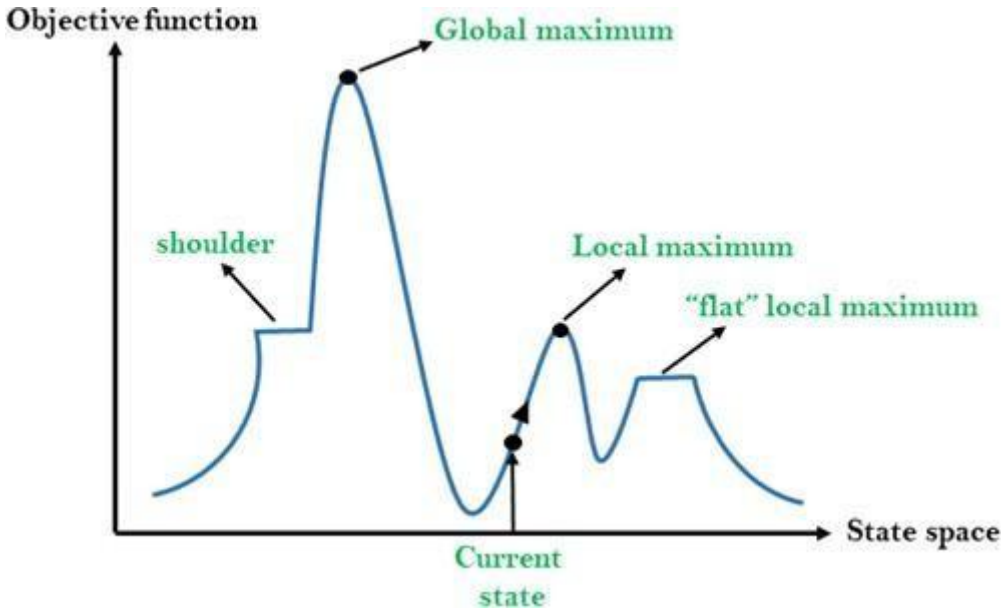
Experiment No.5
Write a Program to Implement Tower of Hanoi using Hill Climbing Algorithm.
Date of Performance: 14/02/2025
Date of Submission: 21/02/2025

Aim: Write a Program to Implement Tower of Hanoi using Hill Climbing Algorithm..

Objective: To apply the Local Search algorithm in game playing.

Theory:

Hill Climbing is a heuristic search algorithm used for optimization problems, where the goal is to find the best possible solution by iteratively improving a candidate solution. It starts from an initial state and moves to a neighboring state that has a higher value based on a heuristic function. The process continues until no better neighboring state is found, at which point the algorithm terminates. Although simple and efficient, Hill Climbing can get stuck in local maxima, plateaus, or ridges, preventing it from finding the global optimum. To overcome these limitations, variations such as random restarts, simulated annealing, and tabu search are used. Hill Climbing is widely applied in fields like artificial intelligence, robotics, machine learning, scheduling, and game optimization, making it a valuable approach for solving real-world optimization problems.

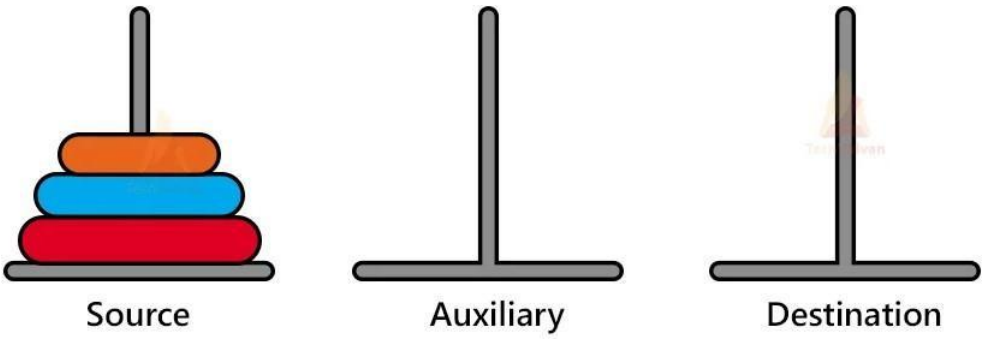


Tower of Hanoi

The Tower of Hanoi is a classic mathematical puzzle involving three pegs and a set of disks of different sizes. The goal is to move all the disks from the source peg to the destination peg, following these rules:

- Only one disk can be moved at a time.
- A disk can only be placed on top of a larger disk or an empty peg.

- The disks must be transferred using an auxiliary peg.



Tower of Hanoi Using Hill Climbing

1. Define the State Representation
A state can be represented as a list of stacks (or arrays) representing the pegs.

2. Define the Heuristic Function

A good heuristic is to measure the "progress" toward the goal:

- **Number of disks on the correct peg**: Count how many disks are in the correct final position.
- **Order of disks on the correct peg**: Prioritize states where disks are stacked correctly.
- **Distance from the goal**: Sum the number of misplaced disks.

3. Generate Possible Moves (Successor States)

- Move the top disk from one peg to another (if valid).
- A move is **valid** if:
  - The source peg is not empty.
  - The destination peg is empty or its top disk is larger than the moving disk.

4. Apply the Hill Climbing Algorithm

1. Start from the initial state.
2. Generate all possible valid successor states.
3. Evaluate each successor using the heuristic function.
4. Choose the best successor (highest heuristic value).
5. Repeat until the goal state is reached or no better moves exist.

5. Handling Local Maxima

- Hill Climbing might get stuck in **local maxima**, meaning it cannot progress further.
- Possible solutions:
  - **Random Restart**: Restart from the initial state if stuck.
  - **Simulated Annealing**: Accept some worse moves with decreasing probability.
  - **Tabu Search**: Keep a memory of previous states to avoid cycling.

Code:

```
import random

# Function to print the current state of the Tower of Hanoi
def print_state(state):
    print("Pole A:", state[0])
    print("Pole B:", state[1])
    print("Pole C:", state[2])

# Heuristic function: the number of disks on the goal pole (Pole C in this case)
def heuristic(state):
    return len(state[2])  # The more disks on pole C, the better

# Function to check if the puzzle is solved (all disks on Pole C)
def is_solved(state):
    return state[0] == [] and state[1] == []  # All disks should be on Pole C

# Function to generate the next possible moves (successors)
def generate_successors(state):
    successors = []

    # Move the top disk from Pole A to Pole B if valid
    if state[0]:
        new_state = [state[0].copy(), state[1].copy(), state[2].copy()]
        disk = new_state[0].pop()
        new_state[1].append(disk)
        successors.append(new_state)

    # Move the top disk from Pole A to Pole C if valid
    if state[0]:
        new_state = [state[0].copy(), state[1].copy(), state[2].copy()]
        disk = new_state[0].pop()
        new_state[2].append(disk)
        successors.append(new_state)

    # Move the top disk from Pole B to Pole A if valid
    if state[1]:
```

```
 new_state = [state[0].copy(), state[1].copy(), state[2].copy()]
        disk = new_state[1].pop()
        new_state[0].append(disk)
        successors.append(new_state)

    # Move the top disk from Pole B to Pole C if valid
    if state[1]:
        new_state = [state[0].copy(), state[1].copy(), state[2].copy()]
        disk = new_state[1].pop()
        new_state[2].append(disk)
        successors.append(new_state)

    # Move the top disk from Pole C to Pole A if valid
    if state[2]:
        new_state = [state[0].copy(), state[1].copy(), state[2].copy()]
        disk = new_state[2].pop()
        new_state[0].append(disk)
        successors.append(new_state)

    # Move the top disk from Pole C to Pole B if valid
    if state[2]:
        new_state = [state[0].copy(), state[1].copy(), state[2].copy()]
        disk = new_state[2].pop()
        new_state[1].append(disk)
        successors.append(new_state)

    return successors

# Hill Climbing Algorithm
def hill_climbing(start_state):
    current_state = start_state
    steps = 0

    while True:
        steps += 1

        # Check if the puzzle is solved
        if is_solved(current_state):
            print("Puzzle Solved!")
            print_state(current_state)
            print(f"Total Steps: {steps}")
            return current_state

        # Generate all possible successors
        successors = generate_successors(current_state)
```

```
    # Evaluate all successors using the heuristic function
    successor_scores = [(succ, heuristic(succ)) for succ in successors]

    # Sort successors based on the heuristic score (descending order)
    successor_scores.sort(key=lambda x: x[1], reverse=True)

    # If the best successor is not better than the current state, stop (local maxima)
    if successor_scores[0][1] <= heuristic(current_state):
        print("Stuck at local maximum.")
        print_state(current_state)
        print(f"Total Steps: {steps}")
        return current_state

    # Choose the best successor and move to it
    current_state = successor_scores[0][0]
    print_state(current_state)

# Initial state setup for Tower of Hanoi (with 3 disks as an example)
start_state = [[3, 2, 1], [], []]  # All disks on Pole A initially

# Run the hill climbing algorithm
hill_climbing(start_state)
```

Output:

```
Pole A: [3, 2]
Pole B: []
Pole C: [1]
Pole A: [3]
Pole B: []
Pole C: [1, 2]
Pole A: []
Pole B: []
Pole C: [1, 2, 3]
Puzzle Solved!
Pole A: []
Pole B: []
Pole C: [1, 2, 3]
Total Steps: 4

=== Code Execution Successful ===
```

Conclusion:

The Hill Climbing algorithm, when applied to the Tower of Hanoi problem, attempts to improve the current state by selecting the best move at each step. While it may make progress toward the goal, it is not guaranteed to find the optimal solution due to the possibility of getting stuck in local optima. The algorithm is efficient in some cases, but for larger numbers of disks or more complex configurations, it may fail to reach the goal or might stop prematurely when no better moves are available. Thus, while Hill Climbing provides a simple and local search approach, it may not always be the best strategy for solving the Tower of Hanoi puzzle optimally.