. Linked lists come in various forms: singly linked lists, where each node points only to the next node; doubly linked lists, where nodes have pointers to both the previous and next nodes; and circular linked lists, where the last node points back to the first node. Although they are slower for random access compared to arrays, linked lists are particularly useful for scenarios where the size of the dataset is unpredictable or frequently changing.

3. Stack

A stack is a linear data structure that operates on the Last In, First Out (LIFO) principle, meaning that the last element added is the first one to be removed. It is akin to a stack of plates: you add new plates on top and remove the top one when needed. Stacks are commonly used in algorithms and programming for situations requiring backtracking, such as undo operations in editors, evaluating expressions, and parsing syntax. Stacks support two primary operations: push (to add an element to the top) and pop (to remove the top element). Additional operations include peek, which allows inspecting the top element without removal, and checking if the stack is empty. Stacks can be implemented using arrays, vectors, or linked lists. They are fundamental to recursion, as function call stacks in most programming languages use this data structure to manage active function calls.

4. Queue

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning the first element added is the first one removed, much like a line of people waiting for service. Queues are widely used in scenarios where order matters, such as managing tasks in a scheduling system, simulating real-world processes (e.g., ticket queues or CPU scheduling), or buffering data streams. The two main operations in a queue are enqueue, which adds an element to the back of the queue, and dequeue, which removes the element from the front. Additional operations include checking if the queue is empty, accessing the front element, or inspecting the size of the queue. Queues can be implemented using arrays, vectors, or linked lists, and they come in different variations, such as circular queues, double-ended queues (deques), and priority queues, which sort elements based on priority instead of the order of insertion.

5. Trees

A tree is a hierarchical data structure consisting of nodes, where each node stores data and has references (or edges) to child nodes. Unlike linear structures such as arrays or linked lists, trees have a branching structure, starting with a root node and expanding outward into subtrees, which can also branch further. Trees are particularly useful for representing hierarchical relationships, such as file systems, organizational structures, and decision-making processes. Common types of trees include binary trees, where each node has at most two children (left and right), and binary search trees (BSTs), which organize data in a way that allows for efficient searching, insertion, and deletion. Other tree structures, such as AVL trees and red-black trees, maintain balance to ensure optimal performance for dynamic datasets. Trees are also essential in various algorithms, such as traversal (preorder, inorder, postorder), pathfinding, and decision trees used in machine learning.

6. Dynamic Programming

Dynamic Programming (DP) is an optimization technique used to solve complex problems by breaking them down into simpler subproblems, storing the results of these subproblems to avoid redundant calculations. DP is particularly effective for problems with overlapping subproblems and optimal substructure properties, where the solution to a problem can be composed of solutions to its subproblems. DP approaches typically involve either a **top-down** (memoization) or **bottom-up** (tabulation) strategy. In memoization, the problem is solved recursively, and results of solved subproblems are stored to be reused. In tabulation, a table is built iteratively from smaller subproblems to larger ones. DP is widely applied in various areas such as algorithm design, game theory, and bioinformatics, with classic problems like the Fibonacci sequence, Knapsack problem, and shortest path problems serving as common examples. It significantly improves performance by reducing the time complexity of recursive solutions.

7. Graphs

Graphs are non-linear data structures consisting of nodes (vertices) connected by edges, which can represent relationships between pairs of nodes. Graphs are widely used to model real-world systems, such as social networks, transportation networks, communication systems, and web page link structures. Graphs can be **directed**, where edges have a direction (pointing from one vertex to another), or **undirected**, where edges have no direction. They can also be **weighted**, where each edge carries a numerical weight, representing costs or distances between nodes. Common graph representations include adjacency matrices and adjacency lists. Graph algorithms are fundamental for solving problems such as finding the shortest path (Dijkstra's or Bellman-Ford algorithms), detecting cycles, or traversing the graph (Depth-First Search and Breadth-First Search). Graph theory is a crucial part of computer science, enabling solutions for complex network-related problems, optimization in routing and logistics, and AI search algorithms.

Chapter-3 Algorithms

Algorithms are step-by-step procedures or formulas for solving problems. They take inputs, perform computations, and produce outputs, with the goal of solving a problem efficiently in terms of time and space. Algorithms are essential to computer science, driving everything from simple operations like sorting and searching to complex tasks like encryption, machine learning, and optimization.

Here's a list of important algorithm definitions:

1. Sorting Algorithms

Sorting algorithms arrange elements of a list or array in a specific order, typically ascending or descending. Common sorting algorithms include:

- **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order.
- **Merge Sort:** Uses a divide-and-conquer approach to divide the list into halves, sort them, and then merge them.
- **Quick Sort:** Selects a "pivot" element and partitions the array around the pivot, recursively sorting the partitions.
- **Heap Sort:** Converts the array into a heap and then repeatedly extracts the maximum element to sort the array.

2. Search Algorithms

Search algorithms find the position of a specific value within a data structure. Examples include:

- **Linear Search:** Scans each element of the array sequentially until the target value is found.
- **Binary Search:** Efficiently searches a sorted array by repeatedly dividing the search interval in half.
- **Depth-First Search (DFS):** Explores as far as possible along each branch before backtracking, commonly used in graph traversal.

- **Breadth-First Search (BFS):** Explores all nodes at the present depth level before moving on to nodes at the next depth level, also used in graph traversal.

3. Greedy Algorithms

Greedy algorithms make the locally optimal choice at each step with the hope of finding a global optimum. They are often used in optimization problems. Key examples include:

- **Dijkstra's Algorithm:** Finds the shortest path from a starting node to all other nodes in a weighted graph.
- **Prim's Algorithm:** Finds a minimum spanning tree for a weighted undirected graph, connecting all vertices with the minimum total edge weight.
- **Huffman Coding:** Creates an optimal prefix code for data compression by selecting the least frequent characters and combining them.

4. Divide and Conquer Algorithms

Divide and conquer algorithms break a problem into smaller subproblems, solve them independently, and combine their results. Examples include:

- **Merge Sort:** Divides the array into two halves, sorts them recursively, and merges them.
- **Quick Sort:** Partitions the array into subarrays around a pivot and sorts the subarrays independently.
- **Binary Search:** Divides the search space in half with each comparison, working on the half that could contain the target.

5. Dynamic Programming

Dynamic programming solves problems by breaking them down into overlapping subproblems, storing their solutions to avoid redundant calculations. Key algorithms include:

- **Fibonacci Sequence:** Computes Fibonacci numbers by storing results of previous computations to avoid exponential time complexity.
- **Knapsack Problem:** Determines the maximum value that can be achieved in a knapsack of limited capacity by considering each item's weight and value.
- **Longest Common Subsequence (LCS):** Finds the longest subsequence common to two sequences by comparing and storing intermediate results.

6. Backtracking Algorithms

Backtracking algorithms incrementally build candidates for solutions and abandon candidates ("backtrack") as soon as it is determined they cannot lead to a valid solution. Examples include:

- **N-Queens Problem:** Places N queens on an N×N chessboard so that no two queens threaten each other, using backtracking to find all valid arrangements.
- **Sudoku Solver:** Fills a partially completed Sudoku grid by trying all possible numbers and backtracking whenever a conflict arises.

7. Graph Algorithms

Graph algorithms solve problems related to graph theory, including traversal, shortest paths, and connectivity. Examples include:

- **Dijkstra's Algorithm:** Computes the shortest paths from a source node to all other nodes in a weighted graph.
- **Kruskal's Algorithm:** Finds a minimum spanning tree for a weighted graph by selecting edges in order of increasing weight.
- **Bellman-Ford Algorithm:** Computes shortest paths from a single source in graphs with negative weight edges.

8. String Matching Algorithms

String matching algorithms find occurrences of a pattern within a text. Examples include:

- **Knuth-Morris-Pratt (KMP) Algorithm:** Searches for a pattern in a text by preprocessing the pattern to skip unnecessary comparisons.
- **Rabin-Karp Algorithm:** Uses hashing to find any of a set of pattern strings in a text efficiently.
- **Boyer-Moore Algorithm:** Searches for a pattern in a text by skipping sections of the text based on mismatched characters.

Chapter-4 Project

Sudoku Solver

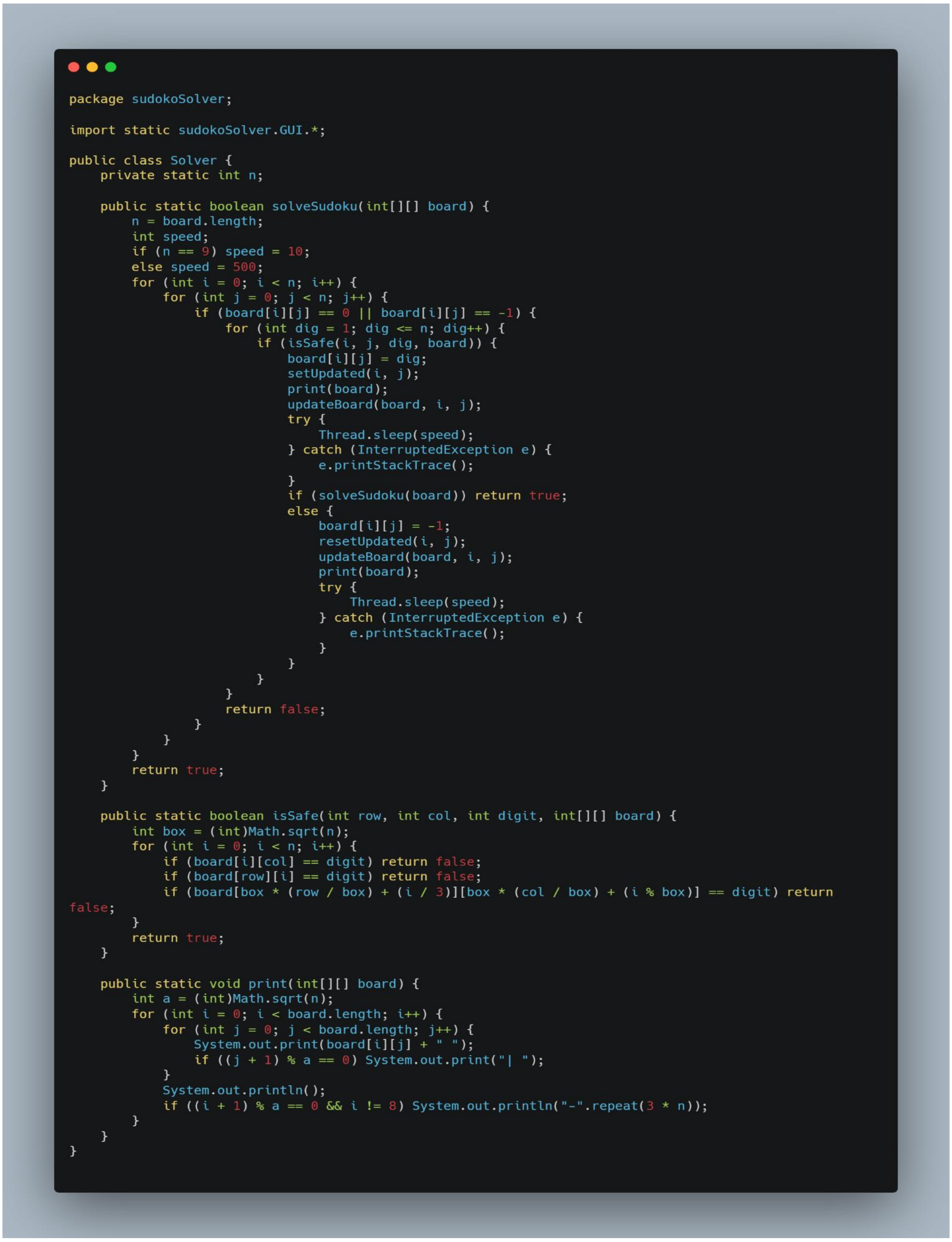
This project involves a Sudoku solver implemented using the Backtracking Algorithm, with a time complexity of O(9^n×n) and a space complexity of O(1). The solver addresses a nxn Sudoku puzzle, where each cell is evaluated recursively to determine the correct number. Backtracking is employed to explore possible numbers for each cell, reverting to previous states if a conflict arises, ensuring that the solution adheres to Sudoku rules.

The frontend of the project is built using Java Swing, providing a user-friendly graphical interface to visualize the Sudoku board. It includes features such as dynamic cell updates. The backend, written in Java, handles the logic for solving the puzzle. The GUI and backend work together to offer an interactive experience, demonstrating the effectiveness of the Backtracking Algorithm in solving complex puzzles.

The backend of this Sudoku solver project is implemented in Java and utilizes the Backtracking Algorithm to solve the Sudoku puzzle. The algorithm operates on a 9x9 grid, where each cell can hold a number from 1 to 9. The core logic involves attempting to fill each cell with a valid number and recursively solving the remaining cells. If a cell's number leads to a conflict or violates Sudoku rules, the algorithm backtracks by reverting to the previous cell and trying the next possible number.

This backend seamlessly integrates with the Java Swing-based GUI, updating the visual representation of the board in real-time as the solver progresses. This combination of Java Swing for UI and Java for backend processing ensures a robust and interactive Sudoku solving experience.

Solver.java



```java
package sudokoSolver;

import static sudokoSolver.GUI.*;

public class Solver {
    private static int n;

    public static boolean solveSudoku(int[][] board) {
        n = board.length;
        int speed;
        if (n == 9) speed = 10;
        else speed = 500;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 0 || board[i][j] == -1) {
                    for (int dig = 1; dig <= n; dig++) {
                        if (isSafe(i, j, dig, board)) {
                            board[i][j] = dig;
                            setUpdated(i, j);
                            print(board);
                            updateBoard(board, i, j);
                            try {
                                Thread.sleep(speed);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }
                            if (solveSudoku(board)) return true;
                            else {
                                board[i][j] = -1;
                                resetUpdated(i, j);
                                updateBoard(board, i, j);
                                print(board);
                                try {
                                    Thread.sleep(speed);
                                } catch (InterruptedException e) {
                                    e.printStackTrace();
                                }
                            }
                        }
                    }
                    return false;
                }
            }
        }
        return true;
    }

    public static boolean isSafe(int row, int col, int digit, int[][] board) {
        int box = (int)Math.sqrt(n);
        for (int i = 0; i < n; i++) {
            if (board[i][col] == digit) return false;
            if (board[row][i] == digit) return false;
            if (board[box * (row / box) + (i / 3)][box * (col / box) + (i % box)] == digit) return
false;
        }
        return true;
    }

    public static void print(int[][] board) {
        int a = (int)Math.sqrt(n);
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board.length; j++) {
                System.out.print(board[i][j] + " ");
                if ((j + 1) % a == 0) System.out.print("| ");
            }
            System.out.println();
            if ((i + 1) % a == 0 && i != 8) System.out.println("-".repeat(3 * n));
        }
    }
}
```

The Above Images are Code of Solver That helps to solve sudoku

The Solver class is the core component for solving the Sudoku puzzle using backtracking in the Java project. It integrates tightly with the GUI for visualizing the solving process in real-time.

Key Methods and Functionality

1. **solveSudoku(int[][] board)**

    - **Purpose:** This is the main recursive function responsible for solving the Sudoku board using the Backtracking Algorithm.
    - **Process:**
        - Iterates through the 9x9 grid to locate the next empty cell (denoted by 0 or -1).
        - For each empty cell, it attempts to place digits from 1 to 9.
        - For each attempted digit, the isSafe method checks if the digit can be legally placed in that position (i.e., it doesn't violate the Sudoku rules).
        - If placing a digit is valid, it updates the board and the GUI, using the setUpdated() and updateBoard() methods to visualize the process.
        - It introduces a delay to slow down the visualization based on the user-controlled slider (Thread.sleep(getDelay())).
        - If the recursive call to solveSudoku returns true, the puzzle is solved, and the function returns true.
        - If placing the digit leads to a dead-end, it backtracks by resetting the cell to -1 and tries the next digit.
        - If no digits fit, the function returns false to backtrack to the previous cell.

2. **isSafe(int row, int col, int digit, int[][] board)**

    - **Purpose:** This method checks if placing a specific digit in the given row and column is allowed according to the rules of Sudoku.
    - **Process:**
        - The function iterates over the respective row and column to ensure that the digit is not repeated.
        - It also checks the corresponding 3x3 subgrid to ensure that the digit is not present there either.

        - The grid boundaries for the subgrid are calculated using integer division and modulus operations.
    - **Return:** Returns true if the digit can be safely placed in the specified cell, otherwise returns false.

3. **print(int[][] board)**
    - **Purpose:** This method prints the current state of the Sudoku board to the console. It's mainly used for debugging and visual inspection outside the GUI.
    - **Process:**
        - Iterates through the 9x9 grid and prints each number in the cells.
        - Adds visual separators to differentiate the 3x3 subgrids by printing vertical lines between subgrids and horizontal lines after every 3 rows.

Integration with GUI

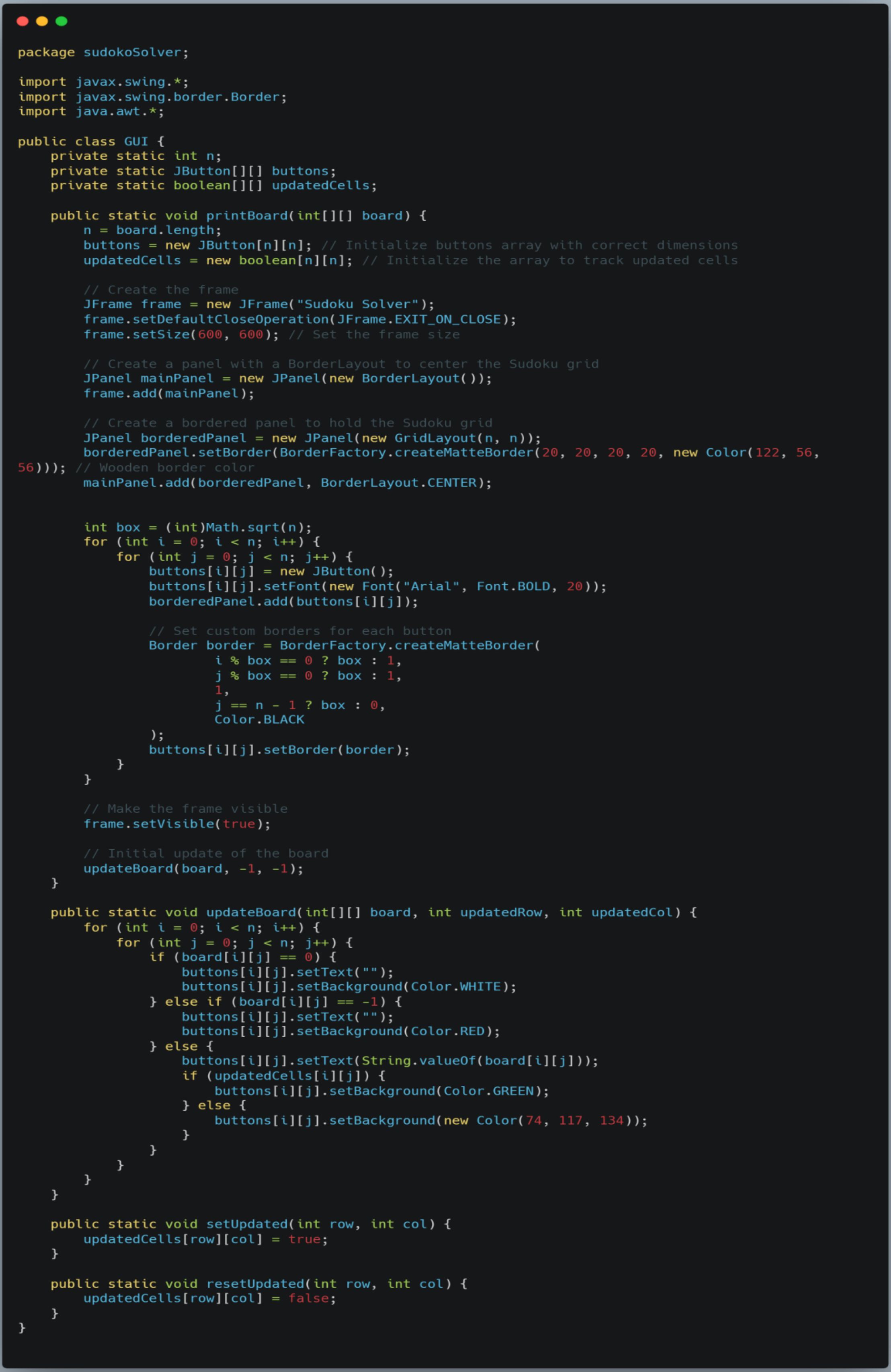
- **Visual Updates:** The Solver class interacts with the GUI class methods like setUpdated, resetUpdated, and updateBoard to reflect changes in the board during the solving process. After each valid placement, the GUI is updated with the new value, and the cell color is changed to highlight updates.

Backtracking Algorithm

The backtracking approach used here attempts to fill in each empty cell by trying all possible digits (1 through 9). It recursively moves forward by filling cells, but if a contradiction arises (i.e., the digit placement is no longer valid), it backtracks by undoing the previous digit and trying the next possibility. This method guarantees finding a solution if one exists.

Summary

The Solver class is an efficient solution for solving Sudoku puzzles. It not only implements a recursive backtracking algorithm but also provides real-time visualization of the solving process, thanks to its tight integration with the Java Swing-based GUI. Through this method, users can visually observe the algorithm's progress, gaining a deeper understanding of how Sudoku can be solved algorithmically



```java
package sudokoSolver;

import javax.swing.*;
import javax.swing.border.Border;
import java.awt.*;

public class GUI {
    private static int n;
    private static JButton[][] buttons;
    private static boolean[][] updatedCells;

    public static void printBoard(int[][] board) {
        n = board.length;
        buttons = new JButton[n][n]; // Initialize buttons array with correct dimensions
        updatedCells = new boolean[n][n]; // Initialize the array to track updated cells

        // Create the frame
        JFrame frame = new JFrame("Sudoku Solver");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(600, 600); // Set the frame size

        // Create a panel with a BorderLayout to center the Sudoku grid
        JPanel mainPanel = new JPanel(new BorderLayout());
        frame.add(mainPanel);

        // Create a bordered panel to hold the Sudoku grid
        JPanel borderedPanel = new JPanel(new GridLayout(n, n));
        borderedPanel.setBorder(BorderFactory.createMatteBorder(20, 20, 20, 20, new Color(122, 56,
56))); // Wooden border color
        mainPanel.add(borderedPanel, BorderLayout.CENTER);


        int box = (int)Math.sqrt(n);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                buttons[i][j] = new JButton();
                buttons[i][j].setFont(new Font("Arial", Font.BOLD, 20));
                borderedPanel.add(buttons[i][j]);

                // Set custom borders for each button
                Border border = BorderFactory.createMatteBorder(
                        i % box == 0 ? box : 1,
                        j % box == 0 ? box : 1,
                        1,
                        j == n - 1 ? box : 0,
                        Color.BLACK
                );
                buttons[i][j].setBorder(border);
            }
        }

        // Make the frame visible
        frame.setVisible(true);

        // Initial update of the board
        updateBoard(board, -1, -1);
    }

    public static void updateBoard(int[][] board, int updatedRow, int updatedCol) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 0) {
                    buttons[i][j].setText("");
                    buttons[i][j].setBackground(Color.WHITE);
                } else if (board[i][j] == -1) {
                    buttons[i][j].setText("");
                    buttons[i][j].setBackground(Color.RED);
                } else {
                    buttons[i][j].setText(String.valueOf(board[i][j]));
                    if (updatedCells[i][j]) {
                        buttons[i][j].setBackground(Color.GREEN);
                    } else {
                        buttons[i][j].setBackground(new Color(74, 117, 134));
                    }
                }
            }
        }
    }

    public static void setUpdated(int row, int col) {
        updatedCells[row][col] = true;
    }

    public static void resetUpdated(int row, int col) {
        updatedCells[row][col] = false;
    }
}
```

The Above Images are Code of GUI For Visualisation of Sudoku Grid:

Overview:

The GUI is designed to visualize a Sudoku board and update it in real-time as the solver works through the puzzle. It includes features for controlling the speed of the updates and visually distinguishing between different cell states.

Key Components

1. Class and Fields
    - GUI: The main class for the graphical interface.
    - n: Stores the dimension of the Sudoku board (typically 9 for a standard Sudoku puzzle).
    - buttons: A 2D array of JButton objects representing each cell in the Sudoku grid.
    - updatedCells: A 2D boolean array used to track which cells have been updated.
    - speedSlider: A JSlider that allows the user to adjust the speed of the solver's updates.
2. printBoard(int[][] board) Method
    - Initializes the Sudoku board by setting up the JFrame and the JPanel with a GridLayout for the Sudoku grid.
    - Configures each cell as a JButton, setting its font and border. Borders are customized to reflect Sudoku grid blocks.
    - Adds a JSlider for controlling the speed of updates and labels it to indicate time in milliseconds.
3. updateBoard(int[][] board, int updatedRow, int updatedCol) Method
    - Updates the visual representation of the board based on the current state of the board array.
    - Clears or sets text and background color for each cell, differentiating between empty cells, cells with values, and cells that have been updated.
4. setUpdated(int row, int col) and resetUpdated(int row, int col) Methods
    - These methods update the updatedCells array to mark cells that have been updated during the solving process.
5. getDelay() Method

- Retrieves the current value from the speedSlider, which represents the delay (in milliseconds) between updates.

6. main(String[] args) Method

    - Initializes an example 9x9 Sudoku board and displays it using the printBoard method. This method serves as an entry point for running the GUI application.

Usage:

Main Frame and Layout:

- The main window features a clean and simple design. It includes a bordered panel where the Sudoku grid is displayed, using a GridLayout to represent each cell in a neat, consistent format. This ensures the grid remains structured regardless of the board size (3x3, 4x4, 9x9, etc.).

Sudoku Grid:

- The grid is made up of JButton elements that represent each cell in the Sudoku board. These buttons are dynamically updated as the algorithm fills in the correct digits. The cells are color-coded:
    - **White Background**: Represents empty cells that haven't been filled yet.
    - **Red Background**: Marks cells that are temporarily set back to empty (used for the backtracking step when the algorithm is undoing previous choices).
    - **Green Background**: Highlights cells that have been updated with a correct value, indicating progress in the solving process.
    - **Default Background**: For cells with pre-filled values that remain static throughout the solution.

Custom Borders:

- Each cell has custom borders to highlight the boxes within the Sudoku grid. For example, in a 9x9 grid, the cells are grouped into 3x3 sub-grids, with thicker borders separating the sub-grids from one another. This visual cue helps the user distinguish between different sections of the board.

Speed Control Slider:

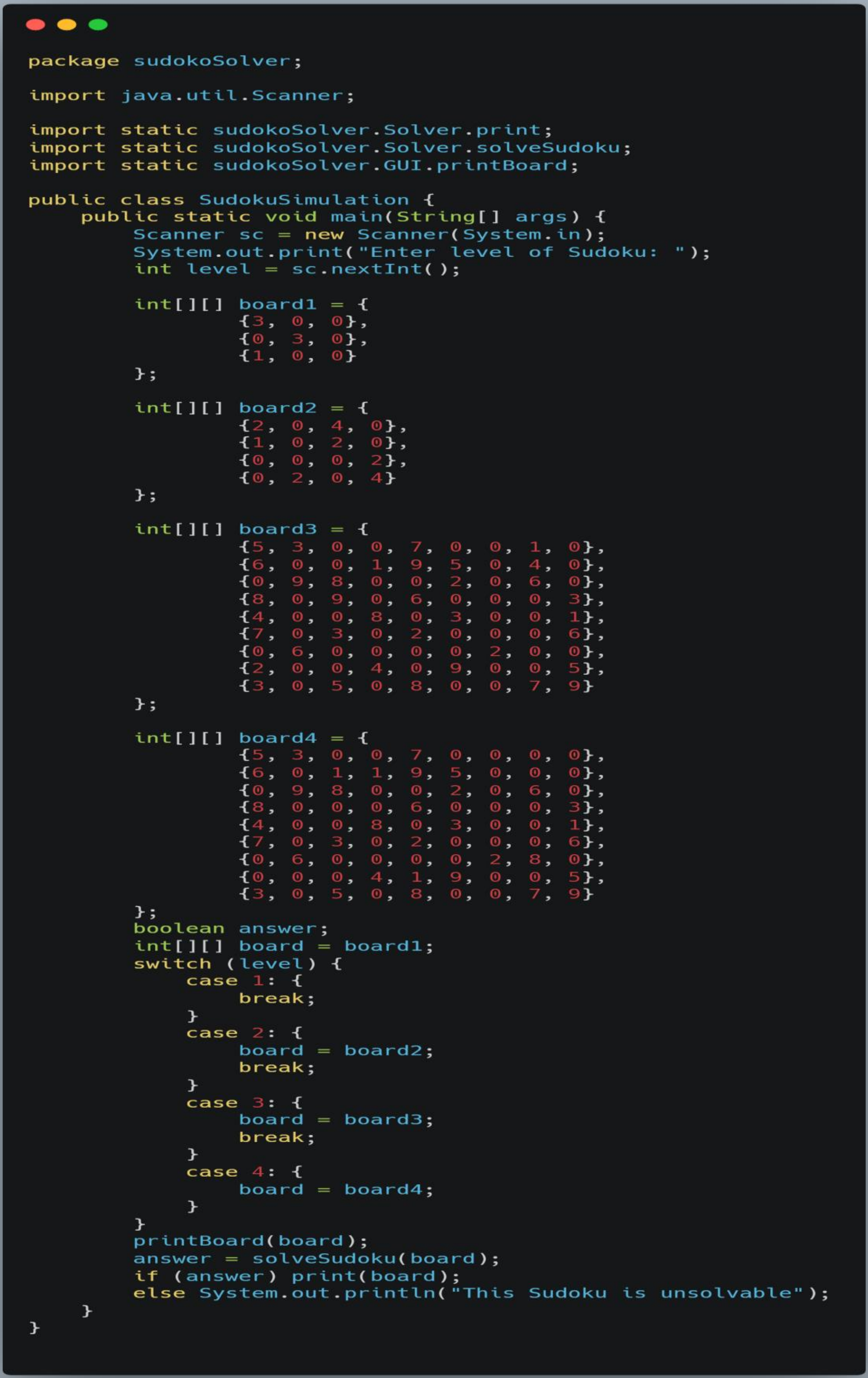
- A JSlider is included at the bottom of the frame, allowing users to control the speed of the Sudoku-solving process. The slider ranges from fast (50ms per step) to slow (2000ms per step), enabling users to adjust the pace of the visualization. The slider is labeled with descriptive text, "Fast <--------> Slow", providing clear guidance for users.

Real-Time Updates:

- The GUI is responsive and updates in real-time as the backtracking algorithm progresses. Each time a cell is updated or reset during the solving process, the board is refreshed to reflect these changes, allowing users to observe the inner workings of the algorithm step by step.

Interactive User Input:

- The GUI is initialized based on user input. The difficulty level (ranging from easy to more complex Sudoku puzzles) is selected at the start, which determines the size of the grid and the initial configuration of the board. This adds an interactive element to the program, allowing users to choose the complexity of the puzzle they want to visualize.



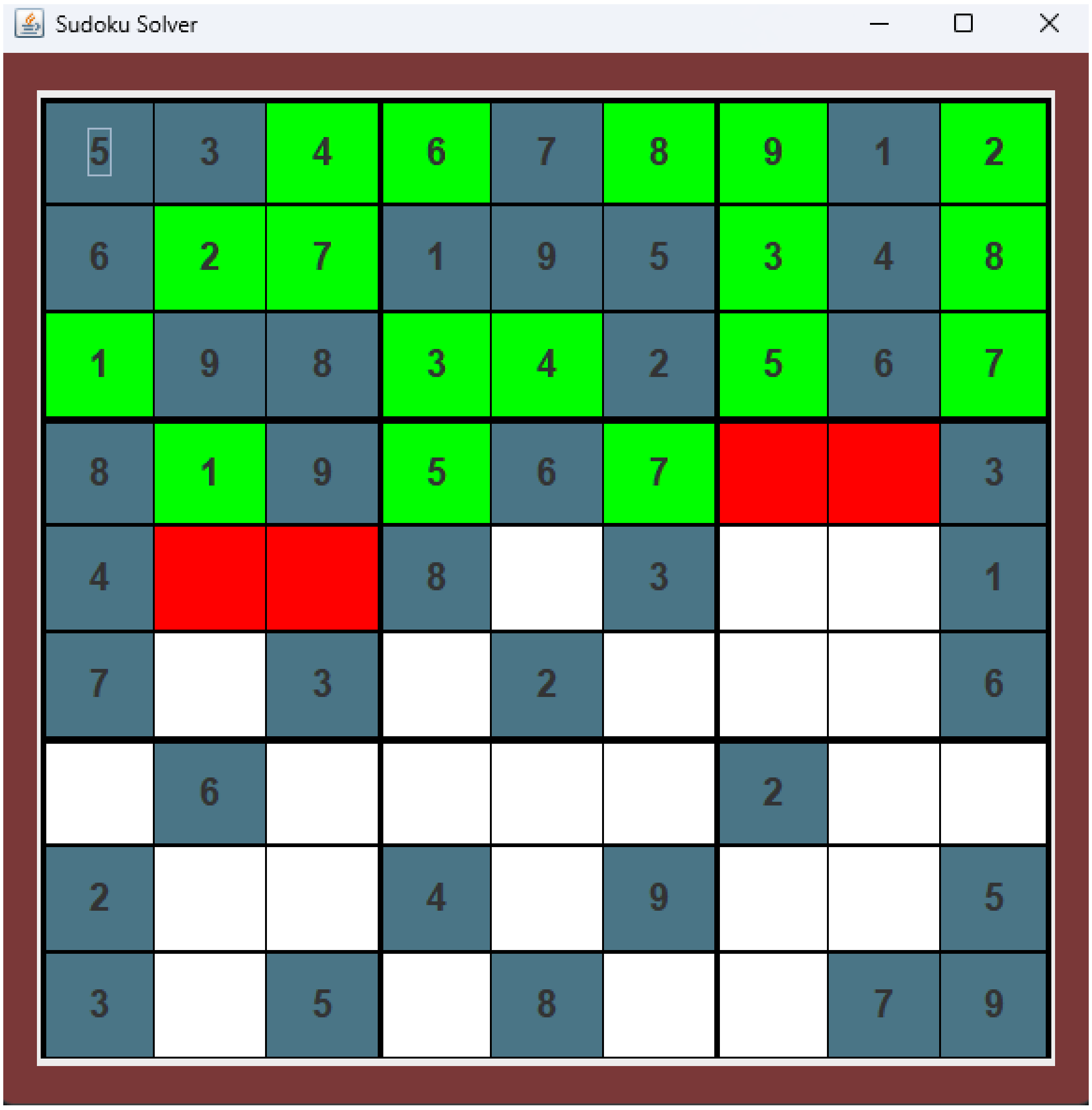
```java
package sudokoSolver;

import java.util.Scanner;

import static sudokoSolver.Solver.print;
import static sudokoSolver.Solver.solveSudoku;
import static sudokoSolver.GUI.printBoard;

public class SudokuSimulation {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter level of Sudoku: ");
        int level = sc.nextInt();

        int[][] board1 = {
                {3, 0, 0},
                {0, 3, 0},
                {1, 0, 0}
        };

        int[][] board2 = {
                {2, 0, 4, 0},
                {1, 0, 2, 0},
                {0, 0, 0, 2},
                {0, 2, 0, 4}
        };

        int[][] board3 = {
                {5, 3, 0, 0, 7, 0, 0, 1, 0},
                {6, 0, 0, 1, 9, 5, 0, 4, 0},
                {0, 9, 8, 0, 0, 2, 0, 6, 0},
                {8, 0, 9, 0, 6, 0, 0, 0, 3},
                {4, 0, 0, 8, 0, 3, 0, 0, 1},
                {7, 0, 3, 0, 2, 0, 0, 0, 6},
                {0, 6, 0, 0, 0, 0, 2, 0, 0},
                {2, 0, 0, 4, 0, 9, 0, 0, 5},
                {3, 0, 5, 0, 8, 0, 0, 7, 9}
        };

        int[][] board4 = {
                {5, 3, 0, 0, 7, 0, 0, 0, 0},
                {6, 0, 1, 1, 9, 5, 0, 0, 0},
                {0, 9, 8, 0, 0, 2, 0, 6, 0},
                {8, 0, 0, 0, 6, 0, 0, 0, 3},
                {4, 0, 0, 8, 0, 3, 0, 0, 1},
                {7, 0, 3, 0, 2, 0, 0, 0, 6},
                {0, 6, 0, 0, 0, 0, 2, 8, 0},
                {0, 0, 0, 4, 1, 9, 0, 0, 5},
                {3, 0, 5, 0, 8, 0, 0, 7, 9}
        };
        boolean answer;
        int[][] board = board1;
        switch (level) {
            case 1: {
                break;
            }
            case 2: {
                board = board2;
                break;
            }
            case 3: {
                board = board3;
                break;
            }
            case 4: {
                board = board4;
            }
        }
        printBoard(board);
        answer = solveSudoku(board);
        if (answer) print(board);
        else System.out.println("This Sudoku is unsolvable");
    }
}
```

The SudokuSimulation class allows users to select the difficulty level of a Sudoku puzzle and solves it using backtracking. It also visualizes the solving process through a graphical interface (GUI).

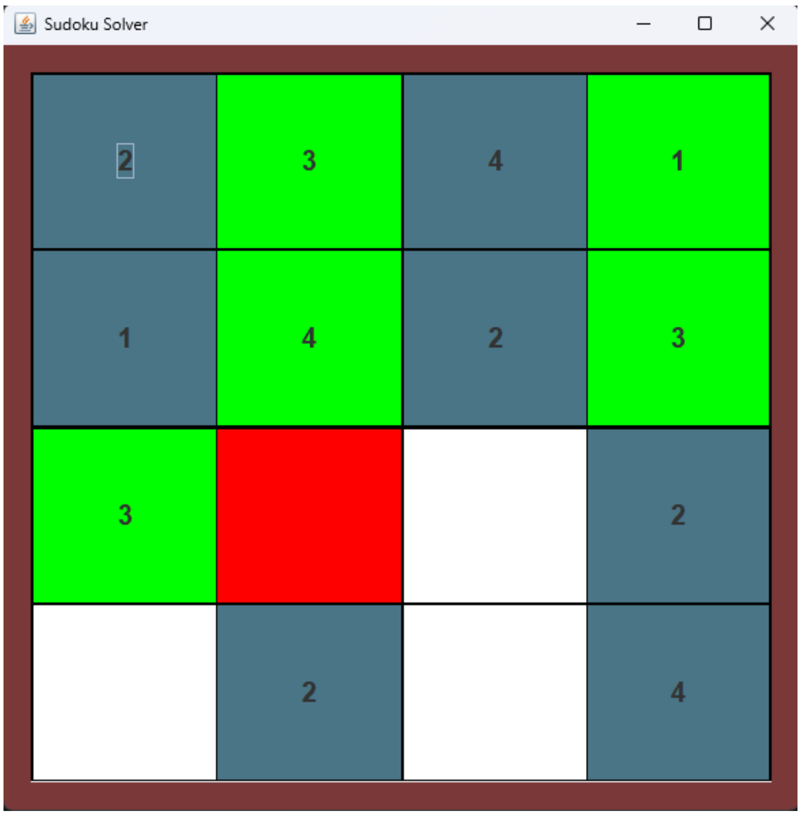
Key Features:

1. **User Input:**
    - The program prompts the user to enter the difficulty level of the Sudoku puzzle.
    - The level variable is an integer input by the user that determines which predefined Sudoku board will be used.
2. **Predefined Sudoku Boards:**
    - The class includes four predefined Sudoku boards (board1, board2, board3, board4) of varying difficulty levels:
        - board1: A 3x3 board for easier visualization and simpler logic.
        - board2: A 4x4 board with a slightly more complex structure.
        - board3 and board4: Standard 9x9 Sudoku boards with varying levels of difficulty.
3. **Board Selection:**
    - Based on the user-selected level, the program selects the appropriate board. The switch statement ensures that the correct board is assigned to the board variable.
4. **GUI Integration:**
    - The printBoard(board) method from the GUI class is called to display the board in the graphical interface. The visualization includes features like colored borders and updated cells.
    - The GUI allows users to watch the solving process in real-time, with the backtracking algorithm dynamically updating the cells on the grid.
5. **Sudoku Solver:**
    - The solveSudoku(board) method from the Solver class attempts to solve the selected Sudoku board using the backtracking algorithm.
    - If the puzzle is solvable, the solved board is printed to the console using the print(board) method.
    - If the puzzle is unsolvable, the program outputs "This Sudoku is unsolvable" to the console.

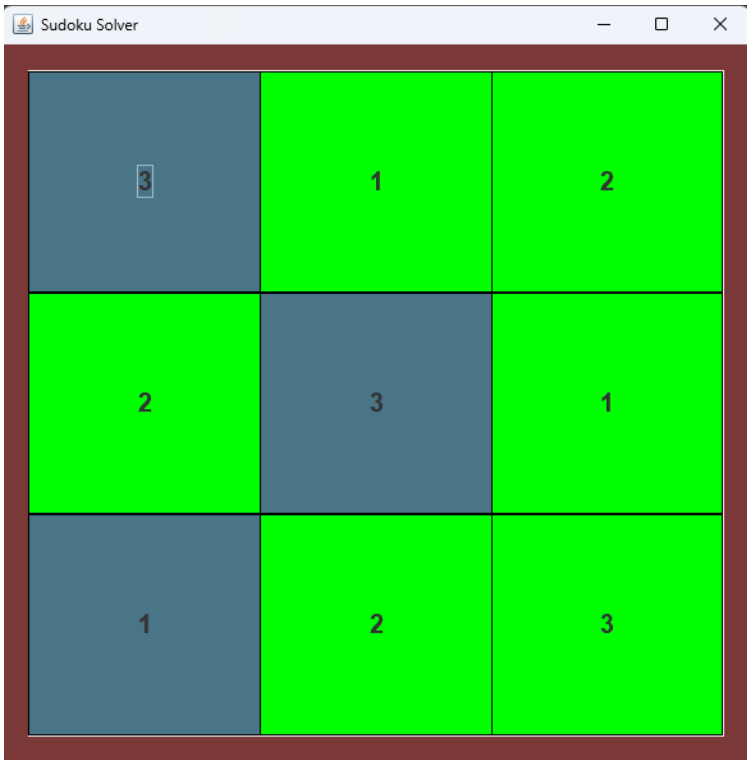
Output:



5	3	4	6	7	8	9	1	2
6	2	7	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	1	9	5	6	7			3
4			8		3			1
7		3		2				6
	6					2		
2			4		9			5
3		5		8			7	9



2	3	4	1
1	4	2	3
3	🟥		2
	2		4



3	1	2
2	3	1
1	2	3

Chapter-5 Learning Outcomes

Understanding Backtracking Algorithms:

- This project provides hands-on experience with backtracking algorithms. Students will learn how backtracking works by recursively trying possible solutions and undoing invalid choices, which is essential for solving constraint-based problems like Sudoku.

Mastering Recursion:

- The project enhances problem-solving skills by developing an understanding of recursion. Students will understand the recursive nature of backtracking and how recursive function calls are used to explore potential solutions step by step.

Graphical User Interface (GUI) Design in Java:

- By integrating Java Swing for the UI, students will learn how to create dynamic and interactive GUIs. This includes the use of components like buttons, panels, sliders, and frames, allowing the visual representation of the Sudoku board and the solving process.

Data Visualization:

- Students will gain experience in visualizing algorithms and data by linking backend logic to frontend displays. The board's real-time updates provide an understanding of how computational processes can be visualized to enhance user engagement and understanding.

Time and Space Complexity Analysis:

- The project provides insight into evaluating time and space complexities. Students will explore the complexity of solving a Sudoku puzzle (O(9^(n*n)) for time and O(1) for space) and understand the challenges of performance optimization in recursive algorithms.

Object-Oriented Programming (OOP) in Java:

- The project reinforces OOP concepts such as encapsulation, modularity, and code reuse. Students will understand how different classes (Solver, GUI, and Simulation) interact with each other, demonstrating real-world application of OOP design principles.

Working with 2D Arrays and Matrix Operations:

- Students will learn to work with 2D arrays to represent Sudoku grids. This project will enhance their skills in matrix operations, which are commonly used in various computational problems, particularly in game development, simulations, and mathematical modeling.

User Interaction and Input Handling:

- The project helps students understand how to capture and process user inputs through the command line and the GUI, an essential skill in developing interactive applications.

Debugging and Error Handling:

- Through the process of building and refining the Sudoku solver, students will gain experience in debugging recursive algorithms, handling edge cases (such as unsolvable puzzles), and ensuring the stability of both the UI and the logic.

Application of Theoretical Concepts to Real Problems:

- This project bridges the gap between theoretical computer science concepts and practical implementation. Students will see how algorithms like backtracking, matrix manipulation, and recursion can be applied to solve real-world problems like Sudoku efficiently.

Chapter-6 Conclusion

This Sudoku Solver project serves as a comprehensive exercise in problem-solving, algorithmic thinking, and software development. By implementing a backtracking algorithm to solve Sudoku puzzles, I gained a deep understanding of how recursive approaches can be applied to constraint-based problems. The integration of Java Swing for the user interface allowed for real-time visualization, enhancing my learning experience by linking backend algorithms to frontend displays.

Throughout the project, I was exposed to important concepts such as time and space complexity analysis, object-oriented programming, and matrix manipulation. Additionally, the project encouraged good software design practices, including modularity and code reuse, while offering an opportunity to work with 2D arrays, graphical components, and user interaction.

Ultimately, this project bridged the gap between theoretical knowledge and practical application, providing me with a solid foundation in both algorithm development and UI programming. It highlights the power of combining different areas of computer science to create functional, interactive software solutions.

References

https://www.w3schools.com/java/

https://www.geeksforgeeks.org/introduction-to-backtracking-2/

https://www.javatpoint.com/java-swing