

**Department of Electronics & Telecommunication Engineering Data Structures & Algorithms Lab (DJ19ECSBL1)**

**Data Structures & Algorithms (DJS22EL505) Mini Project report**

**Academic Year 2024-25**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **SAP ID** | **Name of student** |
| **1.** | **60002220163** | **Siddhant Sanghoi** |
| **2.** | **60002220240** | **Vatsal Maru** |

Aim: To implement a Sudoku solver using multi-dimensional arrays

Programming Language: **C & Java**

Theory:

Sudoku is a logic-based puzzle requiring each row, column, and 3x3 subgrid in a 9x9 grid to contain the numbers 1 to 9 exactly once. Solving it involves balancing three main constraints, as each placement affects allowable numbers in related cells across the grid. The process hinges on constraint propagation every number placed restricts the options for other cells in its row, column, and subgrid, creating a chain reaction that simplifies the puzzle step-by-step. Solvers rely on logical deduction to strategically place numbers, beginning with the most constrained cells and using techniques to reduce possible placements, ultimately revealing the unique solution through logical inference rather than guesswork. This combination of strict rules and logical flexibility is what makes Sudoku engaging and mentally stimulating.

In the C program for solving Sudoku, a 9x9 integer grid represents the puzzle, with 0 indicating empty cells. The solution employs a recursive backtracking approach, where each cell is filled sequentially by testing numbers 1 through 9 and checking constraints with the issafe function. This function ensures no duplicates in the current row, column, or 3x3 subgrid. If a number is safe, it is placed, and the algorithm proceeds to the next cell. If a dead-end is reached, it backtracks by resetting the cell to 0 and testing the next number. This method efficiently explores potential solutions, filling the grid only when a valid configuration is found.

In the Java program, the approach is similar but implemented with a graphical interface using Swing. A JTextField grid lets users input values, and a "Solve" button triggers the solution. The algorithm reads the grid's state, applies a backtracking solution, and updates the display with solved values if a solution is found. The solveSudoku method mirrors the C logic by recursively filling cells and using isSafe to check for duplicates within rows, columns, and subgrids. Through this interface, the Java program provides an interactive way to solve Sudoku, making the solution process visible and accessible within a user-friendly GUI.

Code:

[**\\Code**](file:///\\Code) **in C**

#include <stdio.h>

#include <stdlib.h>

int grid[9][9];

int n=9;

void display(int grid[9][9],int n)

{

int i,j;

printf("\n The sudoku is: \n\n");

for(i=0;i<n;i++)

{

if(i%3 == 0 ) printf("\n");

for(j=0;j<n;j++)

{

if(j%3 == 0 && j != 0)

{

printf(" %d",grid[i][j]);

}

else

{

printf(" %d",grid[i][j]);

}

printf("");

}

printf(" \n");

}

}

int issafe(int grid[9][9],int row,int col,int num)

{

for(int i=0;i<n;i++)

{

if(grid[row][i] == num || grid[i][col] == num)

return 0;

}

int srow = row - row%3;

int scol = col - col%3;

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

if (grid[i + srow][j + scol] == num)

{

return 0;

}

}

}

return 1;

}

int solve(int grid [9][9], int r,int c)

{

if (r == n)

return 1;

if (c == n) {

return solve(grid, r + 1, 0);

}

if (grid[r][c] > 0)

return solve(grid, r, c + 1);

for (int num = 1; num <= n; num++)

{

if (issafe(grid, r, c, num)==1)

{

grid[r][c] = num;

if (solve(grid, r, c + 1)==1)

return 1;

}

grid[r][c] = 0;

}

return 0;

}

void input(int grid[9][9])

{

}

int main()

{

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

display(grid,n);

if (solve(grid, 0, 0)==1)

display(grid,n);

else

printf("No solution exists");

int grid1[9][9] = 0;

input(grid1);

display(grid1,n);

if (solve(grid1, 0, 0)==1)

display(grid1,n);

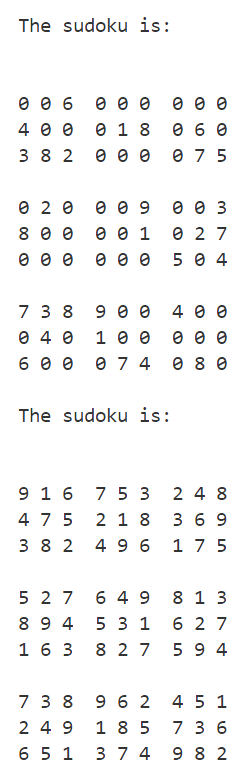
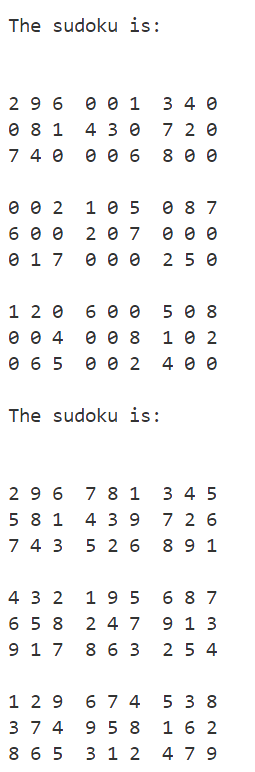
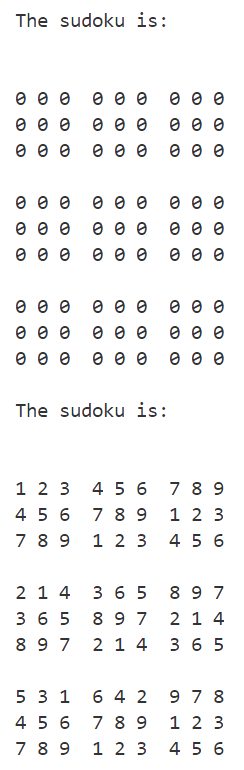
else

printf("No solution exists");

return 0;

}

**OUTPUT:**

****

**//Code in Java**

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class SudokuSolverGUI extends JFrame {

private static final int SIZE = 9;

private JTextField[][] cells = new JTextField[SIZE][SIZE];

private JButton solveButton;

public SudokuSolverGUI() {

setTitle("Sudoku Solver");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(550, 600);

setLayout(new BorderLayout());

// Create the Sudoku grid panel

JPanel gridPanel = new JPanel(new GridLayout(SIZE, SIZE));

gridPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

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));

// Set background color for alternating 3x3 subgrids

boolean isShaded = ((row / 3) + (col / 3)) % 2 == 0;

cells[row][col].setBackground(isShaded ? new Color(220, 220, 220) : Color.WHITE);

// Thicker borders to separate 3x3 subgrids

int top = row % 3 == 0 ? 4 : 1;

int left = col % 3 == 0 ? 4 : 1;

int bottom = row == SIZE - 1 ? 4 : 1;

int right = col == SIZE - 1 ? 4 : 1;

cells[row][col].setBorder(BorderFactory.createMatteBorder(top, left, bottom, right, Color.BLACK));

gridPanel.add(cells[row][col]);

}

}

// Solve button

solveButton = new JButton("Solve Sudoku");

solveButton.setFont(new Font("Arial", Font.BOLD, 16));

solveButton.addActionListener(new SolveButtonListener());

add(gridPanel, BorderLayout.CENTER);

add(solveButton, BorderLayout.SOUTH);

}

private class SolveButtonListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

int[][] grid = new int[SIZE][SIZE];

// Populate the grid with values from the text fields

for (int row = 0; row < SIZE; row++) {

for (int col = 0; col < SIZE; col++) {

String text = cells[row][col].getText();

grid[row][col] = text.isEmpty() ? 0 : Integer.parseInt(text);

}

}

// Solve the Sudoku

if (solveSudoku(grid)) {

// Update the grid with the solved values

for (int row = 0; row < SIZE; row++) {

for (int col = 0; col < SIZE; col++) {

cells[row][col].setText(String.valueOf(grid[row][col]));

}

}

} else {

JOptionPane.showMessageDialog(null, "No solution exists for the provided Sudoku puzzle.", "Error", JOptionPane.ERROR\_MESSAGE);

}

}

}

// Helper function to solve Sudoku using backtracking

private boolean solveSudoku(int[][] grid) {

for (int row = 0; row < SIZE; row++) {

for (int col = 0; col < SIZE; col++) {

if (grid[row][col] == 0) {

for (int num = 1; num <= SIZE; num++) {

if (isSafe(grid, row, col, num)) {

grid[row][col] = num;

if (solveSudoku(grid)) return true;

grid[row][col] = 0;

}

}

return false;

}

}

}

return true;

}

// Helper function to check if a number can be placed in a cell

private boolean isSafe(int[][] grid, int row, int col, int num) {

for (int i = 0; i < SIZE; i++) {

if (grid[row][i] == num || grid[i][col] == num) return false;

}

int startRow = row - row % 3;

int startCol = col - col % 3;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (grid[i + startRow][j + startCol] == num) return false;

}

}

return true;

}

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> {

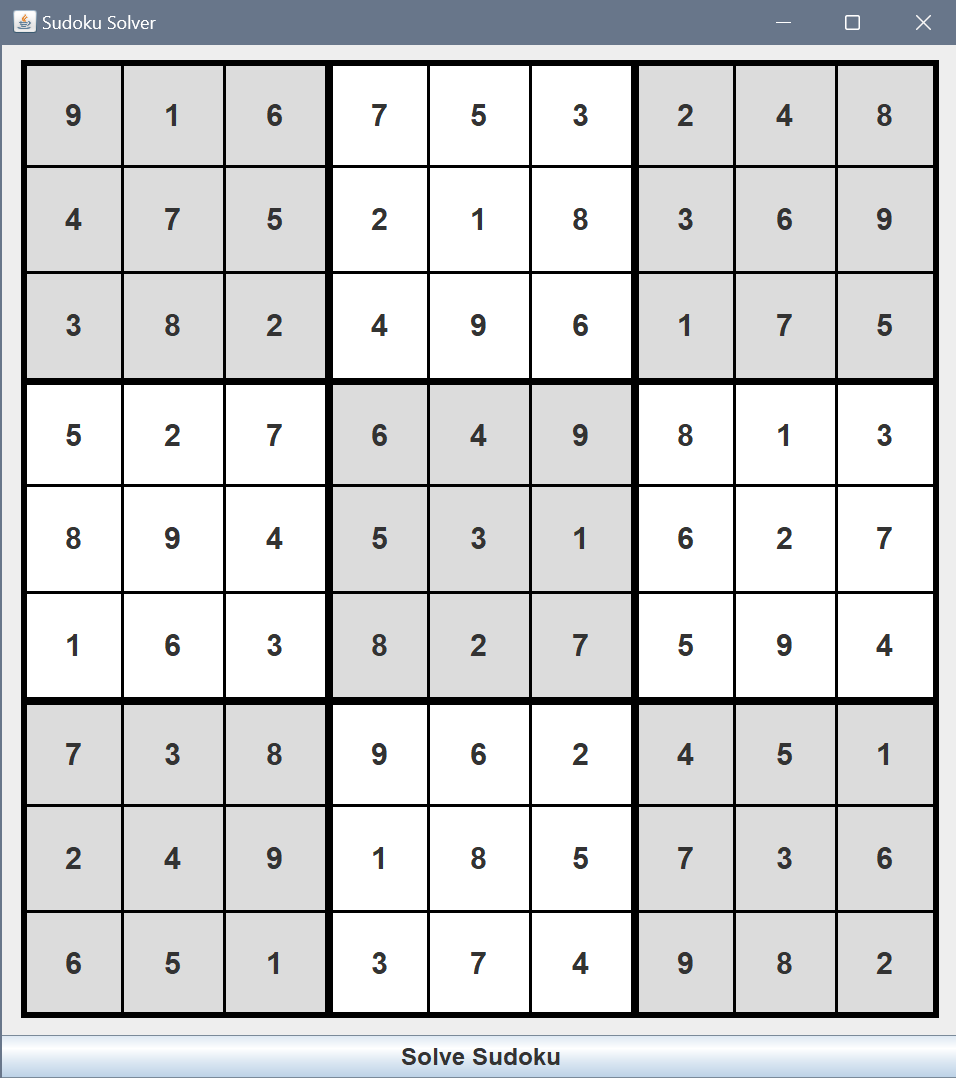
SudokuSolverGUI frame = new SudokuSolverGUI();

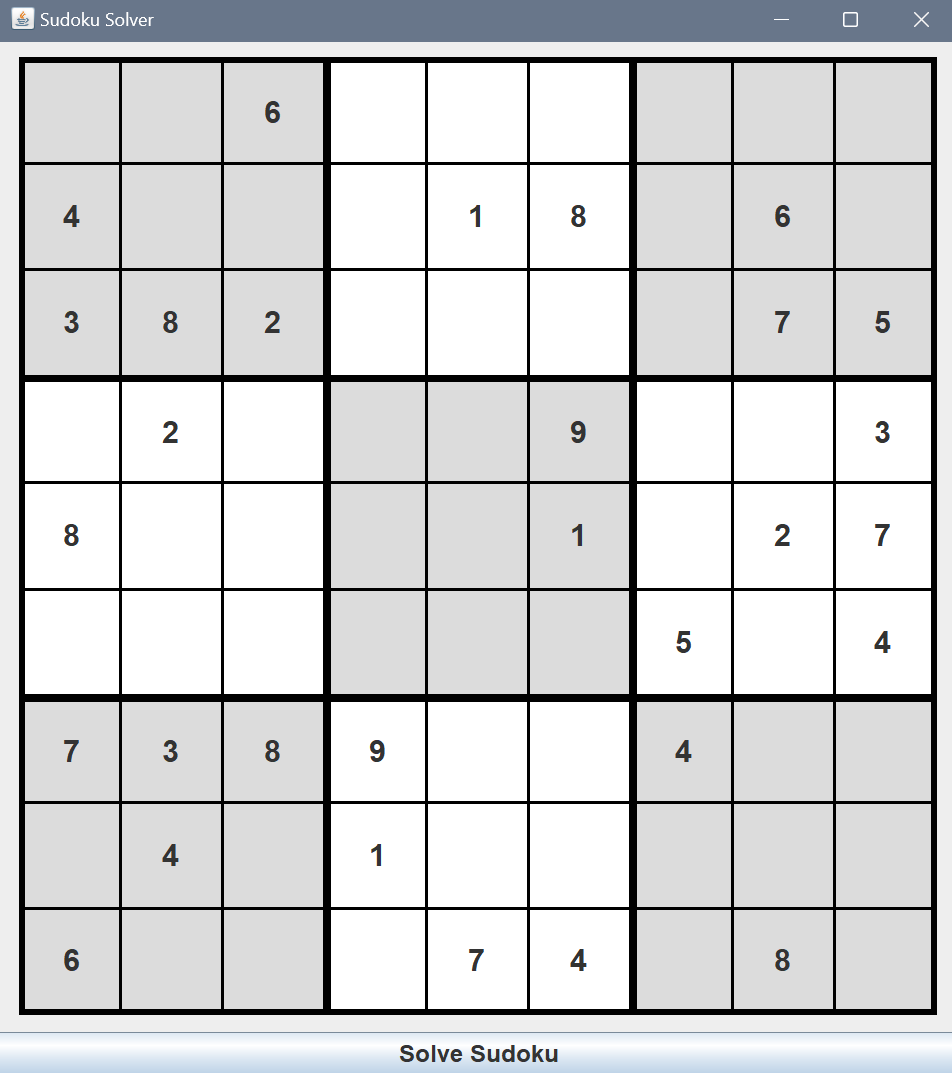
frame.setVisible(true);

});

}

}

**OUTPUT:**

****

Result and Conclusion: In summary, solving Sudoku using both C and Java implementations demonstrates the effectiveness of backtracking algorithms paired with constraint-based logic. The C program efficiently uses recursion and constraint checks to solve the puzzle in a simple console format, while the Java version adds a user-friendly GUI that makes the process interactive and visually accessible. Both implementations rely on constraint propagation and logical deduction to systematically fill the grid, showcasing how algorithmic approaches can be adapted for different interfaces while preserving their core problem-solving capabilities. This comparison illustrates the versatility of backtracking for complex puzzles like Sudoku, balancing computational efficiency with an engaging user experience.