

A Project Report

on

**Sudoku Solver**

**Course Title**: Design and Analysis of Algorithm

**Course Code**: CS3001

**Section**: CSE-44

**Submitted By**

| **Name** | **Roll no** |
| --- | --- |
| Aritra Bag | 22052194 |
| Sraman Dutta Bar | 22052336 |
| Sayan Hazra | 22053277 |

**Submitted to**

Dr. Santwana Sagnika

**Contents**

1. Introduction 3
   1. Overview……………………………………………………………3
   2. Problem……………………………………………………………..3
2. Data Structure 4
   1. 2-D array……………………………………………………………4
   2. 1-D array……………………………………………………………4
   3. Backtracking with Recursion……………………………………….4
   4. Methods for validity checking………………………………………4
3. Algorithm 5
   1. Algorithm 1…………………………………………………………5
   2. Algorithm 2…………………………………………………………5
4. Code 7
5. Result 12
6. Conclusion 14
7. Reference 15

# **INTRODUCTION**

**1.1 Overview**

The Sudoku puzzle is a game in which players have to fill a 9x9 grid with numbers from 1 to 9. This game aims to fill the column, row, and 3x3 subgrids without repeating the numbers. Players give inputs to some boxes and the algorithm finds a compatible solution to fill all the boxes of that puzzle using backtracking. Sudoku puzzles offer an enjoyable and challenging experience that requires strategic thinking and pattern recognition.

This project aims to develop an automated solver for validated sudoku puzzles using Java and a Backtracking approach. It solves the problem instantly and effortlessly.

**1.2**  **Problem Statement**

The Sudoku Puzzle Solver project aims to create algorithmic approaches to solving Sudoku puzzles. The game is based on filling the row, column, and 3x3 subgrids without repeating numbers between 1 and 9. We have used various concepts such as recursion, backtracking, and Graphical user interface (GUI) and diligently incorporated various concepts of Object Oriented Programming (OOP) into this project.

# **DATA STRUCTURES**

This project allows various Data Structures to solve the puzzle, including:

* 1. ***2-D Array:*** To represent the 9x9 matrix of the Sudoku puzzle. Each cell stores 1 to 9 and is blank for empty cells. It gives quick access to any cells, rows, and columns.
  2. ***1-D Array:*** The getBox method returns a 1D array that contains the starting coordinates of a 3x3 subgrid. It helps to locate the starting point of each 3x3 grid.
  3. ***Backtracking with Recursion:*** When we put a number in a particular cell, check the repetition of the number in the 9x9 grid row and column to the corresponding cell, and the corresponding 3x3 subgrid in which the specific cell is obtained. If repetition arises we backtrack to the specific number and change it.
  4. ***Methods for validity checking:*** Methods like checkRow, checkColumn, and checkBox checks the values in the 2D array to ensure that no duplicates occur in the specific rows, columns, or boxes.

In most cases, a Sudoku solver is a combination of the 2D matrix, backtracking, and sets to track constraints. This combination made the Sudoku solver efficient, direct, and organized.

# **ALGORITHM**

The Sudoku-solving algorithm is in charge of solving a given Sudoku puzzle. It enables an efficient approach that systematically discovers all the possible number (1 to 9) placements in the puzzle's empty cells while maintaining that all the Sudoku rules are satisfied. More specific details about the algorithm are explained here:-

* **Backtracking:** The Sudoku-solving algorithm follows a backtracking technique, which is fundamentally a recursive search process that discovers all the possible solutions for placing numbers in empty cells. It also backtracks when an error occurs. This technique guarantees the correct solution by systematically discovering the correct number for all the cells in the puzzle.
* **Recursion:** After placing a number in the designated cell, the algorithm finds the next empty cell and repeats the process by placing another valid number for that cell. This process of recursion keeps on going until all cells are filled with numbers and all the Sudoku rules are not broken, thus finding a valid solution.
* **Finding Empty Cell:** The algorithm starts by finding empty cells in the puzzle to fill with numbers. It starts at the very first cell (0,0) and checks whether it is filled with any number from 1 to 9 or not.
* **Digit Placement:** After finding an empty cell, the algorithm tries to place a number (1 to 9) that follows the Sudoku rules. It inspects if the chosen number is not repeated within the cell’s row or column or in the 3X3 subgrid. If any error occurs, the algorithm backtracks recursively and tries a different number to get the solution for that puzzle.

Algorithm 1: Finding Empty cells and placing the number

/\* creates a boolean recursive function with the parameters board, row, and col \*/

if(row==board.length){

return true, //ensures all the cells are filled

int nrow=0,ncol=0;

if(col != board.length-1) { /\**checks whether we are in a column other than the last column* \*/

nrow= row, ncol= col+1;}

else{

nrow= row+1,ncol=0;}

if(board[row][column] !=0{

*// recursively calls the same function*

return true;}

else{

for( int i=1; i<=9; i++) {

//checks the number whether it is valid or not

board[row][col]= i;  *//if it gets return true, place the number*}

board[row][col]=0; */\* if it gets return false, the algorithm backtracks \*/*

}

Algorithm 2: Checking if the number is valid or not

*/\* creates a boolean function with the parameters board, row, col and number \*/*

for( int i=0;i<board.length; i++){

if(board[i][col]==number){ */\*checks whether number is present in the corresponding row \*/*

return false;}

if(board[row][i]==number){  */\*checks whether number is present in the corresponding column \*/*

return false;}

*//checking whether the number is in the corresponding 3X3 subgrid*

int sr=(row/3)\*3, sc=(col/3)\*3;

for (int i=sr; i<sr+3; i++) {

for (int j=sc; j<sc+3; j++) {

if( board[i][j]==number){

return false;}

}

}

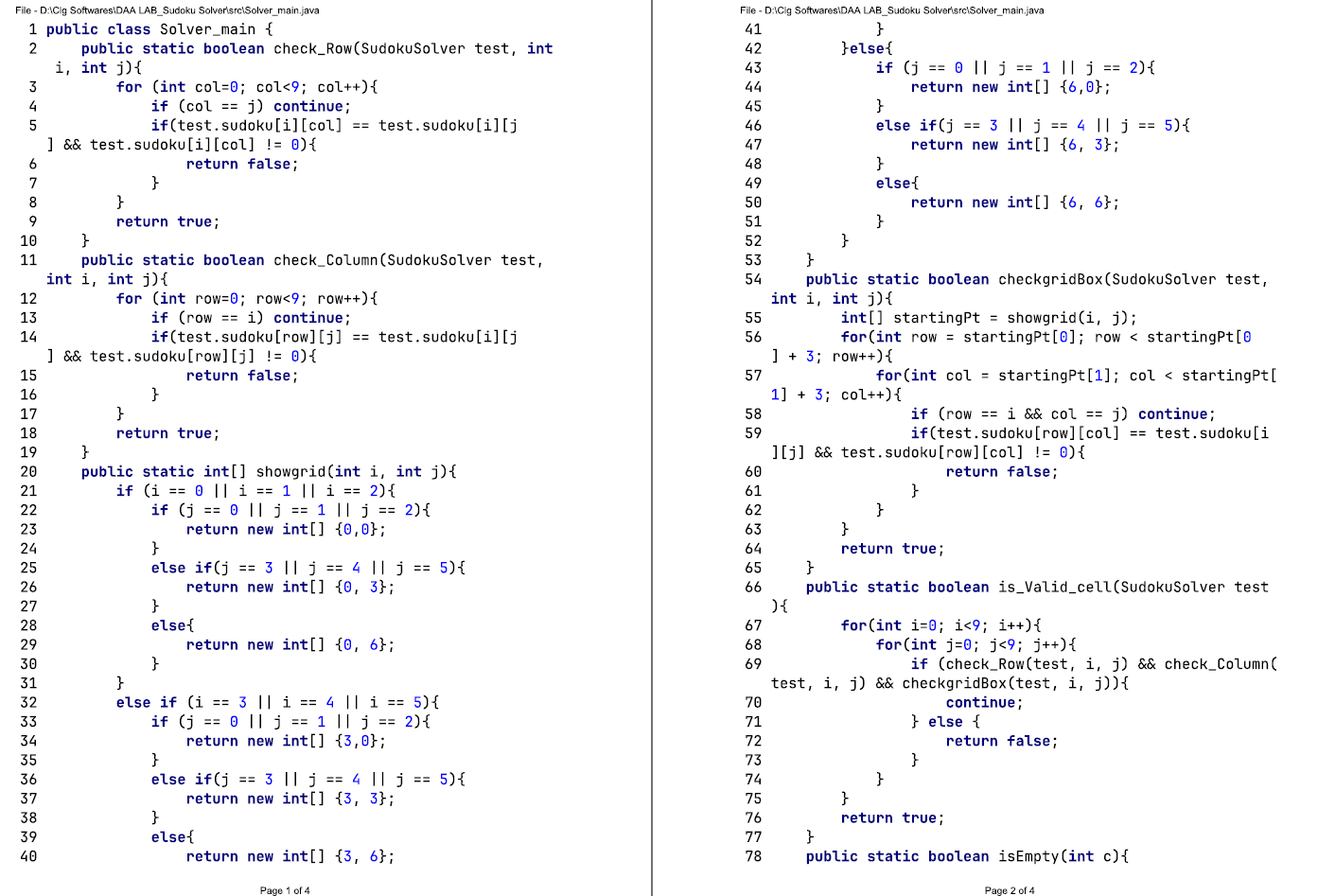
return true;

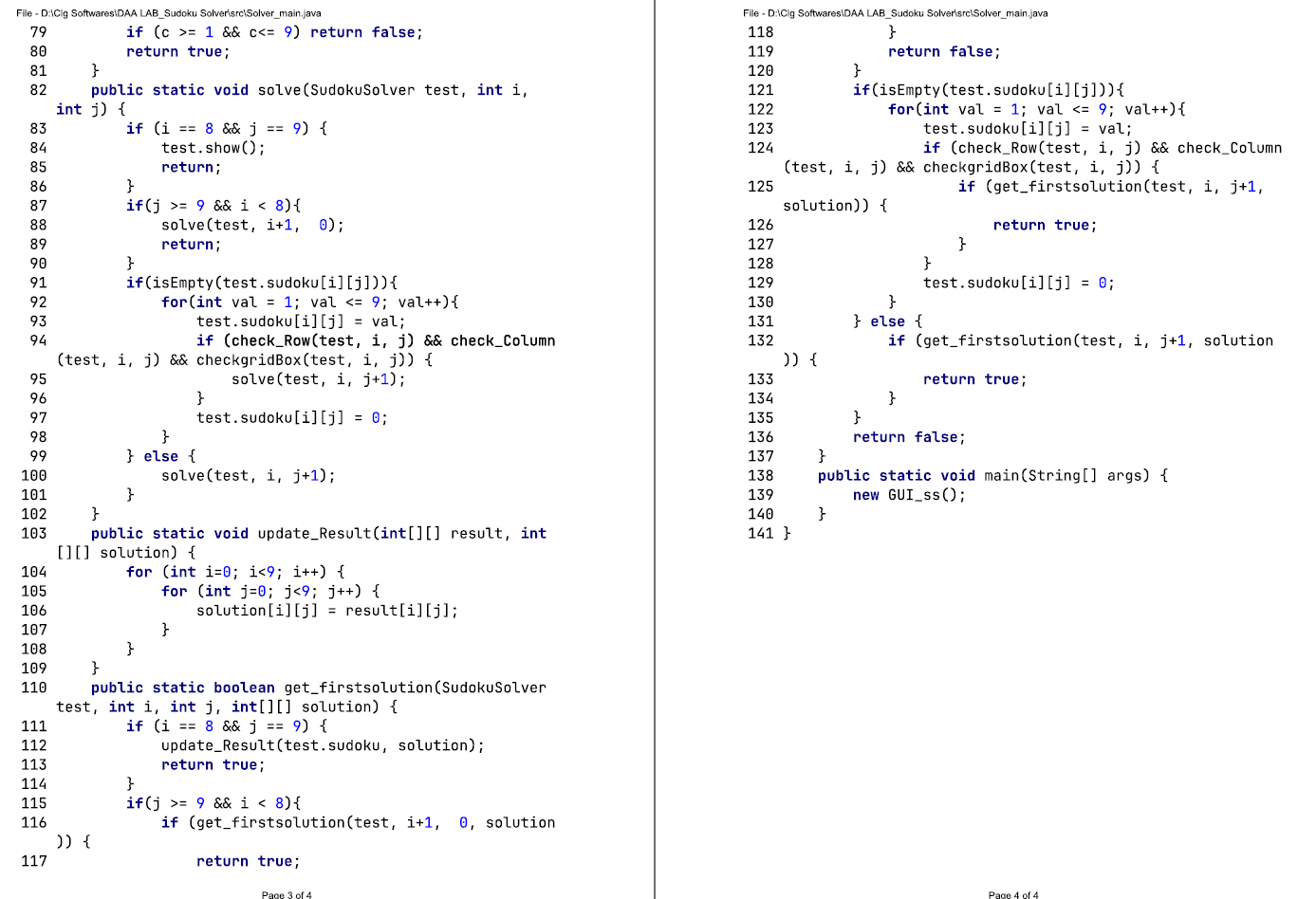
}

# **CODE**

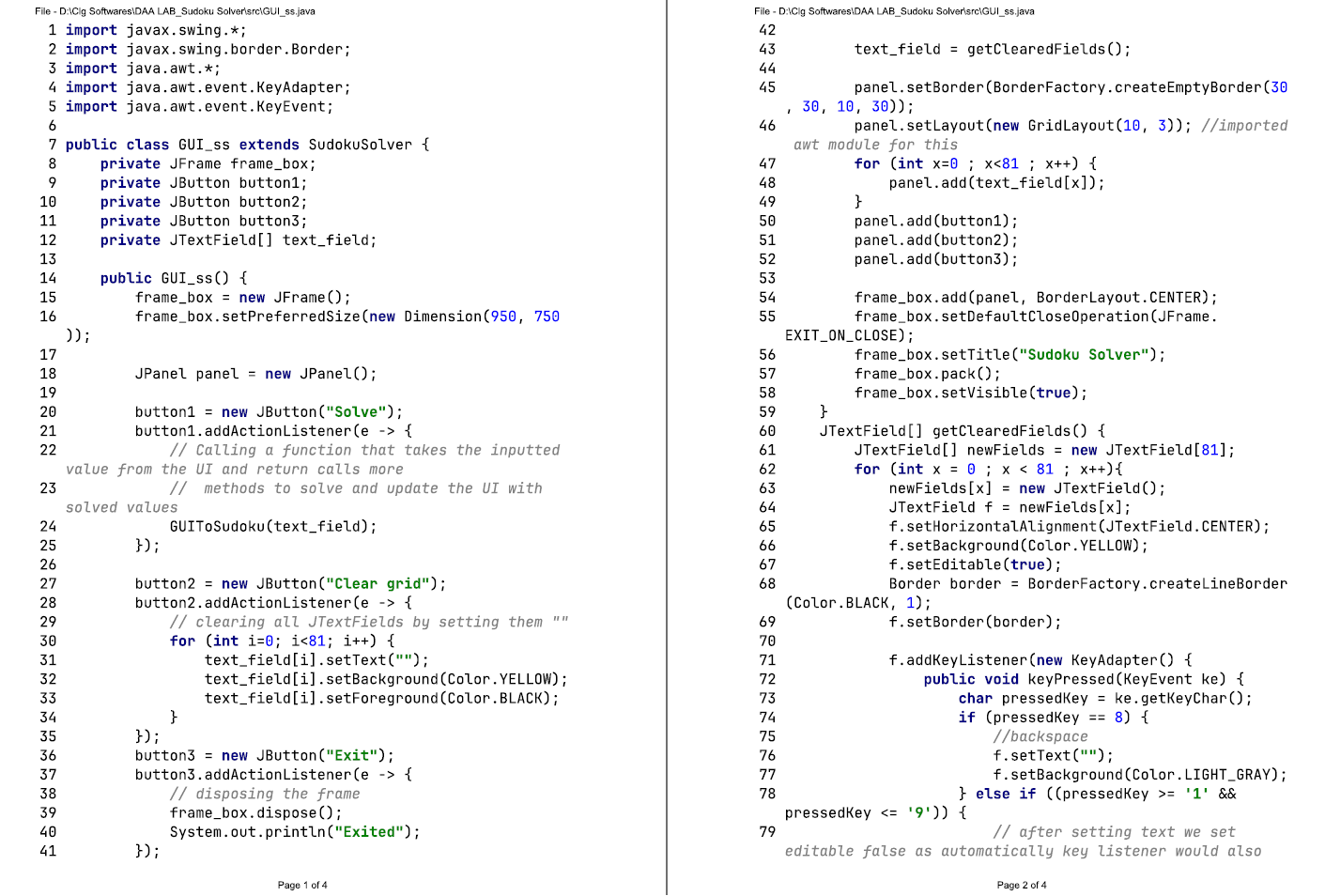
We have created three different segments of code which are SudokoClass, Solver, and GUI\_ss.

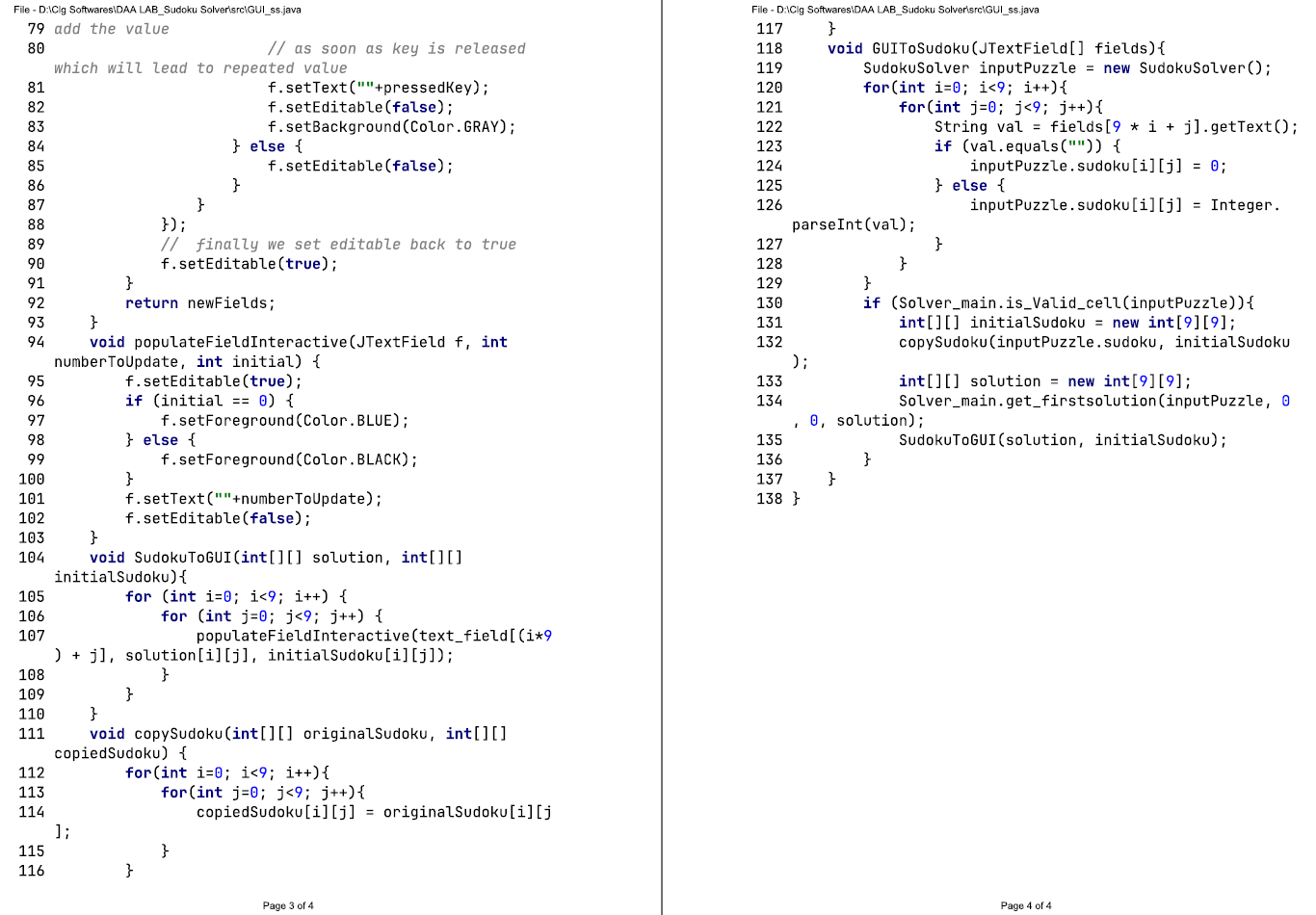
* Code for Solver Class



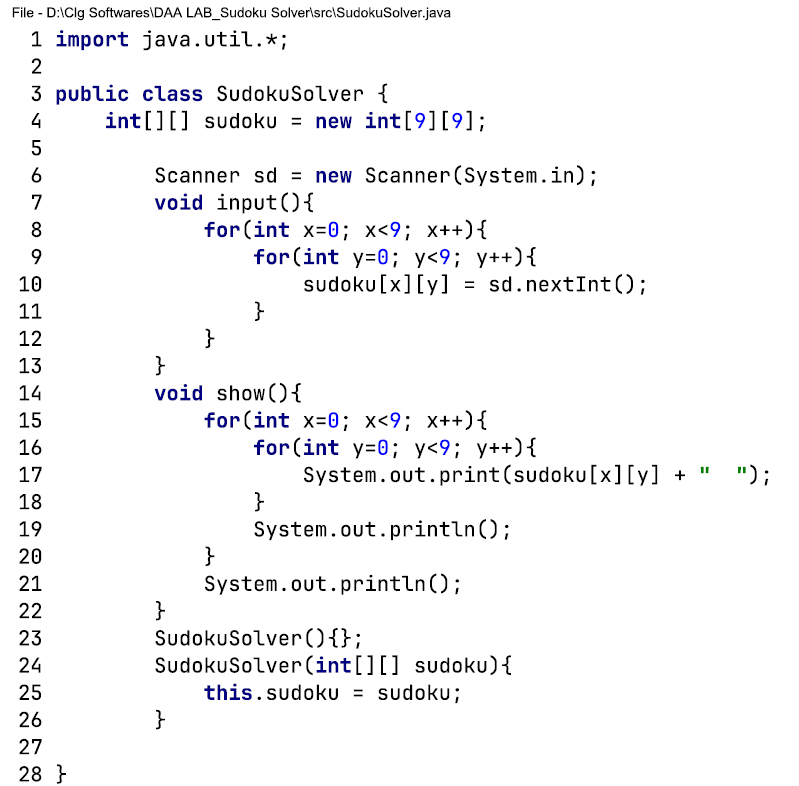


* Code for GUI:





* Code for Sudoku Solver



# **RESULT**

In our Sudoku solver, we have successfully employed a backtracking algorithm, recursion, etc which gave us efficient results for the puzzles provided as input. To evaluate the performance of the solver we tested it against an example and gave us the expected correct output. Below are some screenshots showcasing the working of the GUI and the result after the algorithm is done solving the puzzle.

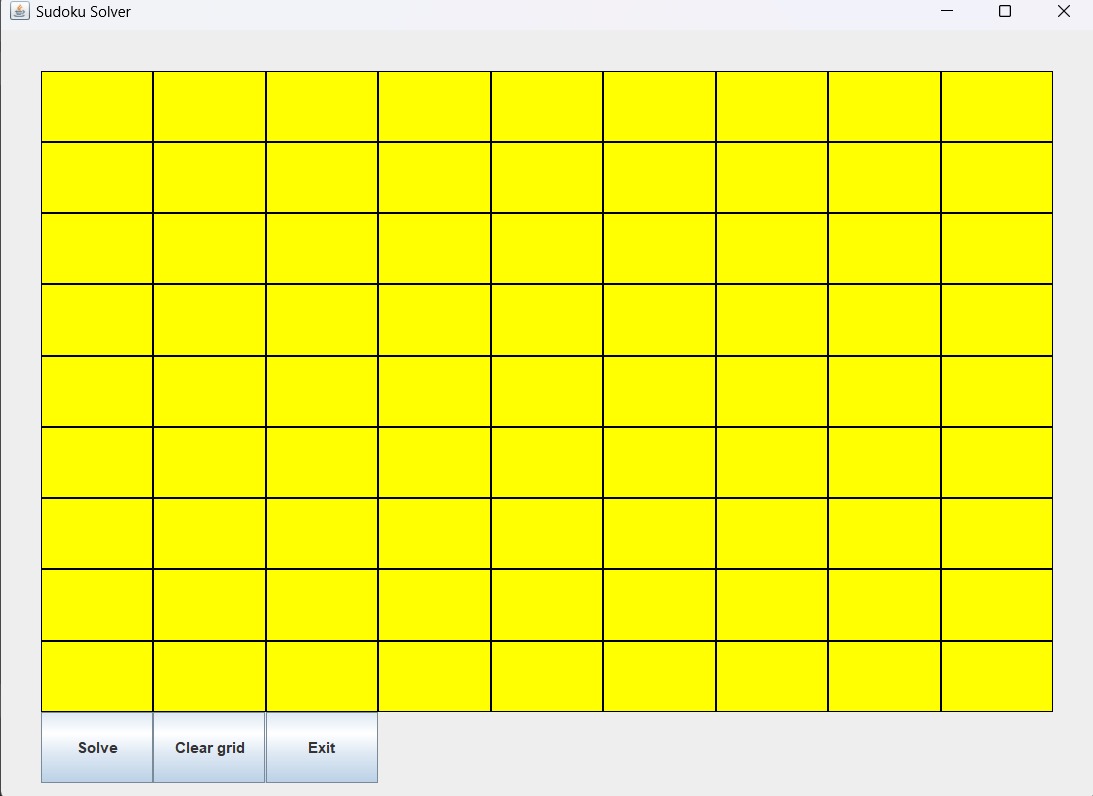


Figure 1.1: 9x9 Sudoku board

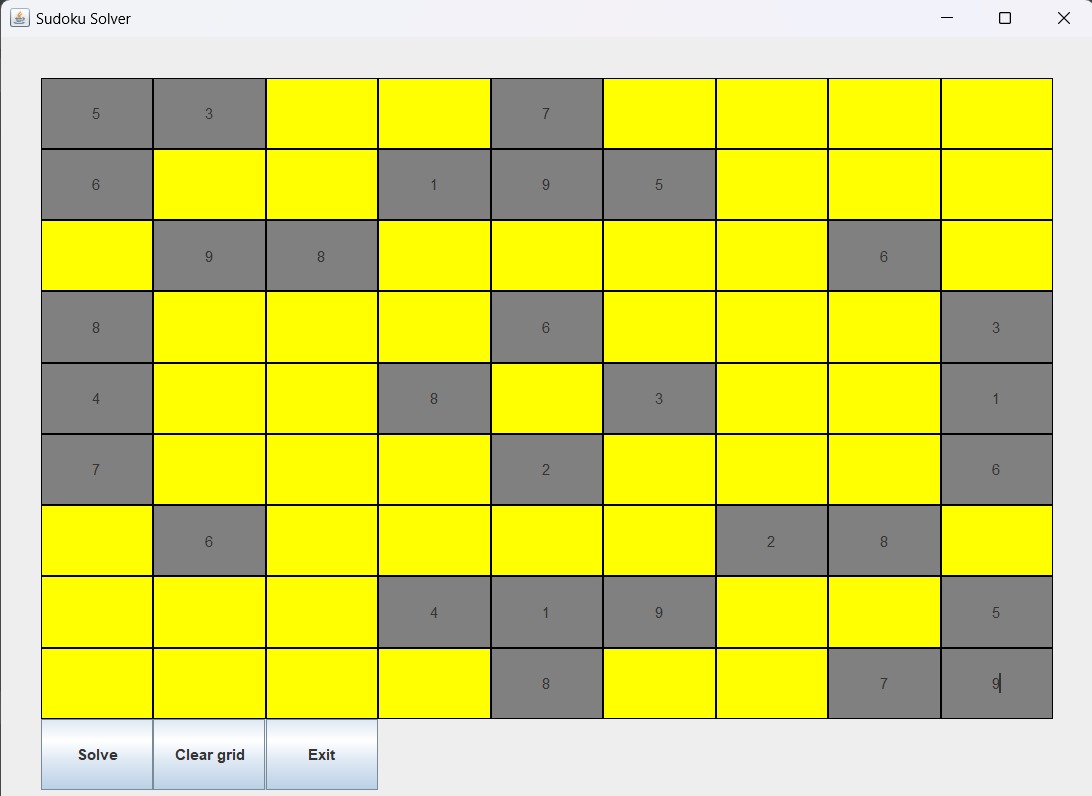


Figure 1.2: Take inputs by user



Figure 1.3: Final output after solving

# **CONCLUSION**

This project provided us the opportunity to explore various aspects of Java and helped us build a strong understanding of concepts such as GUI(Graphical User Interface), OOP(Object Oriented Programming), backtracking, and recursion and simultaneously implement them throughout the project. The GUI provides us with a user-friendly and visually pleasing interface which makes it easy for the users to get a seamless experience while using the algorithm and thus enhances user experience. Additionally, we were able to showcase the real-life applications of recursion and backtracking algorithms through our project.

Though the solver can generate the results efficiently there is always scope for future enhancement of efficiency in terms of time and space complexity with the help of more advanced algorithms and data structures

Overall this project laid a solid foundation for understanding and implementing different steps of Sudoku solving technique in a structured manner, thus showcasing the power of algorithm-based problem solving in solving this real-life problem of Sudoku.

# **REFERANCE**

* <https://www.youtube.com/watch?v=tRj4VlVTat8>
* <https://www.geeksforgeeks.org/sudoku-backtracking-7/>
* <https://sudoku.com/how-to-play/sudoku-rules-for-complete-beginners/>