

**COURSE CODE - CSES003**

**Project Report**

**Submitted by -Sayantan Hutait**

**Reg. No.-12214919**

**Submitted to – Rahul Rajput**

**Acknowledgement**

I would like to express my heartfelt gratitude to all the individuals who have contributed to the successful completion of this C++ programming project on the topic of N-Queens Visuailzer.

I am also immensely grateful to my project team members, who have worked tirelessly to ensure the completion of this project. Their dedication, hard work, and teamwork have been instrumental in achieving our project goals.

I would like to extend my appreciation to all the resources and references that have helped us in the development of this project. We are thankful to all the authors, online resources, and developers whose work we have referred to, to gain a deeper understanding of the various concepts and programming techniques.

Finally, I would like to express my sincere gratitude to my family and friends for their constant support and encouragement throughout the project. Their unwavering support has been the driving force behind my success.

Once again, I extend my heartfelt thanks to all those who have played a part in making this project a reality.

**N-Queens Visualizer Project Report**

### Introduction

The N-Queens problem is a well-known combinatorial problem in computer science and mathematics. It involves placing N chess queens on an N×N chessboard so that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal. The problem was first proposed in 1848 by the chess player Max Bezzel, and it has since become a popular topic for studying algorithms and computer science concepts.

The primary objectives of this project are:

1. To solve the N-Queens problem using a backtracking algorithm.
2. To visualize the step-by-step process of placing queens on the board.
3. To display all possible solutions for a given N.

This report provides a detailed overview of the implementation, visualization, and results of the N-Queens problem solver using C++.

### Objectives

The main goals of the project are:

1. **Algorithm Implementation**: Develop a backtracking algorithm to solve the N-Queens problem.
2. **Step-by-Step Visualization**: Create a visual representation of the algorithm's execution to enhance understanding.
3. **Solution Display**: Display all possible solutions for a given board size N.

The project aims to provide a comprehensive understanding of the N-Queens problem and its solution using a backtracking approach. Additionally, the visualization component helps in understanding the recursive nature of the algorithm and the concept of backtracking.

### Implementation

The solution to the N-Queens problem involves the following key components:

1. **Board Representation**: The chessboard is represented as a one-dimensional vector where the index represents the row, and the value at each index represents the column position of the queen in that row.
2. **Backtracking Algorithm**: A recursive function is used to place queens row by row, backtracking when a conflict is detected.
3. **Visualization**: The board is printed to the console at each step, with a delay to help visualize the placement and backtracking process.

### Board Representation

In this implementation, the chessboard is represented using a one-dimensional vector board. The index of the vector represents the row on the chessboard, and the value at each index represents the column position of the queen in that row. For example, if board[2] = 3, it means that there is a queen placed at the position (2, 3) on the chessboard.

This representation simplifies the process of checking for conflicts, as it allows us to use simple arithmetic operations to determine if two queens are in the same column or diagonal.

**Backtracking Algorithm**

The backtracking algorithm is implemented using a recursive function solveNQueensUtil. The algorithm starts by attempting to place a queen in each column of the first row. For each placement, it checks if the position is safe using the isSafe function. If the position is safe, the algorithm recursively attempts to place queens in the subsequent rows. If a conflict is detected, the algorithm backtracks by removing the queen and trying the next column.

The isSafe function checks for conflicts by ensuring that no two queens are in the same column or diagonal. This is achieved by iterating through the previously placed queens and checking their positions relative to the current queen's position.

**Visualization**

To help visualize the algorithm's execution, the board is printed to the console at each step of the placement and backtracking process. The printBoard function is used to print the current state of the board. It marks the positions of the queens with 'Q' and empty spaces with '.'. A delay of 0.5 seconds is introduced using the usleep function to allow for better visualization of each step.

The visualization helps in understanding how the algorithm explores different configurations and backtracks when conflicts are encountered. It provides a clear view of the recursive nature of the algorithm and the concept of backtracking.

**Code**

The following C++ code implements the N-Queens solver and visualizer:

#include <iostream>

#include <vector>

#include <unistd.h>

using namespace std;

void printBoard(const vector<int>& board, int N) {

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

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

if (board[i] == j) {

cout << "Q ";

} else {

cout << ". ";

}

}

cout << endl;

}

cout << endl;

usleep(500000); // Sleep for 0.5 seconds to visualize the steps

}

bool isSafe(const vector<int>& board, int row, int col, int N) {

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

if (board[i] == col || board[i] - i == col - row || board[i] + i == col + row) {

return false;

}

}

return true;

}

void solveNQueensUtil(vector<int>& board, int row, int N, vector<vector<int>>& solutions) {

 if (row == N) {

        solutions.push\_back(board);

        printBoard(board, N);

        cout << "Solution found!" << endl;

        usleep(500000);

        return;

    }

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

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

board[row] = col;

printBoard(board, N);

solveNQueensUtil(board, row + 1, N, solutions);

board[row] = -1;

printBoard(board, N);

}

}

}

void solveNQueens(int N) {

vector<int> board(N, -1);

vector<vector<int>> solutions;

solveNQueensUtil(board, 0, N, solutions);

if (solutions.empty()) {

cout << "No solution exists for " << N << " queens." << endl;

} else {

cout << "Final Solutions:\n";

for (const auto& sol : solutions) {

printBoard(sol, N);

}

}

}

int main() {

int N;

cout << "Enter the value of N: ";

cin >> N;

solveNQueens(N);

return 0;

}

**Explanation**

The code consists of the following key functions:

1. **printBoard**: This function prints the current state of the board. It marks the positions of the queens with 'Q' and empty spaces with '.'. A delay of 0.5 seconds is introduced to visualize each step.
2. **isSafe**: This function checks whether it is safe to place a queen at the given row and column. It ensures that no other queen can attack the current position. The function checks for conflicts by iterating through the previously placed queens and ensuring that no two queens are in the same column or diagonal.
3. **solveNQueensUtil**: This is the recursive backtracking function. It attempts to place a queen in each column of the current row, checks for safety, and recursively attempts to place queens in subsequent rows. If a conflict arises, it backtracks by removing the queen and trying the next column. The function also prints the board at each step to visualize the process.
4. **solveNQueens**: This function initializes the board and starts the solving process. It also handles the case where no solution exists. The function calls the solveNQueensUtil function and prints the final solutions.
5. **main**: The main function prompts the user to enter the value of N and then calls the solveNQueens function to start the solving process.

**Results**

The program prints each step of the algorithm, showing the placement of queens and the backtracking process. Once all solutions are found, it prints each solution in the same format.For example, for N = 4, the output will show the intermediate steps of placing and removing queens, followed by the final solutions:

\_ Q \_ \_

\_ \_ \_ Q

Q \_ \_ \_

\_ \_ Q \_

\_ \_ Q \_

Q \_ \_ \_

\_ \_ \_ Q

\_ Q \_ \_

The output demonstrates the algorithm's exploration of different configurations and its ability to backtrack when conflicts are encountered. The visualization helps in understanding the recursive nature of the algorithm and the concept of backtracking.

**Conclusion**

This project successfully demonstrates the use of a backtracking algorithm to solve the N-Queens problem. The visualization helps in understanding the recursive nature of the algorithm and the backtracking process. This project can be extended by adding a graphical user interface for better visualization and user interaction.

The N-Queens problem is a classic example of a combinatorial problem that can be solved using a backtracking algorithm. The solution involves exploring different configurations and backtracking when conflicts are encountered. The visualization component helps in understanding the algorithm's execution and the concept of backtracking.

**Future Work**

1. **Graphical Interface**: Implementing a graphical user interface (GUI) using a library like Qt or SFML for better visualization. A GUI would provide a more interactive and visually appealing way to display the board and the algorithm's execution.
2. **Optimizations**: Improving the efficiency of the algorithm for larger values of N. This could involve using more advanced techniques like branch and bound, or parallelizing the algorithm to take advantage of multi-core processors.
3. **Interactive Visualization**: Allowing users to step through the algorithm interactively. This would involve adding controls to pause, resume, and step through each iteration of the algorithm. Interactive visualization would enhance the learning experience by allowing users to explore the algorithm's execution at their own pace.