**LAB – 4**

**1) Implement Hill Climbing search algorithm to solve N-Queens problem**

**import random**

def calculate\_cost(state):

"""Calculate the number of attacking queen pairs."""

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]) == abs(i - j):

cost += 1

return cost

def generate\_neighbors(state):

"""Generate all neighbors by moving one queen in each column to other rows."""

neighbors = []

n = len(state)

for col in range(n):

for row in range(n):

if row != state[col]:

neighbor = state.copy()

neighbor[col] = row

neighbors.append(neighbor)

return neighbors

def hill\_climbing(n):

"""Hill Climbing algorithm to solve N-Queens."""

current\_state = [random.randint(0, n - 1) for \_ in range(n)]

current\_cost = calculate\_cost(current\_state)

while True:

neighbors = generate\_neighbors(current\_state)

neighbor\_costs = [(neighbor, calculate\_cost(neighbor)) for neighbor in neighbors]

best\_neighbor, best\_cost = min(neighbor\_costs, key=lambda x: x[1])

if best\_cost >= current\_cost:

break

current\_state, current\_cost = best\_neighbor, best\_cost

if current\_cost == 0:

break

return current\_state, current\_cost

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()

# Example usage:

N = 4

solution, cost = hill\_climbing(N)

if cost == 0:

print("Solution found:")

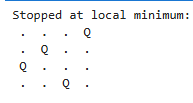
else:

print("Stopped at local minimum:")

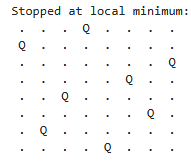
print\_board(solution)

**Output:**

**For 4 queens –**

****

**For 8 queens-**

****

**2)Write a program to implement Simulated Annealing Algorithm**

import random

def calculate\_cost(state):

"""Calculate the number of attacking queen pairs."""

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]) == abs(i - j):

cost += 1

return cost

def generate\_neighbors(state):

"""Generate all neighbors by moving one queen in each column to other rows."""

neighbors = []

n = len(state)

for col in range(n):

for row in range(n):

if row != state[col]:

neighbor = state.copy()

neighbor[col] = row

neighbors.append(neighbor)

return neighbors

def hill\_climbing(n):

"""Hill Climbing algorithm to solve N-Queens."""

current\_state = [random.randint(0, n - 1) for \_ in range(n)]

current\_cost = calculate\_cost(current\_state)

while True:

neighbors = generate\_neighbors(current\_state)

neighbor\_costs = [(neighbor, calculate\_cost(neighbor)) for neighbor in neighbors]

best\_neighbor, best\_cost = min(neighbor\_costs, key=lambda x: x[1])

if best\_cost >= current\_cost:

break

current\_state, current\_cost = best\_neighbor, best\_cost

if current\_cost == 0:

break

return current\_state, current\_cost

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()

N = 8

solution, cost = hill\_climbing(N)

if cost == 0:

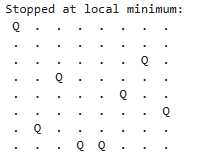
print("Solution found:")

else:

print("Stopped at local minimum:")

print\_board(solution)

**Output:**

****