Universidad Distrital Francisco José De Caldas

Algoritmo Colonia de Hormigas.

Ingeniería en Sistemas

Análisis De Sistemas

Integrante:

JHONATHAN DAVID DE LA TORRE GARCIA 20222020033

Bogotá Colombia

Marzo 2024

INTRODUCCION.

En el presente documento se procederá a realizar un análisis sobre el uso de diversas variables aplicadas en el algoritmo de colonia de hormigas, aplicado a la búsqueda de caminos mas cortos que unen a un conjunto de diversas ciudades.

Algunas de esas variables son: numero de ciudades, cantidad de hormigas, la tasa de evaporación de las hormonas dejadas por las hormigas, el número de iteraciones por ciclo, y otras variables que definen la influencia que sienten las hormigas para verse más atraídas bajo ciertas circunstancias.

Al realizar variaciones en dichas variables se busca ver cómo afectan dichos cambios en el tiempo de ejecución del algoritmo.

Adicionalmente se debe completar una serie de líneas de código faltantes para poder obtener la salida deseada, que para este caso la salida final deseada es un grafico 3D que muestra dichas distancias mínimas y el trayecto que optimiza el recorrido.

DESARROLLO

Para la ejecución de las tareas descritas anteriormente se tomará como punto de partida, añadir las líneas de código faltantes, al algoritmo suministrado inicialmente.

Iniciando de esta primera función:

```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
    return cities
```

Se solicita generar puntos aleatorios en el espacio 3D , usando numpy de Python, la propuesta que cumple con el requisito solicitado es esta:

```python
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.
    """

    cities = []
    for _ in range(number_cities):
        # Generate random coordinates in 3D space
        x, y, z = np.random.rand(3)
        cities.append(np.array([x, y, z]))  # Convertir las coordenadas a un arreglo de NumPy
    return cities
```

generar los puntos usando random de la librería numpy, para posteriormente convertirlos en un arreglo de numpy de esta manera se obtiene la salida deseada de la función.

Continuando con el procedimiento se solicita completar el código para la función que cumple con el calculo de la distancia euclidiana (recordando que es un espacio 3D)

```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
    distance = np.linalg.norm(point_1 - point_2)  # Calcular la distancia euclidiana
    return distance
```

El código anterior aplica una posible forma de calcular la distancia euclidiana entre 2 puntos, usando linalg.norm de numpy ,dicha librería usa principios de calculo de la normal usando algebra lineal y retornando un valor escalar que corresponde a la distancia entre 2 puntos .

Para la siguiente función se requiere completar la implementación del algoritmo de colonia de hormigas, el código implementado se muestra a continuación:

```
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.
    """
    number_cities = len(cities)  # Get number of points
    pheromone = np.ones((number_cities, number_cities))  # 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

            # 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]  # Store only the index of the 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):
```

```
                # calculate the probability of moving to a city based on pheromone, distance
and alpha and beta parameters
                pheromone_factor = pheromone[current_city, unvisited_city] ** alpha
                visibility_factor = 1 / calculate_distance(cities[current_city],
cities[unvisited_city]) ** beta
                probabilities[i] = pheromone_factor * visibility_factor

            # normalize probabilities
            probabilities /= np.sum(probabilities)

            next_city = np.random.choice(unvisited, p=probabilities)
            path.append(next_city)  # Store only the index of the 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  # Store only the indices of the best 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
```

para el Desarrollo de la primera parte se pide especificar el numero de ciudades para el algoritmo, para posteriormente inicializar una matriz con 1’s que corresponden a las feromonas dejadas por las hormigas, dicha solución es la siguiente:

```
    number_cities = len(cities)  # Get number of points
    pheromone = np.ones((number_cities, number_cities))  # Initialize pheromone matrix
with ones
```

se utiliza el tamaño de cities que fue creado anteriormente para conocer la cantidad de ciudades generadas, y para la matriz de feromonas se utiliza la np.ones que suministra numpy y permite llenar una matriz cuadrada con el tamaño del numero de ciudades , donde todos sus componentes son 1's respectivamente.

Posteriormente se solicita normalizar las probabilidades basadas de usar variables como las feromonas, el factor Alpha y beta para ciudades que no han sido visitadas, la propuesta que cumple con la normalización es esta:

```
        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):
                # calculate the probability of moving to a city based on pheromone, distance
and alpha and beta parameters
                pheromone_factor = pheromone[current_city, unvisited_city] ** alpha
                visibility_factor = 1 / calculate_distance(cities[current_city],
cities[unvisited_city]) ** beta
                probabilities[i] = pheromone_factor * visibility_factor

            # normalize probabilities
            probabilities /= np.sum(probabilities)
```

El objetivo del código consiste en utilizar el parámetro Alpha e ir multiplicando la matriz de feromonas , luego usar el factor de visibilidad que no es mas que un cociente que usa la distancia entre ciudades no visitadas , y utilizando el factor beta , de esta manera se obtienen probabilidades para que las hormigas visiten una ciudad , y se irán guardando dichas probabilidades .Finalizando con la normalización de las probabilidades obtenidas usando np.sum que ira sumando las probabilidades obtenidas anteriormente y dividiéndolas entre dicha cantidad, así se obtienen valores de probabilidad dentro del rango [0,1].

Continuando con el desarrollo, es necesario darles valores a las variables para darles un uso al algoritmo, adicionalmente de invocar la función de optimización para el algoritmo de colonia de hormigas, la propuesta es la siguiente:

```
import numpy as np

# Definir parámetros del modelo
number_cities = 20  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 2
```

```
number_iterations = 200
alpha = 0.1
beta = 0.1
evaporation_rate = 0.1
Q = 1

# Crear lista de ciudades
cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants,
number_iterations, alpha, beta, evaporation_rate, Q)
```

En el código anterior se muestra ejemplificado el uso del algoritmo, dándole valores al numero de ciudades, la cantidad de hormigas, el número de iteraciones. Posteriormente se hace el llamado de la función generate_cities con un parámetro de entrada definido anteriormente, y ya con las ciudades creadas aleatoriamente, se ingresan los valores como parámetros a la función de optimización, se solicitó como parámetro de entrada, las ciudades, el número de hormigas, el número de iteraciones, el factor Alpha, el factor beta, el factor de evaporación de feromonas y el factor Q (que es la intensidad de las feromonas dejadas por las hormigas).

Para finalizar esta primera parte, se solicita completar un código usando la librería de Python matplotlib, que es la que nos ayudara a realizar el grafico deseado para la aplicación del algoritmo colonia de hormigas, el código propuesto que resuelve el problema

```
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: list, best_path: list):
    """
    This function plots the cities and the best path found by the ACO algorithm.
```

```
    Parameters:
    - cities (list): 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")

    # Plot the lines between cities in the best path
    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,
        )

    # Connect the first and last cities in the best path
    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,
    )

    # Plot all cities as filled circles
    for city in cities:
        ax.scatter(city[0], city[1], city[2], c="b", 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)
```

Fue necesario modificar levemente el código original a causa del manejo de las listas que obtuve de la aplicación del algoritmo , el problema presentado consistía en que se me solicitaban índices de listas , pero inicialmente estaba retornado tuplas que contenían la información de la posición de las ciudades, así que para evitar el problema de las tuplas como índices de las listas , empecé a tomar las posiciones de dichas tuplas en la coordenada que necesitara (posición 0, 1 y 2), que es homologo a la propuesta original.

De esta manera se obtiene la salida deseada, que para ejemplificar el primer uso dando los valores anteriormente mostrados se obtiene este resultado:

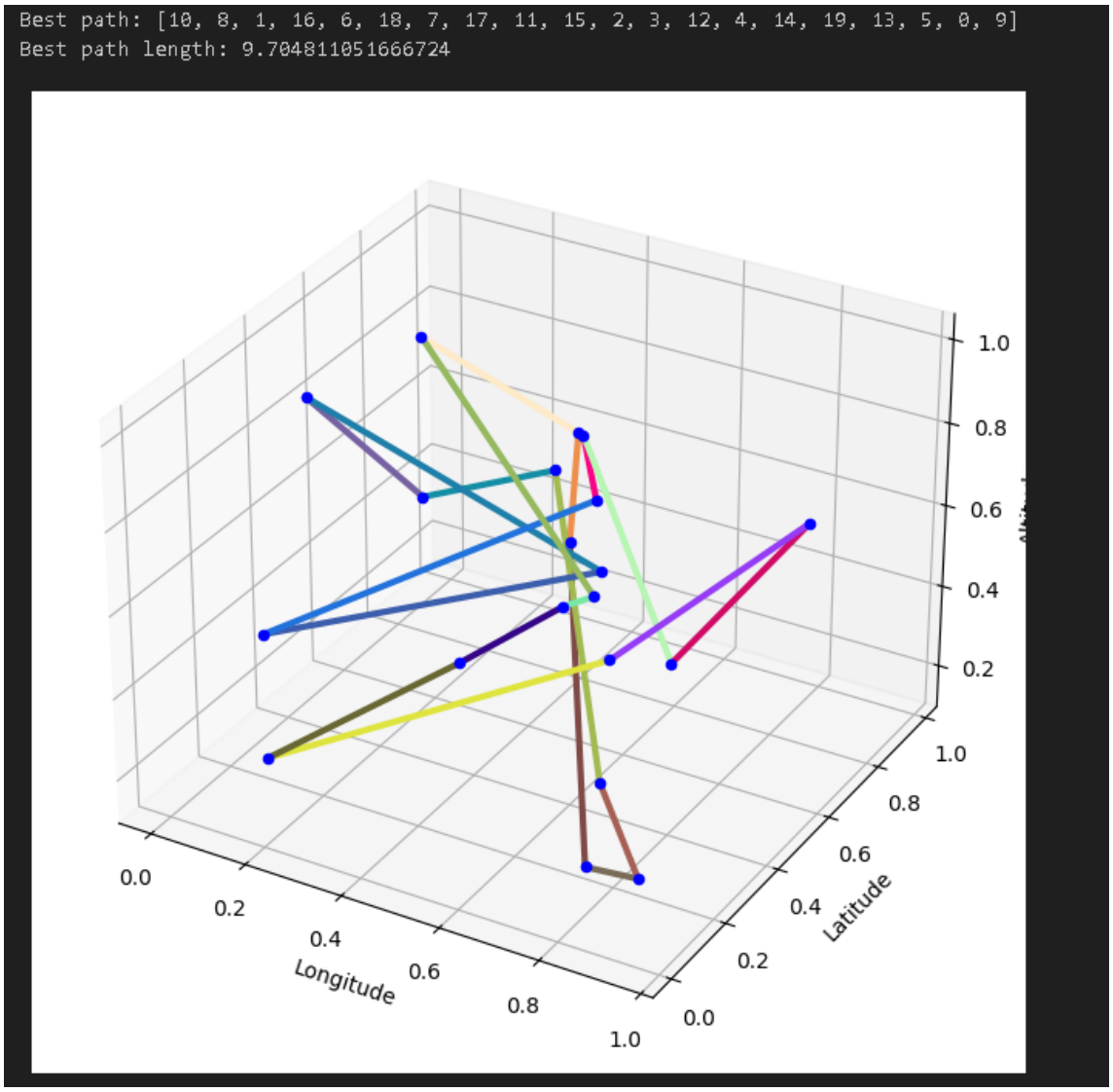


Figura 1. Algoritmo colonia de hormigas para cálculo de distancias mínimas entre un conjunto de ciudades.

Segunda parte

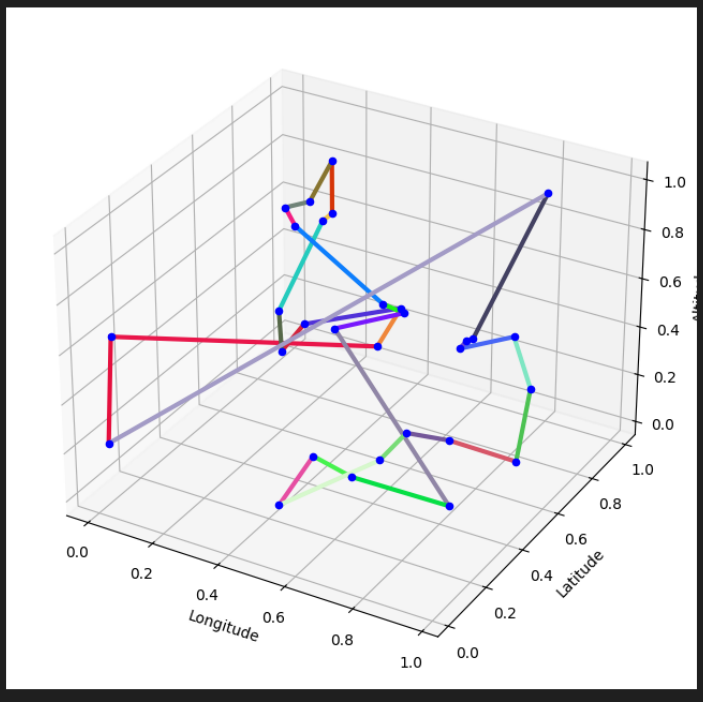
Una vez completado el código necesario para dar uso al algoritmo realizare diversas pruebas enfocándome en el tiempo de ejecución del algoritmo.

Tomando como partida los datos suministrados del código original, dichos valores son los siguientes:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 21.5s
```

Como se puede observar en la imagen anterior el tiempo de ejecución del algoritmo fue de 21.5s y nos arroja la siguiente gráfica:

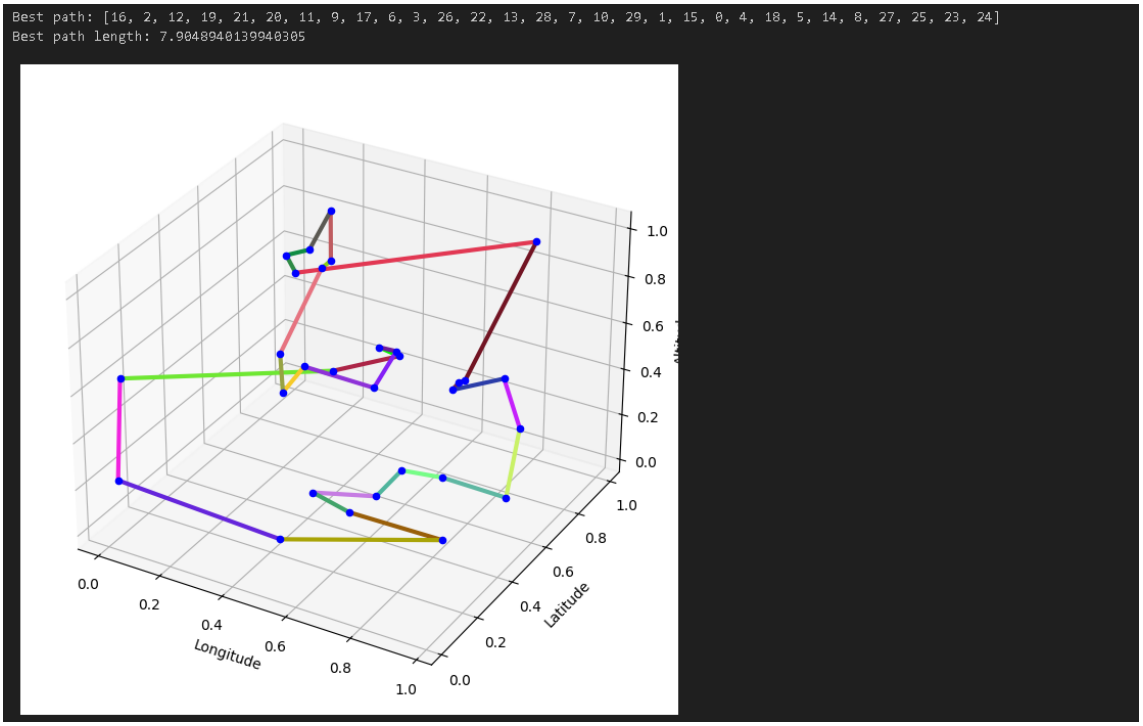
```
Best path: [13, 28, 15, 1, 0, 4, 18, 5, 14, 8, 27, 25, 23, 29, 10, 7, 26, 3, 6, 22, 17, 9, 11, 20, 21, 19, 12, 2, 16, 24]
Best path length: 7.787975258428704
```



Partiendo de esta primera gráfica, inicialmente cambiare la cantidad de hormigas y se observaran las diferencias:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 200
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
43.4s
```

Aumentando la cantidad de hormigas al doble (200) el tiempo de ejecución fue de 43,3s y nos arroja la siguiente gráfica:



De este primer cambio notamos 2 cambios notorios, el mas evidente el aumento del tiempo de ejecución, que corresponde a que existe mayor cantidad de hormigas recorriendo los caminos, y que son afectadas por múltiples variables estocásticas. Y el segundo cambio se relaciona a los caminos recorridos, se modificaron en diversas zonas las conexiones entre ciudades, arrojando un grafo distinto y una distancia mínima diferente.

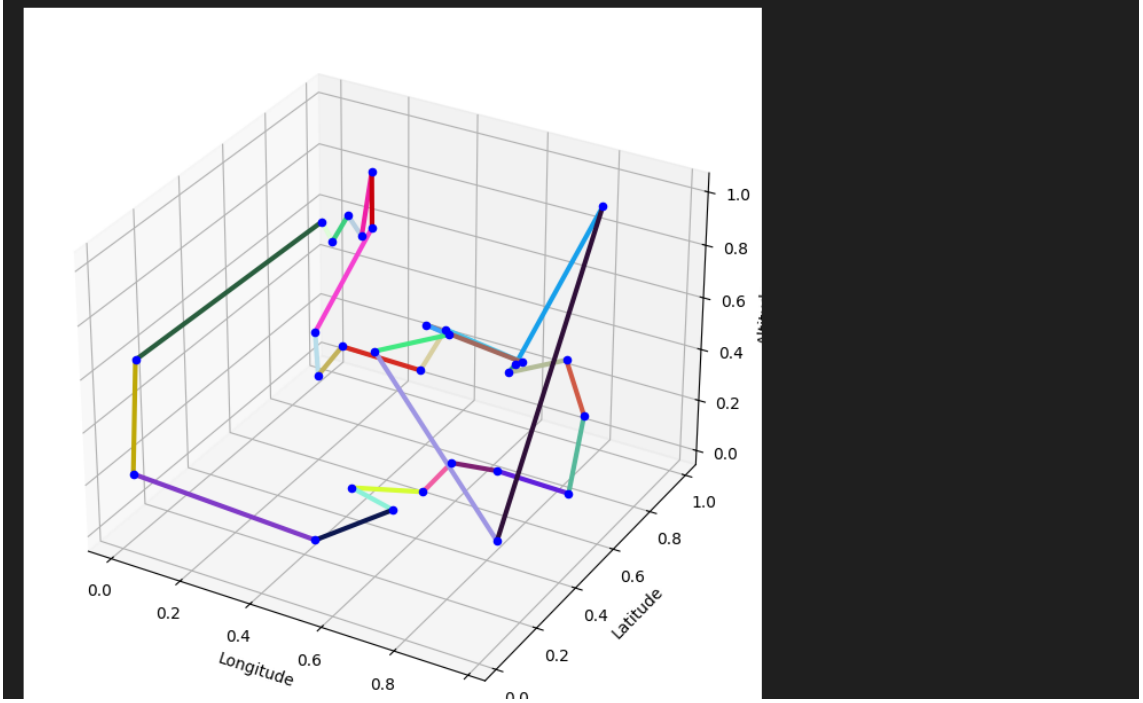
Cabe recalcar que estos cambios son influenciados también a causa de lo descrito anteriormente, ósea a las probabilidades, las hormigas no están iniciando siempre desde la misma ciudad, por consecuencia el camino optimo tendrá cambios para minimizar su distancia.

Ahora vamos a tomar un valor menor de hormigas para ver como esto cambia la ejecución del algoritmo y el resultado en cuestión, los valores son los siguientes:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 50
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 10.8s
```

Al utilizar un valor de 50 hormigas el tiempo de ejecución disminuye hasta 10.8s, y dicha ejecución arroja la siguiente gráfica:

```
Best path: [24, 2, 12, 19, 21, 20, 11, 9, 17, 6, 3, 22, 13, 28, 25, 23, 27, 5, 8, 14, 18, 4, 0, 15, 1, 16, 29, 10, 7, 26]
Best path length: 7.706583886537643
```

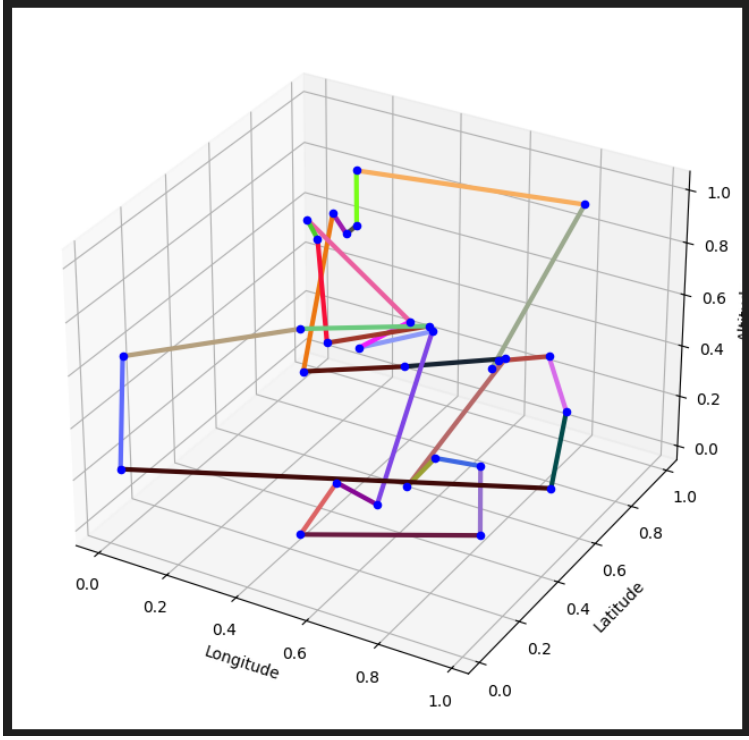


Nuevamente observamos variaciones en el camino resultante, así que una primera conclusión se refuerza en lo expresado en la anterior ejecución, que dichas variaciones son obtenidas a causa de la naturaleza estocástica de la posición y las variables que afectan el comportamiento de las hormigas, así que ahora se van a realizar los siguientes cambios:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 50
number_iterations = 100
alpha = 0.5
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 11.0s
```

Disminuir el valor de Alpha a la mitad (0.5), esto nos da un tiempo de ejecución de 11 s, no es significativamente mayor a la ejecución anterior y da como resultado esta gráfica:

```
Best path: [20, 21, 19, 2, 12, 24, 8, 14, 5, 27, 4, 15, 16, 17, 9, 11, 26, 22, 6, 3, 10, 7, 29, 25, 23, 0, 1, 18, 28, 13]
Best path length: 10.02070962345723
```



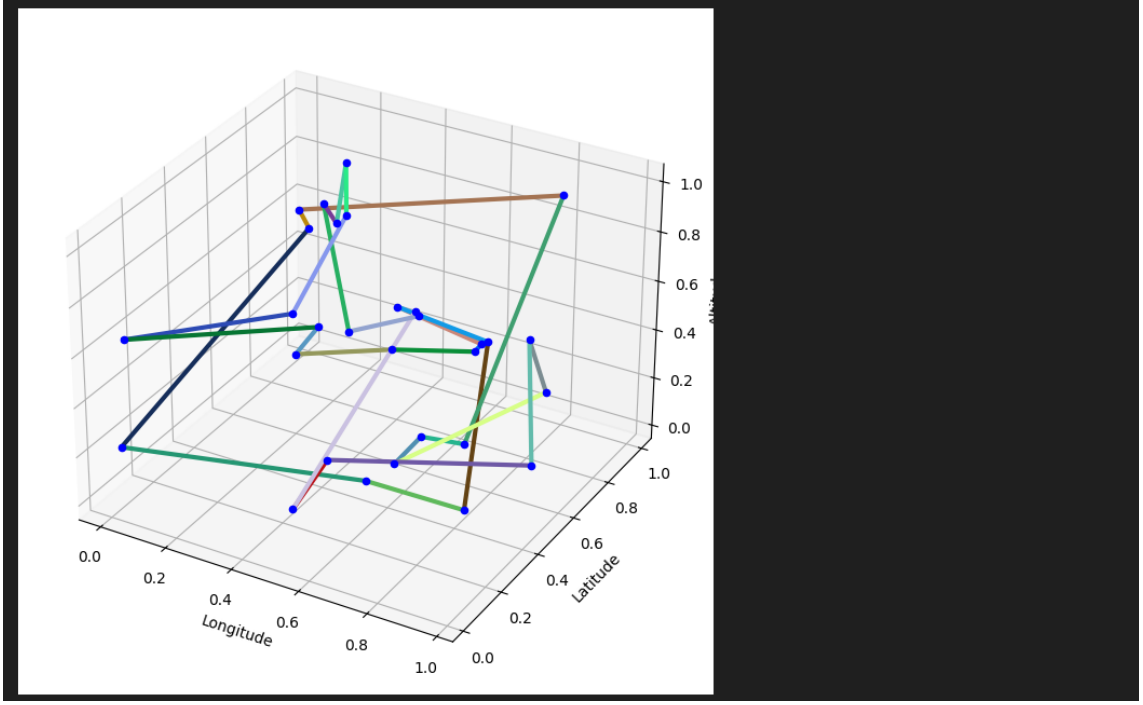
Para esta ejecución en especifico el resultado de la distancia es mayor que en los casos anteriores, sin embargo, mantengo mi postura que es causa de su comportamiento estocástico.

La siguiente prueba toma los siguientes valores:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 50
number_iterations = 100
alpha = 0.1
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 11.2s
```

Se ha disminuido aún más el factor Alpha (0.1) , esto ocasión un tiempo de ejecución levemente mayor y dando la siguiente gráfica:

```
Best path: [0, 4, 15, 12, 2, 29, 16, 26, 3, 13, 23, 25, 24, 11, 9, 17, 21, 19, 20, 6, 22, 1, 10, 7, 27, 5, 8, 14, 18, 28]
Best path length: 11.654778088505088
```



De esta prueba podemos destacar el aumento progresivo del tiempo de ejecución del algoritmo a medida que el factor Alpha baja, y corresponde con la poca influencia que sienten las hormigas de seguir el rastro de feromonas dejados por otras hormigas.

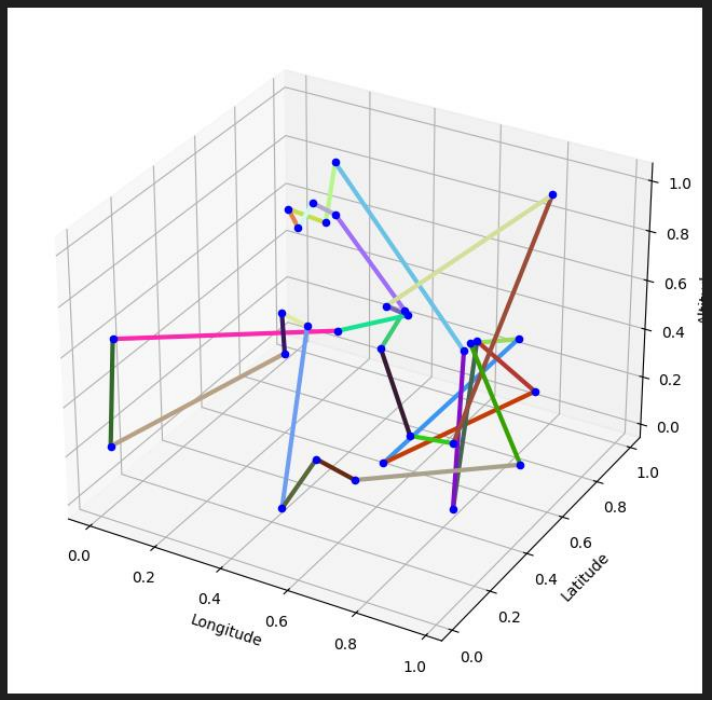
Ahora se va a tomar una variación del factor beta, como se muestra a continuación:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 50
number_iterations = 100
alpha = 1
beta = 0.1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
```

✓ 10.9s

Inicialmente el tiempo de ejecución ha bajado levemente comparado con la prueba anterior (cabe resaltar que se a retornado al valor original de Alpha), y dicha ejecución dio como resultado esta gráfica:

```
Best path: [2, 19, 17, 21, 16, 26, 12, 8, 5, 25, 23, 27, 14, 1, 15, 9, 11, 24, 29, 10, 7, 28, 13, 4, 18, 0, 22, 6, 3, 20]
Best path length: 11.696597103036495
```

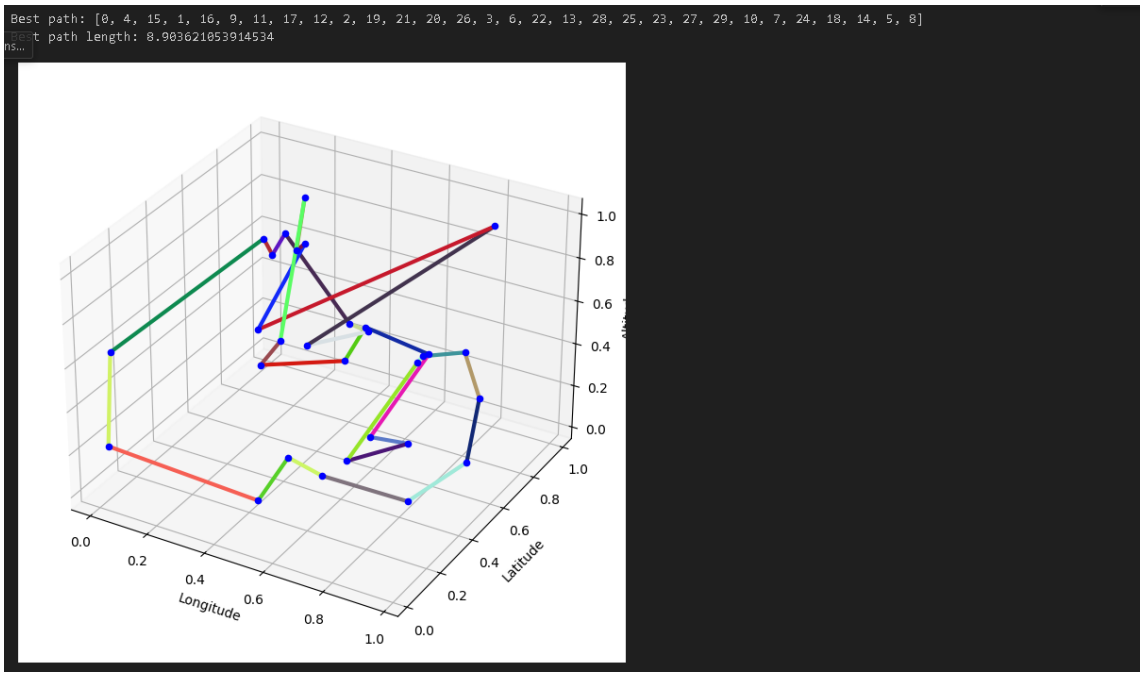


El resultado muestra como las hormigas sienten menos influencia por la distancia que deben recorrer a la siguiente ciudad, dándonos una grafica con un comportamiento visiblemente menos organizado, ya que hay ciudades que se ubican unas más cercas que otras, pero realizan recorridos mas largos entre otras ciudades, así que como el factor Alpha se a mantenido nuevamente alto las hormigas seguirán los recorridos dejados por las feromonas con mayor probabilidad.

Para la siguiente prueba se cambiará el factor de evaporación, como se muestra a continuación:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 50
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 1
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 10.9s
```

A primera vista se observa que el tiempo de ejecución no a cambiado significativamente con las pruebas anteriores, y nos dio la siguiente gráfica:



Una cosa llamativa podemos observar de esta prueba, la distancia a disminuido adicionalmente de ver un comportamiento mas simétrico, la deducción que obtengo observando esta prueba es que a causa del aumento del factor de evaporación , las hormigas no se ven mayormente influenciadas por las feromonas dejadas por sus compañeras y toman caminos diversos usando otros factores.

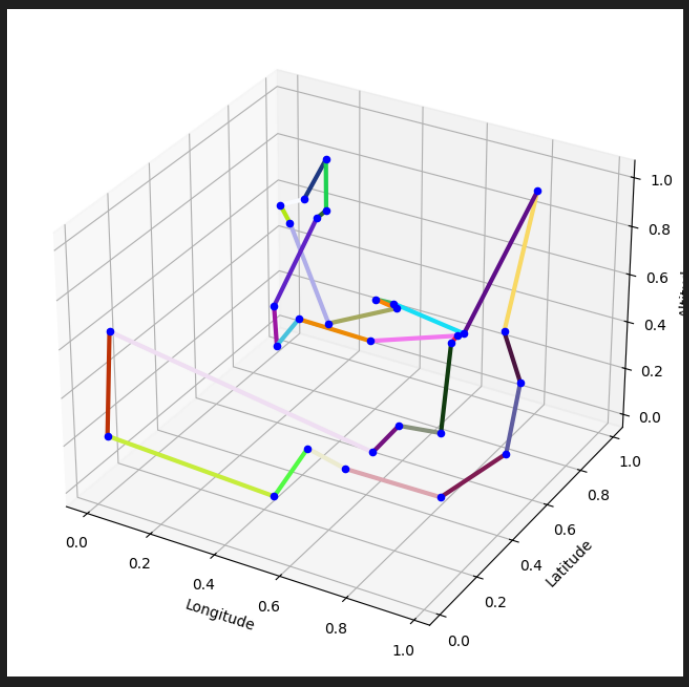
Para la siguiente prueba se busca cambiar el factor Q, para ver como la intensidad de las feromonas dejadas por las hormigas modifica el

comportamiento del algoritmo, los datos son los siguientes:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 30  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 50
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 0.5

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 10.8s
```

Se a disminuido a la mitad el valor de Q (0.5), y el tiempo de ejecución no cambio significativamente comparado con la prueba anterior, dándonos la siguiente gráfica:

```
Best path: [28, 13, 22, 6, 3, 26, 20, 21, 19, 24, 16, 1, 29, 10, 7, 23, 25, 27, 8, 14, 5, 18, 4, 0, 15, 2, 12, 11, 9, 17]
Best path length: 7.860600173053313
```

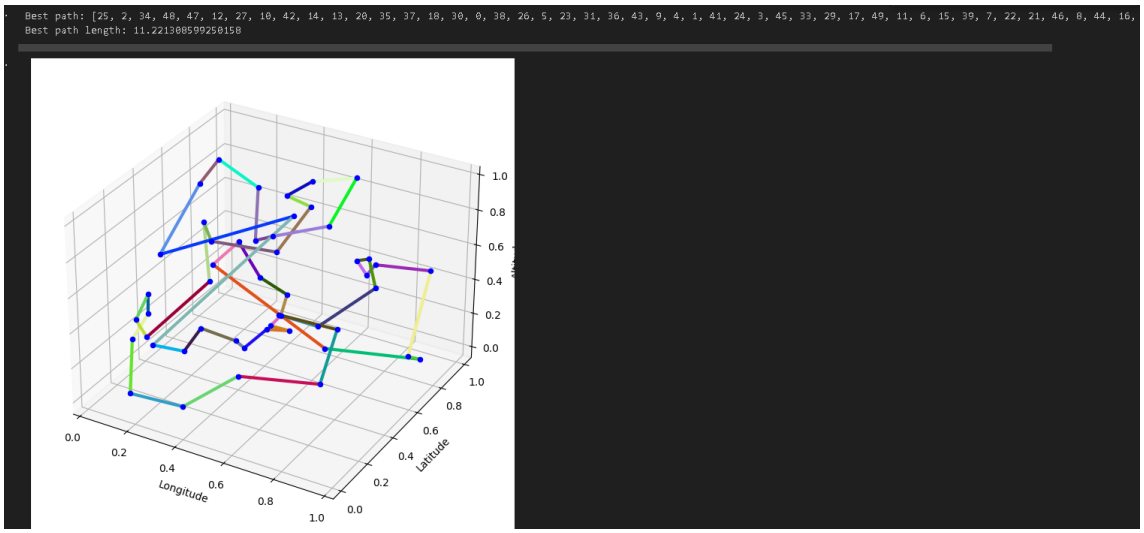


De esta grafica en particular puedo observar una tendencia mayor a un camino “visiblemente” mas ordenado, y esto se comprueba con el valor de la distancia arrojada por el algoritmo, por ende, esto puede inducir que, a causa de poseer una menor intensidad de las feromonas dejadas por las hormigas, las mismas buscan con cierta probabilidad mas variados caminos, dando un resultado ligeramente más optimo.

Ahora buscare forzar la mayor cantidad el uso del procesado, iniciando con una mayor cantidad de ciudades, y dejando los datos restantes como el caso del cual partí, los datos son los siguientes:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 50  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 100
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 53.0s
```

Inicialmente se ve un aumento del tiempo de ejecución, esto tiene sentido debido a que se tienen en cuenta un mayor conjunto de datos, además de contar con la generación de las ciudades para la primera ejecución. y nos dio la siguiente gráfica:

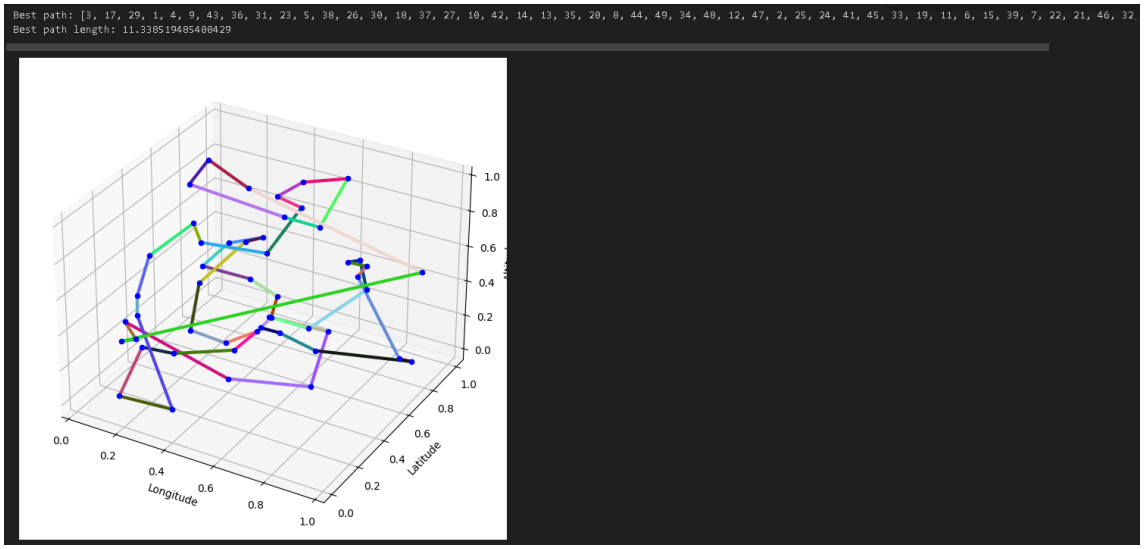


De esta primera prueba podemos realizar una comparación de cuando existían 30 ciudades, como era de esperarse a mayor numero de ciudades mayor tiempo de ejecución.

Realizare una prueba homologa al caso de 30 ciudades, ósea aumentare el número de hormigas iniciales, los datos son los siguientes:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 50  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 200
number_iterations = 100
alpha = 1
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 1m 43.9s
```

Se observa un incremento significativo del tiempo de ejecución, y nos da la siguiente gráfica:



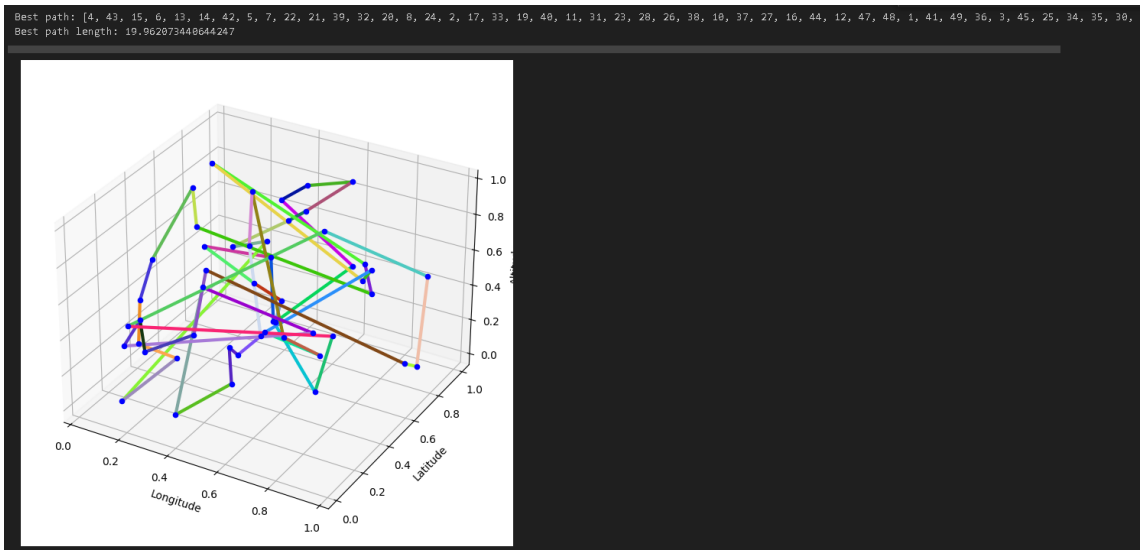
El resultado es el esperado, a mayor cantidad de hormigas el tiempo de ejecución aumentara y esto es influenciado a raíz del conjunto de probabilidades de como toman las decisiones las hormigas.

Ahora tratare de simplificar un poco las pruebas, describiré los cambios que realice y los resultados que puedo deducir.

Para la siguiente prueba cambiare el valor de Alpha, los datos son los siguientes:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 50  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 200
number_iterations = 100
alpha = 0.1
beta = 1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 1m 46.5s
```

El tiempo de ejecución no vario significativamente y nos dio la siguiente gráfica:

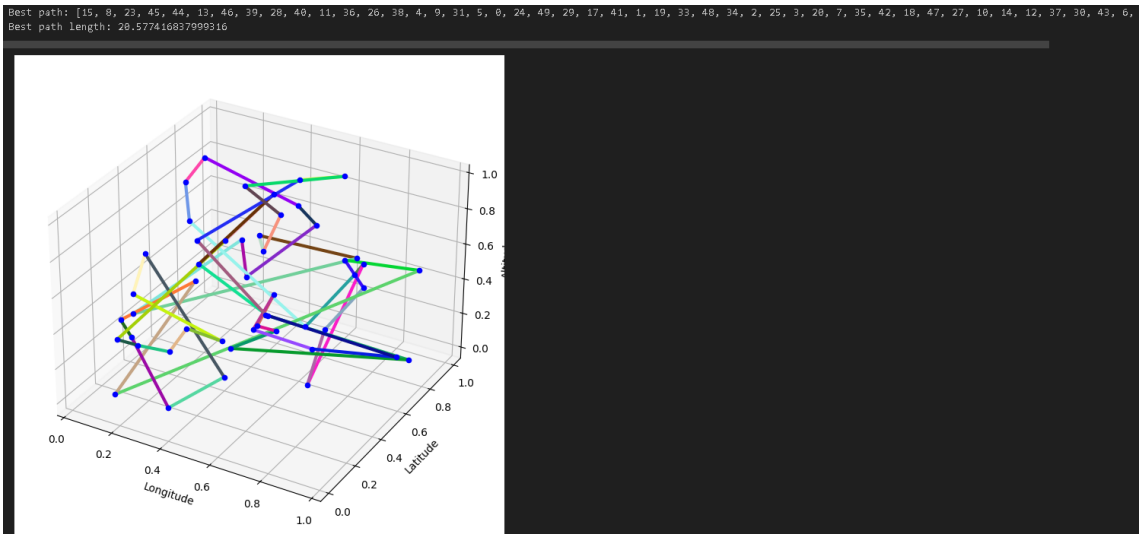


De esta grafica podemos observar un comportamiento visualmente mas desordenado y un aumento de la distancia mínima.

La siguiente prueba se cambia el valor de beta, con los siguientes datos:

```
Click here to ask Blackbox to help you code faster
import numpy as np

# Definir parámetros del modelo
number_cities = 50  # Podrías elegir entre [10, 20, 30, 40, 50]
number_ants = 200
number_iterations = 100
alpha = 1
beta = 0.1
evaporation_rate = 0.5
Q = 1

# Crear lista de ciudades
#cities = generate_cities(number_cities)

# Llamar a la función ant_colony_optimization
best_path, best_path_length = ant_colony_optimization(cities, number_ants, number_iterations, alpha, beta, evaporation_rate, Q)
✓ 1m 45.5s
```

El tiempo de ejecución cuando beta es 0.1 no vario significativamente comparado con la prueba anterior, y nos dio la siguiente gráfica:



De aquí podemos destacar la distancia mínima, a raíz de disminuir el factor beta lo que tiene sentido ya que no fomenta la exploración de nuevas rutas para las hormigas, arrojando ciertos valores con menos optimización para estos casos.

CONCLUSIONES

Ahora realice un conjunto mas amplio de pruebas sin embargo quiero dar ciertas conclusiones que pude obtener, que son las siguientes:

a) Aumentar el número de Hormigas incrementa el tiempo de ejecución del algoritmo, sin embargo, esto fomenta la exploración de mejores rutas, dando como resultado un camino mas optimizado entre mayor sea la cantidad de hormigas, sin embargo, el costo computacional podría ser demasiado elevado para realizar una prueba con un conjunto muy grande de hormigas.
b) Realice diversas pruebas cambiando el numero de iteraciones, dando un resultado homologo al caso de la cantidad de hormigas, si bien aumentar el numero de iteraciones facilitaba la obtención de rutas con mayor optimización, tomar un numero muy alto aumentaba el tiempo significativamente, y esto es a raíz del muy variado conjunto probabilístico del algoritmo
c) Al mantener un valor alto de Alpha aumentaba el peso de los caminos, dando como resultados rutas más optimas. Por otro lado, un valor muy bajo de Alpha causaba que se exploraran caminos que no fueran los más óptimos.
d) Un caso homologo con respecto a Alpha ocurre con beta, al mantener un factor beta alto las distancias toman mayor peso en las decisiones de las hormigas, por ende, los resultados con una beta mayor se encontraban más optimizados.
e) Configurar valores para Alpha y beta me permitía lograr un conjunto variado de optimizaciones, interprete estos dos valores como una balanza que me permitía compensar ciertos resultados y obtener valores mayormente óptimos.

f) Para el caso del factor de evaporación observe que un valor alto permitía resultados más óptimos, sin embargo, un valor demasiado alto causaba una rápida convergencia de soluciones por ende quedaban diversos caminos aun por explorar, así que una tasa de evaporación baja me permitía buscar una mayor combinación de soluciones optimas que dándole valores altos.
g) Un caso similar ocurre con el factor Q, debido a que es la intensidad de las feromonas dejadas por las hormigas, un valor alto causaría que en gran medida las hormigas sigan dichos caminos, dando como consecuencia una pronta convergencia del algoritmo dejando soluciones inexploradas, así que un valor mas bajo para Q puede permitir una mejor optimización.

Con las conclusiones anteriores quiero dejar por sentado que es a raíz de un análisis cuantitativo que genero el razonamiento cualitativo, me resulta particularmente difícil documentar un gran numero de diversas pruebas, asi que opte por realizarlas de manera individual, tomar nota de los resultados y analizarlos al finalizar, como consecuencia en las anteriores conclusiones no doy valores específicos para todos los casos, basta con los análisis mostrados en la primera parte.

Como conclusión general puedo decir que el algoritmo de colonia de hormigas es una buena herramienta para dar soluciones de optimización para un muy variado conjunto de problemas de la vida real y es necesario poder entender su funcionamiento y como lograr aplicar este algoritmo a dichos problemas requiere de un entendimiento de análisis de datos , comprensión de sistemas y conocimientos en procesos estocásticos.