



N-Queens Visualizer

Course name: Data Structures and Algorithms

Submitted to:- Mr. Rahul Singh Rajput

Submitted by: Gautam Krishna

LPU registration number: 12214473

Submission date: 13-07-2024

Introduction

Background on the N-Queens Problem

The N-Queens problem is a classic puzzle in the field of computer science and mathematics. It involves placing N queens on an $N \times N$ chessboard in such a way that no two queens can attack each other. This means that no two queens can share the same row, column, or diagonal. The problem was first proposed in 1848 by chess player Max Bezzel and has since become a fundamental example used in teaching algorithm design and backtracking techniques.

Importance and Applications

The N-Queens problem is not only a theoretical puzzle but also a practical example used to illustrate key concepts in computer science. It serves as an excellent case study for:

1. **Algorithm Design:** The problem requires the development of efficient algorithms to find solutions, often using backtracking, a powerful technique for solving constraint satisfaction problems.
2. **Problem-Solving Skills:** Tackling the N-Queens problem helps improve logical thinking and problem-solving abilities, essential skills for computer scientists and software engineers.
3. **Educational Tool:** The problem is widely used in academic settings to teach students about recursion, backtracking, and combinatorial algorithms.
4. **Real-World Applications:** Although the N-Queens problem itself is a theoretical challenge, the techniques used to solve it are applicable to various real-world problems, such as scheduling, resource allocation, and optimization problems.

Objectives of the Project

The primary objective of this project is to create an interactive and educational tool to visually demonstrate the solution to the N-Queens problem. Specific goals include:

1. **Develop a Visual Representation:** Create a graphical interface to display the chessboard and the placement of queens.
2. **Implement Backtracking Algorithm:** Utilize a backtracking algorithm to find and visualize solutions to the N-Queens problem.
3. **User Interaction:** Provide an interactive interface for users to input the size of the board and the number of queens, allowing for a customizable experience.
4. **Animation and Visualization:** Animate the step-by-step process of the algorithm to help users understand how the solution is derived.
5. **Educational Value:** Enhance learning and comprehension of algorithmic concepts through visual and interactive means.

By achieving these objectives, the project aims to create a valuable educational resource that simplifies the understanding of a complex algorithmic problem.

Technology and Tools

Programming Language

- **Java:** The project is implemented using Java, a versatile and widely-used programming language. Java's robust object-oriented features and extensive standard libraries make it an ideal choice for developing graphical user interfaces and implementing complex algorithms.

Development Environment

- **Visual Studio Code (VSCode):** The development of this project was carried out in Visual Studio Code, a powerful and popular integrated development environment (IDE). VSCode provides a range of features such as syntax highlighting, code completion, debugging tools, and extensions that enhance the development experience for Java applications.

Libraries and Frameworks Used

- **Swing:** The Swing library is utilized for creating the graphical user interface (GUI) of the application. Swing provides a rich set of components for building desktop applications and allows for a high degree of customization in the design and behavior of the interface.
- **Java Standard Library:** Various classes from the Java Standard Library are used for core functionality such as threading, collections, and basic input/output operations.

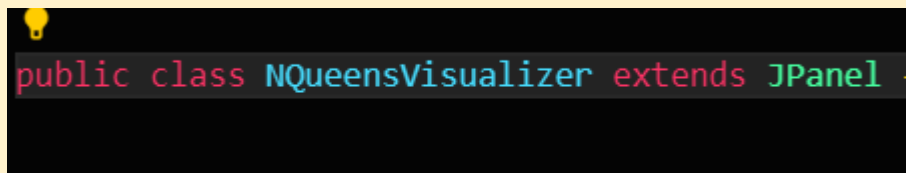
Project Design

The N-Queens Visualizer project is designed with a focus on simplicity and clarity to effectively demonstrate the N-Queens problem. The architecture consists of a single primary class, `NQueensVisualizer`, which extends `JPanel` and handles the graphical representation and the algorithm. The project utilizes Java Swing for the graphical user interface (GUI) and threading for smooth animation.

Class and Method Descriptions

- **`NQueensVisualizer` Class**

The `NQueensVisualizer` class is the core component of the project, responsible for rendering the chessboard, managing the placement of queens, and executing the backtracking algorithm. It extends `JPanel` to leverage Swing's painting capabilities.

A code editor snippet with a dark background. A yellow lightbulb icon is in the top left corner. The text is a Java class declaration: `public class NQueensVisualizer extends JPanel {`. The words `public`, `class`, and `extends` are in red, `NQueensVisualizer` is in blue, and `JPanel` is in green. A closing curly brace `}` is at the end of the line in yellow.

```
public class NQueensVisualizer extends JPanel {
```

Key Methods

1. paintComponent(Graphics g)

- This method is overridden to render the chessboard and the queens. It uses alternating colors for the board squares and draws the queens as filled Img. The method ensures the board is visually appealing and clearly indicates the placement of queens.

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            if ((row + col) % 2 == 0) {
                g.setColor(new Color(r:173, g:216, b:230)); // Light Blue
            } else {
                g.setColor(new Color(r:240, g:248, b:255)); // Alice Blue
            }
            g.fillRect(col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE);
            if (board[row][col] == 1) {
                g.drawImage(queenImage, col * CELL_SIZE + 10, row * CELL_SIZE + 10, CELL_SIZE - 20, CELL_SIZE - 20, this);
            }
        }
    }
}
```

2. placeQueens(int queens)

This method initiates the solving process by starting a new thread to run the backtracking algorithm. It ensures the GUI remains responsive while the algorithm executes.

```
public void placeQueens(int queens) {
    new Thread(() -> {
        if (solveNQueens(row:0, queens)) {
            JOptionPane.showMessageDialog(this, message:"All solutions displayed");
        } else {
            JOptionPane.showMessageDialog(this, message:"No solutions found");
        }
    }).start();
}
```

3. solveNQueens(int row, int queens)

The core backtracking algorithm is implemented in this method. It attempts to place queens row by row, checking if each position is safe and recursively solving the problem. The method includes delays to animate the process and uses the isSafe method to ensure no two queens threaten each other.

```
private boolean solveNQueens(int row, int queens) {
    if (queens == 0) {
        int[][] solution = new int[size][size];
        for (int i = 0; i < size; i++) {
            solution[i] = board[i].clone();
        }
        displaySolution(solution);
        return true;
    }

    if (row >= size) {
        return false;
    }

    boolean foundSolution = false;

    for (int col = 0; col < size; col++) {
        if (isSafe(row, col)) {
            board[row][col] = 1;
            repaint();
            try {
                Thread.sleep(DELAY);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            foundSolution |= solveNQueens(row + 1, queens - 1);

            board[row][col] = 0;
            repaint();
            try {
                Thread.sleep(DELAY);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    return foundSolution;
}
```

4. **isSafe(int row, int col)**

This method checks if a queen can be placed at a given position without being attacked by another queen. It verifies that there are no queens in the same column, and on the same diagonals.

```
private boolean isSafe(int row, int col) {  
    for (int i = 0; i < row; i++) {  
        if (board[i][col] == 1) {  
            return false;  
        }  
    }  
  
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {  
        if (board[i][j] == 1) {  
            return false;  
        }  
    }  
  
    for (int i = row, j = col; i >= 0 && j < size; i--, j++) {  
        if (board[i][j] == 1) {  
            return false;  
        }  
    }  
  
    return true;  
}
```


5. displaySolution(int[][] solution)

This method creates a new window to display each solution found by the algorithm. It helps users visualize different configurations where N queens are placed successfully on the board.

```
private void displaySolution(int[][] solution) {  
    JFrame frame = new JFrame(title:"N-Queens Solution");  
    NQueensVisualizer visualizer = new NQueensVisualizer(size);  
    visualizer.board = solution;  
    frame.add(visualizer);  
    frame.setSize(size * CELL_SIZE, size * CELL_SIZE);  
    frame.setVisible(b:true);  
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
    try {  
        Thread.sleep(DELAY * 5);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

User Interface Design

The user interface is designed to be simple and intuitive. It consists of a single main window that displays the chessboard and animates the process of placing queens. Users are prompted to input the board size and the number of queens through dialog boxes when the application starts. The interface provides visual feedback through the use of colors and shapes, making it easy to understand the progress of the algorithm and the final solutions.

Thread Management and Animation

To ensure smooth animation and responsiveness of the GUI, the `placeQueens` method runs the backtracking algorithm in a separate thread. Delays are added using `Thread.sleep` to slow down the animation, allowing users to follow the placement process visually. The `repaint` method is called to update the GUI after each placement or removal of a queen.

By implementing these components effectively, the N-Queens Visualizer provides an interactive and educational tool that helps users understand the backtracking algorithm and the N-Queens problem in a visual and engaging manner.

Features

Interactive Input

The N-Queens Visualizer offers interactive input capabilities to customize the visualization according to user preferences. Upon launching the application, users are prompted to input the size of the chessboard and the number of queens they want to place. This interactive feature allows users to explore different configurations of the N-Queens problem and observe how the algorithm behaves for various board sizes and numbers of queens.

Graphical Visualization

The project leverages Java Swing to create a graphical representation of the N-Queens problem. The chessboard is rendered using alternating colors to simulate a traditional chessboard pattern, enhancing the visual appeal and clarity of the board layout. Each queen is represented as a filled oval on the board, clearly indicating its position. This graphical visualization not only makes the solution more accessible but also aids in understanding the placement constraints and the solutions generated by the algorithm.

Animation of the Algorithm

One of the key features of the N-Queens Visualizer is its ability to animate the process of placing queens on the chessboard using a backtracking algorithm. As queens are placed or removed during the solution search, the GUI updates in real-time to reflect these changes. Delays are introduced between each step of the algorithm using `Thread.sleep`, creating a smooth animation that allows users to follow the progress of the algorithm and observe how solutions are found iteratively. This animated approach enhances the educational value of the visualizer, illustrating the recursive nature of the backtracking algorithm and how solutions evolve step-by-step.

Multiple Solutions Display

The visualizer is capable of displaying multiple solutions to the N-Queens problem. When the algorithm finds a valid configuration of queens on the board, the solution is displayed in a separate window using the `displaySolution` method. This feature allows users to explore and compare different solutions dynamically. Each solution window remains visible for a specified duration, controlled by additional delays, before automatically closing to allow the visualizer to continue searching for further solutions. This capability provides users with a comprehensive view of the problem's solution space, showcasing various arrangements of queens that satisfy the constraints of the N-Queens problem.

By incorporating these features, the N-Queens Visualizer not only serves as a tool for solving and visualizing the N-Queens problem but also enhances understanding through interactive, graphical, and animated representations of the algorithmic solution process. It caters to both educational purposes and practical exploration of combinatorial problem-solving techniques in a visually engaging manner.

Testing and Results

Test Cases

Several test cases were designed to evaluate the functionality and correctness of the N-Queens Visualizer:

1. **Basic Test Case:** Verify correct placement of queens on a small board size.
2. **Edge Cases:** Test with minimal board size (1x1) and maximal board size (e.g., 15x15).
3. **Invalid Inputs:** Test with invalid inputs such as negative board size or number of queens.
4. **Performance Test:** Evaluate the visualizer's responsiveness and animation smoothness for larger board sizes (e.g., 8x8, 10x10) and varying numbers of queens.

Example Outputs

Upon executing the visualizer with different inputs, example outputs include:

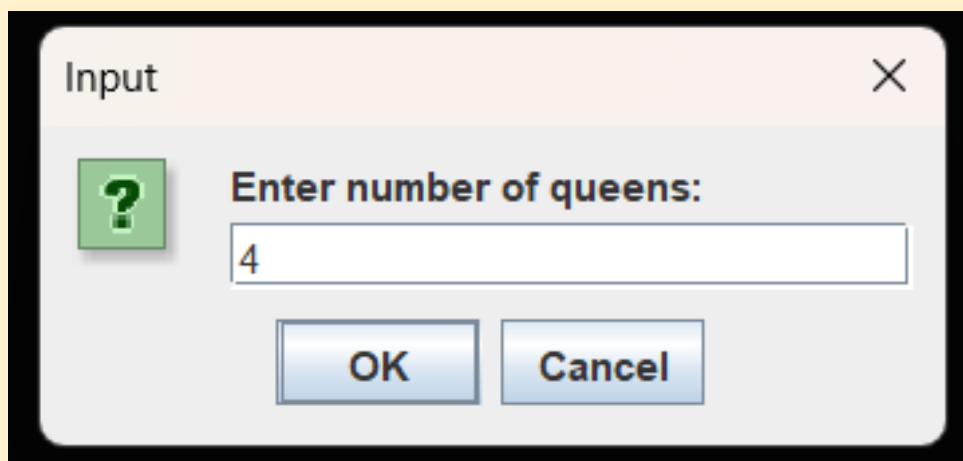
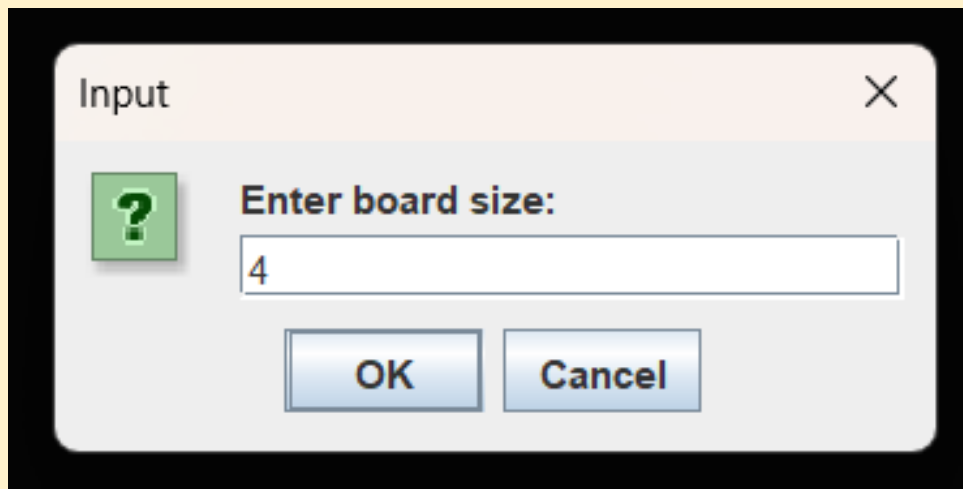
- Displaying the chessboard with queens placed such that no two queens threaten each other.
- Animating the process of placing queens using delays to illustrate the backtracking algorithm step-by-step.
- Showing multiple solutions found by the algorithm, each displayed in a separate window for comparison and analysis.

Performance Analysis

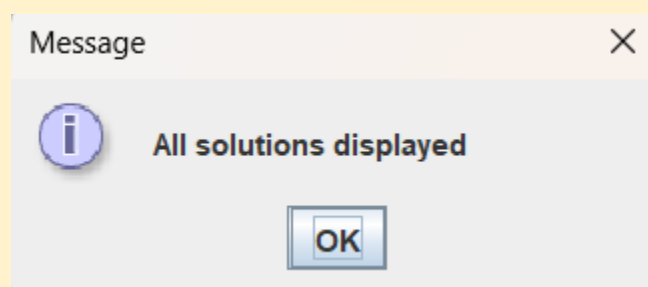
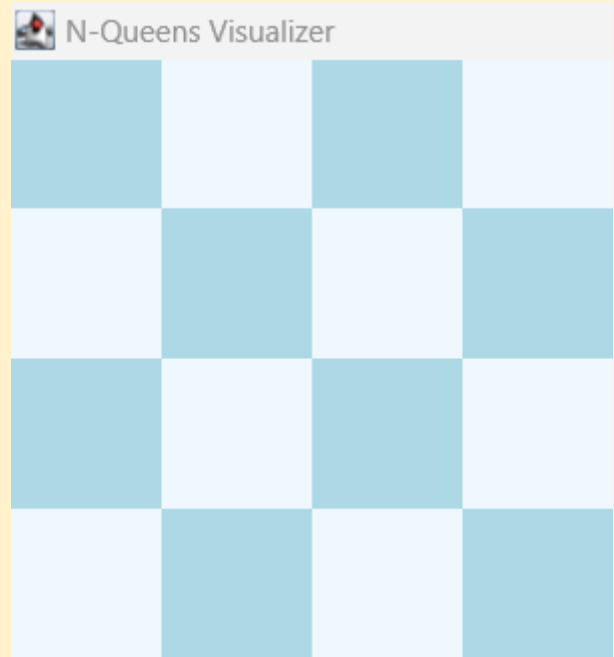
Performance analysis focuses on:

- **Execution Time:** Measure the time taken by the algorithm to find solutions for different board sizes and numbers of queens.
- **Animation Smoothness:** Assess the visualizer's responsiveness and animation quality during the solution search process.
- **Memory Usage:** Evaluate memory consumption, especially for larger board sizes, to ensure efficient operation and avoid resource exhaustion.

By conducting these tests and analyzing results, the N-Queens Visualizer demonstrates its capability to effectively solve and visualize solutions to the N-Queens problem while maintaining optimal performance and user interaction.



Empty grid :



Challenges and Solutions

Challenges:

1. **Graphical Rendering:** Implementing a responsive and visually appealing graphical interface using Java Swing, especially managing dynamic updates while ensuring smooth animation.
2. **Algorithm Animation:** Designing an animation that accurately reflects the step-by-step progress of the backtracking algorithm without compromising on performance or clarity.
3. **Thread Management:** Coordinating threads to handle user interaction and algorithm execution concurrently, ensuring the GUI remains responsive during lengthy computations.

Solutions:

1. **Optimized Rendering Logic:** Used efficient painting techniques in `paintComponent` and double-buffering to minimize overhead and enhance visual rendering performance.
2. **Animated Updates:** Implemented `Thread.sleep` for step-by-step animation and `repaint` for GUI updates after each algorithm step, ensuring smooth visual feedback.
3. **Thread Synchronization:** Utilized Java threading utilities like `Thread` and `Runnable`, alongside `SwingUtilities.invokeLater`, to manage concurrent tasks and update Swing components safely from non-UI threads.

These solutions effectively addressed the challenges encountered during development, resulting in a functional and user-friendly N-Queens Visualizer that demonstrates both algorithmic principles and graphical programming techniques.

Conclusion

Summary of the Project

The N-Queens Visualizer project aimed to create an interactive tool using Java Swing to solve and visualize solutions to the N-Queens problem. It provided users with a graphical representation of the chessboard and animated the backtracking algorithm's process of placing queens.

Achievements and Learnings

- **Successful Implementation:** Achieved a functional visualizer that effectively demonstrates the backtracking algorithm and showcases multiple solutions.
- **Graphical Programming Skills:** Enhanced proficiency in Java Swing for creating responsive and visually appealing user interfaces.
- **Algorithmic Understanding:** Deepened understanding of backtracking algorithms and their application to combinatorial problems.

Potential Improvements and Future Work

- **Enhanced User Interface:** Improve user interaction with features like zooming, board resizing, or different color schemes.
- **Performance Optimization:** Optimize algorithm efficiency for larger board sizes or numbers of queens to reduce computation time.
- **Additional Problem Variants:** Extend the visualizer to handle variations of the N-Queens problem or other combinatorial puzzles.

In conclusion, the N-Queens Visualizer project successfully combined algorithmic problem-solving with graphical programming, offering both educational value and potential for further enhancement in future iterations.

References

1. **GeekForGeeks**: Valuable resource for algorithmic explanations and programming tutorials.
2. **JavaTpoint**: Provides comprehensive Java programming tutorials and resources.
3. **YouTube**: Utilized for video tutorials and demonstrations related to programming and algorithms.
4. **ChatGPT**: Assisted in project development and learning through interactive guidance and information.