Exp: 5: Implementation of Hill Climbing algorithm for 8 Puzzle

Program:

```
import random

class EightPuzzle:
    def __init__(self, initial_state):
        self.state = initial_state

    def generate_random_solution(self):
        # Generate a random solution by shuffling the initial state
        random.shuffle(self.state)

    def generate_neighbors(self):
        neighbors = []
        empty_tile_index = self.state.index(0)
        row, col = divmod(empty_tile_index, 3)

        # Move empty tile left
        if col > 0:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index - 1] = neighbor[empty_tile_index - 1], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        # Move empty tile right
        if col < 2:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index + 1] = neighbor[empty_tile_index + 1], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        # Move empty tile up
        if row > 0:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index - 3] = neighbor[empty_tile_index - 3], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        # Move empty tile down
        if row < 2:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index + 3] = neighbor[empty_tile_index + 3], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        return neighbors
```

```
    def evaluate(self):
        # Evaluate the state based on the number of misplaced tiles
        goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        return sum(x != y for x, y in zip(self.state, goal_state))

def hill_climbing(problem, max_iterations=1000):
    current_solution = problem.state
    current_value = problem.evaluate()

    for _ in range(max_iterations):
        neighbors = problem.generate_neighbors()
        neighbor_values = [problem.evaluate() for _ in
range(len(neighbors))]

        best_neighbor_value = min(neighbor_values)

        if best_neighbor_value >= current_value:
            # If no better neighbor is found, break the loop
            break

        best_neighbor_index =
neighbor_values.index(best_neighbor_value)
        current_solution = neighbors[best_neighbor_index]
        current_value = best_neighbor_value

    return current_solution, current_value

# Example usage:

initial_state = [1, 2, 3, 4, 5, 6, 7, 0, 8]  # Initial state, 0
represents the empty tile
eight_puzzle = EightPuzzle(initial_state)

# Run Hill Climbing algorithm
best_solution, best_value = hill_climbing(eight_puzzle)

print("Best Solution:", best_solution)
print("Best Value:", best_value)
```

Program:

```
import random

class EightPuzzle:
    def __init__(self, initial_state):
        self.state = initial_state
```

```
    def generate_random_solution(self):
        # Generate a random solution by shuffling the initial state
        random.shuffle(self.state)

    def generate_neighbors(self):
        neighbors = []
        empty_tile_index = self.state.index(0)
        row, col = divmod(empty_tile_index, 3)

        # Move empty tile left
        if col > 0:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index - 1]
= neighbor[empty_tile_index - 1], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        # Move empty tile right
        if col < 2:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index + 1]
= neighbor[empty_tile_index + 1], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        # Move empty tile up
        if row > 0:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index - 3]
= neighbor[empty_tile_index - 3], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        # Move empty tile down
        if row < 2:
            neighbor = self.state[:]
            neighbor[empty_tile_index], neighbor[empty_tile_index + 3]
= neighbor[empty_tile_index + 3], neighbor[empty_tile_index]
            neighbors.append(neighbor)

        return neighbors

    def evaluate(self):
        # Evaluate the state based on the number of misplaced tiles
        goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        return sum(x != y for x, y in zip(self.state, goal_state))

def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i + 3])
```

```
# Example usage:

initial_state = [1, 2, 3, 4, 5, 6, 7, 0, 8]  # Initial state, 0
represents the empty tile
eight_puzzle = EightPuzzle(initial_state)

print("Initial State:")
print_state(eight_puzzle.state)

# Run Hill Climbing algorithm
best_solution, best_value = hill_climbing(eight_puzzle)

print("\nGoal State:")
print_state([1, 2, 3, 4, 5, 6, 7, 8, 0])

print("\nBest Solution:")
print_state(best_solution)
print("Best Value:", best_value)
```

```
Initial State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Goal State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]


Best Value: 2
```

Explanation:

Random Module Import:

The program begins by importing the random module, which is used for generating random numbers.

EightPuzzle Class Definition:

The EightPuzzle class represents the 8-puzzle problem.

It has an initializer (__init__) that takes an initial_state argument to set the initial configuration of the puzzle.

generate_random_solution Method:

This method shuffles the current state of the puzzle, creating a random solution by rearranging the tiles.

generate_neighbors Method:

Generates neighboring states by moving the empty tile (represented by 0) in the puzzle in different directions: left, right, up, and down.

evaluate Method:

Evaluates the current state by counting the number of misplaced tiles compared to the predefined goal state ([1, 2, 3, 4, 5, 6, 7, 8, 0]).

print_state Function:

A utility function that prints a given state in a 3x3 grid format.

Example Usage Section:

Creates an instance of the EightPuzzle class with an initial state where 0 represents the empty tile.

Prints the initial state.

Runs the Hill Climbing algorithm (hill_climbing function) to find the best solution.

Prints the goal state, the best solution, and its corresponding value.

Overall Purpose:

The program demonstrates a basic implementation of the Hill Climbing algorithm applied to the 8-puzzle problem.

The algorithm aims to find the optimal arrangement of tiles by iteratively exploring neighboring states and selecting the one that minimizes the evaluation function, which is based on the number of misplaced tiles.

In summary, the program combines the 8-puzzle representation, a Hill Climbing algorithm, and utility functions to demonstrate the process of finding an optimal solution for the 8-puzzle problem.