

PROGRAM TITLE 2

8-QUEEN PROBLEM

AIM:

To write a python program to solve 8-queen problem.

PROCEDURE:

1. Board Representation:

- Define a 2D array or a suitable data structure to represent the chessboard.

2. Constraint Definition:

- Clearly define constraints to ensure that no two queens threaten each other. This includes checking for conflicts in rows, columns, and diagonals.

3. Backtracking Algorithm:

- Implement a recursive backtracking algorithm to explore all possible queen placements on the board. Start with the first row and try placing queens in each column.

4. Constraint Checking:

- At each step, check whether placing a queen in the current position violates any constraints. If a conflict is detected, backtrack to the previous state and try a different position.

5. Base Case and Solution Printing:

- Define a base case that stops the recursion when a valid solution is found or when all possibilities are explored. Print or store the solutions accordingly.

CODING:

```
def is_goal(state):
```

```
    return state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

```
def find_empty(state):
```

```
    for i in range(3):
```

```
for j in range(3):  
    if state[i][j] == 0:  
        return (i, j)
```

```
def get_valid_moves(state, empty_pos):  
    moves = []  
    i, j = empty_pos  
    if i > 0:  
        moves.append("up")  
    if i < 2:  
        moves.append("down")  
    if j > 0:  
        moves.append("left")  
    if j < 2:  
        moves.append("right")  
    return moves
```

```
def solve(state, visited):  
  
    if is_goal(state):  
        return state  
    visited.add(tuple(map(tuple, state)))  
  
    empty_pos = find_empty(state)  
    for move in get_valid_moves(state, empty_pos):
```

```

new_state = [row.copy() for row in state]

i, j = empty_pos

if move == "up":

    new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]

elif move == "down":

    new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]

elif move == "left":

    new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]

else:

    new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]

if tuple(map(tuple, new_state)) not in visited:

    solution = solve(new_state, visited.copy())

    if solution:

        return [move] + solution

return None


initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]

visited = set()

solution = solve(initial_state, visited)


if solution:

    print("Solution found!")

    for move in solution:

        print(move)

```

else:

print("No solution found.")

N = 8

def solveNQueens(board, col):

if col == N:

print(board)

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQueens(board, col + 1):

return True

board[i][col] = 0

return False

def isSafe(board, row, col):

for x in range(col):

if board[row][x] == 1:

return False

for x, y in zip(range(row, -1, -1), range(col, -1, -1)):

if board[x][y] == 1:

return False

for x, y in zip(range(row, N, 1), range(col, -1, -1)):

if board[x][y] == 1:

```
        return False
```

```
    return True
```

```
board = [[0 for x in range(N)] for y in range(N)]
```

```
if not solveNQueens(board, 0):
```

```
    print("No solution found")
```

OUTPUT:

```
up
```

```
right
```

```
down
```

```
down
```

```
left
```

```
up
```

```
up
```

```
right
```

```
down
```

```
down
```

```
left
```

```
up
```

```
up
```

```
right
```

```
down
```

```
down
```

```
left
```

```
up
```

```
up
```

```
right
```

```
down
```

```
down
```

```
right
```

```
[1, 2, 3]
```

```
[4, 5, 6]
```

```
[7, 8, 0]
```

```
[[1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 1, 0, 0, 0]]
```

RESULT:

Hence the program been successfully executed and verified.