Experiment No: 4 Date:

8-Puzzle Problem

Aim: To Implement the 8-Puzzle Problem Using Hill Climbing.

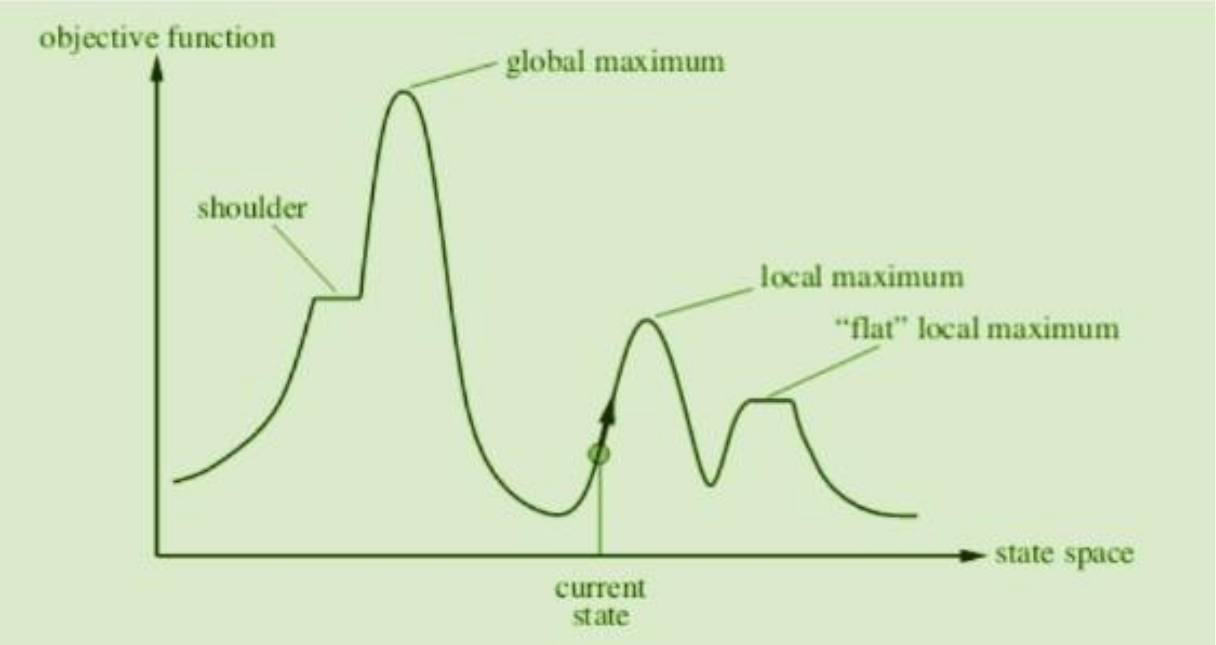
Theory:

- "The 8-puzzle problem revolves around a 3x3 board with 8 numbered tiles and one blank space. Tiles adjacent to the blank space can move into it, allowing players to rearrange the tiles to achieve a specific sequence, known as the goal state. This puzzle, introduced by Noyes Palmer Chapman in 1870, tasks players with arranging the tiles into the correct order.
- In this puzzle, the 3x3 grid consists of numbered tiles from 1 to 8, along with one empty space. Players can shift tiles vertically or horizontally by moving them into the empty square. The objective is to manipulate the tiles through these movements to reach a predetermined arrangement.

Hill Climbing:

- Hill Climbing is a local search algorithm used to find the peak of a "mountain" or the best solution to a problem by continuously moving in the direction of increasing elevation or value.
- It's commonly applied to mathematical optimization problems, such as the Traveling Salesman Problem, aiming to minimize distance.
- Often dubbed as greedy local search, Hill Climbing only considers immediate neighbors, not looking beyond them. Each node in the algorithm consists of a state and a value, representing the current configuration and its quality.
- Hill Climbing leverages heuristics effectively, guiding the search towards promising solution regions.
- Unlike other search algorithms, Hill Climbing maintains only a single current state, eliminating the need for managing complex search trees or graphs.
- Variants of Hill Climbing include Simple Hill Climbing, Steepest-Ascent Hill Climbing, Random Restart Hill Climbing, and Simulated Annealing, each with its own characteristics and applications.
- Hill Climbing is prone to getting stuck in local optima, where it reaches a peak value but not necessarily the global optimum. Techniques like random restarts or simulated annealing can help overcome this limitation.

State Space diagram for Hill Climbing:



- **Local maximum:** It is a state which is better than its neighbouring state however there exists a state which is better than it(global maximum). This stateis better because here the value of the objective function is higher than its neighbours.
- **Global maximum:** It is the best possible state in the state space diagram. This is because, at this stage, the objective function has the highest value.
- **Plateau/flat local maximum:** It is a flat region of state space where neighbouring states have the same value.
- **Ridge:** It is a region that is higher than its neighbours but itself has a slope. It is a special kind of local maximum.
- **Current state:** The region of the state space diagram where we are currentlypresent during the search.
- **Shoulder:** It is a plateau that has an uphill edge.

Algorithm:

```
HillClimbing()
   1 node ← start
   2 newNode ← Head(Sort_h(MoveGen(node)))
   3 while h(newNode) < h(node)
   4    do node ← newNode
   5       newNode ← Head(Sort_h(MoveGen(node)))
   6 return newNode
```

Example: The Hill Climbing algorithm can be effectively applied to solve the Traveling Salesman Problem (TSP). Initially, a solution is determined that visits all the cities exactly once, often resulting in a suboptimal or even poor route. The Hill Climbing algorithm then begins with this initial solution and iteratively improves it. By continuously exploring neighboring solutions and moving towards those with shorter distances, the algorithm gradually refines the route. Eventually, it converges to a much shorter and more efficient path, representing a near-optimal solution to the TSP.

Program:

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

def heuristic(state, goal_state):
    return sum(1 for i in range(9) if state[i] != goal_state[i])

def move(state, direction):
    blank_index = state.index(0)
    new_state = list(state)
    if direction == 'up' and blank_index > 2:
        new_state[blank_index], new_state[blank_index - 3] = new_state[blank_index - 3],
new_state[blank_index]
    elif direction == 'down' and blank_index < 6:
        new_state[blank_index], new_state[blank_index + 3] = new_state[blank_index + 3],
new_state[blank_index]
    elif direction == 'left' and blank_index % 3 != 0:
        new_state[blank_index], new_state[blank_index - 1] = new_state[blank_index - 1],
new_state[blank_index]
    elif direction == 'right' and blank_index % 3 != 2:
        new_state[blank_index], new_state[blank_index + 1] = new_state[blank_index + 1],
new_state[blank_index]
    else:
        return None
    return tuple(new_state)

def generate_neighbors(state):
    neighbors = []
    directions = ['up', 'down', 'left', 'right']
    for direction in directions:
        neighbor = move(state, direction)
        if neighbor is not None:
            neighbors.append(neighbor)
    return neighbors

def solve_puzzle(initial_state, goal_state, max_steps=1000):
    current_state = initial_state
```

```
    current_cost = heuristic(initial_state, goal_state)
    steps = 0
    path = [current_state]
    print_state(current_state)
    print("Current heuristic : ",current_cost)
    print()
    while current_cost > 0 and steps < max_steps:
        neighbors = generate_neighbors(current_state)
        best_neighbor = min(neighbors, key=lambda x: heuristic(x, goal_state))
        best_neighbor_cost = heuristic(best_neighbor, goal_state)
        if best_neighbor_cost < current_cost:
            current_state = best_neighbor
            current_cost = best_neighbor_cost

            print_state(current_state)
            print("Current heuristic : ",current_cost)
            print()
            path.append(current_state)
        else:
            print("No solution found")
            break
        steps += 1
initial_state = tuple(map(int, input("Enter initial state : ").split()))
goal_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

solve_puzzle(initial_state, goal_state)
```

Output:

```
Enter initial state : 1 2 3 4 0 5 7 8 6
(1, 2, 3)
(4, 0, 5)
(7, 8, 6)
Current heuristic :  3

(1, 2, 3)
(4, 5, 0)
(7, 8, 6)
Current heuristic :  2

(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
Current heuristic :  0
```

```
Enter initial state : 1 2 3 4 5 0 7 6 8
(1, 2, 3)
(4, 5, 0)
(7, 6, 8)
Current heuristic :  3

(1, 2, 3)
(4, 5, 8)
(7, 6, 0)
Current heuristic :  2

No solution found
```

Conclusion: Solved 8 puzzle problem using Hill Climbing algorithms with successful execution of programs.