Entrega de Mercancías Perecederas (VRP)

Proyecto 1 - Master IABD

Víctor Angulo | Marcos de Castro | Joshua Winn

1. Descripción de los requisitos del prototipo

a. Requisitos funcionales:

- **Optimización de rutas**:

El principal propósito del proyecto es la optimización de rutas de reparto minimizando el coste que genera usar los vehículos. No todos los vehículos deben ser utilizados si el resultado óptimo no lo requiere.

- **Gestión de flotas de vehículos**:

El proyecto de base cuenta con una flota de 6 vehículos con diferentes datos de capacidad, autonomía y coste por cada km recorrido. Usando estos vehículos se debe repartir todos las órdenes impuestas por los clientes.

- **Restricciones**:

Las principales restricciones del proyecto son:

La capacidad de los vehículos: Cada vehículo tiene su capacidad máxima que puede llevar, las rutas no pueden realizarse si la suma de las órdenes a entregar es superior a la capacidad máxima del vehículo.

La autonomía de los vehículos: Cada vehículo tiene una autonomía máxima que puede recorrer, se asume que el vehículo empieza su ruta con el depósito lleno y en ningún punto de la ruta puede parar a repostar más. Si la ruta del vehículo tiene más kilometraje que la autonomía del vehículo no se puede hacer.

Clientes con distancia 0: Algunos clientes en los datos aparecen con una distancia de 0 km entre ellos, esto significa que esa ruta de un cliente al otro no se puede hacer. Si una ruta contiene esa sucesión de clientes no es posible de hacer.

Los clientes solo se pueden visitar 1 vez: Cada cliente solo puede ser visitado 1 vez y por solo 1 vehículo. Ese vehículo debe satisfacer la demanda del cliente en su totalidad. Si las rutas contienen clientes duplicados no se pueden hacer.

Los vehículos empiezan y acaban en el almacén: Los vehículos deben empezar y acabar sus rutas en el almacén. Las distancias del almacén al cliente y viceversa se cuentan para el kilometraje final. Si una ruta no puede acabar en el almacén no puede ser hecha.

b. Requisitos no funcionales:

- **Escalabilidad**:

El proyecto es escalable en cuanto a la cantidad de vehículos siempre y cuando cada vehículo añadido tenga sus datos de capacidad, autonomía y coste por km. Parte de los casos de uso del proyecto requieren eliminar vehículos y aún con esa restricción el algoritmo debe devolver rutas correctas.

- **Facilidad de uso**:

Tanto si se ejecuta el proyecto como ficheros como si se ejecuta desde la página web el proyecto debe ser fácil de ejecutar. En el caso de los ficheros, deben estar estructurados de una forma lógica y de tal forma que al pulsar el botón de ‘Ejecutar todo’ se lance y de una respuesta clara. En el caso de la página web, debe ser fácil de leer y de entender qué hace cada componente de la página.

- **Tiempo de respuesta**:

El proyecto debe ser rápido al ejecutarse tanto como fichero único como página web. Idealmente esto serían unos pocos segundos y como máximo un minuto de carga.

2. Análisis de las tecnologías de IA usadas

Para la creación de rutas eficientes y que sigan las restricciones impuestas anteriormente hemos probado 3 tecnologías diferentes las cuales se van a explicar a continuación.

a. **ACO (Ant Colony Optimazation)**:

Utilizado para intentar sacar rutas óptimas mediante la creación de una "colonia de hormigas" las cuales mediante repeticiones y con la ayuda de una matriz de "feromonas" se guían hasta poder sacar una ruta eficiente.
El algoritmo se basa en diferentes bucles el primero el número de iteraciones que tiene que hacer el algoritmo, el segundo genera una solución (ruta) para cada vehículo (hormiga) en esa iteración, el tercer bucle actualiza la matriz de feromonas basándose en las soluciones encontradas en esa iteración, el cuarto y quinto bucle se centran en las rutas y por último el sexto bucle actualiza las feromonas en cada segmento de la ruta.
Las feromonas ayudan a las hormigas a ir a puntos y no estancarse siempre con las mismas rutas.

b. **Algoritmo Genético**:

Utilizado para intentar sacar rutas óptimas mediante la creación de una población de rutas, estas se pasan por una función Fitness que las evalúa y penaliza si no cumple las restricciones impuestas. Luego mediante una selección de torneo, un crossover o una mutación se modifica la mejor ruta para crear una nueva población y repetir el proceso hasta que se agote el número de generaciones impuesto al algoritmo.
La ruta final es la mejor que se ha encontrado durante el número de generaciones impuesto, que minimiza los costes siguiendo todas las restricciones impuestas.

c. **PULP (Python Universal Linear Programming)**:

Utilizado para intentar sacar rutas óptimas mediante una serie de variables de decisión, una función objetivo y una serie de restricciones que obligan al método a seguir. Al contrario del algoritmo genético para el PulP no hay que hacer casi código de funciones ya que es una librería que se puede añadir en python. Esta librería te da todos los métodos necesarios para poder crear un problema, 'lpSum' por ejemplo hace la suma de varios términos como las variables de decisión.
Al lanzar el método 'solve' del problema se empezará a buscar soluciones para el mismo, dependiendo de la complejidad de las restricciones y del

problema en si PulP puede tardar bastante tiempo en devolver una solución. Al terminar devolverá si la solución encontrada es “Infeasible” si no es posible hacer una solución, “Not Solved” si no ha tenido suficiente tiempo para encontrarla o “Optimal” si la solución encontrada es la mejor posible.

d. **RandomForestRegression**:

Algoritmo de predicción que hemos usado para sacar, mediantes los datos históricos de órdenes de pedidos, las órdenes de cada cliente para el mes siguiente.

3. Casos de uso

Antes de meternos a los diferentes casos de uso que se han tenido que completar en el proyecto habrá que saber qué datos nos han dado. Todos los datos venían en 5 archivos .csv los cuales hemos leído y trabajado con ellos usando Pandas.

- **DF_Locations:** Archivo excel con datos de los diferentes clientes y el almacén**.**

Nombre columnas	***Tipo de dato***	***Descripción***
Cliente	Object	Variable que da nombre a los 20 clientes y el almacén.
Latitud	Float64	Variable que marca la Latitud en un mapa de cada cliente.
Longitud	Float64	Variable que marca la Longitud en un mapa de cada cliente.

- **DF_Distance_Km**: Archivo excel que compara la distancia en km entre los 20 clientes y el almacén, los datos están en un formato de matriz de distancias.

Nombre columnas	***Tipo de dato***	***Descripción***
Cliente_X	Float64	Distancias en kilómetros entre X cliente o almacén.

- **DF_Historic_Order_Demand**: Archivo excel con el historial de ordenes de pedidos de los diferentes clientes, cada cliente tiene un número de pedidos diferente por cada mes.

Nombre columnas	***Tipo de dato***	***Descripción***
Cliente	Object	Variable que da nombre a los 20 clientes
mes_anio	Object	Variable que da los datos de mes y año para cada cliente, el formato de la fecha dada es Mes-Año.
order_demand	int64	Variable que muestra los pedidos recibidos por cada cliente en la fecha dada

- **DF_Orders**: Archivo excel con los datos de pedidos de los diferentes clientes durante el último mes, cada cliente tiene un número diferente de pedidos.

Nombre columnas	***Tipo de dato***	***Descripción***
Cliente	Object	Variable que da nombre a los 20 clientes
mes_anio	Object	Variable que da los datos de mes y año para cada cliente, el formato de la fecha dada es Mes-Año.
order_demand	int64	Variable que muestra los pedidos recibidos por cada cliente en la fecha dada

- **DF_Vehicles**: Archivo excel con los datos de los vehículos que se usan para el reparto de pedidos, cada vehículo tiene un identificador, su capacidad de almacenamiento, el coste de uso por km y la autonomía en kilómetros.

Nombre columnas	***Tipo de dato***	***Descripción***
vehiculo_id	int64	Variable con el identificador de los vehículos usados para el transporte

		de mercancías.
capacidad_kg	int64	Variable que da los datos de peso máximo de mercancías que puede llevar cada vehículo.
costo_km	float64	Variable que da los datos de coste en euros por cada kilómetro recorrido de cada vehículo.
autonomia_km	int64	Variable que da los datos de la autonomía de cada vehículo sin necesidad de repostar

Para este proyecto hemos tenido que cumplir con 4 casos de uso diferentes usando los datos mencionados anteriormente, los 3 primeros eran fijos y el cuarto a nuestra elección intentando cambiar algún parámetro que haga que el resultado sea diferente.
Antes de explicar los casos en detalle hay que indagar sobre qué algoritmos hemos acabado usando para resolverlos.

De los 3 algoritmos probados acabamos descartando el **Algoritmo Genético**, este algoritmo nos dio bastantes problemas sobretodo a la hora de pasar por la función de Crossover, las rutas la teníamos separadas en un formato de Diccionario en la que la Clave era el vehículo y los Valores eran las rutas que tomaba cada uno. Al pasar por la función Crossover se generaban clientes duplicados y al ser Diccionarios eran bastante difíciles de arreglar con lo cual acabamos decidiendo que no lo usaremos.

Cuando intentamos usar el algoritmo de **PuIP** nos empezó dando muy buenos resultados, tanto para el caso 1, 3 y 4 devolvía una ruta muy coherente y según el método “Óptimo”. Lo único es que al tener varias restricciones y valores que optimizar tardaba considerablemente más en devolver una solución comparado con los otros 2 algoritmos.
El problema lo encontramos cuando tuvimos que trabajar en el caso 2, para este caso al tener solo 3 vehículos no había combinación posible de estos que satisfaga la capacidad total necesaria para entregar todos los pedidos. Para solucionarlo había que implementar la posibilidad de volver al almacén para recoger más órdenes sin que afecte a la

capacidad. Esto significaba añadir varias restricciones más y una variable de decisión extra, al añadir esto el problema se volvió prácticamente imposible de encontrar una solución, probamos a darle 10 minutos y aún así devolvió como resultado “Not Solved”.

El algoritmo de ACO nos ha dado resultados variados por su parte, en los casos en el que PulP funcionaba bien nos daba mejor resultado que ACO, pero al contrario que con PulP con ACO pudimos hacer que los vehiculos pudiesen volver al Almacén y eso nos permitia resolver el caso 2 y 4 con mucha más comodidad y eficiencia.

a. **Caso 1** (Demanda del último mes con 6 vehículos):

Los clientes tienen una demanda fija para el mes y esta se debe repartir eficientemente (reduciendo costes totales) usando el número necesario de vehículos hasta un máximo de 6.
A continuación hay una explicación del código usado para resolver este ejercicio.

Lo primero que hay que hacer (asumiendo que se tienen las librerías descargadas e importadas) sería la lectura de datos, para resolver este caso necesitamos usar 4 excels de datos de los 5 que nos dieron al inicio:

df_orders: Este archivo nos da la demanda de órdenes de cada cliente.

df_vehicle: Este archivo contiene la información de los vehículos.

df_distance_km: Este archivo es la matriz de distancias entre los clientes y el almacén.

df_locations: Este archivo contiene la informacion de Longitud y Latitud de cada cliente y el almacén.

Para la lectura y manejo de los datos utilizamos la librería de Pandas tal y como se muestra en la imagen:

```
# Cargar datos
df_vehicle = pd.read_excel('../../Datos_P1/df_vehicle.xlsx')
df_orders = pd.read_excel('../../Datos_P1/df_orders.xlsx')
df_distance = pd.read_excel('../../Datos_P1/df_distance_km.xlsx')

df_distance.index = df_distance.columns
```

Las primeras 3 líneas son la lectura de los datos de excel y su conversión a DataFrame de Pandas.

La línea final hace que los nombres de las filas y las columnas sean iguales, esto facilita la lectura y comprensión de la matriz de distancias.

Como estamos trabajando con PulP lo siguiente que hay que hacer es definir las variables de decisión para que sean la base del problema. En la imagen se muestra las 3 que usamos:

```
# Crear variables de decisión
x = {(v, i, j): LpVariable(f"x_{v}_{i}_{j}", cat="Binary")
     for v in vehiculos for i in [almacen] + clientes for j in clientes + [almacen] if i != j and df_distance.at[i, j] > 0}

y = {(v, i): LpVariable(f"y_{v}_{i}", cat="Binary")
     for v in vehiculos for i in clientes}

# Variables auxiliares para evitar subciclos
u = {i: LpVariable(f"u_{i}", lowBound=0, upBound=len(clientes), cat="Continuous") for i in clientes}
```

- **Variable x[v, i, j]**: Controla que si x[v, i, j] = 1 el vehículo **v** viaja del punto **i** al punto **j**. Tampoco se permitirá que un nodo viaje a sí mismo **i != j** y que solo se permitirán crear pares (**i , j**) donde haya una distancia válida (En la matriz de distancias hay pares con distancia 0).
- **Variable y[v, i]:** Controla que si y[v, i] = 1 el vehículo ha visitado ese cliente, si por el contrario y[v, i] = 0 significa que no ha sido visitado. Esta variable se usa para garantizar que cada cliente ha sido atendido una sola vez.
- **Variable u[i]:** u[i] es una variable auxiliar para cada cliente la cual ayuda a prevenir subciclos.

Una vez tenemos las variables de decisión definidas lo siguiente que hacemos en el código es definir la función objetivo con el objetivo de minimizar el coste total del recorrido.

```
# Función objetivo: minimizar costo total del recorrido
problem += lpSum(x[v, i, j] * df_distance.at[i, j] * df_vehicle[df_vehicle["vehiculo_id"] == v]["costo_km"].values[0]
                 for v in vehiculos for i in [almacen] + clientes for j in clientes + [almacen] if i != j and df_distance.a
```

Cada parte de esto codigo busca hacer lo siguiente:

- Suma del coste total del recorrido de cada vehiculo
- Cada término multiplica los siguientes parametros.
- **x[v, i, j]**: SI el vehiculo v viaja de i a j
- **df_distance.at[i, j]**: Distancia entre i y j
- **costo_km** del vehiculo v.

Una vez definida la función objetivo lo siguiente es escribir las restricciones que se aplicarán al problema.

```python
# Restricción: Cada cliente debe ser visitado exactamente una vez
for i in clientes:
    problem += lpSum(y[v, i] for v in vehiculos) == 1

# Restricción: Capacidad del vehículo
for v in vehiculos:
    problem += lpSum(y[v, i] * df_orders[df_orders["cliente"] == i]["order_demand"].values[0]
                     for i in clientes) <= df_vehicle[df_vehicle["vehiculo_id"] == v]["capacidad_kg"].values[0]

# Restricción: Autonomía del vehículo
for v in vehiculos:
    problem += lpSum(x[v, i, j] * df_distance.at[i, j]
                     for i in [almacen] + clientes for j in clientes + [almacen] if i != j and df_distance.at[i, j] > 0) <=

# Restricción: Salida y llegada desde el almacén
for v in vehiculos:
    problem += lpSum(x[v, almacen, j] for j in clientes if df_distance.at[almacen, j] > 0) == 1
    problem += lpSum(x[v, i, almacen] for i in clientes if df_distance.at[i, almacen] > 0) == 1

# Restricción: Si un vehículo visita un cliente, debe salir de él
for v in vehiculos:
    for i in clientes:
        problem += lpSum(x[v, i, j] for j in clientes + [almacen] if i != j and df_distance.at[i, j] > 0) == y[v, i]
        problem += lpSum(x[v, j, i] for j in [almacen] + clientes if i != j and df_distance.at[j, i] > 0) == y[v, i]

# Restricción para evitar subciclos
for i in clientes:
    for j in clientes:
        if i != j and df_distance.at[i, j] > 0:
            problem += u[i] - u[j] + (len(clientes) * lpSum(x[v, i, j] for v in vehiculos)) <= len(clientes) - 1

# Resolver el problema con un límite de tiempo
problem.solve(PULP_CBC_CMD(timeLimit=60))
```

a. Cada cliente debe ser visitado exactamente una vez.
b. Restricción de capacidad de los vehiculos
c. Restricción de autonomia de los vehiculos
d. Cada vehiculo debe Salir y Regresar al almacén
e. Si un vehiculo visita un cliente, debe salir de él
f. Restriccion para evitar subciclos

Una vez fijadas todas las restricciones ya se puede lanzar la linea de poblem.solve() la cual le pasamos un parametro ‘timeLimit’ para que solo tarde un minuto en dar la solucion y despues de pruebas con 60 segundos era suficiente para sacar una solucion ‘Optima’.

Con esto ya tendriamos una solucion para el problema, ahora hay que representarla tanto por consola/en el jupyter como en un mapa.
Para lo primero tenemos esta pieza de código:

```
costo_total = 0
rutas_ordenadas_mapa = []
print("\nResumen de rutas:\n" + "="*40)
for v in sorted(vehiculos):  # Ordenar los vehículos
    costo_vehiculo = 0
    peso_total = 0
    print(f"\n🚛 Vehículo {v}:")
    ruta = []
    for i in [almacen] + clientes:
        for j in clientes + [almacen]:
            if i != j and df_distance.at[i, j] > 0 and x[v, i, j].varValue == 1:
                distancia = df_distance.at[i, j]
                costo_km = df_vehicle[df_vehicle["vehiculo_id"] == v]["costo_km"].values[0]
                costo_vehiculo += distancia * costo_km
                if i != almacen:
                    peso_total += df_orders[df_orders["cliente"] == i]["order_demand"].values[0]
                ruta.append((i, j, distancia, distancia * costo_km))


    # Ordenar la ruta en secuencia basándose en el recorrido lógico
    ruta_ordenada = [ruta.pop(0)]

    while ruta:
        for i, j, distancia, costo in ruta:
            if ruta_ordenada[-1][1] == i:
                ruta_ordenada.append((i, j, distancia, costo))
                ruta.remove((i, j, distancia, costo))
                break

    for i, j, distancia, costo in ruta_ordenada:
        print(f"   - De {i} a {j}: {distancia} km, Costo: ${costo:.2f}")
        rutas_ordenadas_mapa.append((i, j))

    capacidad = df_vehicle[df_vehicle["vehiculo_id"] == v]["capacidad_kg"].values[0]
    print(f"    🔹 Peso total transportado: {peso_total} kg / Capacidad: {capacidad} kg")
    print(f"    🔹 Costo total del vehículo: ${costo_vehiculo:.2f}")
    costo_total += costo_vehiculo

print("\n💰 Costo total de todas las rutas: ${:.2f}".format(costo_total))
```

En resumidas cuentas este codigo pasa vehiculo a vehiculo y recoge la ruta que recorren junto con los datos de distancia recorrida y el coste total al multiplicar la distancia por el consumo del vehiculo.
El bucle while pone las rutas en orden y las va metiendo a la lista de ruta_ordenada y en el bucle for siguiente se devuelven los datos con un print.

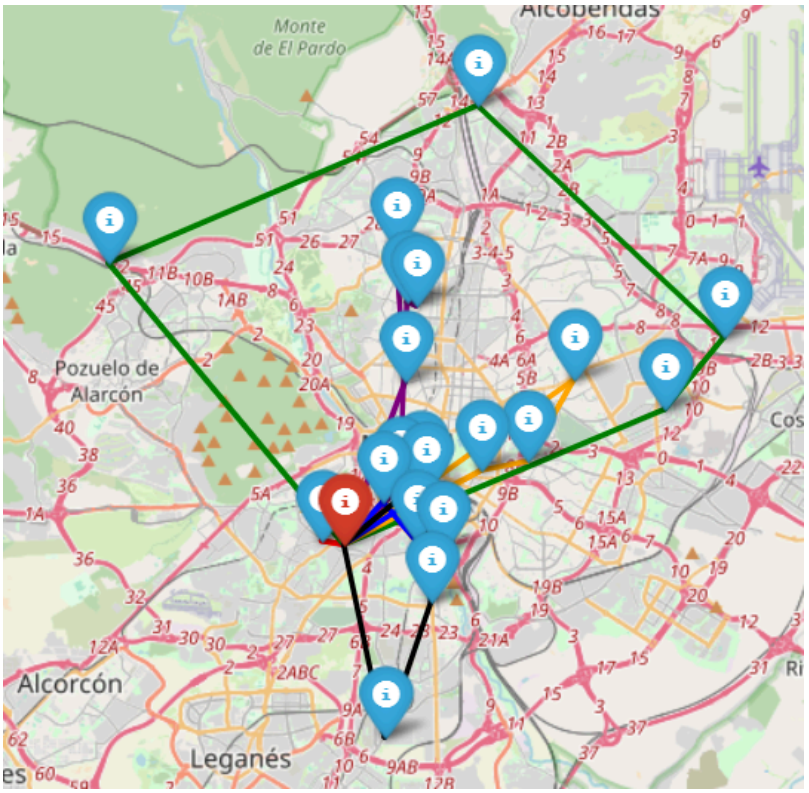
Los diferentes prints acaban formando este texto resultado:

```
🚛 Vehículo 5:
   - De Almacén a Cliente_1: 3.6114 km, Costo: $1.16
   - De Cliente_1 a Cliente_14: 2.2133 km, Costo: $0.71
   - De Cliente_14 a Cliente_11: 1.6099 km, Costo: $0.52
   - De Cliente_11 a Cliente_6: 2.889 km, Costo: $0.92
   - De Cliente_6 a Almacén: 2.6952 km, Costo: $0.86
    🔹 Peso total transportado: 3691 kg / Capacidad: 10000 kg
    🔹 Costo total del vehículo: $4.17

🚛 Vehículo 6:
   - De Almacén a Cliente_17: 8.5221 km, Costo: $1.19
   - De Cliente_17 a Cliente_5: 5.287199999999999 km, Costo: $0.74
   - De Cliente_5 a Cliente_16: 5.434699999999999 km, Costo: $0.76
   - De Cliente_16 a Almacén: 4.155399999999999 km, Costo: $0.58
    🔹 Peso total transportado: 2838 kg / Capacidad: 3129 kg
    🔹 Costo total del vehículo: $3.28

💰 Costo total de todas las rutas: $29.14
```

Usando estas rutas ordenadas y con la ayuda de folium y los datos de localizacion de los clientes en un mapa podemos sacar un mapa con las rutas que sacamos para resolver este caso.



b. **Caso 2** (Demanda del último mes con reducción de flota):

Los clientes tienen una demanda fija para el mes y esta se debe repartir eficientemente (reduciendo costes totales) usando el número necesario de vehículos hasta un máximo de 3.

Para resolver este caso hemos utilizado el algoritmo ACO ya que con PulP no hemos podido conseguir que el camión pueda volver al almacén para recargar y poder hacer más viajes, al añadir las restricciones para ello el problema no podia ser resuelto, o PulP devolvia “Not Solved” despues de 10 minutos pensando o “Infeasible”.
A continuación voy a explicar las diferentes funciones que se usan para resolver el problema.

```python
def ACO(caso, df_distances, df_vehicles, df_customers):
    # Parámetros de ACO
    num_vehiculos = 6
    num_iterations = 100
    rho = 0.5
    Q = 100

    n_nodes = len(df_customers)+1
    pheromone_matrix = np.ones((n_nodes, n_nodes))

    if caso == 2 :
        num_vehiculos = 3

    solucion_optima = None
    mejor_precio = float('inf')

    for iteration in range(num_iterations):
        # Reordenar aleatoriamente los vehículos en cada iteración
        df_vehicles_shuffled = df_vehicles.sample(frac=1).reset_index(drop=True)

        soluciones = []
        costes = []

        for _ in range(num_vehiculos):
            solution, _ = solucion_Vehiculo(df_distances, df_vehicles_shuffled, df_customers["order_demand"],pheromone_matrix, None)
            total_cost, _ = calculo_Coste(solution, df_distances, df_vehicles_shuffled)
            soluciones.append(solution)
            costes.append(total_cost)

            if total_cost < mejor_precio:
                mejor_precio = total_cost
                solucion_optima = solution

        # Actualizar feromonas
        pheromone_matrix *= (1 - rho)
        for solution, cost in zip(soluciones, costes):
            for vehicle_name, routes in solution:
                for route in routes:
                    for i in range(len(route) - 1):
                        pheromone_matrix[route[i]][route[i + 1]] += Q / cost

    print(solucion_optima, mejor_precio)
    return solucion_optima, mejor_precio
```

La función ACO establece los parametros del algoritmo ACO: Número de vehículos, iteraciones, alpha, beta, rho y Q. Inicializa la matriz de feromonas con unos.

El **número de vehículos** se usa para determinar la cantidad de vehículos disponibles que hay.

Alpha y Beta: Controlan el equilibrio entre la exploración (intentar nuevas rutas) y la explotación (seguir rutas conocidas).

- Si alpha es alto y beta es bajo, el algoritmo explota rutas con alta feromona.

- Si alpha es bajo y beta es alto, el algoritmo explora rutas más cortas.

El **rho**: Controla cuánto se "olvida" el algoritmo de las rutas anteriores, permitiendo explorar nuevas soluciones.

Y el **Q**: Afecta la intensidad del refuerzo de feromona en las rutas buenas.

En el **primer bucle** controla el número de iteraciones (ciclos completos) que el algoritmo ejecutará para encontrar una solución óptima:

- En cada iteración, el algoritmo genera nuevas soluciones (rutas) y actualiza la matriz de feromonas.
- Cuantas más iteraciones, más oportunidades tiene el algoritmo de explorar el espacio de búsqueda y mejorar la solución.
- Dentro del primer bucle reordenamos aleatoriamente los vehículos en cada iteración:
    - Esto asegura que el orden en que se procesan los vehículos cambie en cada iteración, lo que ayuda a explorar diferentes combinaciones de rutas.
    - Evita que en pruebas unitarias si se intenta insertar un vehículo con valores atípicos, este evite usarlo si es perjudicial para la salida de los costes.

En el **segundo bucle** generar una solución (ruta) para cada vehículo (hormiga) en la iteración actual.

- Cada vehículo actúa como una "hormiga" que construye una solución (ruta) basada en las feromonas y la heurística.
- Se generan num_vehiculos soluciones en cada iteración.
- Además de apartir de aquí se llama a las funciones solucion_Vehiculo y calculo_Coste para encontrar el mejor vehículo y precio en cada ruta
- Se recoge la mejor solución aportada actualizándose o no en cada iteración.
- Actualiza las feromonas (La evaporación de feromonas reduce la influencia de las rutas antiguas, permitiendo que el algoritmo explore nuevas soluciones.)

En el **tercer bucle** actualiza la matriz de feromonas basándose en las soluciones generadas en la iteración actual.

- Esto guía a las hormigas en iteraciones futuras hacia rutas prometedoras.

El **cuarto bucle** se centra en las rutas, recorre las rutas de cada vehículo en la solución actual.:

- Cada vehículo puede tener múltiples rutas (viajes), y este bucle procesa cada una de ellas.

El **quinto bucle** recorre cada ruta (viaje) del vehículo:

- Siendo una secuencia de nodos (clientes y almacén) que el vehículo visita.

Y en el **sexto bucle** actualiza las feromonas en cada segmento de la ruta:

- Para cada par de nodos consecutivos en la ruta, se refuerza la feromona en la matriz.
- La cantidad de feromona depositada es proporcional a Q / cost, donde Q es una constante y cost es el costo total de la solución.

```python
def calculo_Probabilidad(current_node, unvisited_nodes, pheromones, df_distances):
    #Estas dos variables imprescindibles para ACO
    alpha = 1.0
    beta = 2.0
    probabilities = []
    for node in unvisited_nodes:
        distance = df_distances.iloc[current_node, node]
        if distance > 0:
            tau = pheromones[current_node][node] ** alpha
            eta = (1 / distance) ** beta
            probabilities.append(tau * eta)
        else:
            probabilities.append(0)

    probabilities = np.array(probabilities)
    total = probabilities.sum()

    # Evitar misma posicion entre clientes (DUDA)
    if total == 0:
        probabilities = np.ones(len(unvisited_nodes)) / len(unvisited_nodes)
    else:
        probabilities = probabilities / total

    return probabilities
```

- **calculo_Probabilidad**: determina las probabilidades de moverse a un nodo no visitado basándose en la feromona y la distancia.
    - Calcula la probabilidad de elegir el siguiente nodo basado en la feromona acumulada (tau) y la heurística inversa de la distancia (eta).

- Usa alpha (importancia de la feromona) y beta (importancia de la distancia) mencionados previamente.

```python
def solucion_Vehiculo(df_distances, df_vehicles_shuffled, orders,pheromone_matrix, autonomia_restante=None):
    solution = []
    remaining_customers = set(range(20))
    depot = 20

    if autonomia_restante is None:
        autonomia_restante = {row["vehiculo_id"]: row["autonomia_km"] for _, row in df_vehicles_shuffled.iterrows()}

    for _, vehicle in df_vehicles_shuffled.iterrows():  # Usar el DataFrame reordenado
        vehicle_id = vehicle["vehiculo_id"]
        vehicle_routes = []
        vehicle_capacity = vehicle["capacidad_kg"]
        current_autonomy = autonomia_restante[vehicle_id]

        while remaining_customers and current_autonomy > 0:
            current_route = [depot]  # Inicia en el almacén
            current_capacity = vehicle_capacity
            current_node = depot
            total_distance = 0

            while remaining_customers:
                unvisited_nodes = [
                    node for node in remaining_customers
                    if orders[node] <= current_capacity
                    and (total_distance + df_distances.iloc[current_node, node] + df_distances.iloc[node, depot]) <= current_autonomy
                ]

                if not unvisited_nodes:
                    break

                probabilities = calculo_Probabilidad(current_node, unvisited_nodes, pheromone_matrix, df_distances)
                next_node = np.random.choice(unvisited_nodes, p=probabilities)

                distance_to_next = df_distances.iloc[current_node, next_node]
                total_distance += distance_to_next
                current_autonomy -= distance_to_next

                current_route.append(next_node)
                current_capacity -= orders[next_node]
                remaining_customers.remove(next_node)
                current_node = next_node

            # Verificar si se agregaron clientes a la ruta
            if len(current_route) == 1:  # Solo el almacén, no hay clientes
                break  # Salir del bucle externo para este vehículo
            else:
                # Regresar al almacén y actualizar autonomía
                distance_to_depot = df_distances.iloc[current_node, depot]
                current_autonomy -= distance_to_depot
                current_route.append(depot)
                if len(current_route) > 2:
                    vehicle_routes.append(current_route)

            # Actualizar autonomía restante después de cada ruta
            autonomia_restante[vehicle_id] = current_autonomy

        if vehicle_routes:
            solution.append((vehicle_id, vehicle_routes))

    return solution, autonomia_restante
```

- Luego, **solucion_Vehiculo** intenta construir rutas para cada vehículo, teniendo en cuenta la capacidad y autonomía. Si un vehículo no puede continuar, pasa al siguiente gracias a las modificaciones anteriores.
    - Genera rutas para vehículos en un problema de enrutamiento con capacidad y autonomía limitada.
    - Considera restricciones de capacidad de carga y autonomía de cada vehículo al seleccionar clientes.

- Utiliza feromonas y distancias con calculo_Probabilidad para elegir la mejor siguiente parada.
- Cada vehículo parte del almacén (depot = 20) y regresa una vez completada su ruta.
- Si un vehículo no puede atender más clientes, se pasa al siguiente.

```python
# costo por vehiculo
def calculo_Coste(solution, df_distances, df_vehicles):
    total_cost = 0
    vehicle_costes = []
    for vehicle_name, routes in solution:
        vehicle = df_vehicles.loc[df_vehicles["vehiculo_id"] == vehicle_name]
        price_per_km = vehicle["costo_km"].values[0]
        vehicle_total = 0

        for route in routes:
            route_cost = sum(df_distances.iloc[route[i], route[i + 1]] for i in range(len(route) - 1))
            vehicle_total += route_cost * price_per_km
        total_cost += vehicle_total
        vehicle_costes.append((vehicle_name, vehicle_total))
    return total_cost, vehicle_costes
```

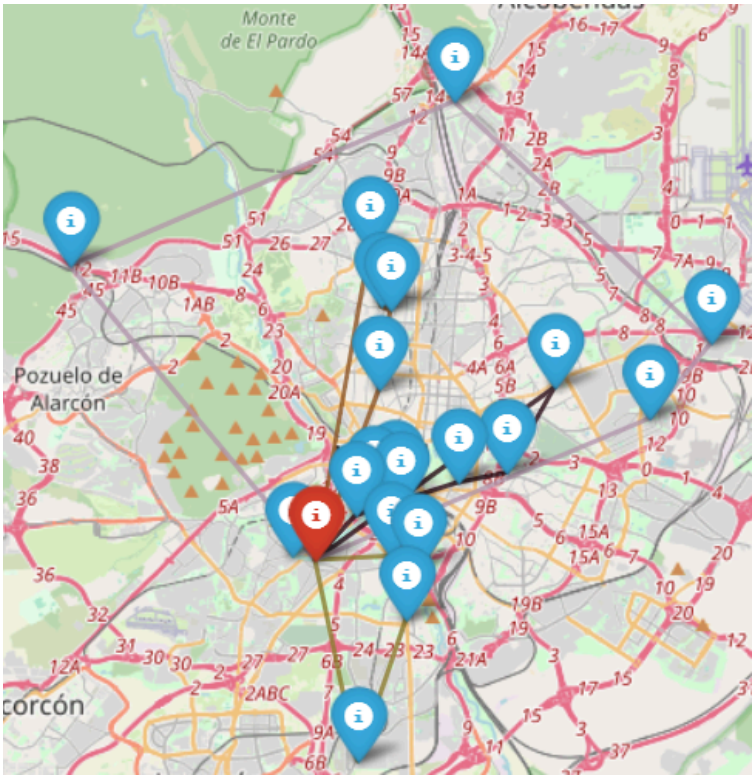
La función **calculo_Coste** calcula el costo total de las rutas generadas. En el bucle principal, se generan soluciones para cada hormiga, se actualiza la feromona y se busca la solución óptima. Finalmente, se imprime y retorna la mejor solución y su costo.

- Usa df_distances para determinar la distancia entre nodos y df_vehicles para obtener el costo por kilómetro de cada vehículo.
- Suma los costos de cada ruta, multiplicando la distancia recorrida por el costo por kilómetro.
- Acumula el costo total y almacena el costo individual de cada vehículo en vehicle_costes.

Una vez pasadas todas las funciones anteriores el algoritmo se queda con la mejor solución de cada iteración y al final de todas las iteraciones mostrará la mejor solucion en cuanto a ruta y coste como la mostrada a continuación.

```
✅ Mejor solución encontrada:
[(2.0, [[20, 8, 0, 3, 15, 20], [20, 5, 9, 11, 17, 20], [20, 10, 13, 4, 16, 20], [20, 14, 7, 18, 19, 20], [20, 12, 1, 6, 2,
20]])]
🏅 Mejor precio: 22.77
```

y tambien se mostrará un mapa con las rutas pintadas:



c. **Caso 3** (Predicción de demanda del mes siguiente):

Usando los datos históricos de cada cliente, se debe hacer una predicción de la demanda del próximo mes y en base a esos datos crear las rutas usando el número necesario de vehículos hasta un máximo de 6.
Para resolver este caso usando PulP hemos seguido las mismas variables y restricciones que para el caso 1. La dificultad de este caso reside en hacer la predicción de los casos del mes siguiente, para ello hemos hecho una prediccion con el algoritmo de **RandomForestRegressor.**

Para este caso vamos a necesitar 4 de los 5 excels que se nos entregaron.

df_historic_order_demand: Este archivo nos da la demanda de órdenes de cada cliente los ultimos años.

df_vehicle: Este archivo contiene la información de los vehículos.

df_distance_km: Este archivo es la matriz de distancias entre los clientes y el almacén.

df_locations: Este archivo contiene la informacion de Longitud y Latitud de cada cliente y el almacén.

Lo primero al estar trabajando con un DataFrame que se va a usar para predecir es limpiarlo de valores nulos y outliers que pueden afectar a la prediccion.

```
# Cargar datos
order = df_order_historic_demand.dropna(subset=['order_demand'])
#IQR
Q1 = order['order_demand'].quantile(0.25)
Q3 = order['order_demand'].quantile(0.75)
IQR = Q3 - Q1

# Esto lo usamos para poner un limite a nuestros datos(vamos a darle chamba de outliers)
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

order = order[(order['order_demand'] >= lower_bound) & (order['order_demand'] <= upper_bound)]
order['mes_anio'] = pd.to_datetime(order['mes_anio'], format='%m-%Y')
order['month'] = order['mes_anio'].dt.month
order['year'] = order['mes_anio'].dt.year
```

En nuestro caso hemos decidido dropear todos los valores que no estan completos con dropna y sacamos el quantil del 25% y 75% de los datos. Usando los quantiles podemos sacar el rango interquantilico IQR y si multiplicamos el IQR por 1.5 y se lo sumamos al Q3 y restamos al Q1 tenemos los valores maximos y minimos para detectar outliers.
Por ultimo eliminamos los outliers con la linea:
order = order[(order['order_demand'] >= lower_bound) & (order['order_demand'] <= upper_bound)]
Cambiamos el tipo de dato de la columna mes_anio a datetime con el formato ‘%m-%Y’ y sacamos 2 columnas separadas con los datos de mes y año.

```python
resultados = []

for cliente in order['cliente'].unique():
    cliente_data = order[order['cliente'] == cliente]

    if len(cliente_data) < 2:
        continue

    X = cliente_data[['month', 'year']]
    y = cliente_data['order_demand']
    model = RandomForestRegressor(n_estimators=100, random_state=42)
    model.fit(X, y)

    #enero de 2025
    next_month = pd.DataFrame({'month': [1], 'year': [2025]})
    predicted_demand = model.predict(next_month)[0]


    resultados.append({
        'cliente': cliente,
        'mes_anio': '2025-01',
        'order_demand': predicted_demand
    })

df_orders = pd.DataFrame(resultados)
```

Lo siguiente es cliente por cliente devolver los datos historicos, separarlos en una variable X que contenga el mes y año, la variable Y que habrá que predecir luego. Se genera un modelo usando la X y la Y de RandomForestRegressor y se hace el fit.
Para sacar los datos nuevos hacemos el .predict al modelo y nos devuelve la demanda predicha, todo esto se guarda en la variable de resultados y se convierte a DataFrame de Pandas.

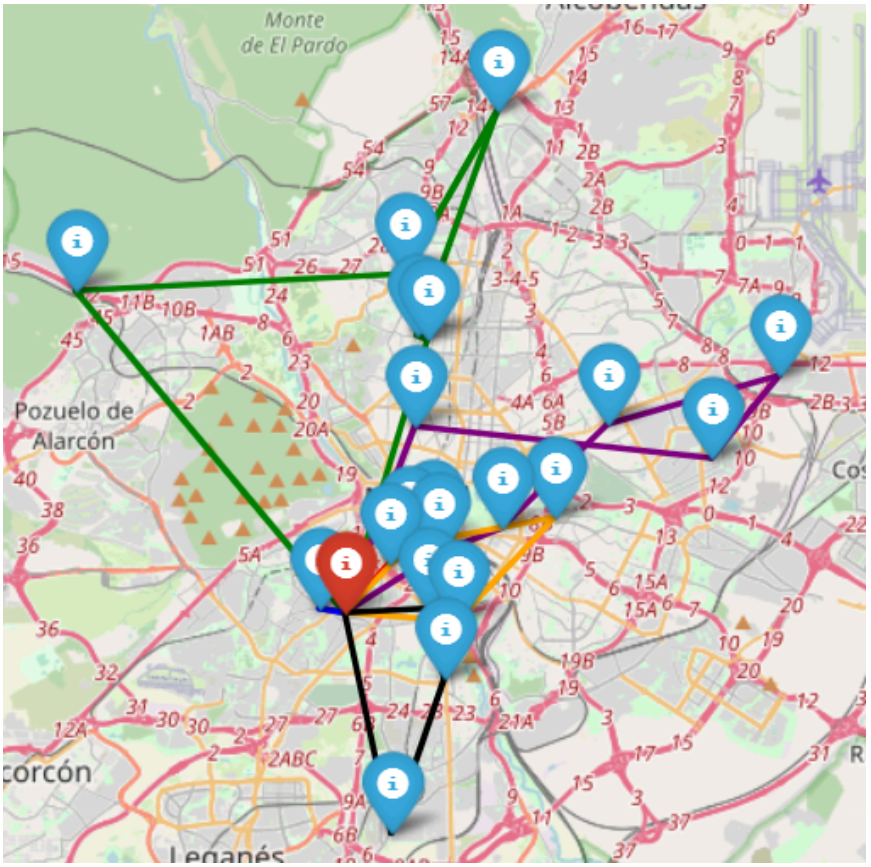
Como el resto del código es similar al del caso 1 no lo voy a explicar, pero pongo por aqui las capturas de que los resultados son diferentes y el mapa final tambien.

🚛 Vehículo 5:
- De Almacén a Cliente_9: 1.0494 km, Costo: $0.34
- De Cliente_9 a Almacén: 1.0494 km, Costo: $0.34
  🔹 Peso total transportado: 749.63 kg / Capacidad: 10000 kg
  🔹 Costo total del vehículo: $0.67

🚛 Vehículo 6:
- De Almacén a Cliente_11: 2.912 km, Costo: $0.41
- De Cliente_11 a Cliente_5: 2.6253 km, Costo: $0.37
- De Cliente_5 a Cliente_17: 5.287199999999999 km, Costo: $0.74
- De Cliente_17 a Almacén: 8.5221 km, Costo: $1.19
  🔹 Peso total transportado: 2203.96 kg / Capacidad: 3129 kg
  🔹 Costo total del vehículo: $2.71

💰 Costo total de todas las rutas: $25.51

y el mapa con las rutas finales:



d. **Caso 4** (Demanda del último mes con 1 vehículo modificado):

Los clientes tienen una demanda fija para el mes y esta se debe repartir eficientemente (reduciendo costes totales) usando un único vehículo el cual estará con datos modificados para que sea viable.
El caso 4 tambien lo hemos resuelto usando ACO ya que con PulP volviamos al problema de crear una restricción para que el vehiculo pueda volver al

almacén a recargar, esto ACO si que lo hace bien.

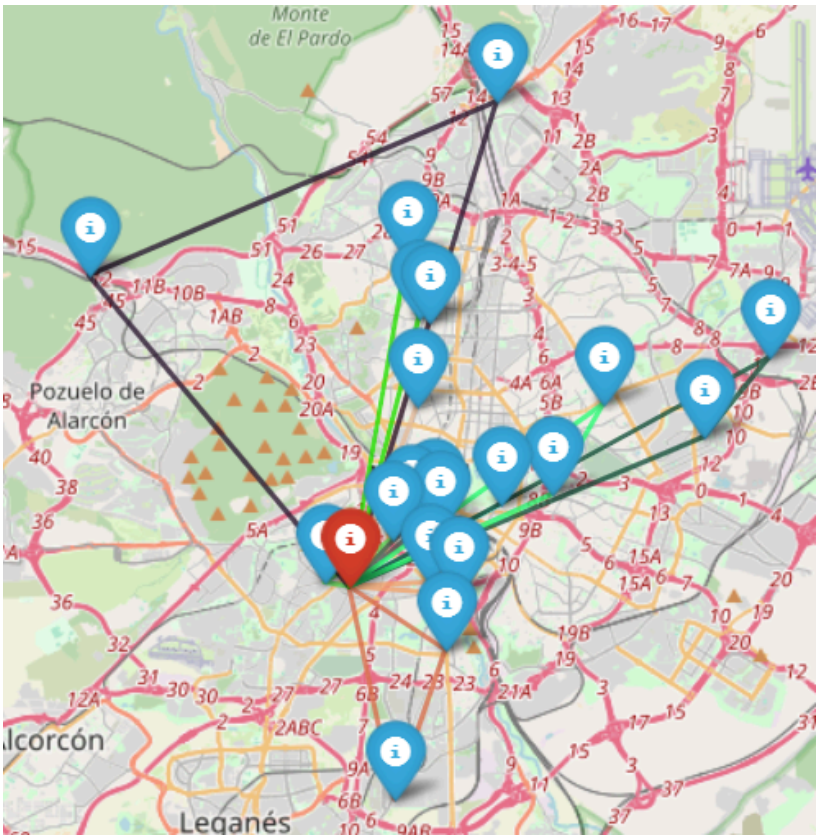
```
# Diferencia clave para Caso 2
df_vehicles = df_vehicles.sample(n=1, random_state=42)
print(df_vehicles)
```

```
   vehiculo_id  capacidad_kg  costo_km  autonomia_km
0            1          2026       0.2           603
```

Usando esta linea podemos sacar un vehiculo aleatorio de los 6 que hay dentro del DataFrame de vehiculos.

Por lo general todo el resto del codigo es similar al explicado en el Caso 2 asi que pondré unas imagenes del resultado final y el mapa de las rutas.

```
✅ Mejor solución encontrada:
[(np.float64(1.0), [[20, np.int64(8), np.int64(5), 20], [20, np.int64(12), np.int64(1), 20], [20, np.int64(15), np.int64
(3), 20], [20, np.int64(10), np.int64(13), 20], [20, np.int64(0), np.int64(9), 20], [20, np.int64(4), np.int64(16), 20], [2
0, np.int64(7), np.int64(14), 20], [20, np.int64(11), np.int64(17), 20], [20, np.int64(19), np.int64(18), 20], [20, np.int6
4(6), np.int64(2), 20]])]
💰 Mejor precio: 46.71
```



Aunque las rutas aparezcan en diferentes colores son todas con el mismo vehiculo.

4. Detalles tecnicos

a. Tecnologías y librerías utilizadas

- Python:

Lenguaje de programación versátil y ampliamente utilizado en desarrollo de software, análisis de datos e inteligencia artificial. Su facilidad de uso y extensa comunidad lo hacen ideal para proyectos de optimización y machine learning.

- Jupyter Notebook:

Entorno interactivo que permite ejecutar código en celdas, visualizar resultados y documentar procesos en un mismo archivo. Es muy útil para el desarrollo y prueba de algoritmos de optimización y análisis de datos.

- Pandas:

Librería de Python especializada en manipulación y análisis de datos. Proporciona estructuras de datos eficientes, como DataFrames, para manejar grandes volúmenes de información de manera estructurada.

- Numpy:

Biblioteca fundamental para cálculos numéricos en Python. Permite realizar operaciones matemáticas eficientes con matrices y arreglos, optimizando el procesamiento de datos en algoritmos de optimización.

- Matplotlib (pyplot):

Librería de visualización de datos en Python. Se utiliza para generar gráficos y representar visualmente los resultados de los análisis y optimización de rutas.

- Folium:

Librería de Python utilizada para la visualización de datos geoespaciales sobre mapas interactivos. Se basa en Leaflet.js, una potente biblioteca de mapas en JavaScript. Folium es ideal para proyectos que requieren representar información geográfica, como análisis de rutas en VRP (Vehicle Routing Problem).

- PULP:

Librería de optimización en Python que permite modelar y resolver problemas de programación lineal y entera. Es clave en la implementación del VRP, ya

que facilita la formulación matemática del problema y su resolución con distintos solvers.

- Flask:

Microframework de Python para el desarrollo de aplicaciones web. En este proyecto, se utiliza para construir una API que permite la interacción entre el sistema de optimización y los usuarios a través de una interfaz web.