# TASK 1:

import numpy as np

import random

import math

import networkx as nx

import matplotlib.pyplot as plt

# Coordinates of the points/cities

coordinates = np.array([[1, 2], [1, 4], [2, 6], [7, 3], [8, 1], [8, 12], [13, 24], [15, 16], [20, 5], [18, 4], [21, 22]])

# Function to calculate distance between two points

def distance(point1, point2):

    return np.linalg.norm(point1 - point2)

# Function to calculate total distance of a path

def total\_distance(path):

    total = 0

    for i in range(len(path)-1):

        total += distance(coordinates[path[i]], coordinates[path[i+1]])

    total += distance(coordinates[path[-1]], coordinates[path[0]])  # Closing the loop

    return total

# Function to generate initial random path

def generate\_random\_path(num\_cities):

    return random.sample(range(num\_cities), num\_cities)

# Simple Hill Climbing algorithm

def simple\_hill\_climbing(path):

    improved = True

    while improved:

        improved = False

        for i in range(len(path)):

            for j in range(i + 1, len(path)):

                new\_path = path[:]

                new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

                if total\_distance(new\_path) < total\_distance(path):

                    path = new\_path

                    improved = True

                    break

            if improved:

                break

    return path

# Hill Climbing (First Search) variant

def hill\_climbing\_first\_search(path):

    improved = True

    while improved:

        improved = False

        for i in range(len(path)):

            for j in range(i + 1, len(path)):

                new\_path = path[:]

                new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

                if total\_distance(new\_path) < total\_distance(path):

                    path = new\_path

                    improved = True

                    break

            if improved:

                break

    return path

# Hill Climbing (Random Restart) variant

def hill\_climbing\_random\_restart(path, max\_iterations):

    best\_path = path

    best\_distance = total\_distance(path)

    for \_ in range(max\_iterations):

        new\_path = generate\_random\_path(len(path))

        new\_distance = total\_distance(new\_path)

        if new\_distance < best\_distance:

            best\_path = new\_path

            best\_distance = new\_distance

    return best\_path

# Hill Climbing (Stochastic) variant

def hill\_climbing\_stochastic(path, max\_iterations):

    current\_path = path

    current\_distance = total\_distance(path)

    for \_ in range(max\_iterations):

        i, j = random.sample(range(len(path)), 2)

        new\_path = current\_path[:]

        new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

        new\_distance = total\_distance(new\_path)

        if new\_distance < current\_distance:

            current\_path = new\_path

            current\_distance = new\_distance

    return current\_path

# Simulated Annealing algorithm

def simulated\_annealing(path, temperature=10000, cooling\_rate=0.003, min\_temperature=1):

    current\_path = path

    current\_distance = total\_distance(path)

    while temperature > min\_temperature:

        i, j = random.sample(range(len(path)), 2)

        new\_path = current\_path[:]

        new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

        new\_distance = total\_distance(new\_path)

        if new\_distance < current\_distance or random.random() < math.exp((current\_distance - new\_distance) / temperature):

            current\_path = new\_path

            current\_distance = new\_distance

        temperature \*= 1 - cooling\_rate

    return current\_path

# Local Beam Search algorithm

def local\_beam\_search(k, num\_iterations):

    current\_paths = [generate\_random\_path(len(coordinates)) for \_ in range(k)]

    for \_ in range(num\_iterations):

        new\_paths = []

        for path in current\_paths:

            for i in range(len(path)):

                for j in range(i + 1, len(path)):

                    new\_path = path[:]

                    new\_path[i], new\_path[j] = new\_path[j], new\_path[i]

                    new\_paths.append(new\_path)

        new\_paths.sort(key=total\_distance)

        current\_paths = new\_paths[:k]

    return current\_paths[0]

# Function to plot the path

def plot\_path(path):

    G = nx.Graph()

    for i in range(len(path)-1):

        G.add\_edge(path[i], path[i+1], weight=distance(coordinates[path[i]], coordinates[path[i+1]]))

    G.add\_edge(path[-1], path[0], weight=distance(coordinates[path[-1]], coordinates[path[0]]))  # Closing the loop

    pos = {i: coordinates[i] for i in range(len(coordinates))}

    nx.draw(G, pos, with\_labels=True, node\_color='skyblue', node\_size=700)

    labels = nx.get\_edge\_attributes(G, 'weight')

    nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels)

    plt.title("Optimal Path")

    plt.show()

# Simple Hill Climbing

initial\_path = generate\_random\_path(len(coordinates))

optimal\_path\_simple\_hill\_climbing = simple\_hill\_climbing(initial\_path)

print("Simple Hill Climbing Optimal Path:", optimal\_path\_simple\_hill\_climbing)

print("Total Distance:", total\_distance(optimal\_path\_simple\_hill\_climbing))

# Hill Climbing (First Search)

initial\_path = generate\_random\_path(len(coordinates))

optimal\_path\_hill\_climbing\_first\_search = hill\_climbing\_first\_search(initial\_path)

print("Hill Climbing (First Search) Optimal Path:", optimal\_path\_hill\_climbing\_first\_search)

print("Total Distance:", total\_distance(optimal\_path\_hill\_climbing\_first\_search))

# Hill Climbing (Random Restart)

initial\_path = generate\_random\_path(len(coordinates))

optimal\_path\_hill\_climbing\_random\_restart = hill\_climbing\_random\_restart(initial\_path, 100)

print("Hill Climbing (Random Restart) Optimal Path:", optimal\_path\_hill\_climbing\_random\_restart)

print("Total Distance:", total\_distance(optimal\_path\_hill\_climbing\_random\_restart))

# Hill Climbing (Stochastic)

initial\_path = generate\_random\_path(len(coordinates))

optimal\_path\_hill\_climbing\_stochastic = hill\_climbing\_stochastic(initial\_path, 1000)

print("Hill Climbing (Stochastic) Optimal Path:", optimal\_path\_hill\_climbing\_stochastic)

print("Total Distance:", total\_distance(optimal\_path\_hill\_climbing\_stochastic))

# Simulated Annealing

initial\_path = generate\_random\_path(len(coordinates))

optimal\_path\_simulated\_annealing = simulated\_annealing(initial\_path)

print("Simulated Annealing Optimal Path:", optimal\_path\_simulated\_annealing)

print("Total Distance:", total\_distance(optimal\_path\_simulated\_annealing))

# Local Beam Search

k = 5  # Beam width

num\_iterations = 100

optimal\_path\_local\_beam\_search = local\_beam\_search(k, num\_iterations)

print("Local Beam Search Optimal Path:", optimal\_path\_local\_beam\_search)

print("Total Distance:", total\_distance(optimal\_path\_local\_beam\_search))

# Plotting optimal path using NetworkX

print("Plotting Optimal Path...")

plot\_path(optimal\_path\_simple\_hill\_climbing)

# OUTPUT:



