****

**A PROJECT REPORT**

**On**

**ALGORITHMIC IN N-QUEENS WITH BACKTRACKING**

SUBMITTED TO

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

In partial fulfillment of the award of the course of

**CSA0697-DESIGN AND ANALYSIS OF ALGORITHMS FOR LOWER BOUND THEORY**

**By**

C. Harsha Vardhan Reddy (192210375)

**SUPERVISOR**

Dr. GNANA SOUNDARI



**SAVEETHA SCHOOL OF ENGINEERING, SIMATS CHENNAI- 602105**

**SEPTEMBER-2024**

**BONAFIDE CERTIFICATE**

Certified that this project report titled **“ALGORITHMIC IN N-QUEENS WITH BACKTRACKING**” is the bonafide work **C. Harsha Vardhan Reddy (192210375)**, who carried out the project work under my supervision as a batch. Certified further, that to the best of my knowledge, the work reported here in does not form any other project report.

Project Supervisor Head of the Department

Date: Date:

**ABSTRACT:**

The N-Queens problem is a classic combinatorial problem that involves placing N queens on an N×N chessboard such that no two queens threaten each other. This problem can be solved using various techniques, and one of the most efficient approaches is through backtracking. This paper presents a C program implementation of the N-Queens problem using the backtracking algorithm.Backtracking is an algorithmic paradigm that builds the solution incrementally, abandoning a solution path as soon as it determines that it cannot lead to a valid solution. In the N-Queens problem, the algorithm places queens one by one in different columns, starting from the leftmost column. For each column, the algorithm checks if placing a queen in a particular row is safe by ensuring that no other queens threaten the position horizontally, vertically, or diagonally.

**KEY WORDS:**

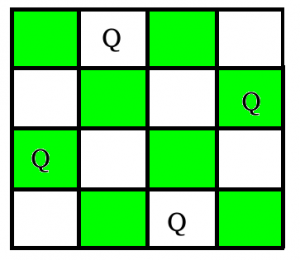
N-Queens problem, Backtracking algorithm, C programming, Combinatorial optimization, Recursion, Chessboard problem

Constraint satisfaction, Algorithm design, Queen placement, Dynamic board size.

**INTRODUCTION:**

The N-Queens problem is a well-known challenge in the field of combinatorial optimization and algorithm design. The task is to place N queens on an N×N chessboard in such a way that no two queens can attack each other. This means that no two queens should share the same row, column, or diagonal. The problem was first posed in the mid-19th century and has since become a classic example of a constraint satisfaction problem. To solve the N-Queens problem efficiently, various techniques can be applied, including brute force and backtracking. Among these, backtracking stands out as a practical and effective method. Backtracking is a depth-first search algorithm that incrementally builds the solution, testing each partial solution for feasibility. If the current configuration leads to a conflict, the algorithm backtracks by removing the last placed queen and tries a different position. This process continues until a solution is found or all possible configurations are exhausted.

This paper discusses the implementation of the N-Queens problem using backtracking in C. The program dynamically accepts the size of the chessboard from the user and employs recursive backtracking to find and print a valid solution. By systematically exploring all potential queen placements and discarding invalid configurations early, the backtracking approach offers a highly optimized solution to this complex problem. The implementation demonstrates how backtracking can efficiently solve the N-Queens problem for varying board sizes, making it a powerful tool in algorithmic design for constraint satisfaction problems.



**Conditions for N- Queens Backtracking:**

* Each queen must be placed in a unique row.
* No two queens should be placed in the same column.
* No two queens should be placed on the same diagonal.
* If at any row, no valid column can be found for placing the queen the algorithm backtracks to the previous row and tries a different column placement for the queen in that row.
* This process continues until a solution is found or all possible configurations are exhausted.

**PROBLEM STATEMENT:**

The N-Queens problem requires placing N queens on an N×N chessboard such that no two queens can attack each other. The challenge lies in ensuring that no two queens occupy the same row, column, or diagonal, which are the three possible ways in which queens can threaten one another.

* Accept a user input for the size of the chessboard, N.
* Dynamically place N queens on an N×N chessboard in such a way that no two queens threaten each other.
* Explore possible configurations using the backtracking technique, which recursively places queens and backtracks when conflicts arise.
* Output one valid arrangement of queens on the chessboard, or indicate if no solution exists.

**CODING:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

void printSolution(int \*\*board, int N) {

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

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

printf("%d ", board[i][j]);

}

printf("\n");

}

}

bool isSafe(int \*\*board, int row, int col, int N) {

int i, j;

for (i = 0; i < col; i++)

if (board[row][i])

return false;

for (i = row, j = col; i >= 0 && j >= 0; i--, j--)

if (board[i][j])

return false;

for (i = row, j = col; j >= 0 && i < N; i++, j--)

if (board[i][j])

return false;

return true;

}

bool solveNQUtil(int \*\*board, int col, int N) {

if (col >= N)

return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1, N))

return true;

board[i][col] = 0; // Remove queen

}

}

return false;

}

bool solveNQ(int N) {

int \*board = (int \*)malloc(N \* sizeof(int \*));

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

board[i] = (int \*)malloc(N \* sizeof(int));

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

board[i][j] = 0;

}

}

if (!solveNQUtil(board, 0, N)) {

printf("Solution does not exist\n");

return false;

}

printSolution(board, N);

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

free(board[i]);

}

free(board);

return true;

}

int main() {

char input[100];

printf("Enter the size of the board (N): ");

fgets(input, 100, stdin);

int N = atoi(input);

if (N < 4) {

printf("No solution exists for N less than 4.\n");

return 1;

}

solveNQ(N);

return 0;

}

**RESULT :**



**TIME COMPLEXITY:**

**Worst-case:** time complexity: O(N!)

**Best-case:** This depends on how early the algorithm finds a valid solution. In the best case (highly optimized pruning), the solution may be found faster, but the worst case remains O(N!).

**FUTURE SCOPE:**

**1. Optimization Techniques:**

Heuristic Improvements: Incorporate advanced heuristics such as constraint propagation or forward checking to further reduce the search space and improve the efficiency of the backtracking algorithm.

**2. Parallel and Distributed Computing:**

**Parallelization:** Implement parallel processing to explore different branches of the search tree simultaneously, leveraging multi-core processors or GPU computing to handle larger board sizes more effectively.

**Distributed Systems:** Develop distributed versions of the algorithm to utilize networked systems for handling extremely large values of N, potentially involving cloud-based solutions.

**3. Extended Problem Variants:**

Generalize the algorithm to handle variations like the K-Queens problem where the objective might involve different constraints or additional pieces on the board.

Explore versions of the N-Queens problem with additional constraints or objectives, such as minimizing conflicts between non-queen pieces or optimizing placement for other strategic goals.

**4. User Interface and Visualization:**

Develop a GUI-based application that allows users to interactively solve the N-Queens problem, visualize solutions, and experiment with different board sizes and constraints.

Create educational tools that use visualizations to demonstrate how the backtracking algorithm explores the solution space and handles constraints, aiding in teaching and learning about algorithmic techniques.

**CONCLUSION:**

The N-Queens problem is a fundamental challenge in combinatorial optimization, demonstrating the complexities and nuances of constraint satisfaction. The backtracking algorithm provides an effective method for solving this problem by incrementally placing queens on an N×N chessboard and using recursive exploration to ensure that no two queens threaten each other. This C program implementation showcases how backtracking can be used to solve the N-Queens problem, dynamically adjusting to different board sizes based on user input. By recursively placing queens and backtracking when conflicts arise, the algorithm efficiently navigates the search space and finds a valid solution if one exists.

The ability to handle various board sizes and the clear demonstration of recursive problem-solving techniques highlight the robustness of backtracking for combinatorial problems. Future improvements could involve integrating optimization techniques, parallel processing, and user-friendly interfaces to further enhance the algorithm's performance and applicability. Overall, this implementation not only solves the N-Queens problem effectively but also serves as a valuable example of how backtracking can be employed to address complex algorithmic challenges.