**Artificial Intelligence: Practical- 4**

**Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.**

#include <iostream>

#include <cstring>

#define N 4 // Define the size of the chessboard (4x4 for this example)

using namespace std;

// Function to print the board configuration

void printPuzzle(int board[N][N]) {

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

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

cout << board[i][j] << " ";

}

cout << '\n';

}

}

// Function to check if placing a queen at (row, col) is safe

bool isSafe(int row, int col, int slash[N][N], int backSlash[N][N],

bool rowLookup[], bool slashLookUp[], bool backSlashLookUp[]) {

// Check if the row, primary diagonal (slash), or secondary diagonal (backslash) is already occupied

if (slashLookUp[slash[row][col]] ||

backSlashLookUp[backSlash[row][col]] ||

rowLookup[row]) {

return false;

}

return true;

}

// Recursive function to solve the N-Queens problem using backtracking

bool solveNqueenUtil(int board[N][N], int col, int slash[N][N], int backSlash[N][N],

bool rowLookUp[N], bool slashLookUp[N], bool backSlashLookUp[N]) {

// If all queens are placed, return true

if (col >= N) return true;

// Try placing a queen in each row for this column

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

// Check if it's safe to place the queen

if (isSafe(i, col, slash, backSlash, rowLookUp, slashLookUp, backSlashLookUp)) {

// Place the queen

board[i][col] = 1;

rowLookUp[i] = true;

slashLookUp[slash[i][col]] = true;

backSlashLookUp[backSlash[i][col]] = true;

// Recursively place queens in the next column

if (solveNqueenUtil(board, col + 1, slash, backSlash, rowLookUp, slashLookUp, backSlashLookUp))

return true;

// Backtrack if placing queen leads to no solution

board[i][col] = 0;

rowLookUp[i] = false;

slashLookUp[slash[i][col]] = false;

backSlashLookUp[backSlash[i][col]] = false;

}

}

// If no position in this column is valid, return false

return false;

}

// Function to initialize the board and start the solving process

void solveNqueen() {

int board[N][N];

memset(board, 0, sizeof(board)); // Initialize board to 0 (no queens placed)

int backSlash[N][N]; // Tracks secondary diagonal (/)

int slash[N][N]; // Tracks primary diagonal (\)

// Lookup tables for row and diagonal conflicts

bool rowLookUp[N] = {false};

bool backSlashLookUp[2 \* N - 1] = {false};

bool slashLookUp[2 \* N - 1] = {false};

// Fill diagonal indices

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

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

backSlash[i][j] = (i - j) + (N - 1); // Secondary diagonal index

slash[i][j] = i + j; // Primary diagonal index

}

}

// Solve the N-Queens problem

if (solveNqueenUtil(board, 0, slash, backSlash, rowLookUp, slashLookUp, backSlashLookUp) == false) {

cout << "Solution does not exist!!\n";

printPuzzle(board); // Print board even if no solution exists

} else {

printPuzzle(board); // Print the valid solution

}

}

int main() {

solveNqueen(); // Run the solver

return 0;

}