**Project Report:**

**N Queen Problem Visualizer Using Java**

**Submitted by: Deepanshu Singh**

**Abstract**

This project report details the development of an interactive N Queen Problem Visualizer implemented in Java using Swing. The primary goal was to create a tool that not only solves the N Queen problem but also provides a graphical representation of the solving process. This project aims to enhance understanding of backtracking algorithms through real-time visualization.

**Introduction**

The N Queen problem is a classic combinatorial problem where N queens must be placed on an N×N chessboard such that no two queens threaten each other. This project focuses on developing a graphical interface using Java Swing to illustrate the placement and backtracking of queens as they are recursively placed and removed to find a valid solution.

**Objectives**

1. **Algorithm Implementation:** Implement the backtracking algorithm to solve the N Queen problem efficiently.
2. **Graphical User Interface:** Develop a Java Swing application to visualize the chessboard and dynamically display queen placements and backtracking steps.
3. **Educational Tool:** Provide a tool that aids in understanding the mechanics of the backtracking algorithm through visual representation.

**Methodology**

1. **Algorithm Selection:** The backtracking algorithm was chosen for its ability to systematically explore potential solutions while efficiently backtracking when conflicts arise. It recursively attempts to place queens on the board, checking each placement's validity against established constraints.
2. **Graphical User Interface Design:** Java Swing was utilized to create a 4×4 grid representing the chessboard. Each cell on the grid was represented by a JLabel, allowing for easy customization of visual properties such as color to denote queen placements and backtracking steps.
3. **Visualization Approach:** Queens' placements and backtracking steps were visually represented using colour changes on the JLabels. This approach provided a clear and intuitive representation of the solving process, with each step animated to aid comprehension.

**Implementation**

Here's the Java code for the N Queen Problem Visualizer, explained step by step:

import javax.swing.\*;

import java.awt.\*;

public class NQueen {

final static int N = 4; // Define the size of the chessboard

static JLabel[][] labels = new JLabel[N][N]; // Array to hold JLabels representing each cell

static int[][] board = new int[N][N]; // 2D array to represent the chessboard

// Method to print the solution matrix (not used in visualization)

static void printSolution() {

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

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

System.out.printf("%d ", board[i][j]);

}

System.out.printf("\n");

}

}

// Method to check if a queen can be placed safely at board[row][col]

static boolean isSafe(int row, int col) {

try {

Thread.sleep(200); // Delay for visualization (optional)

} catch (InterruptedException e) {

e.printStackTrace();

}

// Check if there is a queen in the same row to the left

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

if (board[row][i] == 1)

return false;

// Check upper diagonal on left side

for (int i = row, j = col; i >= 0 && j >= 0; --i, --j)

if (board[i][j] == 1)

return false;

// Check lower diagonal on left side

for (int i = row, j = col; i < N && j >= 0; ++i, --j)

if (board[i][j] == 1)

return false;

return true; // Return true if it's safe to place a queen at board[row][col]

}

// Recursive function to solve N Queen problem starting from column col

static boolean solveNQueen(int col) {

if (col >= N) // If all queens are placed, return true

return true;

// Try placing a queen in each row of the current column col

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

try {

Thread.sleep(100); // Delay for visualization (optional)

} catch (InterruptedException e) {

e.printStackTrace();

}

if (isSafe(i, col)) { // Check if it's safe to place a queen at board[i][col]

board[i][col] = 1; // Place the queen

labels[i][col].setBackground(Color.GREEN); // Visualize queen placement

if (solveNQueen(col + 1)) // Recur to place queens in subsequent columns

return true;

board[i][col] = 0; // Backtrack: Remove queen and backtrack

labels[i][col].setBackground(Color.RED); // Visualize backtracking

}

}

return false; // Return false if no solution exists for the current configuration

}

// Method to initialize Swing components and display the chessboard

NQueen() {

JFrame frame = new JFrame("N Queen Problem Visualizer"); // Create a new JFrame

frame.setLayout(new GridLayout(N, N)); // Set layout to grid layout of size N x N

frame.setSize(400, 400); // Set frame size

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE); // Set default close operation

// Initialize JLabels for each cell in the grid

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

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

labels[i][j] = new JLabel(""); // Create new JLabel with empty text

labels[i][j].setOpaque(true); // Set opaque to true for background color

labels[i][j].setHorizontalAlignment(SwingConstants.CENTER); // Center-align text

labels[i][j].setBackground(Color.BLUE); // Set initial background color to blue

labels[i][j].setBorder(BorderFactory.createLineBorder(Color.BLACK)); // Add border for clarity

frame.add(labels[i][j]); // Add JLabel to the JFrame

}

}

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

}

// Main method to start the application

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> {

new NQueen(); // Initialize Swing components

});

solveNQueen(0); // Solve N Queen problem starting from column 0

}

}

**Explanation**

1. **Imports and Class Declaration:**
   * import javax.swing.\*; and import java.awt.\*; are used to import necessary classes from Swing and AWT libraries.
   * public class NQueen { defines the main class NQueen for the application.
2. **Class Variables:**
   * final static int N = 4; sets the size of the chessboard (N x N).
   * static JLabel[][] labels = new JLabel[N][N]; declares a 2D array labels to hold JLabels for each cell in the grid.
   * static int[][] board = new int[N][N]; declares a 2D array board to represent the chessboard configuration.
3. **printSolution() Method:**
   * This method is currently unused in the visualization but can be used to print the final solution matrix if needed.
4. **isSafe() Method:**
   * isSafe(int row, int col) checks if it's safe to place a queen at board[row][col].
   * It checks for queens in the same row, upper diagonal, and lower diagonal to the left of col.
   * Thread.sleep(200) introduces a delay for visualization purposes.
5. **solveNQueen() Method:**
   * solveNQueen(int col) is a recursive method to solve the N Queen problem starting from column col.
   * It tries to place a queen in each row of the current column col.
   * If placement is safe (isSafe() returns true), it marks the board and updates the JLabel color for visualization.
   * If a solution is found recursively (solveNQueen(col + 1)), it returns true.
   * If no solution is found, it backtracks by removing the queen and updating the JLabel color.
6. **NQueen() Constructor:**
   * NQueen() initializes the Swing components and sets up the JFrame with a grid layout (N x N).
   * It initializes JLabels (labels) for each cell in the grid with default properties (text, alignment, background color).
7. **main() Method:**
   * main(String[] args) is the entry point of the application.
   * SwingUtilities.invokeLater(() -> { new NQueen(); }); initializes the Swing components in the Event Dispatch Thread (EDT).
   * solveNQueen(0); starts the solving process by calling solveNQueen with col set to 0.

**Results**

Upon executing the program, a JFrame titled "N Queen Problem Visualizer" will open, displaying a 4x4 grid representing the chessboard. Queens' placements and backtracking steps will be visualized using color changes on the JLabels within the grid. Green color indicates queen placement, and red color indicates backtracking.

**Conclusion**

The N Queen Problem Visualizer using Java Swing provides an interactive and educational tool for exploring the backtracking algorithm in combinatorial problem-solving. By combining algorithmic complexity with graphical representation, this project provides a valuable resource for learning and understanding the mechanics of solving the N Queen problem.

**Future Work**

1. **Expand Board Size:** Extend the program to handle larger chessboards (e.g., 8 Queen problem) to demonstrate scalability and performance considerations.
2. **Enhance User Interaction:** Implement features for user input, such as allowing customization of board size and dynamically loading different N Queen configurations.
3. **Optimize Visualization:** Improve graphical representation for smoother animation and enhanced user experience, potentially adding features like step-by-step controls and visual statistics.

**References**

* Java Swing Documentation: [Oracle Java Swing Tutorial](https://docs.oracle.com/javase/tutorial/uiswing/)
* Backtracking Algorithm: GeeksforGeeks - N Queen Problem

**Appendix**

* Code Repository: https://github.com/Deepanshu-three/NQUEENS