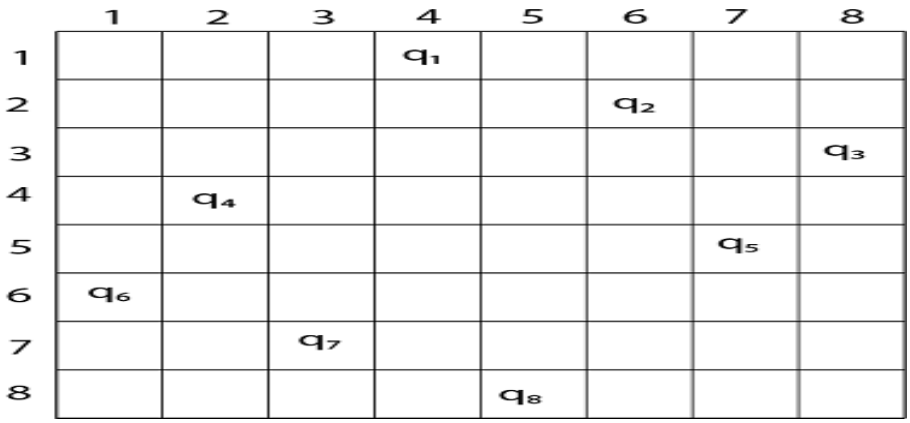
Program 10

Write a Program to Implement 8-Queens Problem using Python.



Description:

The **8-Queens Problem** is a classic example in artificial intelligence and backtracking algorithms. The problem is to place 8 queens on a standard 8x8 chessboard such that no two queens threaten each other. A queen can attack another queen if they are on the same row, column, or diagonal.

Problem Constraints:

- You have 8 queens.
- Each queen must be placed on a different row.
- No two queens can be placed in the same column or diagonal.

Solution Approach:

We can solve the problem using **Backtracking**, which involves placing a queen in a valid position and recursively trying to place the remaining queens. If placing a queen leads to a conflict, we backtrack by removing the queen and trying the next possible position.

Steps to Solve Using Backtracking:

1. Place a queen on the current row in a valid column.
2. Move to the next row and attempt to place a queen in a valid column.
3. If placing a queen results in a valid board configuration, continue.
4. If at any point placing a queen leads to a conflict, backtrack by removing the queen and trying another column.

Step by Step:

1. **is_safe(board, row, col)**:
   - This function checks if it is safe to place a queen in the given position (row, col).
   - We check:
     - If there's a queen in the same column (board[i] == col).
     - If there's a queen on the same diagonal (both main diagonal and anti-diagonal).
     -
2. **solve_n_queens(board, row, n, solutions)**:
   - This is a recursive function that tries to place queens row by row.
   - It places a queen in the current row, checks for safety, and then proceeds to the next row.
   - If all queens are placed (when row == n), the solution is stored in the solutions list.
   - After trying each column in the current row, the function backtracks by removing the queen from the current row (board[row] = -1).
3. **print_solution(board)**:
   - This function prints a valid configuration of queens on the board. Q represents a queen and . represents an empty space.
4. **n_queens(n)**:
   - Initializes the board and calls the solve_n_queens() function to find all solutions.

- There are 92 solutions to the 8-Queens problem. Each solution represents a valid configuration of queens on the chessboard, with no queens threatening each other.

- **Each solution is printed as a grid, where `Q` represents a queen and represents an empty square**

- **SOURCE CODE :**

```
def printSolution(board):
    """Print the chessboard configuration."""
    for row in board:
        print(" ".join("Q" if col else "." for col in row))
    print("\n")

def isSafe(board, row, col, n):
    """Check if placing a queen at board[row][col] is safe."""
    # Check column
    for i in range(row):
        if board[i][col]:
            return False

    # Check upper-left diagonal
    i, j = row, col
    while i >= 0 and j >= 0:
        if board[i][j]:
            return False
        i -= 1
        j -= 1

    # Check upper-right diagonal
    i, j = row, col
    while i >= 0 and j < n:
        if board[i][j]:
            return False
        i -= 1
        j += 1

    return True


def solveNQueens(board, row, n):
    """Use backtracking to solve the N-Queens problem."""
    if row == n:
        printSolution(board)
        return True

    result = False
    for col in range(n):
        if isSafe(board, row, col, n):
```

```
            # Place the queen
            board[row][col] = 1
            # Recur to place the rest of the queens
            result = solveNQueens(board, row + 1, n) or result
            # Backtrack
            board[row][col] = 0

    return result

def nQueens(n):
    """Driver function to solve the N-Queens problem."""
    board = [[0] * n for _ in range(n)]
    if not solveNQueens(board, 0, n):
        print("No solution exists.")
    else:
        print("Solutions printed above.")

# Solve the 8-Queens problem
nQueens(8)
```

OUTPUT :

```
Q . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .


Q . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
```

. . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .

.
.

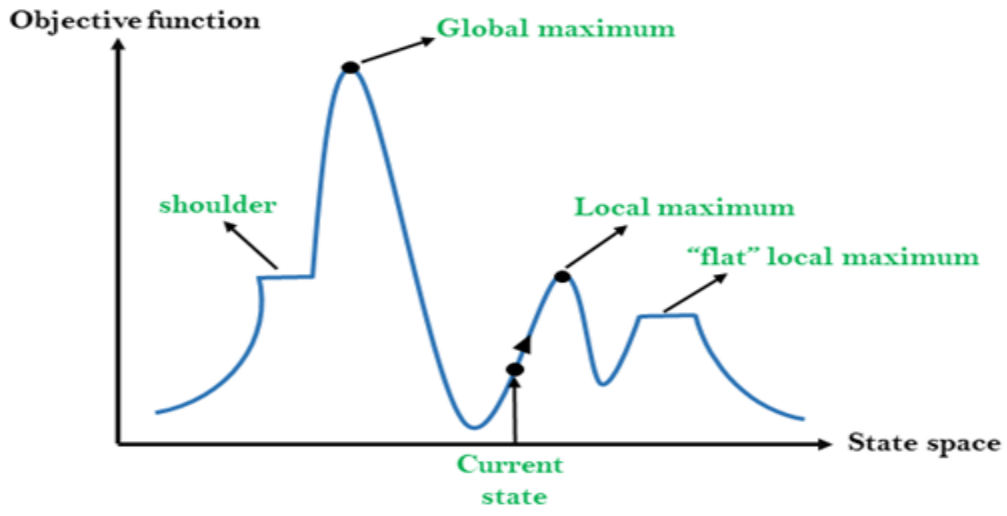
. . . . . Q
. . . Q . . . .
Q . . . . . .
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . Q .
. . . . Q . . .

Solutions printed above.

Additional Programs

Program11

Write a Program to Implement Hill Climbing Problem in AI using python.



Hill climbing is a local search algorithm used for mathematical optimization problems. It starts with an arbitrary solution and then iteratively moves towards the solution that improves the current state by maximizing or minimizing the objective function. This process continues until no further improvement can be made.

Types of Hill Climbing:

1. **Simple Hill Climbing**: Explores one neighbor at a time and moves to a better state if it exists.
2. **Steepest-Ascent Hill Climbing**: Looks at all the neighbors and chooses the best one (the steepest).
3. **Stochastic Hill Climbing**: Chooses a random neighbor and moves to it if it improves the state.

In this example, I will demonstrate the **Simple Hill Climbing** algorithm.

Problem:

Let's assume we are solving a problem where we want to maximize a function (e.g., `f(x) = -x^2 + 10x`), and we start with an initial value of `x` and iteratively try to find a better value of `x` that maximizes this function.

Steps:

Start with a random value.

- Evaluate the neighboring solutions.
- Move to the neighboring solution that provides the highest value.
- Repeat until no neighbors provide a better value.

Python Code Implementation for Simple Hill Climbing:

```
import random
# Objective function (for example, a quadratic function)
def objective_function(x):
    return -x**2 + 10*x
# Hill Climbing Algorithm
def hill_climbing(initial_state, step_size=0.1, max_iterations=100):
    current_state = initial_state
    current_value = objective_function(current_state)

    for iteration in range(max_iterations):
        # Generate neighbors by adding/subtracting step_size
        neighbors = [current_state - step_size, current_state + step_size]

        # Evaluate the neighbors
        next_state = None
        next_value = current_value

        for neighbor in neighbors:
            value = objective_function(neighbor)
            if value > next_value:
                next_state = neighbor
                next_value = value
```

```
        # If no improvement is found, stop the algorithm
        if next_state is None:
            break
            # Move to the better state
        current_state = next_state
        current_value = next_value
        print(f"Iteration {iteration+1}: x = {current_state}, f(x) = {current_value}")
        return current_state, current_value
# Test the algorithm with an initial random state
initial_state = random.uniform(0, 10)  # Start with a random value of x between 0 and 10
print(f"Starting hill climbing with initial state: x = {initial_state}")
optimal_x, optimal_value = hill_climbing(initial_state)
print(f"\nOptimal solution found: x = {optimal_x}, f(x) = {optimal_value}")
```

Output:

```
Starting hill climbing with initial state: x = 2.43723492744764
Iteration 1: x = 2.5372349274476402, f(x) = 18.934788197416168
Iteration 2: x = 2.6372349274476403, f(x) = 19.41734121192664
Iteration 3: x = 2.7372349274476404, f(x) = 19.879894226437116
Iteration 4: x = 2.8372349274476405, f(x) = 20.32244724094759
Iteration 5: x = 2.9372349274476406, f(x) = 20.74500025545806
Iteration 6: x = 3.0372349274476407, f(x) = 21.147553269968533
Iteration 7: x = 3.1372349274476408, f(x) = 21.530106284479004
Iteration 8: x = 3.237234927447641, f(x) = 21.89265929898947
Iteration 9: x = 3.337234927447641, f(x) = 22.235212313499943
Iteration 10: x = 3.437234927447641, f(x) = 22.557765328010422
Iteration 11: x = 3.537234927447641, f(x) = 22.860318342520895
Iteration 12: x = 3.637234927447641, f(x) = 23.142871357031368
Iteration 13: x = 3.7372349274476413, f(x) = 23.405424371541834
Iteration 14: x = 3.8372349274476414, f(x) = 23.647977386052307
Iteration 15: x = 3.9372349274476415, f(x) = 23.870530400562778
```

Iteration 16: x = 4.037234927447641, f(x) = 24.073083415073253

Iteration 17: x = 4.137234927447641, f(x) = 24.25563642958372

Iteration 18: x = 4.23723492744764, f(x) = 24.418189444094196

Iteration 19: x = 4.33723492744764, f(x) = 24.560742458604665

Iteration 20: x = 4.43723492744764, f(x) = 24.683295473115137

Iteration 21: x = 4.537234927447639, f(x) = 24.785848487625607

Iteration 22: x = 4.637234927447639, f(x) = 24.86840150213608

Iteration 23: x = 4.737234927447639, f(x) = 24.93095451664655

Iteration 24: x = 4.837234927447638, f(x) = 24.973507531157026

Iteration 25: x = 4.937234927447638, f(x) = 24.996060545667493

Iteration 26: x = 5.037234927447638, f(x) = 24.998613560177972

Optimal solution found: x = 5.037234927447638, f(x) = 24.998613560177972