22-10-2024 **WEEK - 05**

Implementing Hill climbing search algorithm to solve N-Queens Problem.

ALGORITHM / PSUEDOCODE -

classmate
Date 22|10|24
Page 13

WEEK-4

Implementing Hill climbing search algorithm to solve N-Queens problem.

ALGORITHM | PSUEDOCODE :

step 1 : Initialization
Define initial state by representing position of the 4 queens.

step 2 : define functions
define cost calculation and neighbour state functions

step 3 : Hill climbing function
initialize current state to initial state and calculate cost

step 4 : Iterative search
- generate neighbours and calculate cost
- if best neighbours has lower cost than current state update current state to best neighbour and cost

step 5 : execution
call hill climbing function with initial state and print result.

CODE-

```
import random

def calculate_cost(state):
    """Calculate the cost for a given state, which is the number of pairs of queens attacking each other."""
    attacking_pairs = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacking_pairs += 1
    return attacking_pairs

def get_neighbors(state):
    """Generate all possible neighboring states by swapping two queens' row positions."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = state[:]
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

def hill_climbing(initial_state):
    """Perform the hill climbing search algorithm and store the state space tree."""
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    state_tree = {tuple(current_state): current_cost}  # Dictionary to store state and cost

    step = 0
    while True:
        print(f"\nStep {step}:")
        neighbors = get_neighbors(current_state)

        print("Neighbors:")
        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            print(f"  {neighbor}: Cost = {cost}")
```

```
            state_tree[tuple(neighbor)] = cost  # Store neighbor state and cost

        best_neighbor = None
        best_cost = current_cost

        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:
                best_cost = cost
                best_neighbor = neighbor

        if best_cost >= current_cost:
            # No better neighbor found, return current state
            print(f"\nNo better neighbor found. Final state reached.")
            return current_state, current_cost, state_tree

        current_state = best_neighbor
        current_cost = best_cost
        step += 1

initial_state = [3, 1, 2, 0]

final_state, final_cost, state_space_tree = hill_climbing(initial_state)

print("\nInitial state:", initial_state)
print("Final state:", final_state)
print("Final cost (attacking pairs):", final_cost)
#print("\nState Space Tree (State: Cost):")
#for state, cost in state_space_tree.items():
#    print(f"{list(state)}: {cost}")
```

OUTPUT (Snapshot)-

```
Step 0:
Neighbors:
  [1, 3, 2, 0]: Cost = 1
  [2, 1, 3, 0]: Cost = 1
  [0, 1, 2, 3]: Cost = 6
  [3, 2, 1, 0]: Cost = 6
  [3, 0, 2, 1]: Cost = 1
  [3, 1, 0, 2]: Cost = 1

Step 1:
Neighbors:
  [3, 1, 2, 0]: Cost = 2
  [2, 3, 1, 0]: Cost = 2
  [0, 3, 2, 1]: Cost = 4
  [1, 2, 3, 0]: Cost = 4
  [1, 0, 2, 3]: Cost = 2
  [1, 3, 0, 2]: Cost = 0

Step 2:
Neighbors:
  [3, 1, 0, 2]: Cost = 1
  [0, 3, 1, 2]: Cost = 1
  [2, 3, 0, 1]: Cost = 4
  [1, 0, 3, 2]: Cost = 4
  [1, 2, 0, 3]: Cost = 1
  [1, 3, 2, 0]: Cost = 1

No better neighbor found. Final state reached.
```

```
Initial state: [3, 1, 2, 0]
Final state: [1, 3, 0, 2]
Final cost (attacking pairs): 0
```

STATE SPACE TREE (Snapshot) -