Lab 4

Hill climbing: 4 queens

CODE:

import random

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] or abs(state[i] - state[j]) == abs(i - j):

attacks += 1

return attacks

def get\_neighbors(state):

neighbors = []

n = len(state)

for col in range(n):

for row in range(n):

if row != state[col]:

new\_state = state[:]

new\_state[col] = row

neighbors.append(new\_state)

return neighbors

def hill\_climb\_verbose(state):

print(f"Initial: {state} | Cost: {heuristic(state)}\n")

while True:

h\_current = heuristic(state)

neighbors = get\_neighbors(state)

costs = [heuristic(n) for n in neighbors]

for s, c in zip(neighbors, costs):

print(f"Neighbor: {s} | Cost: {c}")

min\_cost = min(costs)

best\_neighbor = neighbors[costs.index(min\_cost)]

print(f"\nBest Neighbor: {best\_neighbor} | Cost: {min\_cost}\n")

if min\_cost >= h\_current:

print("Stopped: Local Minimum or Goal Reached.")

print(f"Final State: {state} | Cost: {h\_current}")

break

state = best\_neighbor

if heuristic(state) == 0:

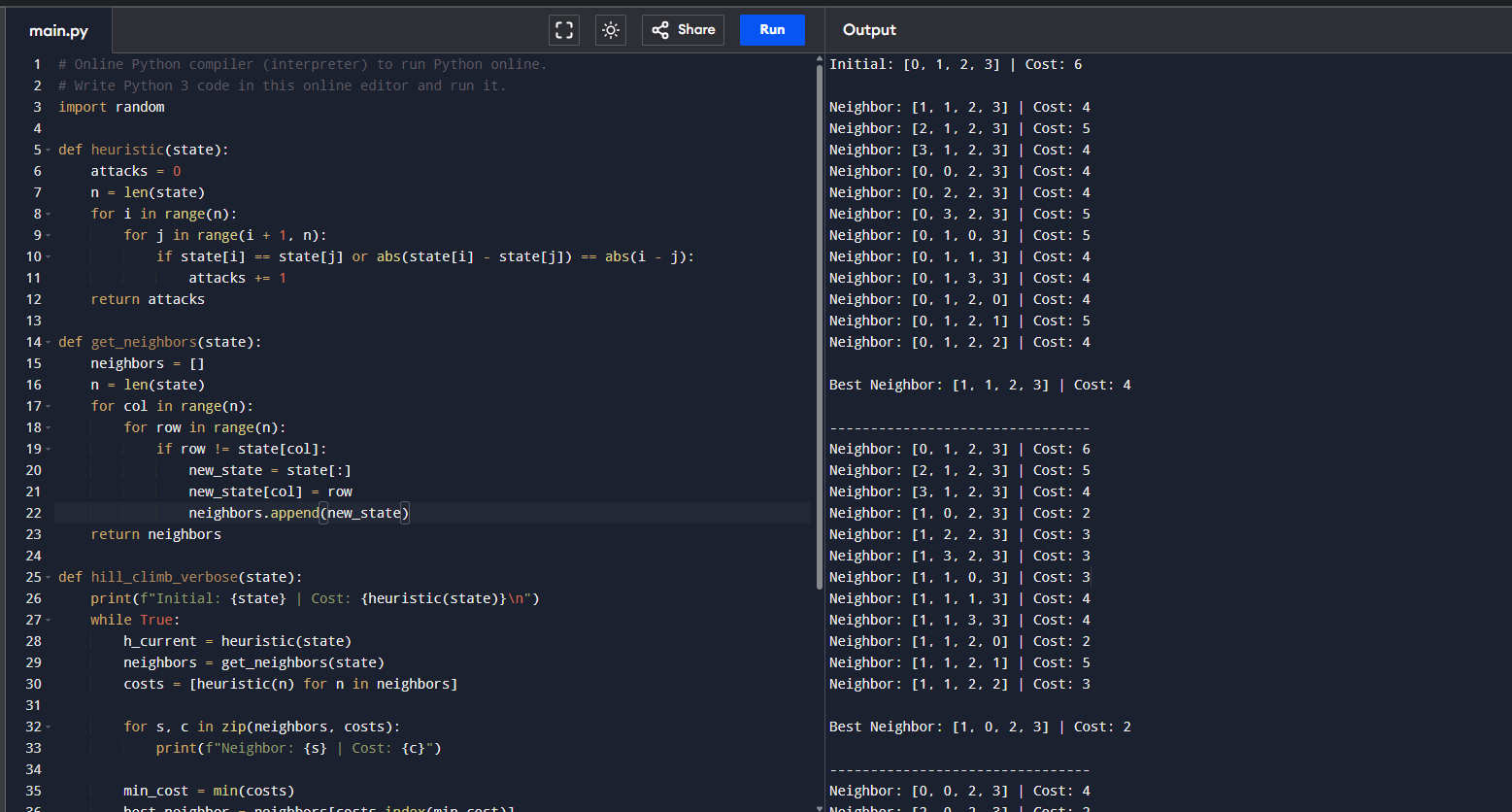
print(f"✅ Goal Reached! State: {state}")

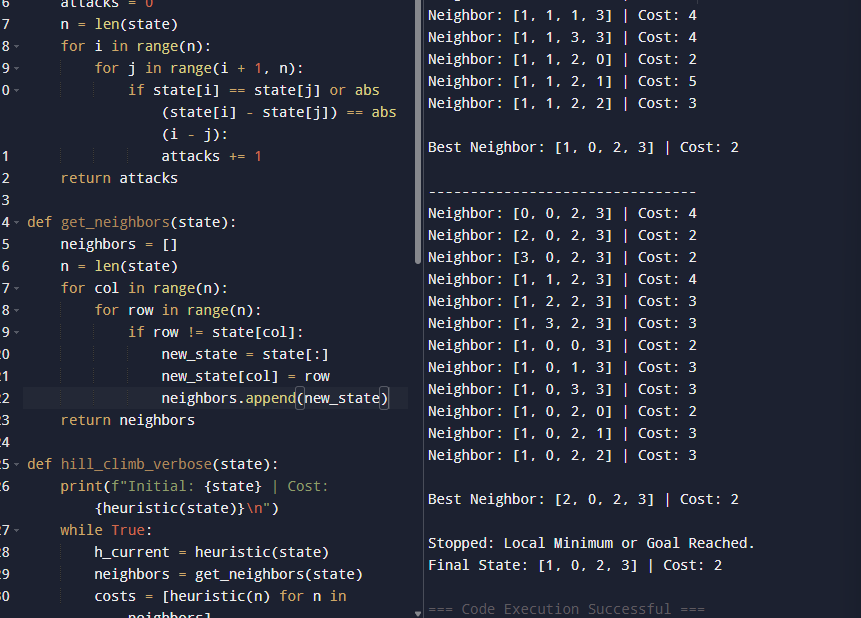
break

print("--------------------------------")

# Run with fixed start

hill\_climb\_verbose([0,1,2,3])

OUTPUT:  
****

****

**SIMULATED ANNEALING: - 4 QUEENS**import random

import math

# Cost function: number of attacking pairs

def cost(state):

attacks = 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):

attacks += 1

return attacks

# Generate neighbor by moving one queen to a new column

def neighbor(state):

n = len(state)

new\_state = state.copy()

row = random.randint(0, n - 1)

col = random.randint(0, n - 1)

while col == new\_state[row]:

col = random.randint(0, n - 1)

new\_state[row] = col

return new\_state

# Simulated Annealing

def simulated\_annealing(n=4, T=1000, alpha=0.95, max\_iter=1000):

# Initial state

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

current\_cost = cost(current)

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

for i in range(max\_iter):

if current\_cost == 0:

print("Solution found!")

break

next\_state = neighbor(current)

next\_cost = cost(next\_state)

print(f"Neighbor: {next\_state}, cost: {next\_cost}")

delta = next\_cost - current\_cost

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

current = next\_state

current\_cost = next\_cost

print(f"Move accepted: {current}, cost: {current\_cost}")

else:

print("Move rejected")

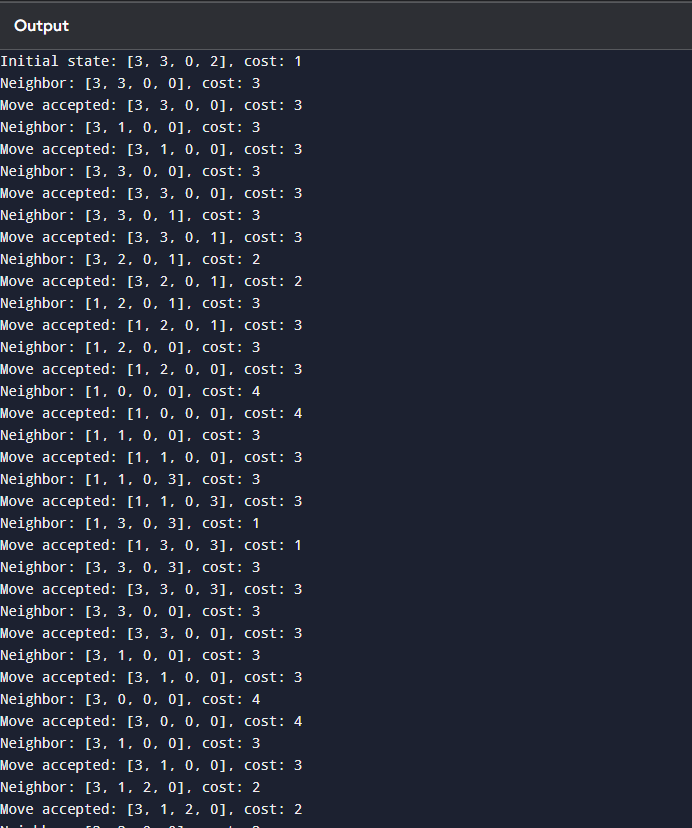
T \*= alpha # Cooling

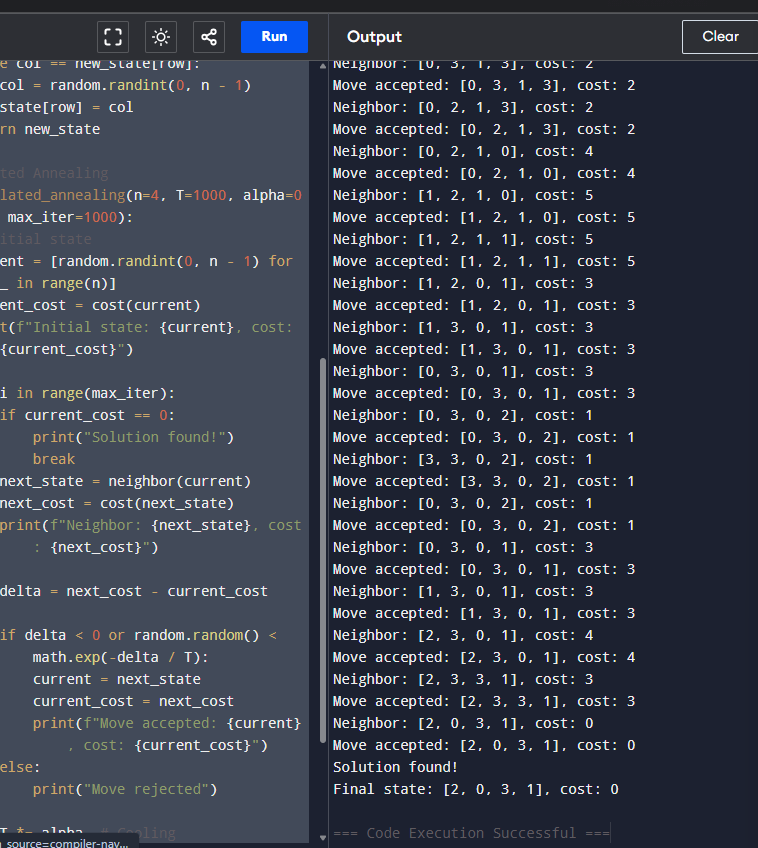
print(f"Final state: {current}, cost: {current\_cost}")

# Run

simulated\_annealing()

**OUTPUT**

****

****

SIMULATED ANNEALING 8 QUEENS:  
import random

import math

# ---------- Helper Functions ----------

def random\_state(n=8):

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

def heuristic(state):

"""Number of attacking pairs of queens (lower = better)."""

attacks = 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):

attacks += 1

return attacks

def random\_neighbor(state):

"""Move one queen to a new random row."""

n = len(state)

neighbor = state[:]

col = random.randint(0, n - 1)

new\_row = random.randint(0, n - 1)

while new\_row == neighbor[col]:

new\_row = random.randint(0, n - 1)

neighbor[col] = new\_row

return neighbor

# ---------- Simulated Annealing ----------

def simulated\_annealing(n=8, initial\_temp=100.0, cooling\_rate=0.99):

current = random\_state(n)

T = initial\_temp

print("Initial State:", current)

print("Initial Cost:", heuristic(current))

print("--------------------------------------")

while T > 0.001:

h\_current = heuristic(current)

if h\_current == 0:

break

neighbor = random\_neighbor(current)

h\_neighbor = heuristic(neighbor)

deltaE = h\_current - h\_neighbor

if deltaE > 0 or math.exp(deltaE / T) > random.random():

current = neighbor

T \*= cooling\_rate

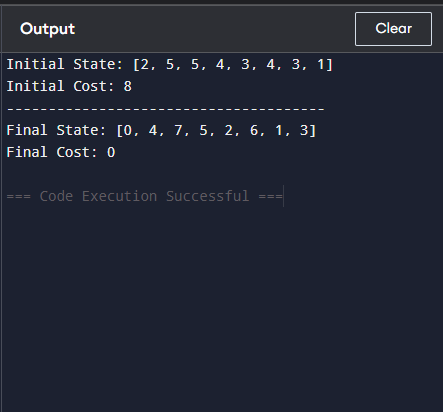
print("Final State:", current)

print("Final Cost:", heuristic(current))

# ---------- MAIN ----------

simulated\_annealing(8)

OUTPUT:



A\* 8 PUZZLE

from heapq import heappush, heappop

# Goal state

goal\_state = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]] # 0 represents the empty space

# Function to find position of a number in the puzzle

def find\_pos(state, value):

for i in range(3):

for j in range(3):

if state[i][j] == value:

return (i, j)

# Heuristic: Manhattan distance

def manhattan(state):

dist = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0:

x, y = find\_pos(goal\_state, state[i][j])

dist += abs(x - i) + abs(y - j)

return dist

# Convert list of lists to a tuple (so it can be stored in sets)

def state\_to\_tuple(state):

return tuple(sum(state, []))

# Generate next possible moves

def get\_neighbors(state):

neighbors = []

i, j = find\_pos(state, 0) # blank position

moves = [(-1,0), (1,0), (0,-1), (0,1)] # up, down, left, right

for dx, dy in moves:

x, y = i + dx, j + dy

if 0 <= x < 3 and 0 <= y < 3:

new\_state = [row[:] for row in state]

new\_state[i][j], new\_state[x][y] = new\_state[x][y], new\_state[i][j]

neighbors.append(new\_state)

return neighbors

# A\* algorithm

def a\_star(start):

open\_list = []

heappush(open\_list, (manhattan(start), 0, start, [])) # (f, g, state, path)

visited = set()

while open\_list:

f, g, current, path = heappop(open\_list)

if current == goal\_state:

return path + [current]

visited.add(state\_to\_tuple(current))

for neighbor in get\_neighbors(current):

if state\_to\_tuple(neighbor) not in visited:

heappush(open\_list, (g + 1 + manhattan(neighbor), g + 1, neighbor, path + [current]))

return None

# Print the puzzle state nicely

def print\_state(state):

for row in state:

print(row)

print()

# Example initial state (you can change this)

start\_state = [[1, 2, 3],

[4, 0, 6],

[7, 5, 8]]

print("Initial State:")

print\_state(start\_state)

print("Solving...\n")

solution = a\_star(start\_state)

if solution:

print("Solution found in", len(solution)-1, "moves:\n")

for step in solution:

print\_state(step)

else:

print("No solution found.")

OUTPUT:

