**Assignment 6**

**Problem Statement:**  
Implement basic search strategies – **8-Queens Problem**.

**Theory**

**1. The N-Queens Problem**

* The **N-Queens problem** is a classic AI and backtracking problem.
* Objective: Place N queens on an **N × N chessboard** such that no two queens attack each other.
* A queen can attack **horizontally, vertically, and diagonally**.
* For N = 8, the problem is called the **8-Queens Problem**.

**2. Search Strategies Used**

* **Backtracking (Depth-First Search)** is used to explore possible board configurations.
* At each row, try placing a queen in one column at a time.
* If the placement is safe, move to the next row.
* If unsafe, backtrack and try another column.

**3. Applications of N-Queens**

* Demonstrates **constraint satisfaction**.
* Used in AI search problems, scheduling, and puzzle solving.
* Illustrates how **DFS + backtracking** can reduce the solution space.

**Algorithm Steps**

1. Start with an empty board.
2. Place a queen in the first row, one column at a time.
3. For each placement, check if it is safe (no queen in the same column, left diagonal, or right diagonal).
4. If safe, place the queen and move to the next row.
5. If all rows are filled, store the solution.
6. If no column is safe in a row, backtrack to the previous row.
7. Repeat until all solutions are found.

**Code (C++ Implementation)**

#include <bits/stdc++.h>

using namespace std;

class Solution {

public:

// Check if placing queen is safe

bool isSafe(vector<string> &board, int row, int col, int n) {

// Check column

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

if (board[i][col] == 'Q') return false;

}

// Check left diagonal

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

if (board[i][j] == 'Q') return false;

}

// Check right diagonal

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

if (board[i][j] == 'Q') return false;

}

return true;

}

// Recursive backtracking

void nQueens(vector<string> &board, int row, int n, vector<vector<string>> &ans) {

if (row == n) {

ans.push\_back(board);

return;

}

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

if (isSafe(board, row, j, n)) {

board[row][j] = 'Q'; // place queen

nQueens(board, row + 1, n, ans);

board[row][j] = '.'; // backtrack

}

}

}

vector<vector<string>> solveNQueens(int n) {

vector<string> board(n, string(n, '.'));

vector<vector<string>> ans;

nQueens(board, 0, n, ans);

return ans;

}

};

int main() {

int n;

cout << "Enter value of N: ";

cin >> n;

Solution solver;

vector<vector<string>> solutions = solver.solveNQueens(n);

cout << "Number of solutions: " << solutions.size() << "\n\n";

for (int k = 0; k < solutions.size(); k++) {

cout << "Solution " << k + 1 << ":\n";

for (string row : solutions[k]) {

cout << row << "\n";

}

cout << "\n";

}

return 0;

}

**Sample Output (N = 4)**

Enter value of N: 4

Number of solutions: 2

Solution 1:

.Q..

...Q

Q...

..Q.

Solution 2:

..Q.

Q...

...Q

.Q..

For **N = 8**, there are **92 distinct solutions**.

**Conclusion**

* The **8-Queens problem** was solved using **backtracking with depth-first search strategy**.
* The program generates **all possible solutions** where queens do not attack each other.
* This problem illustrates **constraint satisfaction and pruning** in AI search.
* The algorithm can be generalized for **N-Queens** (not just 8).