Suduko Solver Visualizer

PROJECT REPORT

LOVELY PROFESSIONAL UNIVERSITY
PHAGWARA, PUNJAB

EXTERNAL EXPERT INPUT - DATA STRUCTURE
CSES003

Submitted by: Madhav Arora
Section: 9SK02
Registration No.: 12217099
Roll No.: 28
Submitted to: Rahul Singh

Prepared for
Spring 2024
Date of submission:-10/7/2024

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to all those who have contributed to the successful completion of this project.

First and foremost, I extend my deepest appreciation to my project supervisor, for their unwavering support, guidance, and valuable insights throughout the entire project.

I would like to acknowledge the support and cooperation received from Lovely Professional University well as the resources and facilities provided, which have significantly contributed to the successful execution of this project.

Last but not least, I would like to express my deep appreciation to my family and friends for their understanding, encouragement, and patience throughout this project journey. Their unwavering support has been a constant source of motivation.

Each of the above individual's contributions has played a crucial role in shaping this report and enhancing my learning experience. I am truly grateful for their involvement and support.

ABSTRACT

In the last decade, solving the Sudoku puzzle has become every one's passion. The simplicity of puzzle's structure and the low requirement of mathematical skills caused people to have enormous interest in accepting challenges to solve the puzzle. Therefore, developers have tried to find algorithms in order to generate the variety of puzzles for human players so that they could be even solved by computer programming. In this essay, we have presented an algorithm called pencil-and-paper using human strategies. The purpose is to implement a more efficient algorithm and then compare it with another Sudoku solver named as back tracking algorithm. This algorithm is a general algorithm that can be employed in to any problems. The results have proved that the pencil-and-paper algorithm solves the puzzle faster and more effective than the back tracking algorithm.

Contents

SI No.	***Details***
1	*Introduction*
2	Analysis
3	Design(UML Diagram)
4	Technologies Used
5	Implementation
	5.1 Sample Code
	5.2 Screenshots
6	Conclusion
	References

Introduction

1.1 Introduction

Sudoku puzzles are becoming increasingly popular worldwide, appearing in newspapers, books, and numerous websites. Developers continuously strive to create more challenging and interesting puzzles.

This project presents a Sudoku Solver using a pencil-and-paper algorithm based on human techniques to solve the puzzles. This algorithm is implemented to mimic human strategies, thus the name "pencil-and-paper algorithm." For comparison, the brute force algorithm, which systematically generates possible solutions until the correct one is found, is also implemented to evaluate the efficiency of the proposed algorithm.

1.2 Problem Statement

Solving Sudoku has been a challenging problem, with efforts focused on developing more effective algorithms to reduce computing time and utilize lower memory space. This project develops an algorithm for solving Sudoku puzzles using the pencil-and-paper method, which mimics human solving techniques. The aim is to implement these techniques in an algorithmic form.

Different variants of Sudoku exist, such as 4x4 grids, 9x9 grids, and 16x16 grids. This project focuses on the classic and regular 9x9 Sudoku board and compares the pencil-and-paper method with the brute force algorithm. The goal is to determine how these algorithms differ, which is more effective, and if there are ways to make these algorithms more efficient.

1.3 Purpose

The aim of this project is to investigate and compare two methods for solving Sudoku puzzles: the pencil-and-paper algorithm and the brute force algorithm. By simulating human strategies, the pencil-and-paper algorithm attempts to solve the puzzle efficiently. The project seeks to find an effective method to solve Sudoku puzzles and analyze the efficiency of the implemented algorithms.

	a	b	c	d	e	f	g	h	i
1			9				7	2	8
2	2	7	8			3		1	
3		9					6	4	
4		5			6		2		
5			6				3		
6		1			5				
7	1			7		6		3	4
8				5		4			
9	7		9	1			8		5

Fig.1 *An example of a Sudoku puzzle.*

1.4 Abbreviations and Definitions

In this project, we use terminology commonly found in journals and research papers on Sudoku. Below is a brief description of some of the abbreviations and definitions used in the text:

- **Sudoku**: A logic-based, combinatorial number placement puzzle. The word "Sudoku" is short for "Su-ji wa dokushin ni kagiru" (in Japanese), meaning "the numbers must be single."
- **Box (Region, Block)**: A region is a 3x3 box within the Sudoku grid. There are 9 such regions in a traditional Sudoku puzzle.
- **Cell (Square)**: The smallest unit of the Sudoku board, equivalent to a single square within the grid.
- **Candidates**: The possible values that can be placed into an empty cell.
- **Clues**: The given numbers in the grid at the beginning of the puzzle.
- **Grid (Board)**: The entire Sudoku board, which consists of a matrix of cells.

Analysis

This section begins with an explanation of Sudoku. It includes research on previous works related to this subject, a discussion on evaluated algorithms, and finally, a description of how this work is carried out.

2.1 Short Introduction to Sudoku

Sudoku is a logic-based puzzle played with numbers from 1 to 9. The puzzle first appeared in newspapers in November 1892 in France and was later presented in its modern form by Howard Garns, an American architect . There are numerous journals, papers, and essays on Sudoku solvers, each presenting different types of algorithms. Sudoku's popularity can be attributed to several factors: it is fun, fascinating, and easy to learn due to its simple rules.

There are various types of Sudoku puzzles. The classic Sudoku consists of a 9x9 grid with given clues in various places. Mini Sudoku features smaller grids, such as 4x4 or 6x6, while Mega Sudoku includes larger grids, such as 12x12 or 16x16 . This text focuses primarily on classic Sudoku with a 9x9 grid. Sudoku's rules are straightforward, making it accessible to a broad audience. It is also believed to improve cognitive abilities, adding to its widespread appeal.

The structure of a classic Sudoku puzzle is simple. Some numbers are already placed on the 9x9 board before the game begins, making the puzzle solvable. The board consists of 81 cells, divided into nine 3x3 sub-boards, each called a "box" or "region." The main objective is to place numbers from 1 to 9 on the board so that each row, column, and box contains each number exactly once, with no repetition. Generally, the puzzle has a unique solution. Certain techniques for solving the puzzle by hand can be implemented into a computer program. These techniques are detailed in section 2.3.

2.2 Previous Research

There is a large volume of published studies on Sudoku problems. Numerous researchers have explored solving Sudoku problems more efficiently . It has been conclusively shown that solving the puzzle using different algorithms is possible. Many developers seek optimization techniques, such as genetic algorithms and simulated annealing, to enhance the efficiency of their solutions.

Several authors have contributed to the study of solving Sudoku puzzles. Nelishia Pillay provided a solution that combines human intuition with optimization. This author investigated the use of genetic programming to

improve a set of programs comprising heuristic moves. In contrast, our approach aims to solve the Sudoku puzzle based on human strategies, using techniques such as the naked single method and the hidden single method.

J.F. Crook also discussed solving Sudoku and presented an algorithm for solving puzzles of varying difficulty levels using the pencil-and-paper algorithm. However, this method has not been implemented, making it difficult to evaluate its performance.

Tom Davis researched "The Mathematics of Sudoku." He described all the techniques people commonly use to solve puzzles, focusing on a mathematical perspective. However, not all the strategies he mentions are required to solve the puzzle. For instance, easier puzzles can often be solved using only one or two strategies.

2.3 Algorithm: Backtracking

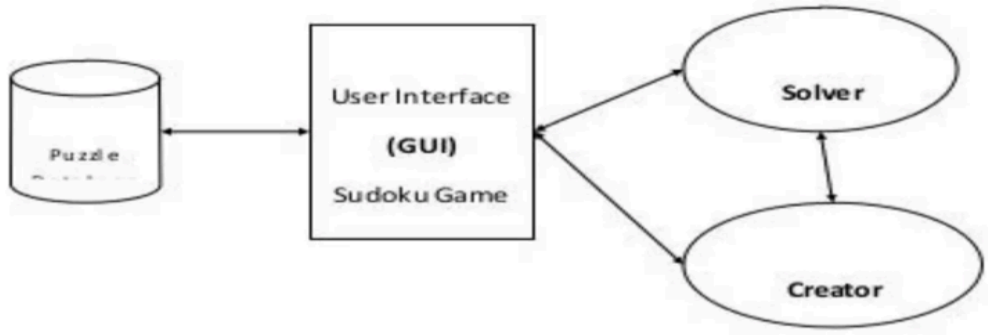
The unique missing method and the naked single method can solve puzzles with easy and medium difficulty levels. However, for more challenging puzzles, such as those classified as hard and evil, the backtracking method is used to complete the algorithm. A human player solves the puzzle using simple techniques. If the puzzle is not solvable using these techniques, the player then tries to fill the remaining empty squares by guessing.

The backtracking method, which resembles the human strategy of guessing, serves as an auxiliary method to the pencil-and-paper algorithm. If the puzzle cannot be solved using the unique missing method and the naked single method, the backtracking method is employed to fill the remaining empty squares. Generally, the backtracking method finds an empty square and assigns the lowest valid number to it, considering the contents of other squares in the same row, column, and box. If none of the numbers from 1 to 9 are valid in a certain square, the algorithm backtracks to the previous square that was recently filled.

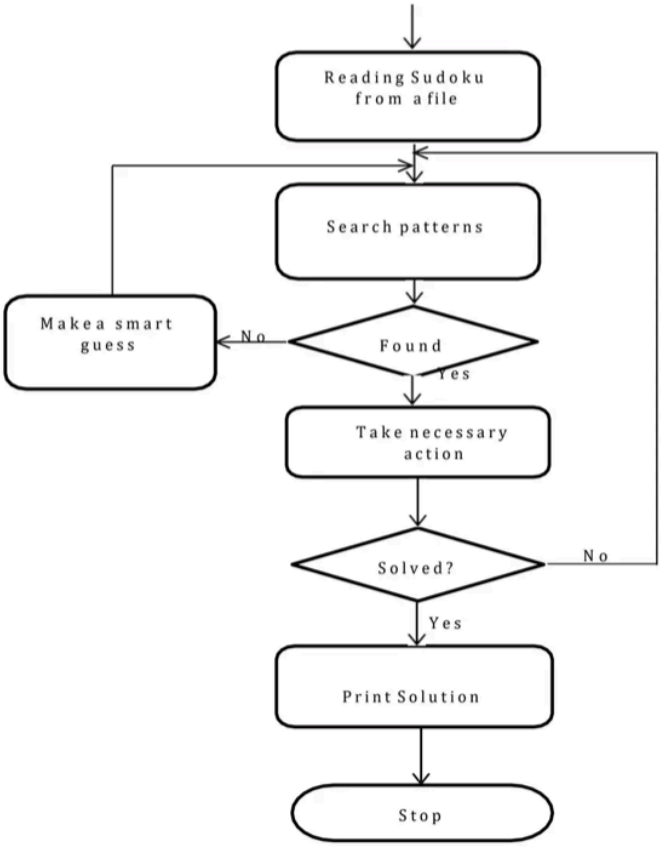
The above-mentioned methods are a suitable combination for solving any Sudoku puzzles. The naked single method can quickly identify single candidates for empty squares that require only one value. As the puzzle nears its solution, the unique missing method can be used to fill the remaining puzzles. Finally, if neither method completes the board, the backtracking method is called to fill the remaining squares.

Design

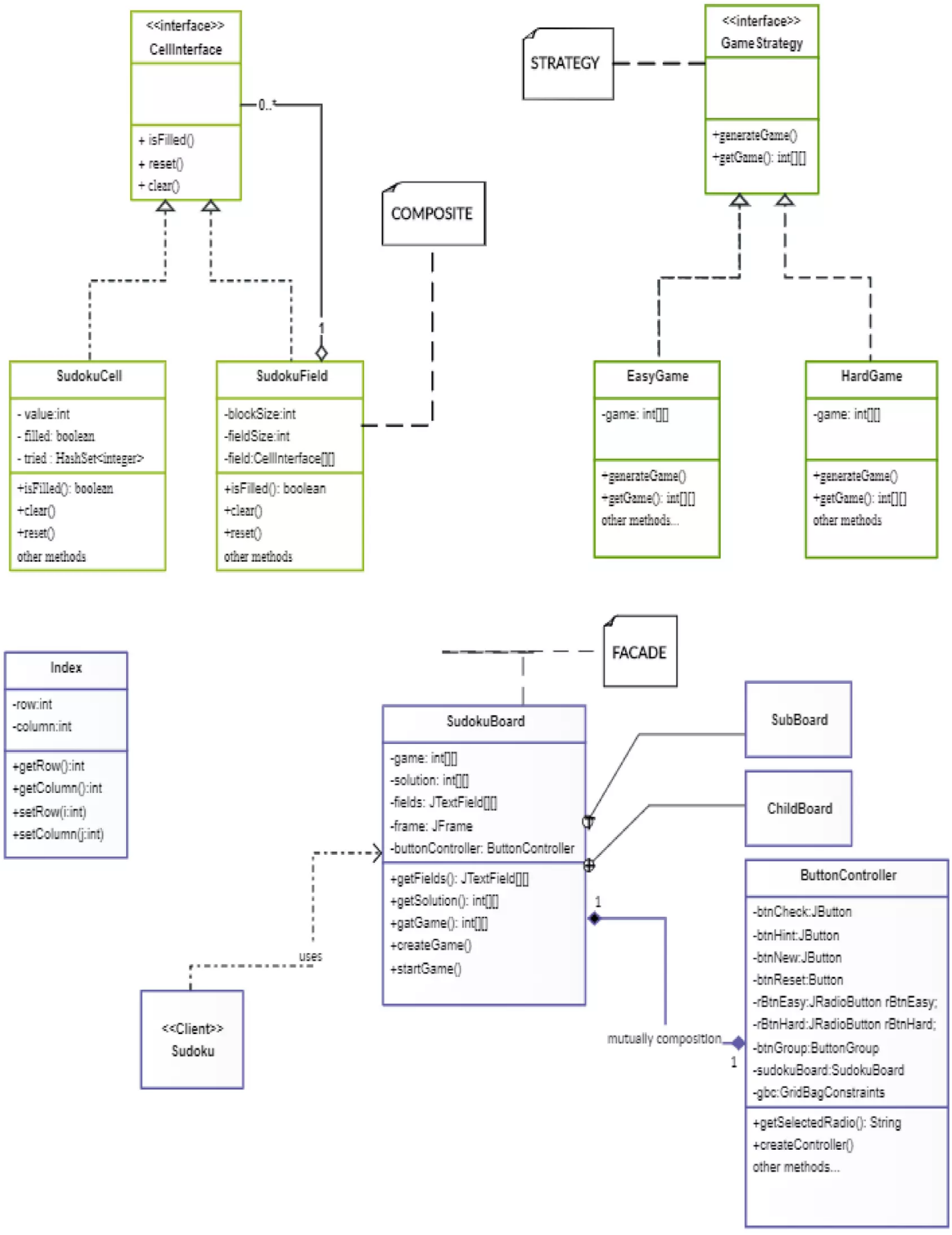
3.1 Block Diagram



3.2 Flow Chart



3.3 Class Diagram



Technologies Used

In this project, the following technologies were utilized to develop the Sudoku solver visualizer:

Java: Java is a versatile and widely-used programming language that provides a robust framework for building complex applications. It was chosen for this project due to its object-oriented capabilities and extensive libraries that simplify the development process.

Swing: Swing is a part of Java Foundation Classes (JFC) used to create window-based applications. It provides a rich set of GUI components and is platform-independent. In this project, Swing is used to create the graphical user interface (GUI) for the Sudoku visualizer, allowing users to interact with the application visually.

Recursion: Recursion is a fundamental programming technique where a method calls itself to solve a problem. It is particularly useful for solving problems that can be broken down into smaller, similar subproblems. In this project, recursion is employed in the Sudoku solving logic to efficiently navigate through the puzzle and find solutions.

Object-Oriented Programming (OOP): OOP is a programming paradigm that uses objects and classes to organize code into reusable and modular components. The Sudoku solver is designed using OOP principles to enhance code readability, maintainability, and scalability. Classes and objects are used to represent different elements of the Sudoku puzzle, such as the board, cells, and solving algorithms.

JUnit: JUnit is a popular testing framework for Java applications. It provides a simple way to write and run repeatable tests. In this project, JUnit is used to test the Sudoku solving logic, ensuring that the implemented algorithms produce correct and efficient solutions.

Implementation

The Sudoku solver uses a backtracking algorithm to find a solution to the puzzle. The core idea behind this approach is to place a number in an empty cell, and then recursively attempt to fill in the remaining cells. If a conflict is detected, the algorithm backtracks and tries the next possible number. This process continues until the puzzle is either solved or determined to be unsolvable. Here is a detailed explanation and the implementation in Java:

Steps of the Backtracking Algorithm

1. **Identify an Empty Cell**:
    - The algorithm scans the Sudoku grid to find the first empty cell (a cell with a 0 value).
2. **Try Possible Numbers**:
    - For each empty cell, the algorithm tries placing each number from 1 to 9, one by one.
    - After placing a number, it checks if it violates any Sudoku rules (no number should repeat in any row, column, or 3x3 sub-grid).
3. **Recursively Solve the Remaining Puzzle**:
    - If placing a number does not violate any rules, the algorithm recursively attempts to solve the remaining puzzle with this partial solution.
    - If the recursive call successfully solves the puzzle, the current placement is part of the solution.
4. **Backtrack If Needed**:
    - If placing a number leads to a dead end (i.e., no valid number can be placed in a future step), the algorithm removes the number (backtracks) and tries the next number.
5. **Terminate When Solved**:
    - The process continues until the entire grid is filled correctly or all options are exhausted, indicating that the puzzle is unsolvable.

5.1 Sample Code

Solver.java

```
public class Solver {
    private int[][] board;
    private static final int SIZE = 9;
    private SolverCallback callback;

    public interface SolverCallback {
        void updateCell(int row, int col, int value);
        void delay(int milliseconds);
    }

    public Solver(int[][] board, SolverCallback callback) {
        this.board = board;
        this.callback = callback;
    }

    public boolean solve() {
        return solveHelper(0, 0);
    }

    private boolean solveHelper(int row, int col) {
        if (col == SIZE) {
            row++;
            col = 0;
        }
        if (row == SIZE) {
            return true;
        }
        if (board[row][col] != 0) {
            return solveHelper(row, col + 1);
        }
        for (int num = 1; num <= SIZE; num++) {
            if (isValid(row, col, num)) {
                board[row][col] = num;
                callback.updateCell(row, col, num);
                callback.delay(50);
                if (solveHelper(row, col + 1)) {
                    return true;
                }
                board[row][col] = 0;
                callback.updateCell(row, col, 0);
                callback.delay(50);
            }
        }
        return false;
    }
```

```
    private boolean isValid(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;
    }

    public int[][] getBoard() {
        return board;
    }
}
```

SudukoClass.java

```
public class SudokuClass {
    private int[][] board;
    private static final int SIZE = 9;

    public SudokuClass(int[][] initialBoard) {
        this.board = initialBoard;
    }

    public int getCell(int row, int col) {
        return board[row][col];
    }

    public void setCell(int row, int col, int value) {
        board[row][col] = value;
    }

    public boolean isEmpty(int row, int col) {
        return board[row][col] == 0;
    }

    public int[][] getBoard() {
        return board;
    }

    public void setBoard(int[][] board) {
        this.board = board;
    }

    public boolean isValid(int row, int col, int num) {
        for (int i = 0; i < SIZE; i++) {
            if (board[row][i] == num || board[i][col] == num ||
```

```
                    board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == num) {
                return false;
            }
        }
        return true;
    }
}
```

Visualizer.java

```
import javax.swing.*;
import javax.swing.border.Border;
import java.awt.*;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Visualizer implements Solver.SolverCallback {
    private JFrame frame;
    private JButton solveButton;
    private JButton clearButton;
    private JButton exitButton;
    private JTextField[] fields;
    private Solver solver;
    private SudokuClass sudoku;
    private JComboBox<String> difficultyDropdown;

    public Visualizer() {
        frame = new JFrame();
        frame.setPreferredSize(new Dimension(800, 700));

        JPanel sudokuPanel = new JPanel();
        sudokuPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        sudokuPanel.setLayout(new GridLayout(9, 9));

        fields = getClearedFields();

        for (int x = 0; x < 81; x++) {
            sudokuPanel.add(fields[x]);
        }

        JPanel controlPanel = new JPanel();
        controlPanel.setLayout(new BoxLayout(controlPanel, BoxLayout.Y_AXIS));
        controlPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        difficultyDropdown = new JComboBox<>(new String[]{"Easy", "Medium",
"Hard"});
```

```
        difficultyDropdown.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                loadSudokuGame(difficultyDropdown.getSelectedItem().toString());
            }
        });
        solveButton = new JButton("Solve");
        solveButton.addActionListener(e -> GUIToSudoku(fields));

        clearButton = new JButton("Clear Grid");
        clearButton.addActionListener(e -> clearGrid());

        exitButton = new JButton("Exit");
        exitButton.addActionListener(e -> frame.dispose());

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new FlowLayout());
        buttonPanel.add(solveButton);
        buttonPanel.add(clearButton);
        buttonPanel.add(exitButton);

        controlPanel.add(difficultyDropdown);
        controlPanel.add(Box.createRigidArea(new Dimension(0, 20)));
        controlPanel.add(createNumberPad());
        controlPanel.add(Box.createRigidArea(new Dimension(0, 20)));
        controlPanel.add(buttonPanel);

        frame.add(sudokuPanel, BorderLayout.CENTER);
        frame.add(controlPanel, BorderLayout.EAST);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setTitle("Sudoku Solver");
        frame.pack();
        frame.setVisible(true);
    }

    private void loadSudokuGame(String difficulty) {
       int[][] game = new int[9][9];
        if (difficulty.equals("Easy")) {
            game = new int[][] {
                {0, 0, 0, 0, 6, 4, 0, 0, 9},
                {6, 0, 0, 0, 7, 0, 0, 8, 4},
                {0, 0, 0, 1, 7, 9, 0, 3, 0},
                {9, 0, 1, 0, 7, 0, 0, 3, 0},
                {0, 0, 2, 0, 6, 0, 0, 9, 0},
                {0, 0, 0, 0, 0, 4, 0, 1, 7},
                {3, 0, 0, 0, 0, 2, 0, 0, 6},
                {0, 0, 0, 0, 0, 7, 3, 0, 1},
                {0, 4, 0, 0, 3, 1, 0, 0, 0}
            };
        } else if (difficulty.equals("Medium")) {
```

```
            game = new int[][] {
                {0, 0, 0, 2, 6, 0, 7, 0, 1},
                {6, 8, 0, 0, 7, 0, 0, 9, 0},
                {1, 9, 0, 0, 0, 4, 5, 0, 0},
                {8, 2, 0, 1, 0, 0, 0, 4, 0},
                {0, 0, 4, 6, 0, 2, 9, 0, 0},
                {0, 5, 0, 0, 0, 3, 0, 2, 8},
                {0, 0, 9, 3, 0, 0, 0, 7, 4},
                {0, 4, 0, 0, 5, 0, 0, 3, 6},
                {7, 0, 3, 0, 1, 8, 0, 0, 0}
            };
        } else if (difficulty.equals("Hard")) {
            game = new int[][] {
                {0, 2, 0, 6, 0, 8, 0, 0, 0},
                {5, 8, 0, 0, 0, 9, 7, 0, 0},
                {0, 0, 0, 0, 4, 0, 0, 0, 0},
                {3, 7, 0, 0, 0, 0, 5, 0, 0},
                {6, 0, 0, 0, 0, 0, 0, 0, 4},
                {0, 0, 8, 0, 0, 0, 0, 1, 3},
                {0, 0, 0, 0, 2, 0, 0, 0, 0},
                {0, 0, 9, 8, 0, 0, 0, 3, 6},
                {0, 0, 0, 3, 0, 6, 0, 9, 0}
            };
        }

        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                fields[i * 9 + j].setText(game[i][j] == 0 ? "" :
String.valueOf(game[i][j]));
                fields[i * 9 + j].setBackground(Color.WHITE);
                fields[i * 9 + j].setForeground(Color.BLACK);
                fields[i * 9 + j].setEditable(game[i][j] == 0);
            }
        }
    }

    private JPanel createNumberPad() {
        JPanel numberPad = new JPanel();
        numberPad.setLayout(new GridLayout(3, 3, 5, 5));
        numberPad.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        for (int i = 1; i <= 9; i++) {
            JButton button = new JButton(String.valueOf(i));
            button.setFont(new Font("Arial", Font.BOLD, 20));
            button.addActionListener(new NumberPadListener());
            numberPad.add(button);
        }

        return numberPad;
    }
```

```
    private class NumberPadListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JButton source = (JButton) e.getSource();
            String number = source.getText();
            for (JTextField field : fields) {
                if (field.isEditable() && field.hasFocus()) {
                    field.setText(number);
                    break;
                }
            }
        }
    }
    JTextField[] getClearedFields() {
        JTextField[] newFields = new JTextField[81];
        for (int x = 0; x < 81; x++) {
            newFields[x] = new JTextField();
            JTextField f = newFields[x];
            f.setHorizontalAlignment(JTextField.CENTER);
            f.setFont(new Font("Arial", Font.BOLD, 20));
            f.setBackground(Color.WHITE);
            f.setEditable(true);
            Border border = BorderFactory.createLineBorder(Color.BLACK, 1);
            f.setBorder(border);

            f.addKeyListener(new KeyAdapter() {
                public void keyPressed(KeyEvent ke) {
                    char pressedKey = ke.getKeyChar();
                    if (pressedKey == 8) {
                        f.setText("");
                        f.setBackground(Color.WHITE);
                    } else if (pressedKey >= '1' && pressedKey <= '9') {
                        f.setText("" + pressedKey);
                        f.setEditable(false);
                        f.setBackground(Color.GRAY);
                    } else {
                        f.setEditable(false);
                    }
                }
            });
            f.setEditable(true);
        }
        applyThickerBorders(newFields);
        return newFields;
    }
    void applyThickerBorders(JTextField[] fields) {
        for (int i = 0; i < 81; i++) {
            int row = i / 9;
            int col = i % 9;

            Border border = fields[i].getBorder();
            Border thickBorder = BorderFactory.createMatteBorder(
```

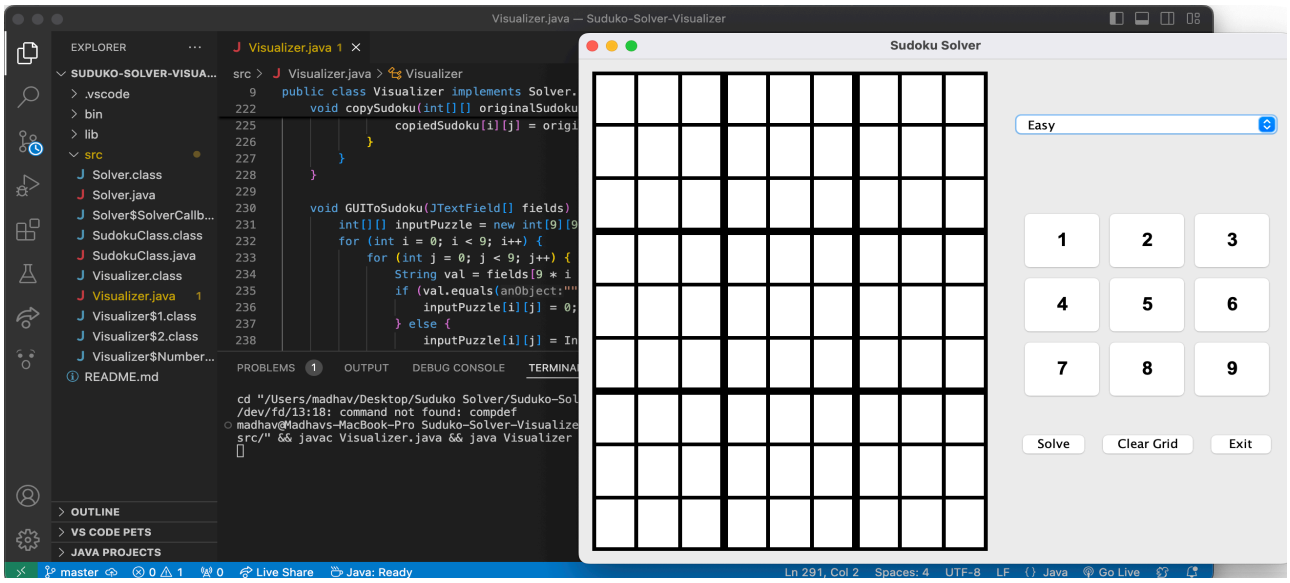
```
                row % 3 == 0 ? 3 : 1,
                col % 3 == 0 ? 3 : 1,
                (row + 1) % 3 == 0 ? 3 : 1,
                (col + 1) % 3 == 0 ? 3 : 1,
                Color.BLACK
            );
            fields[i].setBorder(BorderFactory.createCompoundBorder(thickBorder,
border));
        }
    }
    void populateFieldInteractive(JTextField f, int numberToUpdate, int initial) {
        f.setEditable(true);
        if (initial == 0) {
            f.setForeground(Color.BLUE);
        } else {
            f.setForeground(Color.BLACK);
        }
        f.setText("" + numberToUpdate);
        f.setEditable(false);
    }
    void SudokuToGUI(int[][] solution, int[][] initialSudoku) {
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                populateFieldInteractive(fields[(i * 9) + j], solution[i][j],
initialSudoku[i][j]);
            }
        }
    }
    void copySudoku(int[][] originalSudoku, int[][] copiedSudoku) {
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                copiedSudoku[i][j] = originalSudoku[i][j];
            }
        }
    }
    void GUIToSudoku(JTextField[] fields) {
        int[][] inputPuzzle = new int[9][9];
        for (int i = 0; i < 9; i++) {
            for (int j = 0; j < 9; j++) {
                String val = fields[9 * i + j].getText();
                if (val.equals("")) {
                    inputPuzzle[i][j] = 0;
                } else {
                    inputPuzzle[i][j] = Integer.parseInt(val);
                }
            }
        }

        sudoku = new SudokuClass(inputPuzzle);
        solver = new Solver(inputPuzzle, this);
```

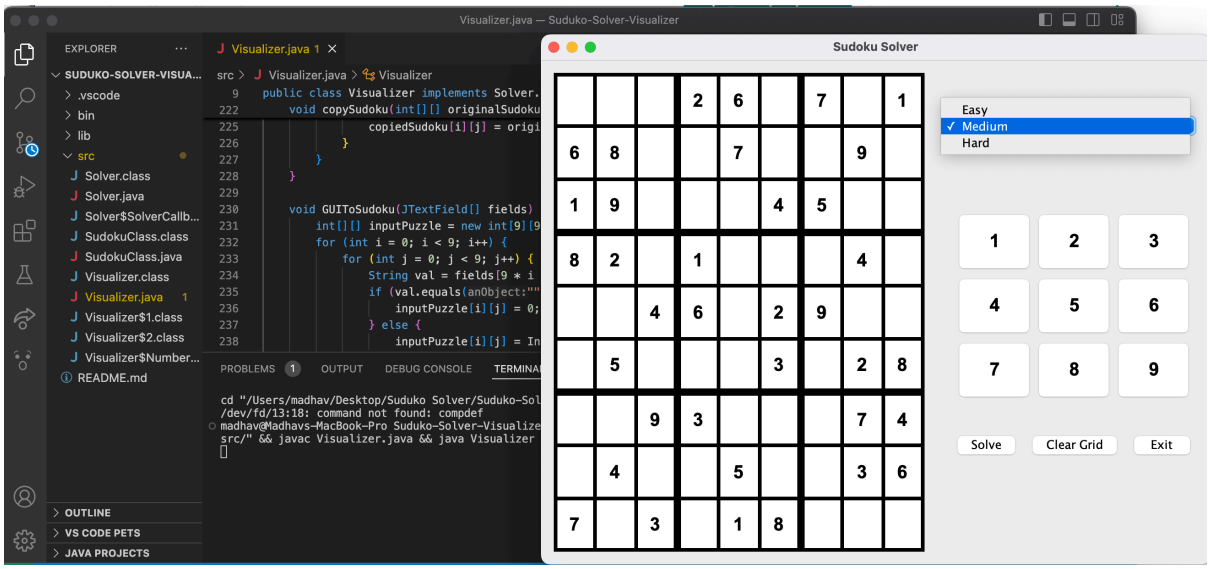
```
        new Thread(() -> {
            if (solver.solve()) {
                int[][] solution = solver.getBoard();
                SwingUtilities.invokeLater(() -> {
                    SudokuToGUI(solution, inputPuzzle);
                    JOptionPane.showMessageDialog(frame, "Sudoku Solved!",
"Success", JOptionPane.INFORMATION_MESSAGE);
                });
            } else {
                SwingUtilities.invokeLater(() -> {
                    JOptionPane.showMessageDialog(frame, "No solution exists for
the given puzzle", "Error", JOptionPane.ERROR_MESSAGE);
                });
            }
        }).start();
    }
    private void clearGrid() {
        for (int i = 0; i < 81; i++) {
            fields[i].setText("");
            fields[i].setBackground(Color.WHITE);
            fields[i].setForeground(Color.BLACK);
            fields[i].setEditable(true);
        }
    }
    @Override
    public void updateCell(int row, int col, int value) {
        SwingUtilities.invokeLater(() -> {
            JTextField field = fields[row * 9 + col];
            field.setText(value == 0 ? "" : String.valueOf(value));
            field.setForeground(value == 0 ? Color.BLACK : Color.BLUE);
        });
    }
    @Override
    public void delay(int milliseconds) {
        try {
            Thread.sleep(milliseconds);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(Visualizer::new);
    }
}
```

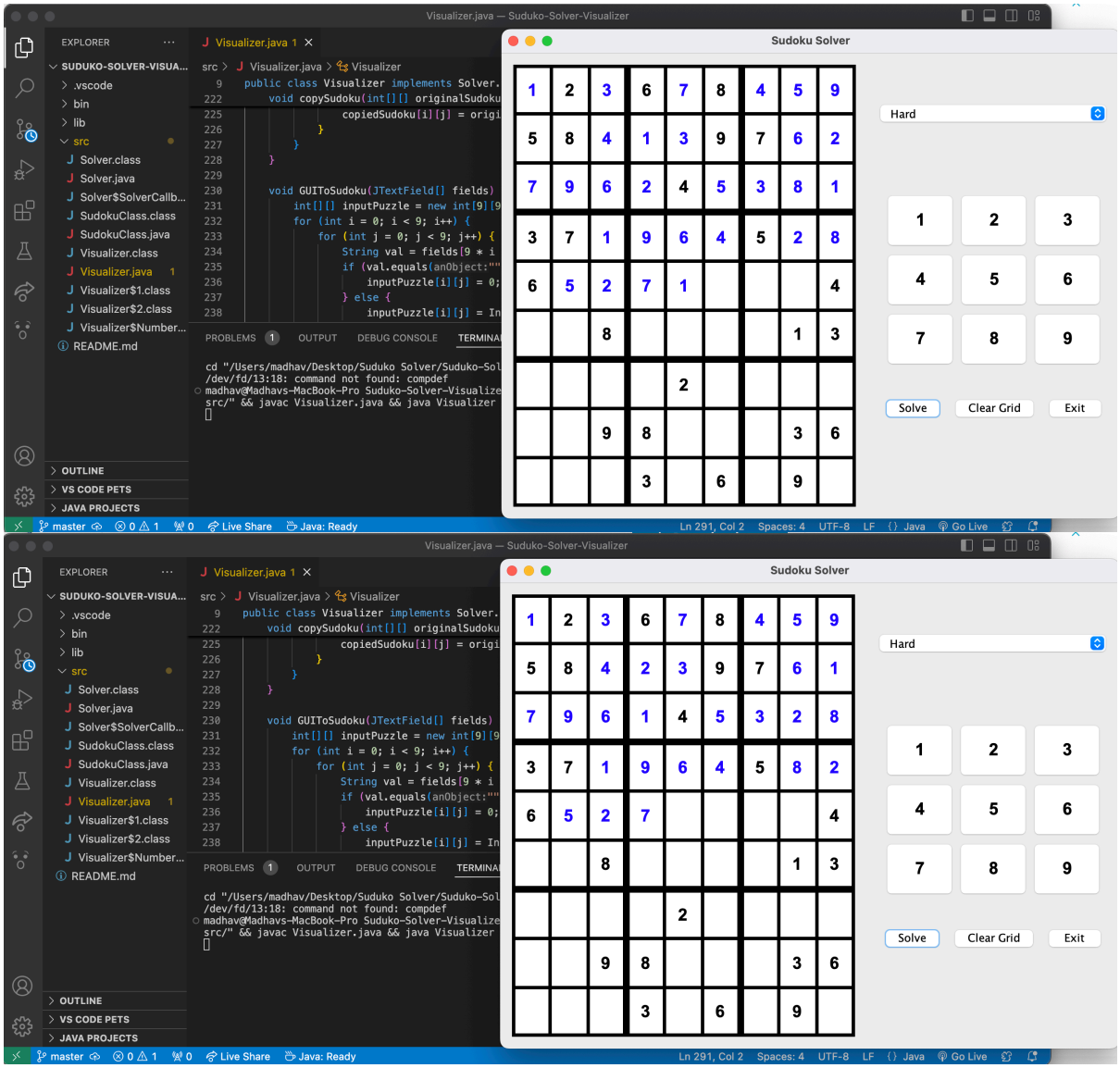
5.2 Output Screenshots



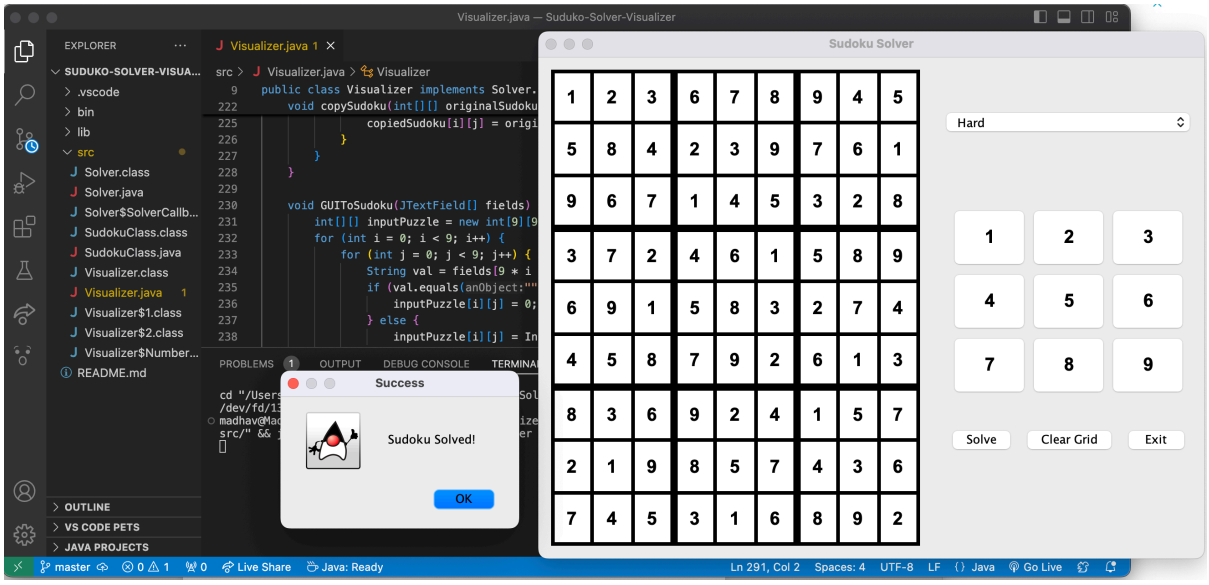
Empty Grid for Manual Inputs



3 Different modes with presaved games.



Backtracking and solving the games and showing each and every thing



Final Solved Suduko

Conclusion

This study has demonstrated that the pencil-and-paper algorithm implemented in the Sudoku solver project is a feasible and efficient method for solving Sudoku puzzles. The algorithm, which simulates human solving techniques, not only finds solutions faster but also handles puzzles of varying difficulty levels effectively. The testing results have shown that the pencil-and-paper algorithm can solve puzzles within a short period (less than one second), outperforming the backtracking algorithm in terms of computing time.

The backtracking algorithm, which is used as a supplementary method in the project, ensures that a solution is found for any Sudoku puzzle by exhaustively checking all possible solutions. However, this method is less efficient because it does not incorporate intelligent strategies and instead relies on brute force. The backtracking algorithm's main advantage is its ability to guarantee a solution, making it a reliable method when the pencil-and-paper algorithm alone is insufficient. Despite its slower performance, the backtracking method provides a robust fallback, ensuring that no puzzle remains unsolved.

References

1. Wikipedia , Web site: http://en.wikipedia.org/wiki/Sudoku
2. Home Of Logic Puzzles ,Web Page : http://www.conceptispuzzles.com/index.aspx?uri=puzzle/sudoku/classic
3. J.F. Crook, A pencil and paper algorithm for solving Sudoku Puzzles
   Webpage: http://www.ams.org/notices/200904/tx090400460p.pdf
4. Sudoku solver using brute force , https://github.com/olav/JavaSudokuSolver
5. The puzzle generator, visited in March 2013, websudoku.com