

Systems Analysis Course
Universidad Distrital Francisco José de Caldas



Swarm Intelligence and Synergy: Ant Colony for the Traveling
Salesman Problem

Submitted by:
Andrés Felipe Vanegas Bogotá

Instructor:
Carlos Andrés Sierra Virguez

April 3rd, 2024

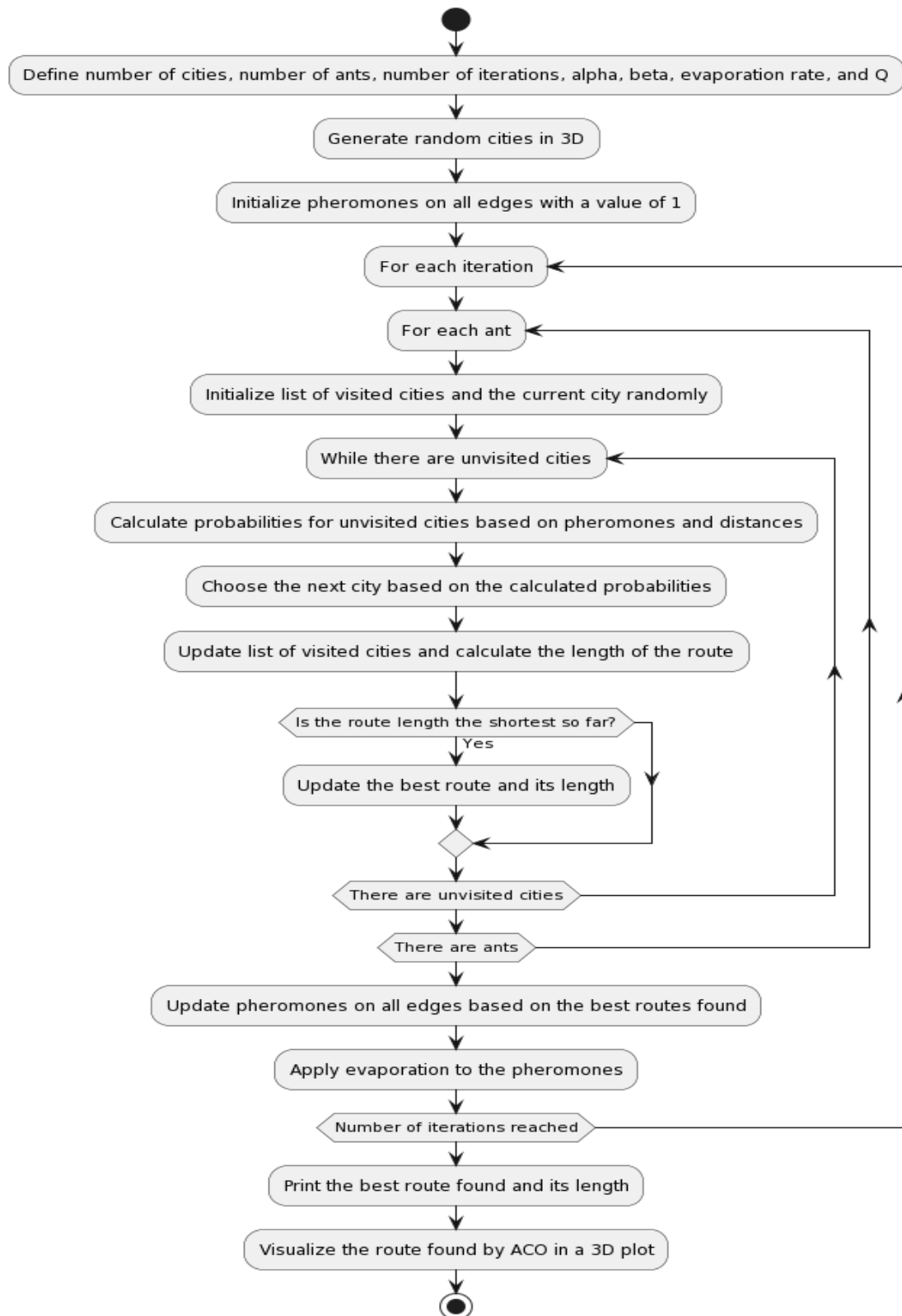
Introduction:

The aim of this workshop is to tackle the Traveling Salesman Problem (TSP) using the Ant Colony Optimization (ACO) algorithm. TSP, a fundamental issue in combinatorial optimization, entails finding the shortest path that visits each city exactly once and returns to the starting point.

As an artificial intelligence engineer within a logistics company, our objective is to devise the most efficient route for product deliveries across multiple cities. ACO, inspired by the foraging behavior of ants, offers a promising strategy for effectively addressing the challenges posed by the TSP.

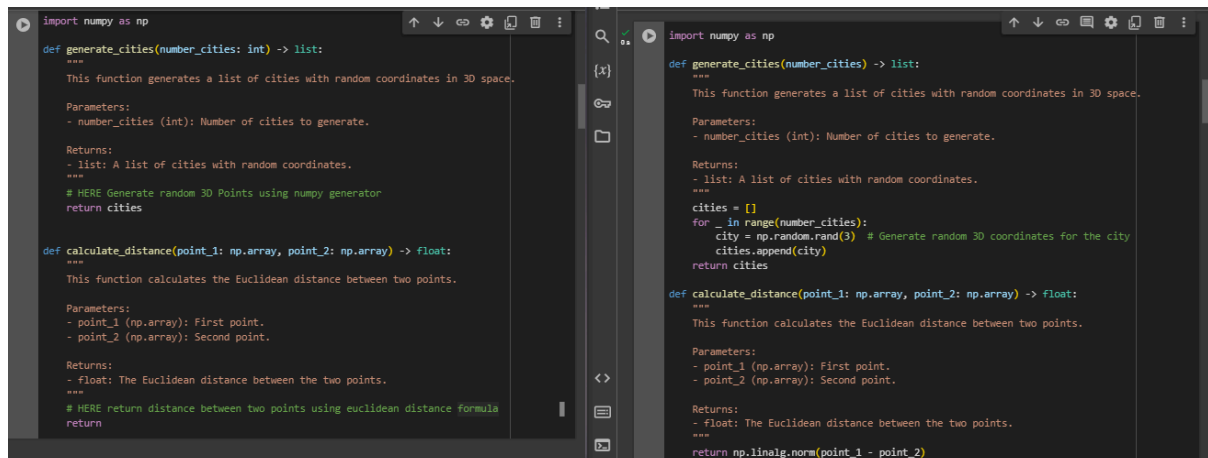
Problem Representation

The flowchart provides a detailed illustration of the methodology employed by the Ant Colony Optimization (ACO) algorithm to address the challenge of optimizing product delivery routes. By strategically leveraging the collective intelligence inspired by the foraging behavior of ants, ACO aims to identify the most efficient path for delivering goods while minimizing travel distances and time. This process involves the iterative exploration of various routes, guided by the interaction between artificial ants and pheromone trails. Through continuous iteration and refinement, ACO progressively converges towards an optimal solution, ensuring the timely and cost-effective delivery of goods to their intended destinations.



Step-by-Step Implementation in the Code

1. Initialize an empty list to store city coordinates.
2. Use a 'for' loop to iterate 'number_cities' times.
3. Generate random coordinates for each city using 'np.random.rand(3)'.
4. Assign generated coordinates to the variable 'city'.
5. Append the generated city to the list using 'append()'.
6. Return the list of cities once coordinates for all cities are generated.
7. Define a function called 'calculate_distance'.
8. The function takes two numpy arrays, 'point_1' and 'point_2', representing coordinates of two points in a multi-dimensional space.
9. It calculates the Euclidean distance between these two points.
10. The function returns the calculated distance as a float value.
11. The Euclidean distance is computed using the 'np.linalg.norm()' function, which computes the Euclidean norm (magnitude) of the vector representing the difference between 'point_1' and 'point_2'.



The image shows two side-by-side code editors with a dark theme. The left editor contains the following code:

```
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
    return cities

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
```

The right editor contains the following code:

```
import numpy as np

def generate_cities(number_cities) -> 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.
    """
    cities = []
    for _ in range(number_cities):
        city = np.random.rand(3) # Generate random 3D coordinates for the city
        cities.append(city)
    return cities

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.
    """
    return np.linalg.norm(point_1 - point_2)
```

12. `number_cities = len(cities)`: Determines the number of cities stored in the list 'cities' and assigns it to the variable 'number_cities'. This is essential for sizing the pheromone matrix in the next line.
13. `pheromone = np.ones((number_cities, number_cities))`: Create an initial pheromone array filled with ones (as the teacher advised). The matrix has dimensions of 'number_cities' by 'number_cities', representing each possible connection between cities. Each entry in the matrix

represents the amount of pheromone present in the connection between two cities. Initializing with ones ensures that all connections start with the same initial amount of pheromone.

```
[ ] 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 trail
    - beta (float): It determines how much the ants are influenced by the distance to the
    - evaporation_rate (float): Evaporation rate.
    - Q (float): It determines the intensity of the pheromone trail left behind by an ant.
    """

    number_cities = # HERE Get number of points
    pheromone = # HERE Initialize pheromone matrix with ones

    # 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

[7] 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 trail
    - beta (float): It determines how much the ants are influenced by the distance to the
    - evaporation_rate (float): Evaporation rate.
    - Q (float): It determines the intensity of the pheromone trail left behind by an ant.

    Returns:
    - tuple: A tuple containing the best path found and its length.
    """

    number_cities = len(cities)
    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 = []
        path_lengths = []
```

14. The formula used to calculate the probability of moving to a city based on pheromone levels, distance, alpha, and beta is prepared for implementation within the 'for' loop. The formula, as specified by the instructor, incorporates the following steps: Utilize the pheromone matrix with indices 'i' and 'j', multiply by alpha, incorporate the distance between cities obtained from the 'calculate_distance' function, and multiply the result by beta.
15. The normalization is calculated via "probabilities /= probabilities.sum()" making sure the probabilities sum to 1

```

    # 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):
        probabilities[i] = (pheromone[current_city, unvisited_city] ** alpha) * \
            [(1 / calculate_distance(cities[current_city], cities[unvisited_city])) ** beta]

    probabilities /= probabilities.sum()
```

16. In this section, I've opted to provide more detailed commentary on the parameters, generate the list of cities, and execute the Ant Colony Optimization (ACO) algorithm. The algorithm takes as input the set of cities along with several parameters that govern its behavior.

The function returns two values:

- 'best_path': This represents the optimal route discovered by the algorithm. It consists of an ordered sequence of cities that denote the shortest path uncovered by the ants.
- 'best_path_length': This means the cumulative distance of the best path determined. It reflects the total distance covered by the ants in the most optimal solution found.

```
# model parameters
number_cities = 30 [10,20,30,40,50]
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1

# HERE create list of cities
cities =

# HERE call ant_colony_optimization function

<class 'numpy.ndarray'>
```

```
[12] # Model Parameters
number_cities = 50 # Number of cities in the traveling salesman problem
number_ants = 20 # Number of ants in the colony
number_iterations = 100 # Number of iterations for optimization
alpha = 1 # Weight for pheromone trail
beta = 1 # Weight for distance between cities
evaporation_rate = 0.5 # Pheromone evaporation rate
Q = 1 # Amount of pheromones deposited by an ant

# Generate a list of cities (this should be defined elsewhere)
cities = generate_cities(number_cities)

# Execute ant colony optimization
best_path, best_path_length = ant_colony_optimization(
    cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q
)
```

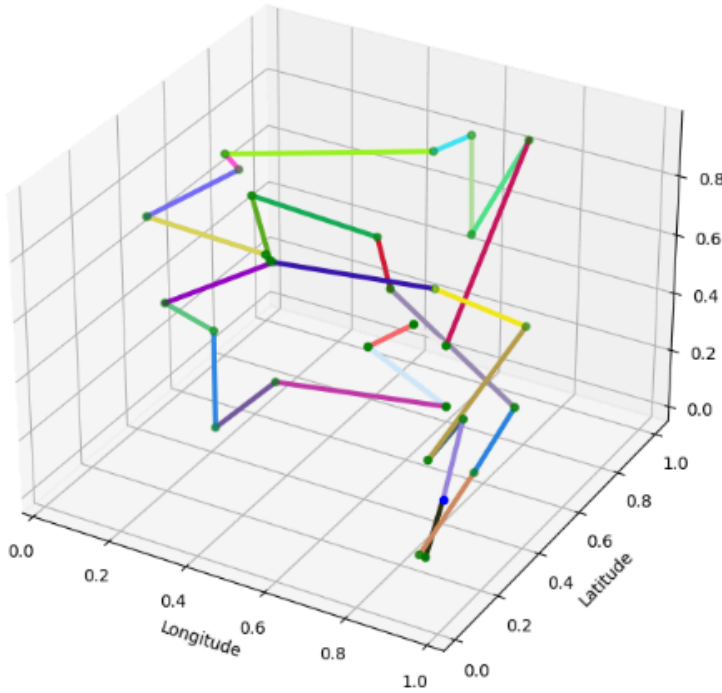
Example of Output

```
# Model Parameters
number_cities = 30 # Number of cities in the traveling salesman problem
number_ants = 20 # Number of ants in the colony
number_iterations = 100 # Number of iterations for optimization
alpha = 1 # Weight for pheromone trail
beta = 1 # Weight for distance between cities
evaporation_rate = 0.5 # Pheromone evaporation rate
Q = 1 # Amount of pheromones deposited by an ant

# Generate a list of cities (this should be defined elsewhere)
cities = generate_cities(number_cities)

# Execute ant colony optimization
best_path, best_path_length = ant_colony_optimization(
    cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q
)
```

Best path: [26, 20, 0, 12, 2, 6, 8, 7, 9, 19, 3, 16, 25, 24, 28, 13, 14, 27, 18, 15, 1, 22, 10, 5, 4, 23, 21, 11, 17, 29]
Best path length: 9.685451002751002



Testing Different Parameter Combinations

To ensure the robustness of our solution, we will test different parameter combinations for the ACO algorithm. This includes parameters such as the number of cities, ants and pheromone evaporation rate, as can be seen in the table below.

number_cities	number_ants	number_iterations	alpha	beta	evaporation_rate	Q	Best path	Best path length
30	20	100	1	1	0.5	1	[1, 10, 6, 2, 29, 20, 22, 21, 7, 13, 18, 23, 5, 8, 17, 14, 16, 9, 19, 4, 24, 0, 26, 3, 11, 27, 28, 25, 15, 12]	8.437.959.802.152.490
40	20	100	1	1	0.5	1	[29, 13, 2, 34, 10, 36, 25, 21, 8, 7, 27, 38, 18, 15, 23, 35, 20, 33, 12, 9, 3, 39, 14, 0, 28, 31, 24, 16, 37, 11, 30, 22, 4, 26, 5, 19, 32, 17, 6, 1]	10.390.692.795.308.600
50	20	100	1	1	0.5	1	[19, 29, 22, 31, 49, 16, 26, 33, 21, 39, 12, 6, 24, 41, 46, 25, 20, 7, 9, 47, 30, 34, 4, 5, 35, 37, 43, 11, 18, 3, 0, 36, 40, 10, 1, 44, 14, 48, 45, 28, 8, 23, 38, 32, 13, 15, 2, 17, 27, 42]	1.148.478.137.351.780
30	30	100	1	1	0.5	1	[29, 9, 22, 3, 21, 27, 28, 1, 16, 0, 11, 15, 26, 5, 23, 18, 12, 13, 14, 25, 2, 8, 10, 7, 24, 17, 19, 4, 20, 6]	9.066.040.779.737.290
30	40	100	1	1	0.5	1	[9, 29, 21, 11, 15, 7, 17, 18, 13, 27, 14, 3, 6, 12, 28, 4, 24, 22, 8, 10, 1, 19, 20, 25, 2, 23, 5, 16, 26, 0]	7.662.768.511.845.580
30	50	100	1	1	0.5	1	[8, 28, 24, 5, 15, 26, 20, 21, 4, 18, 22, 19, 7, 13, 29, 14, 3, 25, 16, 9, 11, 2, 17, 0, 23, 6, 1, 10, 12, 27]	7.870.905.744.188.010
30	20	100	1	1	2	1	[28, 1, 23, 8, 21, 12, 5, 17, 9, 18, 16, 19, 11, 20, 22, 4, 24, 29, 2, 27, 6, 7, 10, 25, 13, 26, 14, 15, 0, 3]	13.307.800.513.941.200
30	20	100	1	1	0.1	1	[7, 2, 26, 11, 28, 14, 21, 18, 6, 29, 5, 1, 3, 12, 0, 19, 16, 13, 24, 15, 4, 10, 8, 20, 9, 23, 17, 27, 22, 25]	7.101.254.820.017.390

Analysis and Conclusions

Based on the results obtained by running the algorithm several times, changing the parameters, we can deduce that:

- As the number of cities increases, the total path length also tends to increase, indicating that the problem becomes more complex and optimization becomes more difficult with a larger number of cities.
- These results suggest that the number of ants also affects the total length of the shortest path found by the ACO algorithm. In this case, an increase in the number of ants appears to lead to a decrease in the total length of the shortest path up to a certain point, after which the effect stabilizes or reverses slightly. This may indicate that an optimal number of ants can lead to better exploration and exploitation of the search space in the algorithm, this reminds me of an economics concept "Law of diminishing marginal returns"
- An increase in Evaporation Rate leads to greater pheromone evaporation, which can result in fewer pheromones at the edges and therefore an increase in the total length of the shortest path. On the other hand, a decrease in Evaporation Rate can result in a greater amount of pheromones at the edges and, therefore, a decrease in the total length of the shortest path.