**N-Queens Problem Visualization with Java and Swing**

**Introduction**

The N-Queens problem is a classic example in the field of computer science and mathematics, especially in combinatorial optimization and artificial intelligence. It involves placing N queens on an N×N chessboard such that no two queens threaten each other. The challenge is to find all possible ways to position the queens. This project aims to provide a visual representation of the N-Queens problem using Java and the Swing framework.

Specifically, we want to find arrangements where no two queens share the same row, column, or diagonal. Let’s delve into the details:

**Problem Definition**

The objective is to place N queens on an NxN chessboard such that no two queens can attack each other. This means:

1. No two queens share the same row.
2. No two queens share the same column.
3. No two queens share the same diagonal.

* **Backtracking Approach**:
  + Start in the leftmost column.
  + If all queens are placed, return true (base case).
  + Try all rows in the current column.
  + For each row:
    - If the queen can be safely placed in this row (no clashes with existing queens), mark this position as part of the solution.
    - Recursively check if placing the queen here leads to a valid solution.
    - If yes, return true.
    - Otherwise, backtrack by unmarking the position and trying other rows.
  + If no valid solution is found, return false to trigger backtracking.
* **Time Complexity**: O(N!) (factorial)
* **Space Complexity**: O(N^2)

Given these constraints, the N-Queens problem becomes a combinatorial challenge, which grows exponentially with the size of the board.

**Methodology**

**Tools and Technologies**

1. **Java**: The programming language used to develop the application.
2. **Swing**: A Java-based GUI toolkit used for creating the graphical user interface.
3. **AWT (Abstract Window Toolkit)**: Provides the core graphical functionalities.
4. **ImageIO**: Used to read image files for displaying the queen's icon.

**Implementation Steps**

1. **Setup the Environment**: Initialize the main class and GUI components.
2. **Input Handling**: Allow users to specify the number of queens.
3. **Solve the Problem**: Implement the backtracking algorithm to solve the N-Queens problem.
4. **Visualization**: Use Swing to visually display the solution on a chessboard.

**Detailed Code Explanation**

**Class Structure**

The main class N\_Queens extends JPanel and implements the core functionality of solving the N-Queens problem and visualizing the solutions.

**Key Methods and Functionalities**

1. **setN(int nValue)**:
   * Sets the value of N (the number of queens and the size of the board).

java

public static void setN(int nValue) {

n = nValue;

}

1. **solve(int arr[][], int k)**:
   * This is the recursive method that implements the backtracking algorithm. It attempts to place queens row by row, ensuring that no two queens threaten each other.
   * For each possible position in a row, it checks if placing a queen there would be safe.
   * If placing a queen is safe, it proceeds to place the next queen in the following row.
   * If a solution is found, it stores the board configuration in a queue for later visualization.

java

void solve(int arr[][], int k) {

for(int i=0; i<arr[k].length; i++){

int t = 0;

int x = i+1, y = i-1;

for(int j = k-1; j >= 0; j--){

if(arr[j][i] == 1) {

t = 1;

}

if(y >= 0 && arr[j][y] == 1) {

t = 1;

}

if(x < arr[j].length && arr[j][x] == 1) {

t = 1;

}

x++;

y--;

}

if(t == 0){

if(k == arr.length-1){

arr[k][i] = 1;

int a1[][] = new int[n][n];

for(int l = 0; l < n; l++){

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

a1[l][j] = arr[l][j];

}

}

a.add(a1);

arr[k][i] = 0;

}

else {

arr[k][i] = 1;

int a1[][] = new int[n][n];

for(int l = 0; l < n; l++){

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

a1[l][j] = arr[l][j];

}

}

a.add(a1);

solve(arr, k+1);

arr[k][i] = 0;

}

}

}

}

1. **paint(Graphics g)**:
   * This method overrides the paint method of JPanel. It initializes the graphical components and starts the solution visualization thread.

public void paint(Graphics g){

super.paint(g);

N\_Queens.g.setColor(new Color(150,20,0));

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

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

if((i+j)%2 == 0)

g.fillRect(i\*100,j\*100,100,100);

}

}

int arr[][] = new int[n][n];

solve(arr,0);

Runnable r = new draw();

Thread t = new Thread(r);

t.start();

}

1. **startApplication()**:
   * Initializes and displays the main application window, sets up the board size, and reads the queen image.

java

public static void startApplication() {

SwingUtilities.invokeLater(() -> {

JFrame f = new JFrame();

f.setVisible(true);

g = f.getGraphics();

f.setSize(n \* 100 + 100, n \* 100 + 100);

try {

queenImage = ImageIO.read(new File("—Pngtree—gold chess queen 3d render\_13004591.png"));

} catch (IOException e) {

e.printStackTrace();

}

N\_Queens q = new N\_Queens();

f.add(q);

});

}

1. **main method**:
   * Sets up the initial GUI for user input and starts the application based on user input.

java

public static void main(String args[])throws Exception{

JFrame f = new JFrame();

f.setVisible(true);

f.setSize(400,300);

JPanel p = new JPanel();

JTextField jt = new JTextField(10);

p.add(jt);

JLabel lj = new JLabel("Enter the number of queens");

p.add(lj);

JButton jb = new JButton("Submit");

p.add(jb);

p.setLayout(new BoxLayout(p, BoxLayout.Y\_AXIS));

f.add(p);

p.add(Box.createVerticalStrut(300));

jb.addActionListener(e -> {

setN(Integer.parseInt(jt.getText()));

f.setVisible(false);

startApplication();

});

}

**Draw Class**

This separate class draw implements Runnable to handle the visualization in a separate thread, ensuring smooth updates of the GUI.

java

class draw implements Runnable{

public void run(){

while(!N\_Queens.a.isEmpty()){

int ar[][] = new int[N\_Queens.n][N\_Queens.n];

ar = N\_Queens.a.poll();

int k = 0;

for(int i = 0; i < N\_Queens.n; i++){

for(int j = 0; j < N\_Queens.n; j++){

if(ar[i][j] == 1){

k++;

N\_Queens.g.setColor(Color.RED);

N\_Queens.g.fillRect(8+i\*100, 30+j\*100, 100, 100);

N\_Queens.g.drawImage(N\_Queens.queenImage, 10 + i \* 100, 30 + j \* 100, 100, 100, null);

try {

if(k == N\_Queens.n)

Thread.sleep(3000);

Thread.sleep(150);

} catch (InterruptedException ie) {

ie.printStackTrace();

}

} else {

if((i+j)%2 == 0){

N\_Queens.g.setColor(new Color(150,20,150));

N\_Queens.g.fillRect(8+i\*100, 30+j\*100, 100, 100);

} else {

N\_Queens.g.setColor(Color.WHITE);

N\_Queens.g.fillRect(8+i\*100, 30+j\*100, 100, 100);

}

}

}

}

}

}

}

**Visualization**

The visualization part involves:

* Drawing the chessboard using alternating colors.
* Placing the queens as per the solutions generated by the backtracking algorithm.
* Displaying the queens using an image file.

**Conclusion**

This project effectively demonstrates how to solve and visualize the N-Queens problem using Java and Swing. The implementation showcases the use of backtracking for problem-solving and Swing for creating a rich graphical user interface. The ability to visualize each step provides a better understanding of the algorithm and the problem itself.

**Future Enhancements**

1. **Optimization**: Improve the algorithm to handle larger values of N efficiently.
2. **User Interface**: Enhance the GUI to be more interactive and user-friendly.
3. **Additional Features**: Add features such as saving the solutions to a file, step-by-step solving mode, and customizable board and queen colors.

By implementing these enhancements, the application can become a more robust and educational tool for understanding the N-Queens problem and backtracking algorithms in general.