N-Queens Visualizer

Using Backtracking algorithm



**Made By:** Pariyank Sikarwar

**Registration Number:** 12211436

**Submitted to:** Rahul Singh Rajput

**1. Introduction**

The N-Queens problem is a classic challenge in computer science where the goal is to place N queens on an NxN chessboard such that no two queens threaten each other. This problem is not only of theoretical interest but also has practical applications in constraint satisfaction problems and combinatorial optimization. This report explores the implementation of a solution to the N-Queens problem using a backtracking algorithm and visualizes the process of finding all possible solutions for different board sizes.

**2. Problem Description**

The N-Queens problem can be formally defined as follows: Given an NxN chessboard, place N queens such that no two queens threaten each other. Queens threaten each other if they share the same row, column, or diagonal. This constraint makes the problem both challenging and computationally expensive to solve for larger values of N. The report discusses the constraints, challenges, and various approaches to solving this problem.

**3. Solution Approach**

**Algorithm Overview**

The chosen approach to solve the N-Queens problem is based on backtracking. Backtracking is a systematic method to find all possible configurations by exploring each potential solution incrementally and abandoning paths that lead to violations of constraints.

**Recursive Strategy**

The core of the solution lies in a recursive function (solve) that attempts to place queens column by column. It uses the isSafe function to check if placing a queen in a particular cell maintains the validity of the board configuration. Upon finding a valid placement, the function recursively attempts to place queens in subsequent columns until a solution is found or all possibilities are exhausted.

**Function Explanation**

The Java implementation includes functions such as isSafe to check the safety of placing a queen, saveBoard to store valid board configurations, printBoard to visualize board states, and solution to orchestrate the solving process. Each function plays a crucial role in achieving the goal of finding all solutions to the N-Queens problem.

**4. Implementation Details**

**Code Walkthrough**

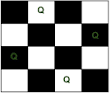
The provided Java code initializes an NxN chessboard with empty cells represented by '.' and attempts to place queens ('Q') in valid positions using the backtracking approach. It handles user input to determine the board size (n) and systematically explores all possible placements using recursion and backtracking.

**Execution Flow**

Upon initialization, the program starts the solving process from the first column of the board and recursively attempts to place queens while ensuring the board remains valid. The solve function backtracks when a placement leads to an invalid configuration, exploring alternative placements until all possibilities are exhausted or a solution is found.

**5. Visual Representation**

The report includes visual representations of board configurations during the execution of the N-Queens Visualizer. Screenshots or diagrams illustrate the intermediate steps of placing queens and demonstrate how the board evolves with each recursive call. Final solutions for different board sizes are presented to showcase the effectiveness of the backtracking algorithm in solving the problem.

[](https://www.google.com/search?rlz=1C1CHBF_enIN844IN844&q=What%2Bis%2Bmeant%2Bby%2BN-Queen%2Bproblem%3F&tbm=isch&source=iu&ictx=1&vet=1&fir=WKQpzcW_s5QGQM%2CuZRkUSBnn8BjYM%2C_&usg=AI4_-kRTUARXSPCe1sfo8RTO8rCwrGQrYA&sa=X&sqi=2&ved=2ahUKEwiH_5rn5ZX3AhWxSLgEHbOGCTEQ9QF6BAgPEAE&imgrc=WKQpzcW_s5QGQM)

A grid with numbers and letters

Description automatically generated

Since, we have to place 4 queens such as q1 q2 q3 and q4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q1 in the very first acceptable position (1, 1). Next, we put queen q2 so that both these queens do not attack each other. We find that if we place q2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q3' safely. So we backtrack one step and place the queen 'q2' in (2, 4), the next best possible solution.

Then we obtain the position for placing 'q3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q4' can be placed safely. Then we have to backtrack till 'q1' and place it to (1, 2) and then all other queens are placed safely by moving q 2 to (2, 4), q3 to (3, 1) and q4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

A grid with numbers and letters

Description automatically generated

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

A black background with a black square

Description automatically generated with medium confidence

Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.

# Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

Code

import java.util.\*;

public class Main {

public static boolean isSafe(int row, int col, char[][] board) {

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

if (board[row][j] == 'Q') {

return false;

}

}

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

if (board[i][col] == 'Q') {

return false;

}

}

int r = row;

for (int c = col; c >= 0 && r >= 0; c--, r--) {

if (board[r][c] == 'Q') {

return false;

}

}

r = row;

for (int c = col; c < board.length && r >= 0; r--, c++) {

if (board[r][c] == 'Q') {

return false;

}

}

r = row;

for (int c = col; c >= 0 && r < board.length; r++, c--) {

if (board[r][c] == 'Q') {

return false;

}

}

for (int c = col; c < board.length && r < board.length; c++, r++) {

if (board[r][c] == 'Q') {

return false;

}

}

return true;

}

public static void saveBoard(char[][] board, List<List<String>> allBoards) {

List<String> newBoard = new ArrayList<>();

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

StringBuilder row = new StringBuilder();

for (int j = 0; j < board[0].length; j++) {

row.append(board[i][j]);

}

newBoard.add(row.toString());

}

allBoards.add(newBoard);

}

public static void solve(char[][] board, List<List<String>> allBoards, int col) {

if (col == board.length) {

saveBoard(board, allBoards);

return;

}

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

if (isSafe(row, col, board)) {

board[row][col] = 'Q';

System.out.println("Iteration " + (col + 1) + ":");

printBoard(board);

solve(board, allBoards, col + 1);

board[row][col] = '.';

}

}

}

public static void printBoard(char[][] board) {

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

for (int j = 0; j < board[0].length; j++) {

System.out.print(" " + board[i][j] + " ");

}

System.out.println();

}

System.out.println();

}

public static void solution(int n) {

List<List<String>> allBoards = new ArrayList<>();

char[][] board = new char[n][n];

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

Arrays.fill(board[i], '.');

}

solve(board, allBoards, 0);

System.out.println("All solutions:");

for (List<String> aboard : allBoards) {

for (String row : aboard) {

System.out.println(row);

}

System.out.println();

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the number of queens (board size): ");

int n = scanner.nextInt();

scanner.close();

solution(n);

}}

**6. Complexity Analysis**

**Time Complexity**

The time complexity of the backtracking solution for the N-Queens problem is analyzed in terms of the number of recursive calls and operations performed at each call. The exponential growth of solutions with increasing n is discussed, highlighting the computational challenges for larger board sizes.

**Space Complexity**

The space complexity focuses on the memory usage of the solution, including the stack space consumed by recursive calls and the auxiliary data structures used to store board configurations. The impact on memory requirements is evaluated alongside theoretical analysis.

**7. Conclusion**

In conclusion, the report summarizes the findings and insights gained from implementing the N-Queens Visualizer using a backtracking algorithm. It reflects the efficiency and effectiveness of the approach in generating all valid configurations for different board sizes. Future directions for improving performance or exploring alternative algorithms, such as bit manipulation techniques, are suggested to further optimize the solution.