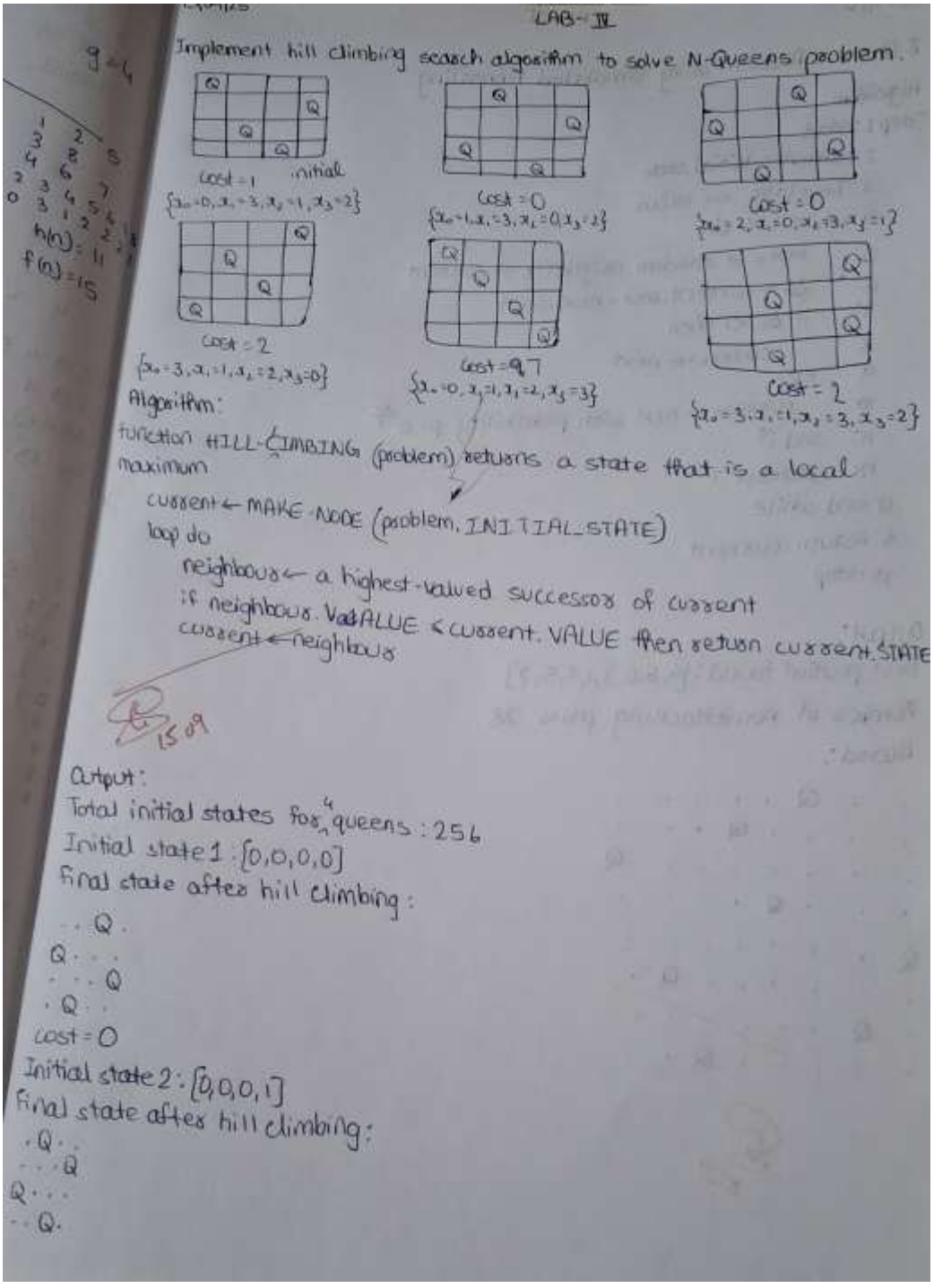
# Lab 4

Implement hill climbing search algorithm to solve N-Queens problem.

Algorithm:



LAB - II

Implement hill climbing search algorithm to solve N-Queens problem.

cost = 1 initial
{x0=0, x1=3, x2=1, x3=2}

cost = 0
{x0=1, x1=3, x2=0, x3=2}

cost = 0
{x0=2, x1=0, x2=3, x3=1}

cost = 2
{x0=3, x1=1, x2=2, x3=0}

cost = 97
{x0=0, x1=1, x2=2, x3=3}

cost = 2
{x0=3, x1=1, x2=3, x3=2}

Algorithm:

function HILL-CLIMBING (problem) returns a state that is a local maximum

current ← MAKE-NODE (problem. INITIAL_STATE)
loop do
neighbour ← a highest-valued successor of current
if neighbour. VALUE < current. VALUE then return current. STATE
current ← neighbour

1509

Output:

Total initial states for 4 queens : 256
Initial state 1 : [0,0,0,0]
final state after hill climbing :
. . Q .
Q . . .
. . . Q
. Q . .
cost = 0
Initial state 2 : [0,0,0,1]
final state after hill climbing :
. Q . .
. . . Q
Q . . .
. . Q .

Code:

```
import copy
print("Shreya Raj 1BM23CS317")
def print_board(state):
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            line += "Q " if state[col] == row else ". "
        print(line)
    print()

def heuristic(state):
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:  # same row
                attacks += 1
            if abs(state[i] - state[j]) == abs(i - j):  # same diagonal
                attacks += 1
    return attacks

def get_neighbors(state):
    neighbors = []
    n = len(state)
```

```
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(start_state):
    current = copy.deepcopy(start_state)
    while True:
        current_h = heuristic(current)
        if current_h == 0:
            return current, 0

        neighbors = get_neighbors(current)
        neighbor_h = [heuristic(neigh) for neigh in neighbors]

        min_h = min(neighbor_h)
        if min_h >= current_h:
            # No improvement possible
            return current, current_h

        current = neighbors[neighbor_h.index(min_h)]

def generate_all_states(n):
```

```
    states = []
    def backtrack(col=0, state=[]):
        if col == n:
            states.append(state.copy())
            return
        for row in range(n):
            state.append(row)
            backtrack(col+1, state)
            state.pop()
    backtrack()
    return states


if __name__ == "__main__":
    n = 4
    all_states = generate_all_states(n)

    print(f"Total initial states for {n} queens: {len(all_states)}\n")

    for i, start in enumerate(all_states, start=1):
        final_state, cost = hill_climbing(start)
        print(f"Initial state {i}: {start}")
        print(f"Final state after hill climbing:")
        print_board(final_state)
        print(f"Cost (heuristic): {cost}")
        print("="*30)
```

Output:

```
Shreya Raj 1BM23CS317
Total initial states for 4 queens: 256

Initial state 1: [0, 0, 0, 0]
Final state after hill climbing:
. . Q .
Q . . .
. . . Q
. Q . .

Cost (heuristic): 0
==============================
Initial state 2: [0, 0, 0, 1]
Final state after hill climbing:
. Q . .
. . . Q
Q . . .
. . Q .

Cost (heuristic): 0
==============================
Initial state 3: [0, 0, 0, 2]
Final state after hill climbing:
. . Q .
Q . . .
. . . Q
. Q . .

Cost (heuristic): 0
==============================
Initial state 4: [0, 0, 0, 3]
Final state after hill climbing:
. . Q .
Q . . .
. Q . .
. . . Q
```