HILL-CLIMBING SEARCH ALGORITHM

CODE:-

```
print(f"SHREYASGOWDA C (1BM23CS319)")

def print_board(state):
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

def calculate_cost(state):
    cost = 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]) == j - i:
                cost += 1
    return cost

def get_neighbors(state):
    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]
            neighbors.append((neighbor, (i, j)))
    return neighbors

def hill_climbing(initial_state):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    print("Start State:")
    print_board(current_state)
    print(f"Cost: {current_cost}\n")

    path = [(current_state, current_cost, None)]

    while True:
```

HILL-CLIMBING SEARCH ALGORITHM

```
        neighbors = get_neighbors(current_state)
        neighbor_costs = [(tuple(neighbor), calculate_cost(neighbor), swap) for neighbor, swap in
neighbors]

        neighbor_costs.sort(key=lambda x: (x[1], x[2]))

        best_neighbor, best_cost, best_swap = neighbor_costs[0]

        print(f"Neighbors of {current_state} with costs:")
        for neighbor, cost, swap in neighbor_costs:
            print(f"Swap columns {swap}:")
            print_board(neighbor)
            print(f"Cost: {cost}\n")

        if best_cost < current_cost:
            print(f"Moving to better neighbor by swapping columns {best_swap}:")
            print_board(best_neighbor)
            print(f"Cost: {best_cost}\n")
            current_state, current_cost = best_neighbor, best_cost
            path.append((current_state, current_cost, best_swap))
        else:
            print("/nReached goal state.")
            break

    return path

def get_initial_state():
    print("Enter the initial positions of the 4 queens (row for each column, 0-indexed):")
    positions = []
    for col in range(4):
        while True:
            try:
                pos = int(input(f"Column {col}: "))
                if 0 <= pos < 4:
                    positions.append(pos)
                    break
                else:
                    print("Invalid input. Enter a number between 0 and 3.")
            except ValueError:
                print("Invalid input. Please enter an integer.")
    return tuple(positions)


initial_state = get_initial_state()
```

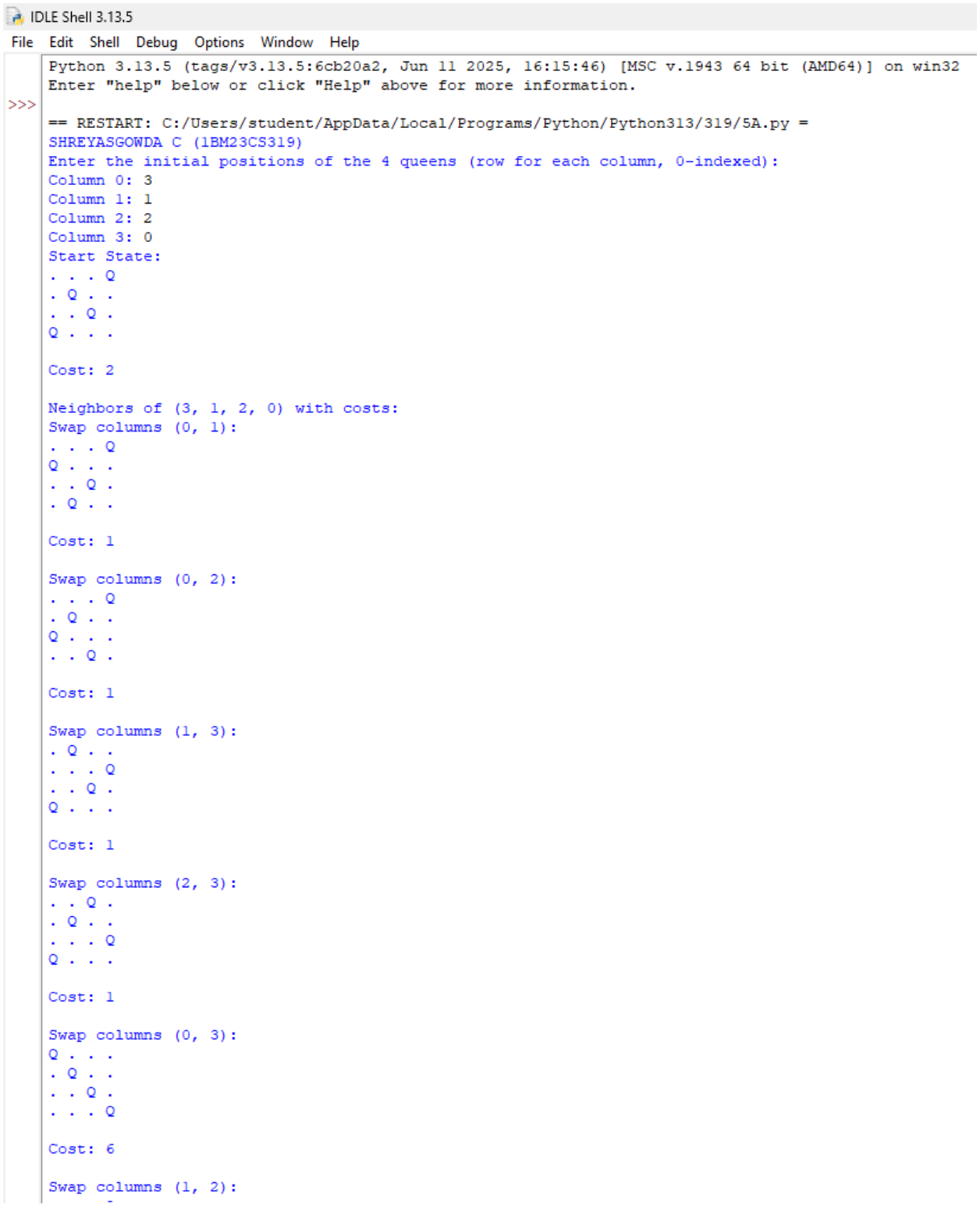
HILL-CLIMBING SEARCH ALGORITHM

```
path = hill_climbing(initial_state)

print("Final path:")
for i, (state, cost, swap) in enumerate(path):
    print(f"Step {i}:")
    print_board(state)
    print(f"Cost: {cost}")
    if swap is not None:
        print(f"Swap columns: {swap}")
    print("--------------------")
```

OUTPUT:-

HILL-CLIMBING SEARCH ALGORITHM



HILL-CLIMBING SEARCH ALGORITHM