| **Experiment No. 9** |
| --- |
| **To implement N -Queen problem** |
| Date of Performance: 28/3/24 |
| Date of Submission:4/4/24 |

## Experiment No. 9

**Title**: To implement N -Queen problem

**Aim**: To study, implement and Analyze N queen Problem.

**Objective:** To introduce the N queen Problem and analyzing algorithms

**Theory**:

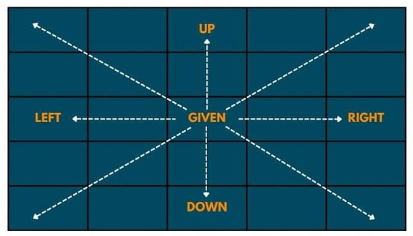
Backtracking is a problem-solving technique that involves recursively trying out different solutions to a problem, and backtracking or undoing previous choices when they don’t lead to a valid solution. It is commonly used in algorithms that search for all possible solutions to a problem, such as the famous eight-queens puzzle. Backtracking is a powerful and versatile technique that can be used to solve a wide range of problems.

The N Queen problem demands us to place N queens on a N x N chessboard so that no queen can attack any other queen directly.

**Problem Statement:**

Find out all the possible arrangements in which N queens can be seated in each row and each column so that all queens are safe.

The queen moves in 8 directions and can directly attack in these 8 directions only.



#### Example:

**4 - Queen Problem:**

* This problem demands us to put 4 queens on 4 X 4 chessboard in such a way that no 2 or more queens can be placed in the same diagonal or row or column.
* 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**.

**Solution to 4 Queen Problem** A diagram of a network

Description automatically generated



#### Algorithm and Complexity:

**Code:**

**Algorithm:**

Step 1: Start in the leftmost column.

Step 2: If all queens are placed return true.

Step 3: Try all rows in the current column. Do the following for every row.

Step 3.1: If the queen can be placed safely in this row.

Step 3.1.1: Then mark this [row, column] as part of the solution and recursively

check if placing queen here leads to a solution.

Step 3.1.2: If placing the queen in [row, column] leads to a solution then

return true.

Step 3.1.3: If placing queen doesn’t lead to a solution then unmark this [row,

column] then backtrack and try other rows.

Step 4: If all rows have been tried and valid solution is not found return false to trigger

backtracking.

**Time Complexity -  O(N!)**

* For the first row, we check N columns; for the second row, we check the N - 1 column and so on. Hence, the time complexity will be N \* (N-1) \* (N-2) …. i.e. O(N!)

**Space Complexity - O(N^2)**

* O(N^2), where ‘N’ is the number of queens.
* We are using a 2-D array of size N rows and N columns, and also, because of Recursion, the recursive stack will have a linear space here. So, the overall space complexity will be O(N^2).

**Program:**

**#include <stdio.h>**

**#include <stdbool.h>**

**#define N 10 // Maximum board size**

**bool isSafe(int board[][N], int row, int col, int size) {**

**int i, j;**

**// Check for queens in the same row**

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

**if (board[row][i])**

**return false;**

**// Check upper diagonal on left side**

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

**if (board[i][j])**

**return false;**

**// Check lower diagonal on left side**

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

**if (board[i][j])**

**return false;**

**return true;**

**}**

**bool solveNQUtil(int board[][N], int col, int size) {**

**// All queens are placed**

**if (col >= size)**

**return true;**

**// Try placing queen in each row of this column**

**for (int i = 0; i < size; i++) {**

**if (isSafe(board, i, col, size)) {**

**// Place the queen**

**board[i][col] = 1;**

**// Recur to place remaining queens**

**if (solveNQUtil(board, col + 1, size))**

**return true;**

**// If placing queen at board[i][col] doesn't lead to a solution, backtrack**

**board[i][col] = 0;**

**}**

**}**

**return false;**

**}**

**bool solveNQ(int size) {**

**int board[N][N] = {{0}}; // Initialize the board to 0**

**if (solveNQUtil(board, 0, size) == false) {**

**printf("Solution does not exist");**

**return false;**

**}**

**// Printing the solution**

**for (int i = 0; i < size; i++) {**

**for (int j = 0; j < size; j++)**

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

**printf("\n");**

**}**

**return true;**

**}**

**int main() {**

**int size;**

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

**scanf("%d", &size);**

**solveNQ(size);**

**return 0;**

**}**

**Output:**

**Enter the size of the board (N x N): 5\*5**

**1 0 0 0 0**

**0 0 0 1 0**

**0 1 0 0 0**

**0 0 0 0 1**

**0 0 1 0 0**

**Conclusion:**

**The N Queens problem, while seemingly simple, poses a complex computational challenge. Its solution involves finding all possible arrangements of N queens on an N×N chessboard without any queen threatening another. The most common approach to solving this problem is through backtracking algorithms, like the recursive depth-first search. The time complexity of this approach is approximately O(N!), where N is the size of the board.**

**This is because each queen placement is tested against every previous queen's position, resulting in an exponential growth of possibilities. However, with optimizations such as pruning invalid branches and symmetry exploitation, the actual runtime can be significantly reduced, making the problem tractable even for moderately large values of N.**