Project Report

![Lovely Professional University logo — Transforming Education Transforming India]

N QUEEN

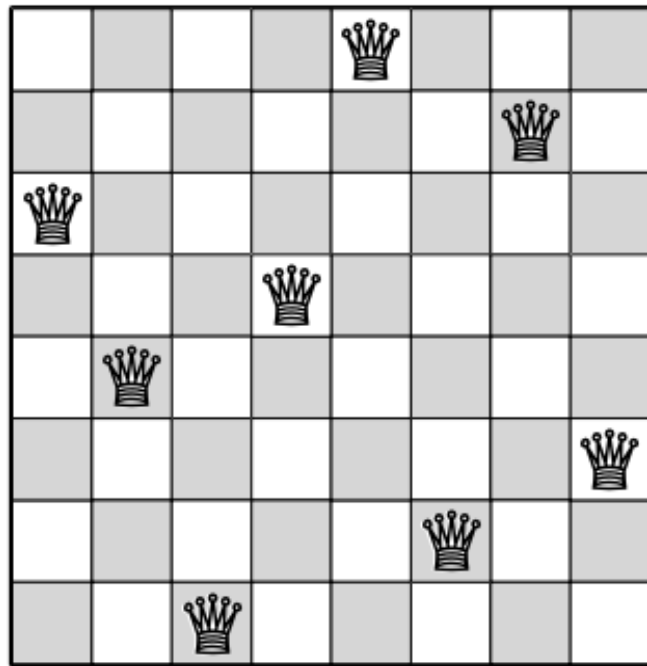Backtracking Problem

Submitted By

Tarun Kumar

12222850

Submitted to

Rahul Singh Rajput

1. Introduction

The N-queens problem is a classic chess puzzle where N queens must be placed on an N×N chessboard so that no two queens threaten each other. This project implements a solution to the N-queens problem with a visual component, allowing users to see the solving process in real-time.

Key objectives:

- Implement an efficient backtracking algorithm to solve the N-queens problem

- Create a visual representation of the solving process

- Allow user interaction for customizable solving parameters

1. Problem Statement and Algorithm

The N-queens problem requires placing N queens on an N×N chessboard so that no two queens can attack each other. This means no two queens can share the same row, column, or diagonal.

We use a backtracking algorithm to solve this problem:

1. Start in the leftmost column
2. If all queens are placed, return true
3. Try placing a queen in all rows of the current column
4. For each row, check if the queen can be placed safely
5. If safe, mark this position and recursively check if placing this queen leads to a solution
6. If placing the queen doesn't lead to a solution, backtrack and try the next row

2. Implementation Details

2.1 Key Components

- **board**: The board is a 2D vector of integers where 0 represents an empty square and 1 represents a queen. This N x N structure efficiently models the chessboard, allowing for easy manipulation and checking of queen positions. It provides a balance between simplicity and performance, crucial for the algorithm's implementation and visualization.

- **solveNQueens()**: This function serves as the entry point for the solving algorithm. It calls solveNqueensUtil(0) to start the recursive process, handles the output of results, and displays the final chessboard state. It manages the overall flow of the solving process and user interaction.

- **solveNQueensUtil()**: The core of the solving process, this recursive function implements the backtracking algorithm. It attempts to place queens column by column, backtracking when necessary. It also handles the visualization by calling printBoard() after each move, allowing users to see the solving process step- by-step.

- **isSafe()**: This function determines if it's safe to place a queen at a given position. It checks for conflicts in the row, upper-left diagonal, and lower- left diagonal. By only checking

to the left of the current column, it optimizes the process, crucial for the algorithm's efficiency, especially on larger boards.

● **printBoard():** Visualizes the current board state

2.2 Visualization Features

- Uses Unicode chess queen symbol (♕) to represent queens

- Alternating white and gray squares for chessboard pattern

- Red color for queens
- Real-time updates with customizable delay between moves

2.3 User Interaction

- Input parameters: board size, option to find all solutions, delay between steps

- Output: current board state, iteration count, solution count

3. Code Analysis

```cpp
bool solveNQueensUtil(int col) {
    recursiveCalls++;

    if (col >= boardSize) {
        solutionCount++; printBoard();

this_thread::sleep_for(chrono::milliseconds(delay)); return
        findAllSolutions ? false : true;

    }

    for (int i = 0; i < boardSize; i++) { iterationCount++;

        if (isSafe(i, col)) {
            board[i][col] = 1;

            printBoard();

this_thread::sleep_for(chrono::milliseconds(delay));

            if (solveNQueensUtil(col + 1)) return true;
```

```
        board[i][col] = 0; // Backtrack printBoard();


this_thread::sleep_for(chrono::milliseconds(delay));
        }
    }


    return false;
}
```
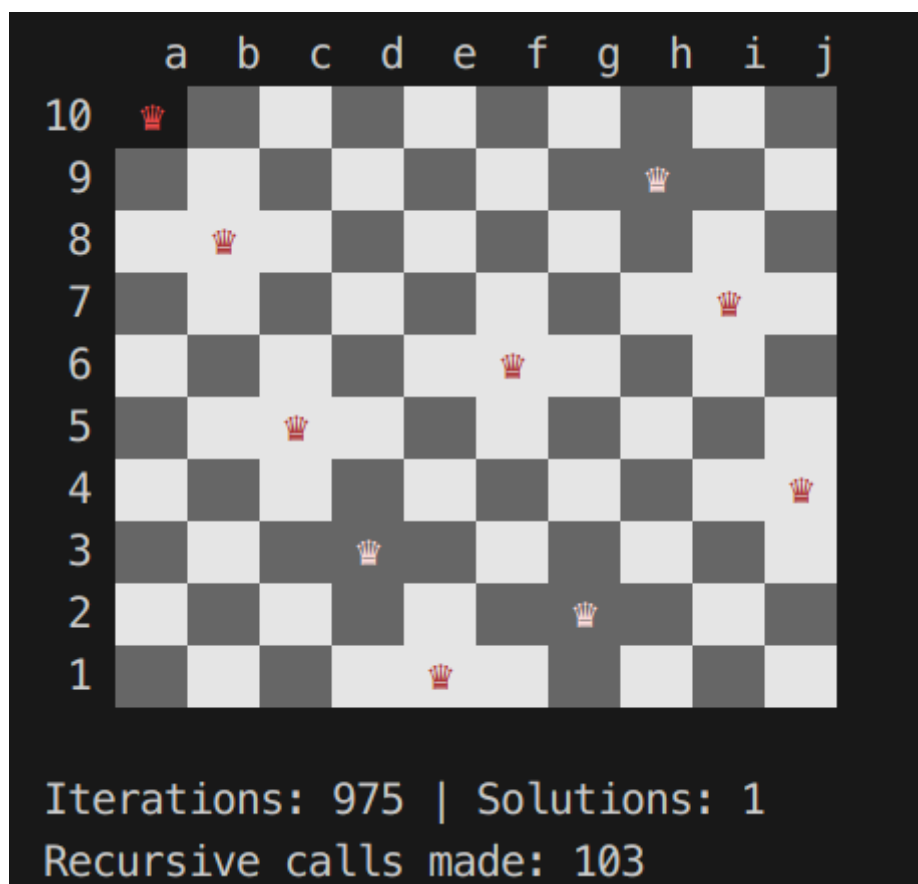

Iterations: 975 | Solutions: 1
Recursive calls made: 103

4. Code Analysis

This function is the core of the backtracking algorithm. It attempts to place a queen in each row of the current column, recursively moving to the next column if successful. If a solution is found or all possibilities are exhausted, it backtracks.

5. Performance Analysis

- Time Complexity: O(N!), but practically faster due to backtracking

- Space Complexity: O(N^2) for the board representation • Performance metrics tracked: iteration count,

  recursive calls, solution count

6. Results and Discussion

The program successfully solves the N-queens problem for various board sizes. The visualization helps in understanding the backtracking process, making it an effective educational tool.

Performance observations:

- Smaller boards (N ≤ 8) are solved almost instantly • Larger boards (N > 12) can take significant time to find all solutions

- The number of recursive calls grows exponentially with board size

7. Conclusion

This N-queens Visualizer successfully combines an efficient solving algorithm with an informative visualization. It serves both as a problem solver and an educational tool, helping users understand the intricacies of backtracking algorithms.