Week 4

import random

# Heuristic: number of pairs of queens attacking each other

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]) == abs(i - j):

cost += 1

return cost

# Generate all neighbors of the current state

def get\_neighbors(state):

neighbors = []

n = len(state)

for col in range(n):

for row in range(n):

if state[col] != row: # move queen in col to new row

neighbor = state.copy()

neighbor[col] = row

neighbors.append(neighbor)

return neighbors

# Hill climbing algorithm

def hill\_climb(initial\_state):

current = initial\_state

current\_cost = calculate\_cost(current)

print(f"Initial state: {current}, Cost = {current\_cost}")

while True:

neighbors = get\_neighbors(current)

best\_neighbor = None

best\_cost = current\_cost

# Find the best neighbor

for neighbor in neighbors:

cost = calculate\_cost(neighbor)

if cost < best\_cost:

best\_cost = cost

best\_neighbor = neighbor

# If no better neighbor is found → stop

if best\_neighbor is None:

print(f"Final state: {current}, Cost = {current\_cost}")

return current, current\_cost

# Move to the better neighbor

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

print(f"Move to: {current}, Cost = {current\_cost}")

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

n = int(input("Enter number of queens (N): "))

print("Enter initial state as space-separated row positions for each column.")

print("Example for N=4: '1 3 0 2' means queen at (0,1), (1,3), (2,0), (3,2).")

initial\_state = list(map(int, input("Initial state: ").split()))

if len(initial\_state) != n:

print("Invalid input: Length of initial state must be N.")

else:

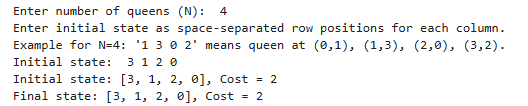
solution, cost = hill\_climb(initial\_state)

if cost == 0:

print("Goal state reached!")

else:

print("Stuck in local minimum.")  
  
Output:



import random

import math

def cost(state, N):

"""Compute number of attacking queen pairs."""

conflicts = 0

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):

conflicts += 1

return conflicts

def random\_neighbor(state, N):

"""Generate a neighbor by moving one queen to another row in a random column."""

neighbor = state.copy()

col = random.randrange(N)

new\_row = random.randrange(N-1)

if new\_row >= neighbor[col]:

new\_row += 1

neighbor[col] = new\_row

return neighbor

def simulated\_annealing(N, T0=5.0, alpha=0.995, Tmin=1e-6, max\_iters=50000):

"""Solve N-Queens using simulated annealing."""

# Random initial state: one queen per column

state = [random.randrange(N) for \_ in range(N)]

current\_cost = cost(state, N)

T = T0

it = 0

while T > Tmin and it < max\_iters and current\_cost != 0:

neighbor = random\_neighbor(state, N)

neighbor\_cost = cost(neighbor, N)

delta = neighbor\_cost - current\_cost

if delta <= 0 or random.random() < math.exp(-delta / T):

state, current\_cost = neighbor, neighbor\_cost

T \*= alpha

it += 1

return state, current\_cost

def print\_board(state, N):

"""Pretty-print the board with row and column numbers."""

print(" " + " ".join(str(c) for c in range(N))) # column indices

for r in range(N):

row = f"{r} " # row index

for c in range(N):

row += "Q " if state[c] == r else ". "

print(row)

print()

# --------------------------

# User input

# --------------------------

N = int(input("Enter number of queens (N): "))

solution, c = simulated\_annealing(N)

print(f"\nFinal state (col -> row): {solution}")

print("Cost:", c)

print("\nBoard:")

print\_board(solution, N)  
  
Output:

