```python
import random

def cost(state):
    """Calculate the number of attacking pairs of queens in the current state."""
    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 print_board(state):
    """Represent the state as a 4x4 board."""
    n = len(state)
    board = [['.' for _ in range(n)] for _ in range(n)]
    for i in range(n):
        board[state[i]][i] = 'Q'

    for row in board:
        print(" ".join(row))

def get_neighbors(state):
    """Generate all possible neighbors by swapping two queens."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = list(state)
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  # Swap queens
            neighbors.append(tuple(neighbor))
    return neighbors

def hill_climbing(initial_state):
    """Hill climbing algorithm to solve the N-Queens problem."""
    current = initial_state
    print(f"Initial state:")
    print_board(current)
    print(f"Cost: {cost(current)}")
    print('-' * 20)

    while True:
        neighbors = get_neighbors(current)
        # Select the neighbor with the lowest cost
        next_state = min(neighbors, key=lambda x: cost(x))
```

```
        print(f"Next state:")
        print_board(next_state)
        print(f"Cost: {cost(next_state)}")
        print('-' * 20)

        if cost(next_state) >= cost(current):
            # If no better state is found, return the current state as
the solution
            print(f"Solution found:")
            print_board(current)
            print(f"Cost: {cost(current)}")
            return current
        current = next_state

if __name__ == "__main__":
    # Initial state for 4-Queens, random placement
    initial_state = (3, 1, 2, 0)  # Example initial state, where each
index represents a column

    # Run Hill Climbing algorithm
    solution = hill_climbing(initial_state)
```

→ Hill Climbing Search Algorithm

function Hill Climbing (problem) returns a state that is a local maximum curr ← MakeNode (problem.Initial-State)

loop do

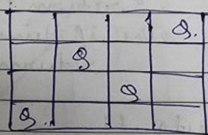
neighbor ← a highest-valued successor of curr if neighbor.value ≤ curr.value then

return curr.State.

curr ← neighbor.

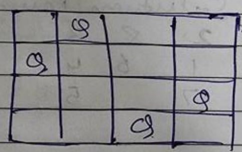
- $x_0 = 3,\ x_1 = 1,\ x_2 = 2,\ x_3 = 0$

cost = 2



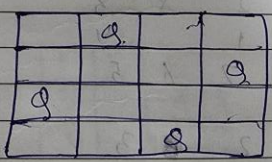
- $x_0 = 1,\ x_1 = 0\ ;\ x_2 = 3,\ x_3 = 2$

cost = 2 + 1 + 1 = 4



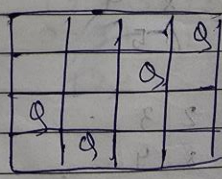
- $x_0 = 1,\ x_1 = 3,\ x_2 = 0,\ x_3 = 2$

$\boxed{C = 0}$



- $x_0 = 3,\ x_1 = 2,\ x_2 = 0,\ x_3 = 1$

C = 2



Hill Climbing Algorithm 1BM23CS316.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[1]

```python
initial_state = (3, 1, 2, 0)  # Example initial state, where each index represents a column

# Run Hill Climbing algorithm
solution = hill_climbing(initial_state)
```

```
Initial state:
. . . Q
. Q . .
. . Q .
Q . . .
Cost: 2
--------------------
Next state:
. . . Q
Q . . .
. . Q .
. Q . .
Cost: 1
--------------------
Next state:
. . Q .
Q . . .
. . . Q
. Q . .
Cost: 0
--------------------
Next state:
. . Q .
. Q . .
. . . Q
Q . . .
Cost: 1
--------------------
Solution found:
. . Q .
Q . . .
. . . Q
. Q . .
Cost: 0
```