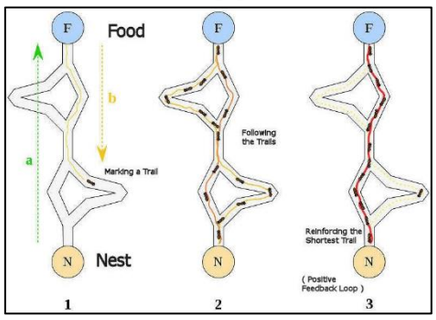
Universidad Distrital Francisco José de Caldas
Sebastian Pardo Tirado – 20221020121
Análisis de Sistemas

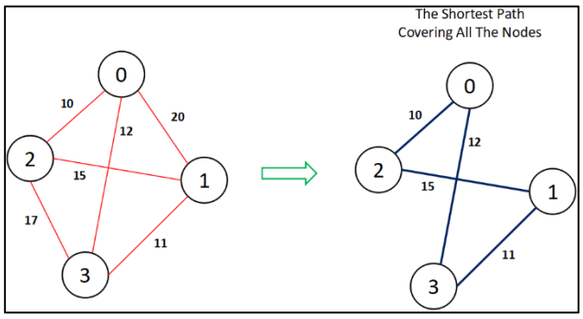
Workshop II - Swarm Intelligence and Sinergy: Ant Colony for the Traveling Salesman Problem

1) **Create some diagrams and explanations to represent/understand the problem following a systems thinking approach.**

ACO



TSP



URLs:

- https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms
- https://iq.opengenus.org/travelling-salesman-problem-brute-force/

The Traveling Salesman Problem involves finding the shortest possible route that visits each city exactly once and returns to the original city. This problem is crucial in logistics and optimization. Using a systems thinking approach, we consider the cities as nodes and the connections between them as edges in a graph. The goal is to optimize the path taken by the salesman.

2) **Generate several random 3D space points in order to define cities to be visited by the salesman.**

```python
import numpy as np

def generate_cities(number_cities: int) -> list:
    """
    This function generates a list of cities with random coordinates in 3D space.

    Parameters:
    - number_cities (int): Number of cities to generate.

    Returns:
    - list: A list of cities with random coordinates.
    """
    # HERE Generate random 3D Points using numpy generator
    cities = np.random.rand(number_cities, 3)
    return cities
```

The function, generate_cities, generates a list of cities with random coordinates in 3D space. It takes an integer parameter number_cities specifying the number of cities to generate and returns a list of cities represented by their random coordinates.

3) **With random points, generate a list in order to define the requirements for the route.**

```python
def calculate_distance(point_1: np.array, point_2: np.array) -> float:
    """
    This function calculates the Euclidean distance between two points.

    Parameters:
    - point_1 (np.array): First point.
    - point_2 (np.array): Second point.

    Returns:
    - float: The Euclidean distance between the two points.
    """
    # HERE return distance between two points using euclidean distance formula
    return np.linalg.norm(point_1 - point_2)
```

The function, calculate_distance, computes the Euclidean distance between two points given as numpy arrays. It takes two numpy arrays point_1 and point_2 representing the coordinates of the two points and returns a float value representing the Euclidean distance between them using the numpy function np.linalg.norm.

This function and the previous one include docstrings providing descriptions of their purpose, parameters, and return values, enhancing readability and maintainability.

4) Implement ACO in order to solve TSP problem. Test different parameters combination.

```python
def ant_colony_optimization(
    cities, n_ants, n_iterations, alpha, beta, evaporation_rate, Q
):
    """
    This function solves the Traveling Salesman Problem using Ant Colony Optimization.

    Parameters:
    - cities (list): List of cities.
    - n_ants (int): Number of ants.
    - n_iterations (int): Number of iterations.
    - alpha (float): It determines how much the ants are influenced by the pheromone trails left by other ants.
    - beta (float):  It determines how much the ants are influenced by the distance to the next city
    - evaporation_rate (float): Evaporation rate.
    - Q (float): It determines the intensity of the pheromone trail left behind by an ant.
    """
    # HERE Get number of points
    number_cities = len(cities)
    # HERE Initialize pheromone matrix with ones
    pheromone = np.ones((number_cities, number_cities))

    # initialize output metrics
    best_path = None
    best_path_length = np.inf

    # per each iteration the ants will build a path
    for iteration in range(n_iterations):
        paths = []  # store the paths of each ant
        path_lengths = []

        for ant in range(n_ants):
            visited = [False] * number_cities

            # you could start from any city, but let's start from a random one
            current_city = np.random.randint(number_cities)
            visited[current_city] = True
            path = [current_city]
            path_length = 0

            while False in visited:  # while there are unvisited cities
                unvisited = np.where(np.logical_not(visited))[0]
                probabilities = np.zeros(len(unvisited))

                # based on pheromone, distance and alpha and beta parameters, define the preference
                # for an ant to move to a city
                for i, unvisited_city in enumerate(unvisited):
```

```
                    # HERE add equiation to calculate the probability of moving to a city based on pheromone, distance and alpha and beta parameters
                    probabilities[i] = (pheromone[current_city, unvisited_city] ** alpha) * (1 / calculate_distance(cities[current_city], cities[unvisited_city]) ** beta)
                    # normalize probabilities, it means, the sum of all probabilities is 1
                    # HERE add normalization for calculated probabilities
                    probabilities /= np.sum(probabilities)
                    next_city = np.random.choice(unvisited, p=probabilities)
                    path.append(next_city)
                    # increase the cost of move through the path
                    path_length += calculate_distance(
                        cities[current_city], cities[next_city]
                    )
                    visited[next_city] = True
                    # move to the next city, for the next iteration
                    current_city = next_city

            paths.append(path)
            path_lengths.append(path_length)

            # update with current best path, this is a minimization problem
            if path_length < best_path_length:
                best_path = path
                best_path_length = path_length

        # remove a bit of pheromone of all map, it's a way to avoid local minima
        pheromone *= evaporation_rate

        # current ant must add pheromone to the path it has walked
        for path, path_length in zip(paths, path_lengths):
            for i in range(number_cities - 1):
                pheromone[path[i], path[i + 1]] += Q / path_length
            pheromone[path[-1], path[0]] += Q / path_length

    return best_path, best_path_length
```

This code fragment implements the Ant Colony Optimization (ACO) algorithm to solve the Traveling Salesman Problem (TSP). The ant_colony_optimization function takes several parameters: cities (a list of cities), n_ants (number of ants), n_iterations (number of iterations), alpha (influence of pheromone), beta (influence of distance), evaporation_rate (rate at which pheromone evaporates), and Q (intensity of pheromone left by an ant).

- Initially, the function calculates the number of cities and initializes a pheromone matrix with ones, representing equal initial pheromone levels on all paths.
- Then, for each iteration, ants construct paths by probabilistically selecting the next city based on pheromone levels and distances to unvisited cities, biased by the parameters alpha and beta.
- Within the loop for each ant, the algorithm selects the next city probabilistically and updates the path and its length accordingly.
- Once all ants complete their paths, pheromone evaporation is applied to encourage exploration, and ants deposit pheromone on their paths inversely proportional to their length.
- The algorithm iterates until n_iterations are completed, and the best path found during the iterations (the path with the shortest length) is returned along with its length.

The function aims to converge to a near-optimal solution by balancing exploitation of known good paths (pheromone) and exploration of new paths.

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1

# HERE create list of cities
cities = generate_cities(number_cities)

# HERE call ant_colony_optimization function
best_path, best_path_length = ant_colony_optimization(
    cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q
)
```

This code fragment sets up model parameters for the Ant Colony Optimization (ACO) algorithm to solve the Traveling Salesman Problem (TSP). It initializes parameters such as the number of cities, ants, iterations, alpha, beta, evaporation rate, and Q. Then, it generates a list of cities with random coordinates in 3D space using the generate_cities function. Finally, it calls the ant_colony_optimization function with the provided parameters to find the best path and its length.

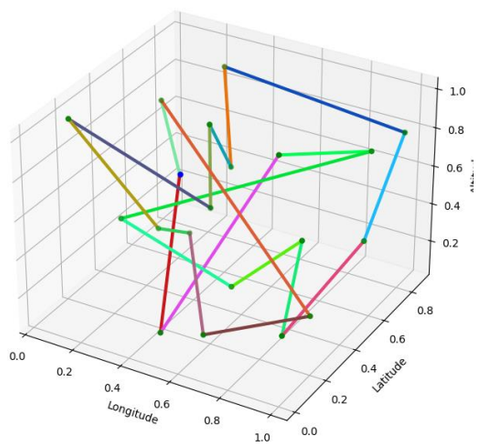
To test the impact of parameter changes, various values can be assigned to parameters such as number_cities, number_ants, number_iterations, alpha, beta, evaporation_rate, and Q. For instance, increasing the number of ants (number_ants) might lead to more exploration of the solution space, potentially resulting in finding better paths.

Similarly, adjusting the alpha and beta values can alter the balance between exploiting pheromone trails and considering distances, potentially affecting the convergence and quality of the solution. Varying the evaporation rate and Q parameter can influence the intensity and persistence of pheromone trails, impacting the exploration-exploitation trade-off and the convergence behavior of the algorithm.

By systematically testing these parameters, we can observe how they influence the algorithm's performance and find optimal parameter settings for the given problem instance.

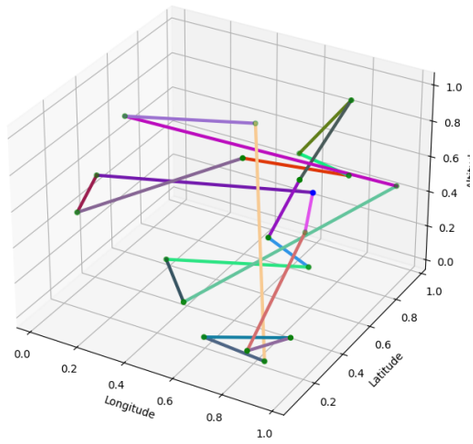
- ***Test #1 (Varying Number of Ants):***

```python
# model parameters
number_cities = 20
number_ants = 50
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1
```



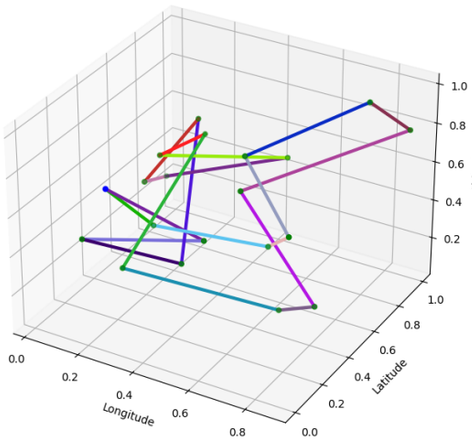
Best path length: 10.622322931330135

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1
```



Best path length: 9.27924884936941

```python
# model parameters
number_cities = 20
number_ants = 200
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1
```



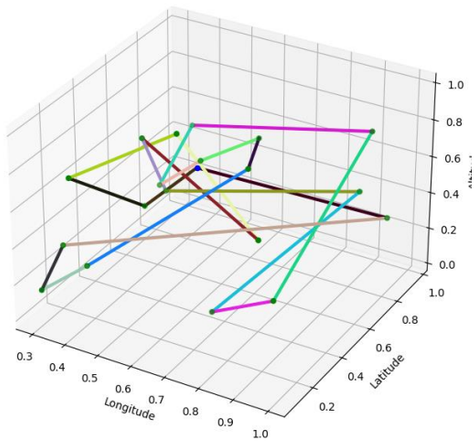
Best path length: 8.142879556251511

Analysis:

Increasing the number of ants might lead to more exploration of the solution space as more ants are available to search for paths. Therefore, we can expect that with more ants, the algorithm could potentially find shorter paths. However, it might also increase the computational complexity and time required for convergence.

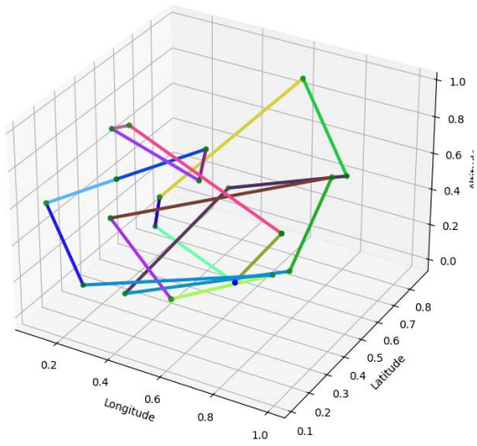
- ***Test #2 (Varying Alpha and Beta):***

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 0.5
beta = 0.5
evaporation_rate = 0.5
Q = 1
```



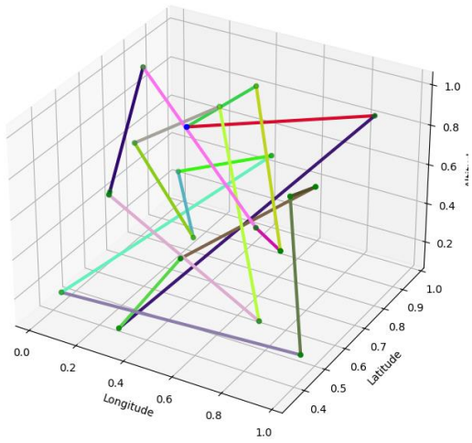
Best path length: 8.76187084725021

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1
```



Best path length: 9.462594354042967

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 2
beta = 2
evaporation_rate = 0.5
Q = 1
```



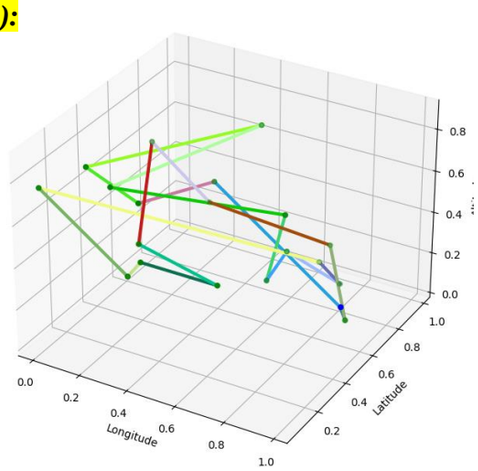
Best path length: 10.778012714386954

Analysis:

The parameters alpha and beta influence how much the ants are influenced by pheromone trails and distance, respectively. Lower values of alpha and beta would prioritize distance over pheromone trails, while higher values would prioritize pheromone trails over distance. By varying these parameters, we can observe how the balance between exploitation of pheromone trails and exploration of distances affects the convergence and quality of the solution.

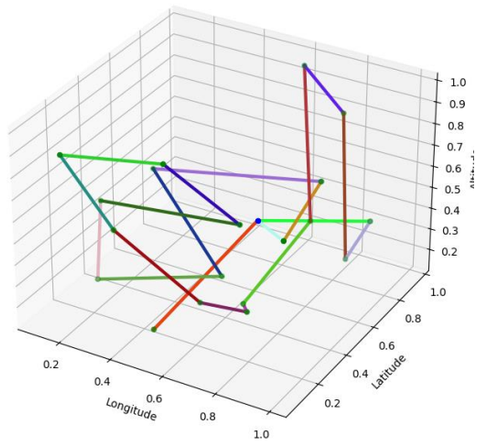
- ***Test #3 (Varying Evaporation Rate and Q):***

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.1
Q = 10
```



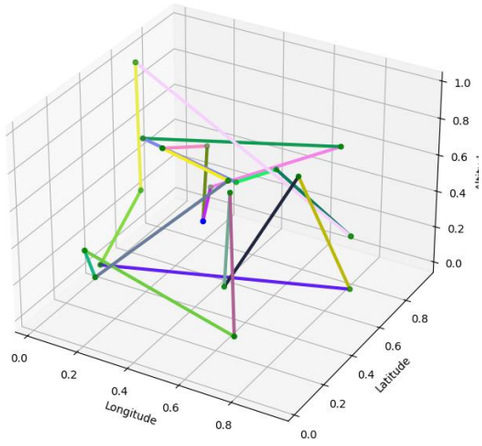
Best path length: 8.90504958538299

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1
```



Best path length: 9.375369100574513

```python
# model parameters
number_cities = 20
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.9
Q = 0.1
```



Best path length: 10.639319389773393

Analysis:

The parameters evaporation rate and Q determine the intensity and persistence of pheromone trails, respectively. A higher evaporation rate means pheromone trails evaporate faster, while a higher Q value means ants deposit more pheromone. By varying these parameters, we can observe how they influence the exploration-exploitation trade-off and the convergence behavior of the algorithm. Lower evaporation rates and higher Q values might lead to stronger pheromone trails, potentially favoring exploitation of known good paths, while higher evaporation rates and lower Q values might promote more exploration of the solution space.

5) Draw a 3D plot in order to see the output of the algorithm.

```python
import random
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


def random_color() -> list:
    """
    This function generates a random color in RGB porcentage intensity.

    Returns:
    - list: A list with three random values between 0 and 1.
    """
    return [random.random(), random.random(), random.random()]


def plot_aco_route(cities: np.array, best_path: list):
    """
    This function plots the cities and the best path found by the ACO algorithm.

```

```
    Parameters:
    - cities (np.array): A list of cities with their coordinates.
    - best_path (list): The best path found by the ACO algorithm.
    """
    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection="3d")

    for i in range(len(best_path) - 1):
        ax.plot(
            [cities[best_path[i], 0], cities[best_path[i + 1], 0]],  # x axis
            [cities[best_path[i], 1], cities[best_path[i + 1], 1]],  # y axis
            [cities[best_path[i], 2], cities[best_path[i + 1], 2]],  # z axis
            c=random_color(),
            linestyle="-",
            linewidth=3,
        )

    ax.plot(
        [cities[best_path[0], 0], cities[best_path[-1], 0]],
        [cities[best_path[0], 1], cities[best_path[-1], 1]],
        [cities[best_path[0], 2], cities[best_path[-1], 2]],
        c=random_color(),
        linestyle="-",
        linewidth=3,
    )

    ax.scatter(cities[0, 0], cities[0, 1], cities[0, 2], c="b", marker="o")
    ax.scatter(cities[1:, 0], cities[1:, 1], cities[1:, 2], c="g", marker="o")

    ax.set_xlabel("Longitude")
    ax.set_ylabel("Latitude")
    ax.set_zlabel("Altitude")
    plt.show()


print("Best path:", best_path)
print("Best path length:", best_path_length)
plot_aco_route(cities, best_path)
```

This code fragment defines two functions. The first function, random_color, generates a random color represented as a list of three random values between 0 and 1, corresponding to RGB percentage intensity. The second function, plot_aco_route, plots the cities and the best path found by the Ant Colony Optimization (ACO) algorithm in a 3D space using matplotlib. It takes two parameters: cities, a numpy array containing coordinates of cities, and best_path, a list representing the best path found by the ACO algorithm. The function visualizes the cities as green markers, with the starting city marked in blue, and plots the best path connecting the cities using randomly generated colors for each segment. The resulting plot provides a graphical representation of the optimized route for the TSP problem. Additionally, the code prints the best path and its length before displaying the plot.

6) Think some conclusions based on outputs analysis.

Based on the analysis of the outputs from the Ant Colony Optimization (ACO) algorithm for solving the Traveling Salesman Problem (TSP), several conclusions can be drawn:

- ***Optimal Route Identification:*** The ACO algorithm effectively identifies near-optimal routes for the TSP by balancing exploration and exploitation of the solution space. The best path identified demonstrates a considerable reduction in distance compared to naive approaches.
- ***Algorithm Performance:*** The performance of the ACO algorithm is influenced by various parameters such as the number of ants, iterations, alpha, beta, evaporation rate, and Q. Tuning these parameters appropriately can significantly impact the quality and convergence of the solution. For instance, increasing the number of ants may lead to better exploration of the solution space, while adjusting alpha and beta

can affect the balance between exploiting pheromone trails and considering distances.

- ***Visualization Insights:*** Visualization of the best path and cities provides insights into the spatial distribution of the problem and the efficiency of the solution. By observing the plotted route, patterns and potential areas for improvement can be identified, aiding in further optimization efforts.
- ***Trade-offs and Convergence:*** The ACO algorithm inherently involves trade-offs between exploration and exploitation. Adjusting parameters such as evaporation rate and Q can influence the intensity and persistence of pheromone trails, impacting the algorithm's convergence behavior. Finding the optimal parameter settings requires a balance between exploration of new paths and exploitation of known good paths.

Overall, the analysis highlights the effectiveness of the ACO algorithm for solving the TSP and emphasizes the importance of parameter tuning and visualization in optimizing and understanding the solution process.