1. Introducción

El comercio electrónico ha experimentado una transformación significativa en los últimos cinco años, consolidándose como un canal fundamental para las empresas a nivel global. Entre los últimos años, con la evolución tecnológica, los cambios en los hábitos de consumo y la aceleración provocada por la pandemia de COVID-19 han sido factores determinantes. Las empresas han adoptado soluciones digitales avanzadas, como inteligencia artificial, análisis de datos y plataformas multicanal, para mejorar la experiencia del cliente. Además, se ha incrementado la competencia en el sector, con la entrada de nuevos actores y la diversificación de modelos de negocio, como el comercio social y las suscripciones personalizadas.

En este periodo, los consumidores han mostrado una creciente preferencia por las compras en línea debido a la comodidad, la personalización y la disponibilidad de opciones. Este cambio ha impulsado a las empresas a desarrollar estrategias de marketing digital más sofisticadas, integrando SEO, redes sociales y campañas automatizadas. Paralelamente, la importancia de la logística y las entregas rápidas se ha intensificado, fomentando la colaboración con terceros. Este contexto ha consolidado el comercio electrónico como un pilar estratégico para las empresas, tanto para capturar nuevos mercados como para adaptarse a las expectativas cambiantes de los consumidores.

La empresa IA DELIVERY S.L. con sede en Madrid, se dedica a la manufactura de productos alimenticios perecederos de corta duración. Han identificado que una parte significativa de sus costes operativos proviene de la externalización de servicio de distribución a terceros. Tras realizar un análisis exhaustivo de estos costes, han decidido internalizar este proceso y crear su propia flota de vehículos, optimizando así la distribución de sus productos desde sus instalaciones centrales.

Con el fin de reducir costos y mejorar la eficiencia, la empresa busca desarrollar una herramienta de optimización de rutas. Esta herramienta deberá calcular las rutas más eficientes para la distribución de sus productos, teniendo en cuenta factores como la cantidad mínima de vehículos necesaria, las distancias a recorrer, el tiempo de entrega y la capacidad de los vehículos. El objetivo final es minimizar el uso de recursos y garantizar una distribución efectiva con la menor inversión de flota propia.

2. Descripción general

2.1 Objetivos

2.1.1 Objetivo general

Se busca minimizar los costes totales generados por el reparto de artículos vendidos y maximizar el número de entregas con la flota de vehículos disponibles, con el objetivo de alcanzar el máximo beneficio disponible.

2.1.2 Objetivos específicos

- Realizar el análisis del histórico de datos de entregas y clientes de la empresa para conocer el contexto de ventas de la empresa.
- Desarrollar un modelo que planifique las rutas de entrega, minimizando el coste total y maximizando el número de entregas realizadas cumpliendo las restricciones de capacidad, autonomía y vehículos disponibles.
- Suponiendo que la empresa tenga que llevar acabo una reducción de su flota de vehículos disponibles para la entrega de pedidos (por causas de economía o mantenimiento), optimizar las rutas para mantener el máximo de los beneficios.
- Desarrollar un modelo de demanda de pedidos que predice el número de pedidos esperados para el próximo mes y desarrollar las rutas para cumplir con la demanda prevista y ajustarse a los pedidos futuros.

2.2 Business Criteria

Para conocer si con el paso del tiempo el modelo está funcionando correctamente, se podrá comparar los datos de valores de tiempo y distancias entre las rutas, teniendo en cuenta que se busca el mínimo de tiempo invertido para la entrega de pedidos, y el máximo de entregas realizadas.

3. Data understanding

Proporciona la base para tomar decisiones informadas sobre cómo procesar, transformar y analizar los datos en las etapas posteriores del proyecto. Una comprensión sólida de los datos asegura un análisis más efectivo y resultados de mayor calidad. Para el proyecto nos ceñiremos a los datos proporcionados únicamente.

3.1 Data exploration report

Para el desarrollo del proyecto, disponemos de 6 archivos .csv con el contenido de datos a estudiar para la resolución y obtención del objetivo.

- **df_distance_km:** contiene los datos entre las localizaciones en km

Nombre	Tipo	Descripción
cliente_k	Numérico	Contiene la distancia de Cliente_k a Cliente_X
almacen	Numérico	Contiene la distancia de Almacen a Cliente_X

- **df_distance_min:** contiene las datos de tiempo entre las localizaciones en minutos

Nombre	Tipo	Descripción
cliente_k	Numérico	Contiene el tiempo de Cliente_k a Cliente_X
almacen	Numérico	Contiene el tiempo de Almacen a Cliente_X

- **df_historic_order_demand:** registro del histórico de pedidos realizados

Nombre	Tipo	Descripción
cliente	Categórica	Contiene el cliente que realiza el pedido
mes_anio	Fecha	Contiene el dato de mes y año en que se realiza el pedido
order_demand	Numérico	Contiene el peso en kg del pedido

- **df_location:** contiene los datos de localizaciones

Nombre	Tipo	Descripción
cliente	Categórica	Contiene el cliente que realiza el pedido
latitud	Numérico	Contiene el valor de la latitud del cliente
longitud	Numérico	Contiene el valor de la longitud del cliente

- **df_orders:** registro del mes de diciembre de pedidos realizados

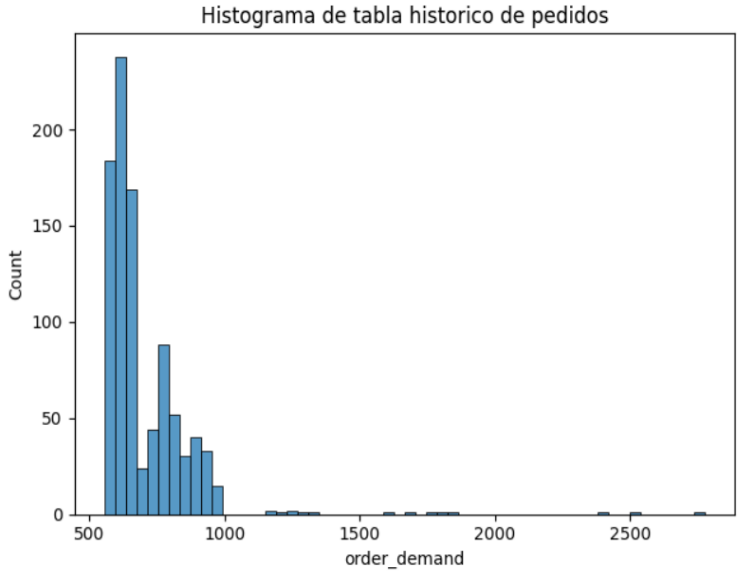
Nombre	Tipo	Descripción
cliente	Categórica	Contiene el cliente que realiza el pedido
mes_anio	Fecha	Contiene el dato de mes y año en que se realiza el pedido
order_demand	Numérico	Contiene el peso en kg del pedido

- **df_vehicle:** contiene los datos de los vehículos disponibles

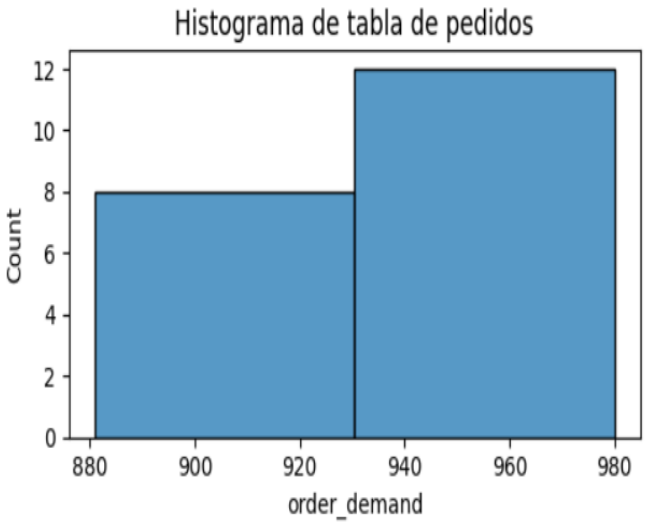
Nombre	Tipo	Descripción
vehiculo_id	Numérica	Contiene el valor del id único del vehículo
capacidad_kg	Numérica	Contiene el valor de la capacidad máxima que permite
costo_km	Numérico	Contiene el valor de coste por km de cada vehículo
autonomia_km	Numérico	Contiene el valor de la autonomía disponible de cada vehículo

3.2 Infographics

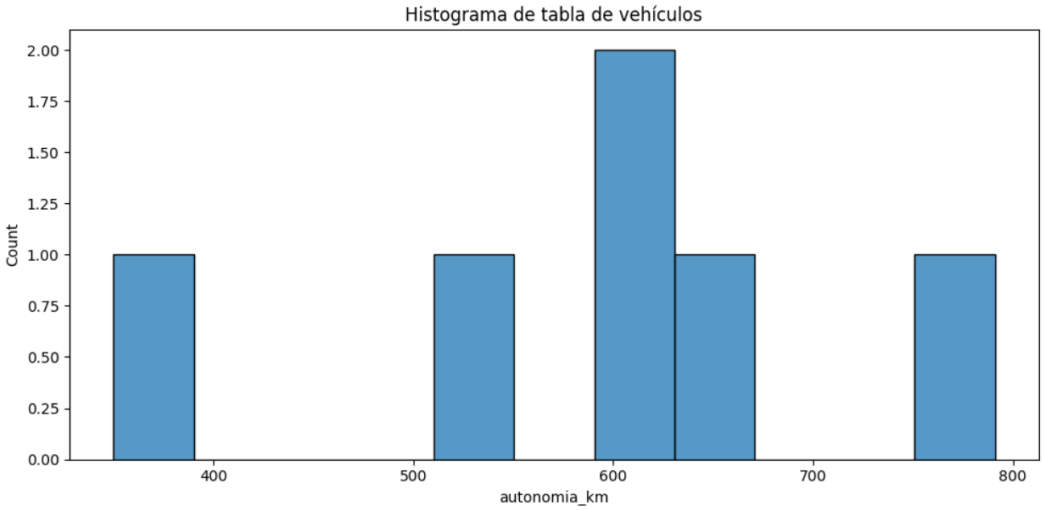
- **Histograma de tablas historico de pedidos:** Encontramos la comparación entre el peso y la cantidad del pedido de los clientes, se observa que la mayoria de pedidos, tienen un peso entre 500 y 1000 kg.



- **Historico de tabla de pedidos:** Encontramos la comparación entre el peso y la cantidad de los pedidos realizados por los clientes en el último mes y se observa que la mayoria de pedidos, tienen un peso cercano a los 1000 kg.

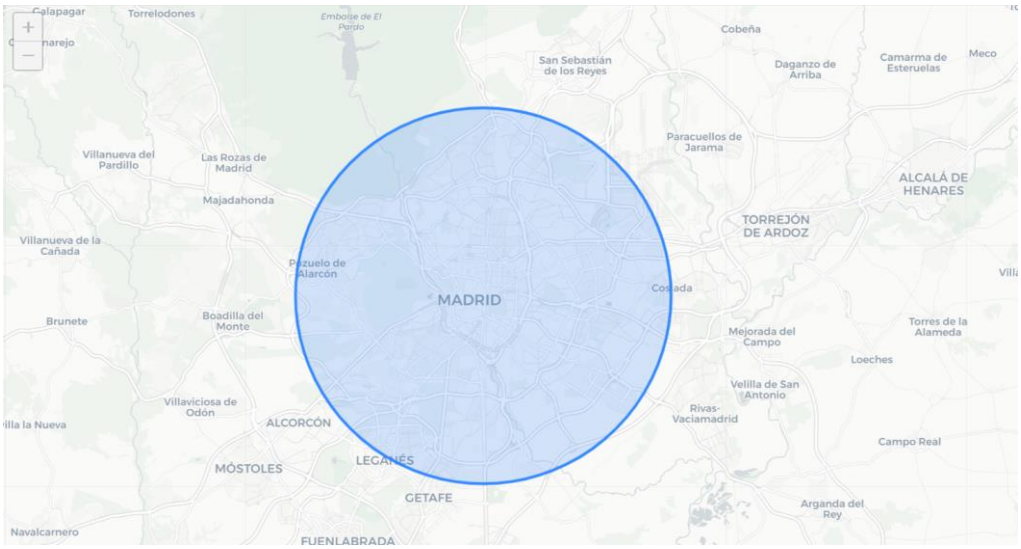


- **Histograma de tabla de vehículos:** Se muestra la autonomia de los diferentes vehiculos.

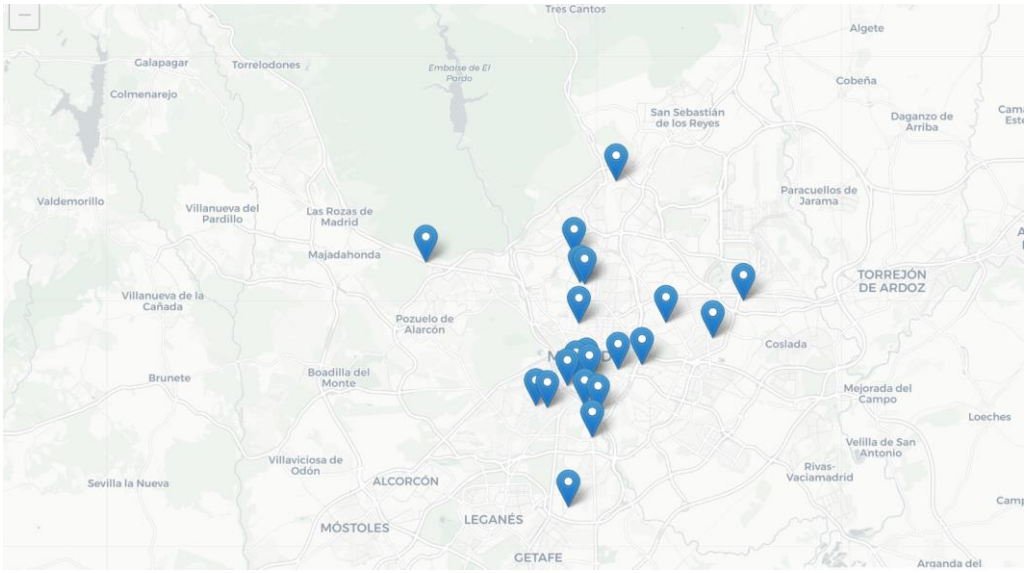


- **Mapa de ubicación de clientes:** Se muestran los distintas ubicaciones de los clientes en un mapa, primero se ve el radio de donde se encuentran las ubicaciones y por último los puntos exactos de las locaclizaciones.

Radio de ubicaciones



Radio de locaclizaciones



4. Data Preparation

4.1 Clean Data and Construct Data

La preparación de datos consiste en la limpieza y manipulación de los datos en crudo para mejorar el procesamiento y el análisis de dichos datos.

Para el caso número 3 del proyecto (predicción de demanda de clientes de enero de 2025) se ha tenido que realizar un análisis exploratorio del archivo **df_historic_order_demand** para comprobar que los datos estaban correctamente.

Se ha encontrado que la columna order_demand tenía valores nulos por lo que se ha procedido a su eliminación.

4.2 Format Data

Para el caso número 3 del proyecto (predicción de demanda de clientes de enero de 2025) se aplica un algoritmo de regresión (random forest regressor) para poder predecir la demanda y dicho algoritmo no acepta valores categóricos, por lo que las columnas mes_anio ha tenido que ser transformada a numérica.

Para el caso de la columna mes_anio, cuando se hizo el análisis exploratorio se vio que dicha columna estaba compuesta por mes y año en este formato “MM-YYYY” y todos los valores de dicha columna eran únicos por lo que aplicar get_dummies no era la mejor solución.

Se optó por separar mes y año en dos columnas distintas y así poder convertirlas a numéricas.

5. Modeling

A. Clean Data and Construct Data

La exploración de los modelos se lleva a cabo con el indicado RMSE (Error medio), en el caso de los algoritmos Djikstra, PuLP y genético no se mide el RMSE ya que no es un algoritmo de predicción sino de optimización por lo que el resultado será la ruta más rápida entre una serie de puntos dados.

B. Build Model

1) Algoritmo Dijkstra

Para el modelo 1, se va a aplicar el algoritmo de optimización Djikstra que es un método utilizado para encontrar el camino más corto en un grafo ponderado (las aristas tienen un peso, que puede representar costo, distancia o cualquier métrica relevante) con pesos no negativos.

Funciona de la siguiente manera:

- El algoritmo de Dijkstra básicamente inicia en el nodo que escojas (el nodo de origen) y analiza el grafo para encontrar el camino más corto entre ese nodo y todos los otros nodos en el grafo.
- El algoritmo mantiene un registro de la distancia conocida más corta desde el nodo de origen hasta cada nodo y actualiza el valor si encuentra un camino más corto.
- Una vez que el algoritmo ha encontrado el camino más corto entre el nodo de origen y otro nodo, ese nodo se marca como "visitado" y se agrega al camino.
- El proceso continúa hasta que todos los nodos en el grafo han sido añadidos al camino. De esta forma, tenemos un camino que conecta al nodo de origen con todos los otros nodos siguiendo el camino más corto posible para llegar a cada uno de ellos.

2) Algoritmo de Programación Lineal

Para el modelo 2, se va a aplicar la librería PuLP de Python, que encuentra soluciones óptimas al problema como restricciones lineales.

PuLP permite indicar el tipo de problema que hay que optimizar mediante palabras reservadas de la propia librería, maximización (LpMaximize) o minimización (LPMinimize), que deberán usarse cuando comenzamos a definirlo. Acto seguido se definirá la función objetivo y sus restricciones para que se pueda iniciar la búsqueda de posibles soluciones.

3) Algoritmo de Genético

Para el modelo 3 se va a aplicar el algoritmo genético que se empleará para encontrar rutas para el modelo. Este algoritmo se basa en la teoría evolutiva de selección natural propuesta por Darwin en la que los individuos de una población se reproducen generando descendencia, cuyas características, son combinación de las características de los progenitores (más alguna mutación). De todos ellos, únicamente sobreviven los mejores individuos y pueden volver a reproducirse y transmitir sus características a la siguiente generación.

Explicada la teoría, la estructura de dicho algoritmo es la siguiente:

1. Crear una población inicial aleatoria de P individuos. En este caso, cada individuo representa una combinación de valores de las variables.

2. Calcular la fortaleza (fitness) de cada individuo de la población. El fitness está relacionado con el valor de la función para cada individuo. Si se quiere maximizar, cuanto mayor sea el valor de la función para el individuo, mayor su fitness. En el caso de minimización, ocurre lo contrario.

3. Crear una nueva población vacía y repetir los siguientes pasos hasta que se hayan creado P nuevos individuos.

3.1 Seleccionar dos individuos de la población existente, donde la probabilidad de selección es proporcional al fitness de los individuos.

3.2 Cruzar los dos individuos seleccionados para generar un nuevo descendiente (crossover).

3.3 Aplicar un proceso de mutación aleatorio sobre el nuevo individuo.

3.4 Añadir el nuevo individuo a la nueva población.

4. Reemplazar la antigua población por la nueva.

5. Si no se cumple un criterio de parada, volver al paso 2.

C. Access Model

1) Algorimo Djikstra

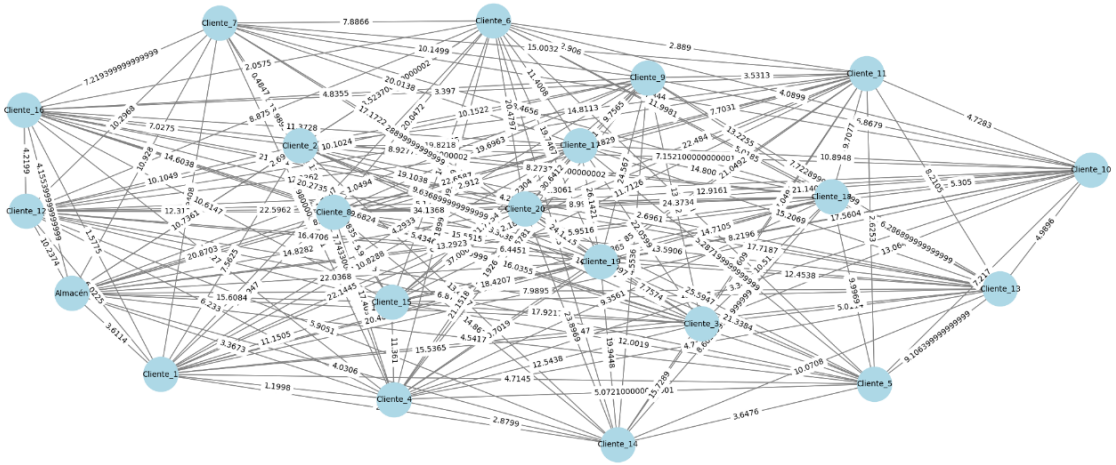
Partimos de los datos de distancias entre los distintos clientes (medida en kilómetros) en forma de matriz.

	Cliente_1	Cliente_2	Cliente_3	Cliente_4	Cliente_5	Cliente_6	Cliente_7	Cliente_8	Cliente_9	Cliente_10	...	Cliente_12
0	0.0000	7.5625	15.5365	1.1998	4.7145	1.7407	7.9408	17.1947	4.2933	3.2659	...	6.0225
1	7.5625	0.0000	3.3838	7.7433	14.5720	8.5237	0.4847	13.7974	10.1522	7.1521	...	10.1049
2	15.5365	3.3838	0.0000	12.5438	0.0000	0.0000	0.0000	16.0355	13.9120	13.0649	...	12.3430
3	1.1998	7.7433	12.5438	0.0000	5.0721	0.9119	7.5798	17.4095	3.5781	3.3451	...	6.2330
4	4.7145	14.5720	0.0000	5.0721	0.0000	4.8187	0.0000	0.0000	0.0000	7.2170	...	6.8738
5	1.7407	8.5237	0.0000	0.9119	4.8187	0.0000	7.8866	20.0472	2.9060	4.0899	...	8.8750
6	7.9408	0.4847	0.0000	7.5798	0.0000	7.8866	0.0000	13.9893	10.1499	7.3440	...	10.2968
7	17.1947	13.7974	16.0355	17.4095	0.0000	20.0472	13.9893	0.0000	19.6963	12.9161	...	12.3127
8	4.2933	10.1522	13.9120	3.5781	0.0000	2.9060	10.1499	19.6963	0.0000	6.8679	...	10.1024
9	3.2659	7.1521	13.0649	3.3451	7.2170	4.0899	7.3440	12.9161	6.8679	0.0000	...	2.3061
10	2.1866	14.8113	17.0494	2.8532	2.6253	2.8890	15.0032	16.1829	3.5313	4.7283	...	6.5845
11	6.0225	10.1049	12.3430	6.2330	6.8738	8.8750	10.2968	12.3127	10.1024	2.3061	...	0.0000
12	5.4470	2.6961	5.0114	4.7117	9.1064	5.0185	0.0000	13.5906	7.7229	4.9896	...	6.4451
13	2.2133	13.4907	15.7289	2.8799	3.6476	3.2185	0.0000	14.8623	4.5536	0.0000	...	5.9051

En la imagen se puede observar que la diagonal principal son todos ceros ya que la distancia entre Cliente_1 y el mismo es cero.

Por otro lado, se aprecia que hay más ceros aparte de los de la diagonal ya que es posible que no haya una ruta directa entre dos clientes, es decir, que no hay camino.

Para hacernos una idea de cómo están relacionados todos los clientes, creamos un diagrama que muestra la conexión entre todos los nodos y sus correspondientes distancias como se muestra en la siguiente imagen.



- Nuestro primer caso de uso consiste en calcular la ruta más corta (km) entre el almacén y los distintos clientes. Estos han sido los resultados:

```
Caminos más cortos desde el nodo de inicio:
Camino más corto a Almacén: ['Almacén']
Camino más corto a Cliente_1: ['Almacén', 'Cliente_1']
Camino más corto a Cliente_2: ['Almacén', 'Cliente_6', 'Cliente_13', 'Cliente_2']
Camino más corto a Cliente_3: ['Almacén', 'Cliente_6', 'Cliente_13', 'Cliente_3']
Camino más corto a Cliente_4: ['Almacén', 'Cliente_4']
Camino más corto a Cliente_5: ['Almacén', 'Cliente_5']
Camino más corto a Cliente_6: ['Almacén', 'Cliente_6']
Camino más corto a Cliente_7: ['Almacén', 'Cliente_6', 'Cliente_7']
Camino más corto a Cliente_8: ['Almacén', 'Cliente_16', 'Cliente_8']
Camino más corto a Cliente_9: ['Almacén', 'Cliente_9']
Camino más corto a Cliente_10: ['Almacén', 'Cliente_10']
Camino más corto a Cliente_11: ['Almacén', 'Cliente_11']
Camino más corto a Cliente_12: ['Almacén', 'Cliente_10', 'Cliente_12']
Camino más corto a Cliente_13: ['Almacén', 'Cliente_6', 'Cliente_13']
Camino más corto a Cliente_14: ['Almacén', 'Cliente_14']
Camino más corto a Cliente_15: ['Almacén', 'Cliente_10', 'Cliente_12', 'Cliente_15']
Camino más corto a Cliente_16: ['Almacén', 'Cliente_16']
Camino más corto a Cliente_17: ['Almacén', 'Cliente_17']
Camino más corto a Cliente_18: ['Almacén', 'Cliente_10', 'Cliente_18']
Camino más corto a Cliente_19: ['Almacén', 'Cliente_6', 'Cliente_13', 'Cliente_2', 'Cliente_19']
Camino más corto a Cliente_20: ['Almacén', 'Cliente_20']
```

Se observa que hay casos en los que la ruta más corta es directamente ruta que hay entre ellos mismos y en otros casos pasa por otros clientes hasta llegar al nodo final.

Por ejemplo, en el camino más corto para llegar al Cliente_2 la distancia entre dicho cliente y el almacén es de **10.7361 km,** pero Djikstra ha encontrado una ruta más rápida, pero pasando antes por el Cliente_6 y el Cliente_13 y la distancia total recorrida por este camino es de **10.4098 km,** que es ligeramente menor que la primera distancia.

En la siguiente imagen se muestra una lista de distancias entre ambos puntos:

```
Distancias desde el nodo de inicio:
Distancia a Almacén: 0
Distancia a Cliente_9: 1.0494
Distancia a Cliente_6: 2.6952
Distancia a Cliente_11: 2.912
Distancia a Cliente_4: 3.3673
Distancia a Cliente_1: 3.6114
Distancia a Cliente_14: 4.0306
Distancia a Cliente_16: 4.155399999999999
Distancia a Cliente_5: 4.5417
Distancia a Cliente_10: 5.9516
Distancia a Cliente_13: 7.7137
Distancia a Cliente_12: 8.2577
Distancia a Cliente_17: 8.5221
Distancia a Cliente_2: 10.4098
Distancia a Cliente_7: 10.5818
Distancia a Cliente_18: 11.256599999999999
Distancia a Cliente_3: 12.725100000000001
Distancia a Cliente_15: 13.8995
Distancia a Cliente_20: 14.8282
Distancia a Cliente_8: 18.7592
Distancia a Cliente_19: 20.0467
```

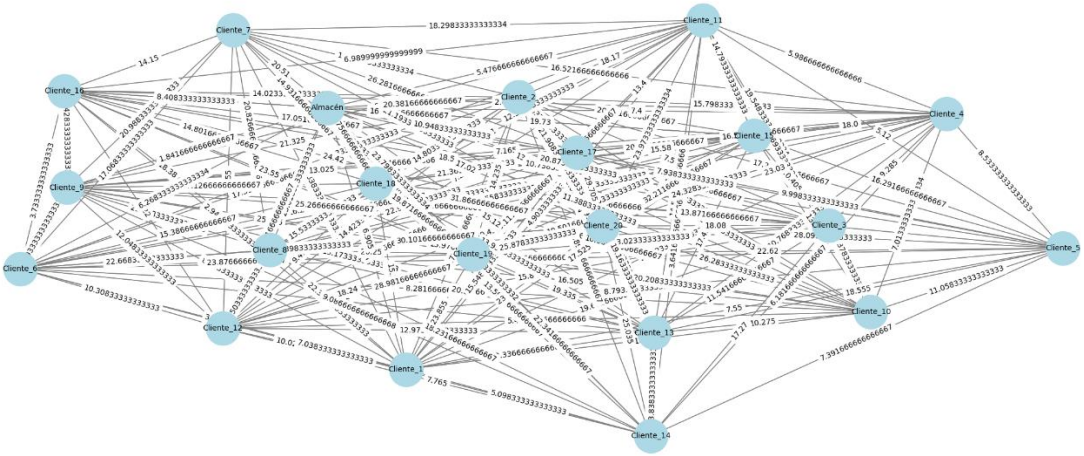
- Nuestro segundo caso de uso consiste en calcular la ruta más rápida en término de tiempo entre el almacén y los distintos clientes. Partimos de estos datos:

	Cliente_1	Cliente_2	Cliente_3	Cliente_4	Cliente_5	Cliente_6	Cliente_7	Cliente_8	Cliente_9	Cliente_10	...	Cliente_12
0	0.000000	15.731667	19.651667	2.543333	7.550000	3.760000	15.838333	22.388333	8.563333	7.185000	...	10.028333
1	15.731667	0.000000	7.505000	15.798333	17.241667	17.316667	1.248333	20.700000	21.325000	13.503333	...	14.423333
2	19.651667	7.505000	0.000000	19.285000	0.000000	0.000000	0.000000	22.076667	19.981667	17.378333	...	15.800000
3	2.543333	15.798333	19.285000	0.000000	8.533333	1.878333	16.521667	22.863333	7.165000	7.013333	...	10.501667
4	7.550000	17.241667	0.000000	8.533333	0.000000	9.711667	0.000000	0.000000	0.000000	11.058333	...	8.793333
5	3.760000	17.316667	0.000000	1.878333	9.711667	0.000000	17.068333	22.668333	6.033333	8.281667	...	10.308333
6	15.838333	1.248333	0.000000	16.521667	0.000000	17.068333	0.000000	20.826667	20.988333	13.631667	...	14.550000
7	22.388333	20.700000	22.076667	22.863333	0.000000	22.668333	20.826667	0.000000	25.133333	16.505000	...	15.503333
8	8.563333	21.325000	19.981667	7.165000	0.000000	6.033333	20.988333	25.133333	0.000000	13.976667	...	12.048333
9	7.185000	13.503333	17.378333	7.013333	11.058333	8.281667	13.631667	16.505000	13.976667	0.000000	...	5.411667
10	4.903333	18.170000	19.548333	5.986667	5.120000	6.233333	18.298333	20.510000	6.815000	9.693333	...	7.295000
11	10.028333	14.423333	15.800000	10.501667	8.793333	10.308333	14.550000	15.503333	12.048333	5.411667	...	0.000000
12	12.336667	6.695000	11.541667	10.768333	18.555000	11.315000	0.000000	20.850000	15.173333	10.275000	...	12.973333
13	5.098333	15.891667	17.270000	6.181667	7.391667	7.038333	0.000000	18.231667	9.086667	0.000000	...	7.765000
14	17.525000	20.061667	20.405000	18.000000	16.291667	17.806667	0.000000	10.558333	19.546667	13.880000	...	10.435000
15	2.943333	14.023333	18.500000	2.465000	10.738333	3.733333	14.150000	18.380000	9.428333	0.000000	...	8.220000
16	15.548333	26.153333	27.531667	16.546667	9.998333	16.240000	26.281667	22.963333	13.025000	18.623333	...	18.026667
17	15.105000	14.803333	16.181667	15.580000	13.871667	15.386667	14.931667	15.533333	17.126667	10.886667	...	9.490000

En la imagen se puede observar que la diagonal principal son todos ceros ya que la distancia en tiempo entre Cliente_1 y el mismo es cero.

Por otro lado, se aprecia que hay más ceros aparte de los de la diagonal ya que es posible que no haya una ruta directa entre dos clientes por lo que el tiempo es 0.

Para hacernos una idea de cómo están relacionados todos los clientes, creamos un diagrama que muestra la conexión entre todos los nodos y sus correspondientes distancias en minutos como se muestra en la siguiente imagen.



Aplicado el algoritmo se han obtenido los siguientes resultados:

```
Caminos más cortos desde el nodo de inicio:
Camino más corto a Almacén: ['Almacén']
Camino más corto a Cliente_1: ['Almacén', 'Cliente_1']
Camino más corto a Cliente_2: ['Almacén', 'Cliente_2']
Camino más corto a Cliente_3: ['Almacén', 'Cliente_3']
Camino más corto a Cliente_4: ['Almacén', 'Cliente_4']
Camino más corto a Cliente_5: ['Almacén', 'Cliente_5']
Camino más corto a Cliente_6: ['Almacén', 'Cliente_6']
Camino más corto a Cliente_7: ['Almacén', 'Cliente_7']
Camino más corto a Cliente_8: ['Almacén', 'Cliente_8']
Camino más corto a Cliente_9: ['Almacén', 'Cliente_9']
Camino más corto a Cliente_10: ['Almacén', 'Cliente_10']
Camino más corto a Cliente_11: ['Almacén', 'Cliente_11']
Camino más corto a Cliente_12: ['Almacén', 'Cliente_11', 'Cliente_12']
Camino más corto a Cliente_13: ['Almacén', 'Cliente_13']
Camino más corto a Cliente_14: ['Almacén', 'Cliente_14']
Camino más corto a Cliente_15: ['Almacén', 'Cliente_15']
Camino más corto a Cliente_16: ['Almacén', 'Cliente_16']
Camino más corto a Cliente_17: ['Almacén', 'Cliente_17']
Camino más corto a Cliente_18: ['Almacén', 'Cliente_11', 'Cliente_18']
Camino más corto a Cliente_19: ['Almacén', 'Cliente_19']
Camino más corto a Cliente_20: ['Almacén', 'Cliente_20']
```

Al contrario que en el primer caso de uso, nos encontramos con que en general la ruta más rápida en término de tiempo es la ruta entre ambos puntos sin intermediarios.

Se observa que los resultados no son los mismo que en el anterior caso de uso ya que una distancia en kilómetros más corta no significa que el tiempo sea menor ya que puede haber obstáculos como semáforos o muchas curvas etc.

2) Algoritmo de Programación Lineal

Partimos de los siguientes datos de prueba:

```
# Matriz de distancias y datos iniciales
matriz_distancias = np.array([
    [0, 7.5625, 15.5365, 1.1998, 4.7145, 3.6614],  # Cliente 0 a todos
    [7.5625, 0, 3.3838, 7.7433, 14.572, 10.7361],  # Cliente 1 a todos
    [15.5365, 3.3838, 0, 12.5438, 0, 13.9021],     # Cliente 2 a todos
    [1.1998, 7.7433, 12.5438, 0, 5.0721, 3.3673],  # Cliente 3 a todos
    [4.7145, 14.572, 0, 5.0721, 0, 4.5417],        # Cliente 4 a todos
    [3.6614, 10.7361, 13.9021, 3.3673, 4.5417, 0], # Almacén a todos
])

vehiculos = [
    {"id": 1, "capacidad": 2026, "costo_km": 0.2, "autonomia": 603},
    {"id": 2, "capacidad": 4362, "costo_km": 0.14, "autonomia": 630},
    {"id": 3, "capacidad": 4881, "costo_km": 0.2, "autonomia": 664},
    {"id": 4, "capacidad": 3321, "costo_km": 0.19, "autonomia": 514},
    {"id": 5, "capacidad": 10000, "costo_km": 0.32, "autonomia": 350},
    {"id": 6, "capacidad": 3129, "costo_km": 0.14, "autonomia": 791},
]

demandas = [909, 959, 960, 980, 979]
```

- La matriz de distancias representa la distancia en kilómetros entre clientes y el almacén. Los registros que son ceros significan que no hay ruta directa entre dichos puntos
- Los vehículos contienen información como su capacidad máxima en kilogramos, su autonomía en kilómetros y el costo por kilómetro.
- Las demandas contienen la demanda de pedido de cada cliente en kilogramos.

Nuestro caso de uso consiste en calcular las posibles rutas de cada vehículo que pasen por el máximo número de clientes y con el menor costo posible.

Para ello definimos nuestra función objetivo que en este caso es la de minimizar costes y creamos nuestras variables decisión que serán si un vehículo viaja entre dos clientes, como se muestra en la imagen:

```python
# Parámetros iniciales
num_clientes = len(demandas) # Número de clientes
num_vehiculos = len(vehiculos) # Número de vehículos
almacen_idx = num_clientes    #Indice del almacén en la matriz

# Crear el modelo de optimización
modelo = lp.LpProblem("Minimizar_Costo_Transporte", lp.LpMinimize)

# Variables de decisión: si un vehículo viaja entre i y j
x = lp.LpVariable.dicts(
    "ruta",
    [
        (i, j, v["id"])
        for i in range(num_clientes + 1)
        for j in range(num_clientes + 1)
        if i != j and matriz_distancias[i, j] > 0  # Excluir rutas con distancia 0
        for v in vehiculos
    ],
    cat=lp.LpBinary,
)

# Función objetivo: Minimizar el costo total
modelo += lp.lpSum(
    matriz_distancias[i, j] * x[(i, j, v["id"])] * v["costo_km"]
    for v in vehiculos
    for i in range(num_clientes + 1)
    for j in range(num_clientes + 1)
    if i != j and matriz_distancias[i, j] > 0
)
```

Acto seguido procedemos a crear las restricciones del modelo que serán las relativas a capacidad máxima y autonomía de cada vehículo y que cada cliente debe ser visitado como mínimo una vez.

```python
# Restricción: cada cliente debe ser visitado una vez
for j in range(1, num_clientes + 1):
    modelo += lp.lpSum(x[(i, j, v["id"])] for v in vehiculos for i in range(num_clientes + 1) if i != j and matriz_distancias[i, j] > 0) >= 1

# Restricción: capacidad de los vehículos
for v in vehiculos:
    modelo += lp.lpSum(
        demandas[j - 1] * lp.lpSum(x[(i, j, v["id"])] for i in range(num_clientes + 1) if i != j and matriz_distancias[i, j] > 0)
        for j in range(1, num_clientes + 1)
    ) <= v["capacidad"]

# Restricción: autonomía de los vehículos
for v in vehiculos:
    modelo += lp.lpSum(
        matriz_distancias[i, j] * x[(i, j, v["id"])]
        for i in range(num_clientes + 1)
        for j in range(num_clientes + 1)
        if i != j and matriz_distancias[i, j] > 0
    ) <= v["autonomia"]
```

Ahora que tenemos nuestro objetivo y restricciones definidos pasaremos a resolver nuestro problema y estas han sido las rutas generadas:

```
Estado del modelo: Optimal
Costo total: 2.222696
Ruta vehículo 1: {5}, distancia recorrida = 0.00 km
Ruta vehículo 2: {0, 3, 1, 2, 2, 1, 3, 5, 5}, distancia recorrida = 11.33 km
Ruta vehículo 3: {5}, distancia recorrida = 0.00 km
Ruta vehículo 4: {5}, distancia recorrida = 0.00 km
Ruta vehículo 5: {5}, distancia recorrida = 0.00 km
Ruta vehículo 6: {5, 4, 5}, distancia recorrida = 4.54 km
```

Como se puede observar, el estado del modelo es óptimo por lo que nuestras restricciones y objetivo han funcionado, pero el resultado no es el esperado ya que en una misma ruta se repiten los clientes y seguidos.

Hay cuatro vehículos que no tienen rutas asignadas y el vehículo dos es el que más clientes recorre. También se observa que el vehículo dos no pasa por el cliente 4 (que en realidad es el 5) y el vehículo 6 solo pasa por dicho cliente.

3) Algoritmo Genético

Partimos de los mismos datos de prueba del anterior algoritmo, para la realización del algoritmo genético.

- Nuestro caso de uso es un método de optimización y búsqueda de las mejores rutas, que utiliza procesos como la selección, cruzamiento y mutación para encontrar las rutas óptimas de los vehículos para que repartan a más clientes con el menor coste.

```python
# Matriz de distancias (datos iniciales reales)
matriz_distancias = np.array([
    [0, 7.5625, 15.5365, 1.1998, 4.7145, 3.6614],
    [7.5625, 0, 3.3838, 7.7433, 14.572, 10.7361],
    [15.5365, 3.3838, 0, 12.5438, 0, 13.9021],
    [1.1998, 7.7433, 12.5438, 0, 5.0721, 3.3673],
    [4.7145, 14.572, 0, 5.0721, 0, 4.5417],
    [3.6614, 10.7361, 13.9021, 3.3673, 4.5417, 0]
])

# Vehículos disponibles
vehiculos = [
    {"id": 1, "capacidad": 2026, "costo_km": 0.2, "autonomia": 603},
    {"id": 2, "capacidad": 4362, "costo_km": 0.14, "autonomia": 630},
    {"id": 3, "capacidad": 4881, "costo_km": 0.2, "autonomia": 664},
    {"id": 4, "capacidad": 3321, "costo_km": 0.19, "autonomia": 514},
    {"id": 5, "capacidad": 10000, "costo_km": 0.32, "autonomia": 350},
    {"id": 6, "capacidad": 3129, "costo_km": 0.14, "autonomia": 791}
]
```

Definimos una función de aptitud que minimiza la distancia total recorrida por los vehículos, asignando clientes a rutas según la capacidad de cada uno. Las soluciones se generan aleatoriamente en clientes y las variables de decisión son las rutas asignadas a los vehículos.

```python
def fitness_function(routes):
    total_distance = 0
    for route in routes:
        if len(route) == 0:
            continue
        distance = 0
        prev_location = 0  # Comienza en el almacén
        for client in route:
            distance += matriz_distancias[prev_location][client]
            prev_location = client
        distance += matriz_distancias[prev_location][0]  # Regresa al almacén
        total_distance += distance
    return total_distance

def generate_initial_population():
    population = []
    clients = list(range(1, num_clients + 1))
    for _ in range(POPULATION_SIZE):
        random.shuffle(clients)
        population.append(split_routes(clients))
    return population

def split_routes(clients):
    routes = []
    current_route = []
    current_capacity = 0
    for client in clients:
        demand = demands[client]
        if current_capacity + demand <= vehicle_capacities[len(routes) % num_vehicles]:
            current_route.append(client)
            current_capacity += demand
        else:
            routes.append(current_route)
            current_route = [client]
            current_capacity = demand
    if current_route:
        routes.append(current_route)
    return routes
```

-Aquí tenemos la solución de nuestro algoritmo y las rutas y la distancia generada:

```
Vehículo 1: Ruta -> Almacén -> Cliente 5 -> Cliente 3 -> Almacén
Vehículo 2: Ruta -> Almacén -> Cliente 1 -> Cliente 4 -> Cliente 2 -> Almacén

Distancia total recorrida: 45.90 km
```

4) Algoritmo heurístico

Mediante el uso de la biblioteca ORTools se utilizará como solución de ruteo con restricciones.

Primero realizamos la asignación de los valores a utilizar en el código, teniendo en cuenta una ligera modificación sobre los datos de entrada, de manera que quedará lo siguiente:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	Almacén	Cliente_1	Cliente_2	Cliente_3	Cliente_4	Cliente_5	Cliente_6	Cliente_7	Cliente_8	Cliente_9	Cliente_10	Cliente_11	Cliente_12	Cliente_13	Cliente_14	Cliente_15	Cliente_16	Cliente_17	Cliente_18	Cliente_19	Cliente_20
2	0	3.6114	10.7361	13.9021	3.3673	4.5417	2.6952	10.928	20.8703	1.0494	5.9516	2.912	10.2374	7.9895	4.0306	15.6084	4.1554	8.5221	13.2923	22.0368	14.8282
3	3.6114	0	7.5625	15.5365	1.1998	4.7145	1.7407	7.9408	17.1947	4.2933	3.2659	2.1866	6.0225	5.447	2.2133	11.1505	1.5775	10.8288	9.1456	20.4871	22.1445
4	10.7361	7.5625	0	3.3838	7.7433	14.572	8.5237	0.4847	13.7974	10.1522	7.1521	14.8113	10.1049	2.6961	13.4907	18.0835	7.0275	19.8218	8.2737	9.6369	19.1038
5	13.9021	15.5365	3.3838	0	12.5438	0	0	0	16.0355	13.912	13.0649	17.0494	12.343	5.0114	15.7289	17.9217	9.6824	22.0599	10.5118	7.7574	16.5997
6	3.3673	1.1998	7.7433	12.5438	0	5.0721	0.9119	7.5798	17.4095	3.5781	3.3451	2.8532	6.233	4.7117	2.8799	11.361	1.3127	11.1926	9.3561	20.7019	21.1518
7	4.5417	4.7145	14.572	0	5.0721	0	4.8187	0	0	0	7.217	2.6253	6.8738	9.1064	3.6476	12.0019	5.4347	5.2872	9.9969	21.3384	25.5947
8	2.6952	1.7407	8.5237	0	0.9119	4.8187	0	7.8866	20.0472	2.906	4.0899	2.889	8.875	5.0185	3.2185	14.003	2.0575	11.4008	11.9981	19.7467	20.4797
9	10.928	7.9408	0.4847	0	7.5798	0	7.8866	0	13.9893	10.1499	7.344	15.0032	10.2968	0	0	0	7.2194	20.0138	8.4656	9.8289	17.1722
10	20.8703	17.1947	13.7974	16.0355	17.4095	0	20.0472	13.9893	0	19.6963	12.9161	16.1829	12.3127	13.5906	14.8623	5.9742	14.6038	22.6587	8.9984	15.5515	34.1368
11	1.0494	4.2933	10.1522	13.912	3.5781	0	2.906	10.1499	19.6963	0	6.8679	3.5313	10.1024	7.7229	4.5536	15.2304	4.8355	9.7565	13.2255	24.567	22.9781
12	5.9516	3.2659	7.1521	13.0649	3.3451	7.217	4.0899	7.344	12.9161	6.8679	0	4.7283	2.3061	4.9896	0	8.2196	0	10.8948	5.305	17.5604	21.1407
13	2.912	2.1866	14.8113	17.0494	2.8532	2.6253	2.889	15.0032	16.1829	3.5313	4.7283	0	6.5845	8.2105	1.6099	11.7126	3.397	7.7031	9.7077	21.0492	22.484
14	10.2374	6.0225	10.1049	12.343	6.233	6.8738	8.875	10.2968	12.3127	10.1024	2.3061	6.5845	0	6.4451	5.9051	5.6418	4.2199	12.2362	5.1248	16.4706	22.5962
15	7.9895	5.447	2.6961	5.0114	4.7117	9.1064	5.0185	0	13.5906	7.7229	4.9896	8.2105	6.4451	0	10.0708	10.0675	4.2809	15.2069	6.2869	12.4538	17.7187
16	4.0306	2.2133	13.4907	15.7289	2.8799	3.6476	3.2185	0	14.8623	4.5536	0	1.6099	5.9051	10.0708	0	0	0	0	8.6033	19.9448	23.8969
17	15.6084	11.1505	18.0835	17.9217	11.361	12.0019	14.003	0	5.9742	15.2304	8.2196	11.7126	5.6418	10.0675	0	0	10.6147	21.7754	5.1865	18.4207	37.006
18	4.1554	1.5775	7.0275	9.6824	1.3127	5.4347	2.0575	7.2194	14.6038	4.8355	0	3.397	4.2199	4.2809	0	10.6147	0	11.3728	8.9277	20.2735	21.0351
19	8.5221	10.8288	19.8218	22.0599	11.1926	5.2872	11.4008	20.0138	22.6587	9.7565	10.8948	7.7031	12.2362	15.2069	0	21.7754	11.3728	0	14.8006	26.1421	30.6412
20	13.2923	9.1456	8.2737	10.5118	9.3561	9.9969	11.9981	8.4656	8.9984	13.2255	5.305	9.7077	5.1248	6.2869	8.6033	5.1865	8.9277	14.8006	0	14.7105	24.3734
21	22.0368	20.4871	9.6369	7.7574	20.7019	21.3384	19.7467	9.8289	15.5515	24.567	17.5604	21.0492	16.4706	12.4538	19.9448	18.4207	20.2735	26.1421	14.7105	0	24.1215
22	14.8282	22.1445	19.1038	16.5997	21.1518	25.5947	20.4797	17.1722	34.1368	22.9781	21.1407	22.484	22.5962	17.7187	23.8969	37.006	21.0351	30.6412	24.3734	24.1215	0

Partiendo del archivo modificado y los Excel de df_orders y df_vehicle, generamos la definición de las variables a utilizar en la función create_data_model(), siendo data la variable diccionario de retorno que contiene los datos a utilizar.

```python
import pandas as pd
from ortools.constraint_solver import pywrapcp, routing_enums_pb2
import os

path = "C:/Users/galla/Dropbox/PC/Documents/1 Pedro G Gallardo/Master IA y Big Data/Clases/octubre-24/2024-10-01/jupyter/Proyecto1/Datos_P1/"
os.chdir(path)

def create_data_model():
    data = {}

    # Cargar los datos desde el archivo Excel
    df = pd.read_excel('df_distance_km.xlsx', index_col=0)

    # Convertir los datos del DataFrame a una lista de listas para la matriz de distancias
    data['distance_matrix'] = df.values.tolist()

    # Debug: Imprimir la matriz de distancias
    print("Matriz de distancias:")
    print(data['distance_matrix'])

    # Actualizar data['demands'] con la columna 'order_demand' del archivo Excel
    df_demands = pd.read_excel('df_orders.xlsx')
    data['demands'] = df_demands['order_demand'].tolist()

    # Debug: Imprimir la lista de demandas
    print("Demandas:")
    print(data['demands'])

    # Actualizar data['vehicle_capacities'] con la columna 'vehicle_capacity' del archivo Excel
    df_vehicles = pd.read_excel('df_vehicle.xlsx')
    data['vehicle_capacities'] = df_vehicles['capacidad_kg'].tolist()  # Capacidades de los vehículos
    data['vehicle_autonomy'] = df_vehicles['autonomia_km'].tolist()  # Autonomías de los vehículos
    data['num_vehicles'] = len(df_vehicles['vehiculo_id'])  # Número de vehículos
    data['depot'] = 20  # Índice del almacén

    # Debug: Imprimir capacidades y autonomías de los vehículos, y número de vehículos
    print("Capacidades de los vehículos:")
    print(data['vehicle_capacities'])
    print("Autonomías de los vehículos:")
    print(data['vehicle_autonomy'])
    print("Número de vehículos:")
    print(data['num_vehicles'])

    return data
```

A continuación se llama a la función main() que ejecutará la totalidad del código, estructurado de la siguiente manera:

1. Almacena la variable retorno de la función previamente comentada
2. Mapea nodos reales a índices internos para OR-Tools y representa las rutas posibles y restricciones
3. La función distance_callback devuelve la distancia entre nodos usando data['distance_matrix']. Si no existe una conexión directa, asigna una penalización alta
4. La función demand_callback garantiza que los vehículos no excedan su capacidad
5. Añadir dimensión de autonomía para verificar que las rutas respeten la autonomía de los vehículos.

```python
def main():
    data = create_data_model()
    manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                           data['num_vehicles'], data['depot'])
    routing = pywrapcp.RoutingModel(manager)

    def distance_callback(from_index, to_index):
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        distance = data['distance_matrix'][from_node][to_node]
        if distance == 0 and from_node != to_node:
            return 999999  # Penalización alta para caminos inexistentes
        return distance

    transit_callback_index = routing.RegisterTransitCallback(distance_callback)
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    def demand_callback(from_index):
        from_node = manager.IndexToNode(from_index)
        return data['demands'][from_node]

    demand_callback_index = routing.RegisterUnaryTransitCallback(demand_callback)
    routing.AddDimensionWithVehicleCapacity(demand_callback_index, 0, data['vehicle_capacities'], True, 'Capacity')

    # Añadir dimensión de autonomía
    routing.AddDimension(transit_callback_index, 0,  # No slack
        max(data['vehicle_autonomy']),  # Autonomía máxima de los vehículos
        True,  # Start cumulative to zero
        'Autonomy')

    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
    search_parameters.local_search_metaheuristic = (routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH)
    search_parameters.time_limit.seconds = 30  # Tiempo límite de búsqueda

    assignment = routing.SolveWithParameters(search_parameters)

    if assignment:
        print_solution(data, manager, routing, assignment)
```

Finalmente la función print_solution() muestra el resultado de las rutas generadas para que resulten interpretables y claras al usuario

```python
def print_solution(data, manager, routing, assignment):
    total_distance = 0
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        plan_output = 'Route for vehicle {}:\n'.format(vehicle_id)
        route_distance = 0
        while not routing.IsEnd(index):
            plan_output += ' {} ->'.format(manager.IndexToNode(index))
            previous_index = index
            index = assignment.Value(routing.NextVar(index))
            route_distance += routing.GetArcCostForVehicle(previous_index, index, vehicle_id)
            print(routing.GetArcCostForVehicle(previous_index, index, vehicle_id), previous_index, index, vehicle_id)
        plan_output += ' {}\n'.format(manager.IndexToNode(index))
        plan_output += 'Distance of the route: {}km\n'.format(route_distance)
        print(plan_output)
        total_distance += route_distance
    print('Total distance of all routes: {}km'.format(total_distance))
```

Para la ejecución del código se hace una llamada a la función main

```python
if __name__ == '__main__':
    main()
```

Actualmente aún existen errores en la solución generada que deben ser retocados y por tanto las rutas obtenidas no son óptimas ni cumplen todos los requisitos.

Ejemplo de resultado:

```
Route for vehicle 0:
 20 -> 20
Distance of the route: 0km
```

```
Route for vehicle 1:
 20 -> 20
Distance of the route: 0km
```

```
Route for vehicle 2:
 20 -> 3 -> 2 -> 1 -> 0 -> 20
Distance of the route: 0km
```

```
Route for vehicle 3:
 20 -> 6 -> 5 -> 4 -> 20
Distance of the route: 0km
```

```
Route for vehicle 4:
 20 -> 16 -> 15 -> 14 -> 13 -> 12 -> 11 -> 10 -> 9 -> 8 -> 7 -> 20
Distance of the route: 0km
```

```
Route for vehicle 5:
 20 -> 19 -> 18 -> 17 -> 20
Distance of the route: 0km
```

6. Casos de Uso y Escenarios

Después de haber aplicado a todos los algoritmos realizados una serie de pruebas para comprobar su eficacia y explorar los distintos resultados obtenidos, se ha optado por escoger el algoritmo genético ya que en los demás algoritmos no se ha llegado a una solución óptima ni eficiente por lo que se utilizará el algoritmo genético para su integración en los casos de uso reales del proyecto.

A continuación, se explicarán las distintas funciones que componen este algoritmo, ya que en todos los casos de usos las funciones son las mismas, pero en algunas su contenido cambia, pero eso se explicará detalladamente en cada escenario. Las funciones son las siguientes:

- *Generate_initial_population*: genera una población aleatoria de individuos y llama a la función Split_routes.
- *Split_routes*: crea rutas con la población de la anterior función cumpliendo con las restricciones de capacidad y autonomía de los vehículos.
- *Fitness_function*: recibe las rutas creadas en la anterior función y comprueba que sean correctas, es decir, que las distancias entre clientes sean distintas de 0 y además añade al principio y al final el almacén y devuelve la distancia total recorrida. Si no son correctas devuelve un número muy alto para que no lo tenga en cuenta en los siguientes pasos.
- *Select_parents*: se escoge aleatoriamente dos individuos de la población y sus respectivos fitness.
- *Crossover*: se recogen los individuos elegidos anteriormente y aleatoriamente se escoge un numero entre 0 y 1. Si es mayor que 0.9, que en nuestro caso, es la tasa de cruce nos quedamos con el primer individuo y no hay cruce, sino procedemos a coger las listas de rutas donde están involucrados y aleatoriamente cortamos en un punto de dichas rutas y el hijo será la combinación de la parte que hemos cortado del primer padre y de la que no está contenida del primer padre en el segundo padre y se llama a la función Split_routes para comprobar que se cumplen las restricciones.
- *Mutate*: ya obtenido el resultado de la anterior función, se genera un número aleatorio y si es mayor a MUTATION_RATE que en nuestro caso es 0.1 no surge mutación ya que buscamos que haya más crossover que mutaciones, en caso contrario se coge la ruta de dicho cliente y se mezclan aleatoriamente todos los clientes de dicha ruta para llamar a la función Split_routes y crear una ruta nueva.
- *Info_ruta*: devuelve la información del output final como la ruta completa, la distancia recorrida en cada ruta junto con su vehículo, la capacidad utilizada y el coste de cada ruta.

- *Genetic_algorithm_multiple_runs:* función que engloba todas las funciones anteriores. Como el algoritmo genético funciona con muchos valores aleatorios, lo que hace esta función es repetir 'x' veces el algoritmo genético para comparar todas las soluciones y quedarnos con la que menos distancia total (suma de las distancias recorridas en cada ruta) recorra.

Ya aclarado su funcionamiento, se explicarán los distintos casos de uso en los que se verá involucrado y una descripción del mismo:

Casos de Uso	Descripción
Caso 1: Optimización de rutas para los pedidos del último mes disponible	Utiliza los datos actuales de clientes, vehículos y pedidos para planificar las rutas de entrega, minimizando el coste total y maximizando el número de entregas realizadas dentro de las restricciones de capacidad, autonomía y vehículos disponibles.
Caso 2: Optimización de rutas para los pedidos del último mes disponible reduciendo la flota	Por decisión de la empresa, solo se pueden usar la mitad de los vehículos disponibles. Simula las rutas con una flota más pequeña (disminuir la flota en un 50% de vehículos) y evalúa el impacto en el número de entregas realizadas, el coste y la eficiencia general. ¿Qué cambios observas?
Caso 3: Predicción de demanda de pedidos para el próximo mes	Utilizando el histórico de pedidos, predice el número de pedidos esperados para el próximo mes. Con esta información, simula las rutas necesarias para cumplir con la demanda prevista. Para resolver este caso de uso es necesario usar la tabla de histórico de pedidos.
Caso 4: Optimización de rutas reduciendo la duración del trayecto	Para poner a prueba nuestras capacidades logísticas y demostrar nuestra eficiencia, se ha garantizado a nuestros clientes que su pedido será entregado en un plazo inferior a una hora desde que el vehículo sale del almacén.

I. Caso 1

Para la aplicación de este escenario primero se procede a leer los datos que van a ser necesarios para dicho caso, que son: datos históricos de pedidos, datos de los vehículos disponibles y datos de las distancias entre los clientes y el almacén que se obtienen de los csv df_distance_km, df_historic_order_demand y df_vehicle.

Al aplicar el algoritmo en situaciones diferentes, nos damos cuenta de que para generar rutas escoge los vehículos en orden por lo que, por norma general, los últimos vehículos no suelen tener muchos clientes y las rutas

son más baratas. Siguiendo esta idea, ordenamos los vehículos de menor a mayor consumo para que los vehículos de menor consumo tengan más clientes y las rutas, aunque más extensas, saldrán más baratas.

Al no especificar qué fecha de demanda escoger, se algoritmo está preparado para utilizar cualquier fecha que esté en dicho csv, lo que conlleva reforzar el algoritmo para que si en algún mes hay algún cliente que no haga pedido de producto, en la solución final no aparezca dicho cliente.

Mencionada la anterior información se muestra el resultado de aplicar el algoritmo a los pedidos realizados en el mes de febrero de 2021 en el que el Cliente_2 no realiza ningún pedido:

```
Mejor solución encontrada:
Vehículo 2:
  Ruta: Almacén -> Cliente 14 -> Cliente 8 -> Cliente 15 -> Cliente 5 -> Cliente 1 -> Cliente 12 -> Cliente 9 -> Almacén
  Distancia recorrida: 58,76 km
  Demanda total: 4294.0/[4362]kg
  Coste: 8,23€
Vehículo 6:
  Ruta: Almacén -> Cliente 16 -> Cliente 4 -> Cliente 18 -> Cliente 19 -> Cliente 17 -> Almacén
  Distancia recorrida: 64,2 km
  Demanda total: 3012.0/[3129]kg
  Coste: 8,99€
Vehículo 4:
  Ruta: Almacén -> Cliente 11 -> Cliente 10 -> Cliente 20 -> Cliente 13 -> Cliente 3 -> Almacén
  Distancia recorrida: 65,41 km
  Demanda total: 3084.0/[3321]kg
  Coste: 12,43€
Vehículo 1:
  Ruta: Almacén -> Cliente 6 -> Cliente 7 -> Almacén
  Distancia recorrida: 21,51 km
  Demanda total: 1163.0/[2026]kg
  Coste: 4,3€
------------------------------------------------------------
Distancia total recorrida: 209,88 km
Coste total de la ruta: 33,95€
```

Se puede observar que, como antes se ha mencionado, los primeros vehículos se llevan la mayor parte del trabajo viendo como el vehículo 1 solo reparte a dos clientes además de que en el csv de df_vehicle hay seis vehículos y en esta solución solo hay cinco, lo que indica que no se ha necesitado dicho vehículo.

Por otro lado, el Cliente_2 no aparece en ninguna ruta porque al no haber realizado ningún pedido no se incluye.

En relación con la capacidad máxima de cada vehículo, se ve nunca se ve superada ya que, por ejemplo, la suma de todos los pedidos realizados por los clientes de la primera ruta es de 4294 kg y la capacidad máxima del vehículo es de 4362 kg por lo que la restricción funciona además de la de autonomía.

Por último, se ve que el precio de cada ruta va de menos a más ya que antes se dijo que los vehículos se ordenaron de menos a más costo por km.

II. Caso 2

Para la aplicación de este escenario primero se procede a leer los datos que van a ser necesarios para dicho caso, que son: datos históricos de pedidos, datos de los vehículos disponibles y datos de las distancias entre los clientes y el almacén que se obtienen de los csv df_distance_km, df_historic_order_demand y df_vehicle.

En ambos escenarios, nos vemos obligados a reducir la flota de vehículos a la mitad, que en nuestro caso se ha hecho aleatoriamente para ver el impacto en cada resultado. Al ser aleatorio, el resultado final dependerá del vehículo que se haya escogido porque cada vehículo tiene su costo por km y su capacidad máxima además de su autonomía. Para los siguientes escenarios, se escogerá los pedidos que se realizaron en diciembre de 2024 en el que todos los clientes hacen pedidos.

En nuestra primera ejecución vemos como se han escogido los vehículos con las siguientes características:

```
Id vehiculo: 2, costo_km: 0.14, capacidad_max: 4362
Id vehiculo: 3, costo_km: 0.2, capacidad_max: 4881
Id vehiculo: 5, costo_km: 0.32, capacidad_max: 10000
```

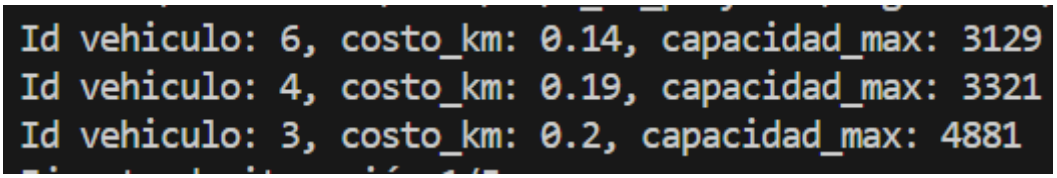
En este escenario, al haber solo tres vehículos disponibles el número de rutas va a aumentar ya que es casi imposible que con las restricciones de capacidad todos los pedidos se puedan entregar en solo tres rutas y al haber un vehículo con una carga de 10000 kg hará que este sea el que más pedidos entregue, además de que es el más caro por kilómetro y eso afectará al precio final como se muestra en la siguiente imagen:

```
Mejor solución encontrada:
Vehículo 2:
  Ruta: Almacén -> Cliente 11 -> Cliente 6 -> Cliente 9 -> Cliente 14 -> Almacén
  Distancia recorrida: 17,29 km
  Demanda total: 3668.0/[4362]kg
  Coste: 2,42€
Vehículo 3:
  Ruta: Almacén -> Cliente 7 -> Cliente 2 -> Cliente 19 -> Cliente 13 -> Cliente 4 -> Almacén
  Distancia recorrida: 41,58 km
  Demanda total: 4714.0/[4881]kg
  Coste: 8,32€
Vehículo 5:
  Ruta: Almacén -> Cliente 10 -> Cliente 5 -> Cliente 17 -> Cliente 12 -> Cliente 15 -> Cliente 16 -> Cliente 3 -> Cliente 8 -> Cliente 18 -> Cliente 20 ->
Almacén
  Distancia recorrida: 120,87 km
  Demanda total: 9447.0/[10000]kg
  Coste: 38,68€
Vehículo 2:
  Ruta: Almacén -> Cliente 1 -> Almacén
  Distancia recorrida: 7,22 km
  Demanda total: 909.0/[4362]kg
  Coste: 1,01€
------------------------------------------------------------
Distancia total recorrida: 186,96 km
Coste total de la ruta: 50,43€
```

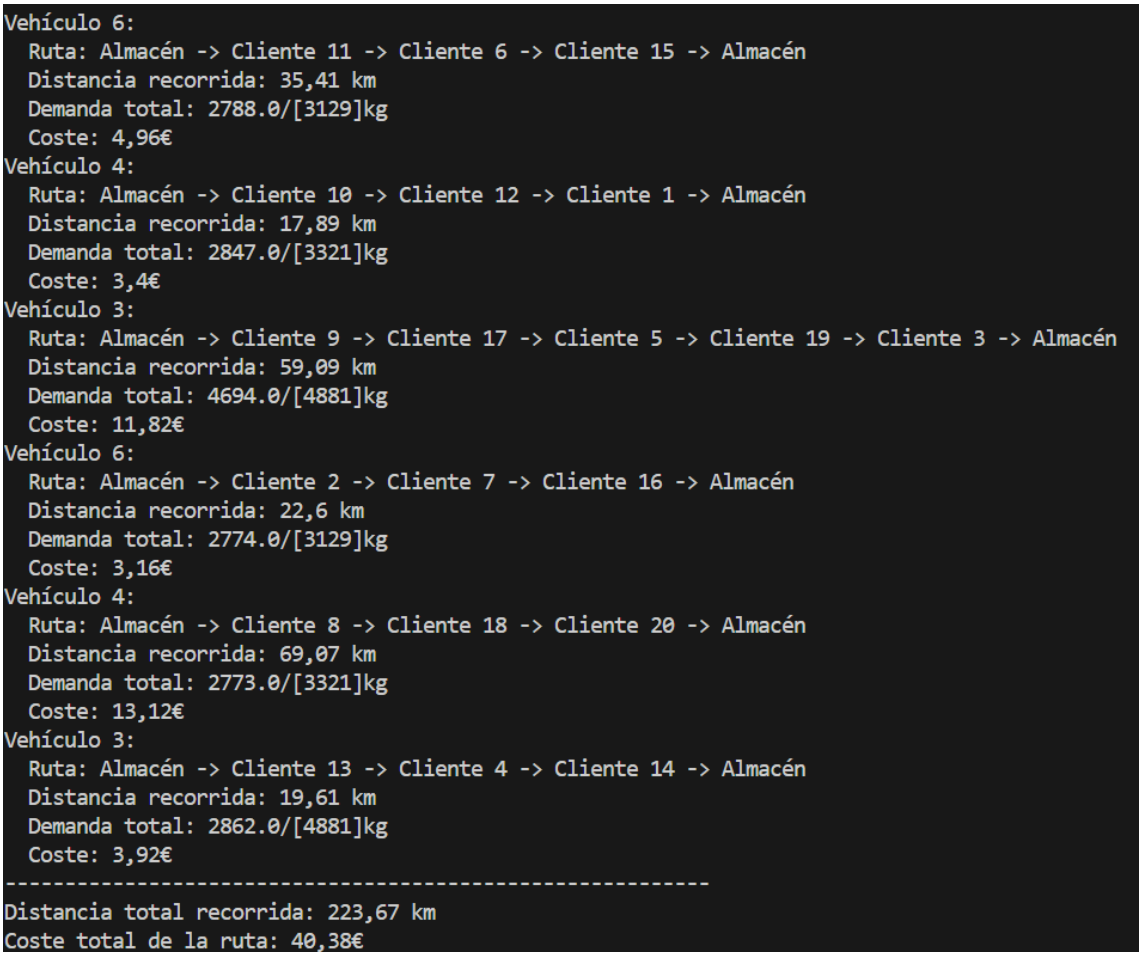
Se observa como las rutas de los vehículos 2 y 3 son más baratas debido a su precio por kilómetros, pero su capacidad se ve limitada. Por el mismo motivo, el vehículo 5 al tener mucha capacidad se lleva toda la demanda de clientes, pero su precio por kilómetro es mucho mayor lo que lleva a que el

precio final de las rutas sea de 50 € aunque la distancia recorrida no sea muy elevada. Cabe mencionar que los veinte clientes que hicieron pedidos en la fecha anteriormente mencionada reciben su pedido.

En este otro escenario se han elegido estos vehículos:



Como se ve dichos vehículos no tiene un costo por kilómetro muy elevado pero su capacidad es menor por lo que el precio final será barato pero el número de rutas crecerá:



Comparado con el anterior escenario, se observa que se han generado más rutas debido a la capacidad máxima de cada vehículo, pero su precio es mucho más barato por cada ruta.

En conclusión, el resultado final se verá afectado en gran medida en la elección aleatoria de los vehículos, pero por norma general los vehículos se verán más cargados de pedidos de lo normal para intentar suplir la demanda lo que llevará a más rutas y más kilómetros recorridos.

III. Caso 3

Para la aplicación de este escenario primero se procede a leer los datos que van a ser necesarios para dicho caso, que son: datos de pedidos de enero de 2025, datos de los vehículos disponibles y datos de las distancias entre los clientes y el almacén que se obtienen de los csv df_distance_km, pedidos_2025 y df_vehicle.

En este escenario, ya no tenemos los datos de la demanda de cada cliente, ya que el objetivo es realizar las rutas para los pedidos realizados en enero de 2025 pero el histórico de pedidos termina en diciembre de 2024 por lo que tenemos que predecirlos a partir del histórico de demandas.

Se procede a realizar un pequeño exploratorio de los datos que ya se mencionó en los puntos anteriores y se encuentra que el csv está compuesto por tres columnas que son:

- **Cliente:** aquí se encuentran los clientes.
- **Mes_anio:** la fecha compuesta por mes y el año de tipo string.
- **Order_demand:** la demanda de producto realizada por cada cliente.

La columna order_demand contenía nulos por lo que se procedió a su eliminación y la columna mes_anio al ser de tipo string se tuvo que separar en dos columnas una de mes y otra de año y casteada a numérica.

Realizados los ajustes necesarios de los datos, se procedió a dividir los datos en entrenamiento (80 %) y test (20%) para realizar la predicción correctamente.

Para aplicar a la predicción se tienen en cuenta dos algoritmos:

- *Random Forest Regressor*: en este algoritmo se combinan varios árboles de regresión, cuya predicción final será la media de los resultados de cada árbol. Se ha tenido en cuenta por su posibilidad de personalizar muchos parámetros para realizar una mejor predicción y por estar mejor preparado para el overfitting que, por ejemplo, un árbol de regresión.
- *XGBRFRegressor:* este algoritmo también utiliza árboles de regresión como el anterior algoritmo, pero trata de mejorar el resultado teniendo en cuenta el resultado obtenido por el árbol anterior.

Finalmente se optó por la utilización del Random Forest Regressor al obtener mejores resultados.

Tras dividir los datos procedemos a entrenar el modelo. Para encontrar los mejores parámetros, se opta por crear una lista de valores por cada

parámetro que influya en la predicción que son la profundidad de los árboles, el criterio utilizado y el min_samples_split que cuyo valor será el valor mínimo de muestras necesario para hacer una división en un nodo.

Hechas las listas se procede a crear un modelo por cada cliente para realizar una correcta predicción de los valores y el resultado obtenido para enero de 2025 es el siguiente:

	cliente	order_demand
0	Cliente_1	725.0
1	Cliente_2	721.0
2	Cliente_3	768.0
3	Cliente_4	743.0
4	Cliente_5	679.0
5	Cliente_6	705.0
6	Cliente_7	713.0
7	Cliente_8	741.0
8	Cliente_9	707.0
9	Cliente_10	690.0
10	Cliente_11	753.0
11	Cliente_12	692.0
12	Cliente_13	726.0
13	Cliente_14	712.0
14	Cliente_15	740.0
15	Cliente_16	726.0
16	Cliente_17	708.0
17	Cliente_18	720.0
18	Cliente_19	694.0
19	Cliente_20	848.0

Ya obtenidas las predicciones, se procede a implementarlas al algoritmo genético y el resultado obtenido es el siguiente:

```
Mejor solución encontrada:
Vehículo 2:
  Ruta: Almacén -> Cliente 20 -> Cliente 6 -> Cliente 15 -> Cliente 8 -> Cliente 9 -> Almacén
  Distancia recorrida: 76,03 km
  Demanda total: 3741.0/[4362]kg
  Coste: 10,64€
Vehículo 6:
  Ruta: Almacén -> Cliente 17 -> Cliente 1 -> Cliente 10 -> Cliente 13 -> Almacén
  Distancia recorrida: 35,6 km
  Demanda total: 2849.0/[3129]kg
  Coste: 4,98€
Vehículo 4:
  Ruta: Almacén -> Cliente 19 -> Cliente 2 -> Cliente 3 -> Cliente 18 -> Almacén
  Distancia recorrida: 58,86 km
  Demanda total: 2903.0/[3321]kg
  Coste: 11,18€
Vehículo 1:
  Ruta: Almacén -> Cliente 16 -> Cliente 11 -> Almacén
  Distancia recorrida: 10,46 km
  Demanda total: 1479.0/[2026]kg
  Coste: 2,09€
Vehículo 3:
  Ruta: Almacén -> Cliente 4 -> Cliente 7 -> Cliente 12 -> Cliente 14 -> Cliente 5 -> Almacén
  Distancia recorrida: 35,34 km
  Demanda total: 3539.0/[4881]kg
  Coste: 7,07€
------------------------------------------------------------
Distancia total recorrida: 216,29 km
Coste total de la ruta: 35,96€
```

Como se observa, es un escenario parecido al del caso 1 en el sentido de que se utilizan cinco vehículos de los seis disponibles y las rutas más largas la obtienen vehículos con costo por kilómetro barato obteniendo finalmente un precio muy parecido.

IV. Caso 4

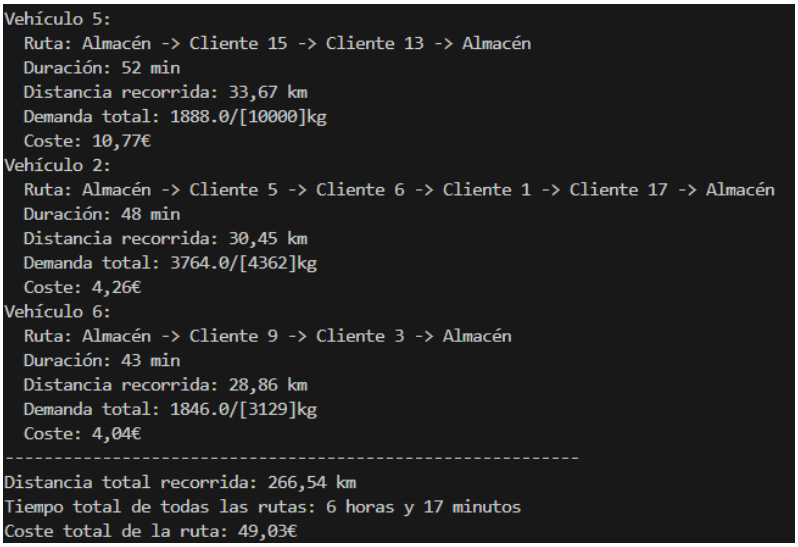
Para la aplicación de este escenario primero se procede a leer los datos que van a ser necesarios para dicho caso, que son: datos históricos de pedidos, datos de los vehículos disponibles, datos de las distancias entre los clientes en kilómetros y datos de las distancias entre clientes y el almacén en minutos que se obtienen de los csv df_distance_km, df_distance_min, df_historic_order_demand y df_vehicle.

En este último caso de uso, para poner a prueba nuestras capacidades logísticas y demostrar nuestra valía en situaciones límite a nuestros clientes y a la competencia, se ha garantizado a nuestros clientes que su pedido será entregado en un plazo inferior a una hora desde que el vehículo sale del almacén.

Para ello, se han cambiado el contenido de algunas funciones para este escenario especial como Split_routes que ahora trabaja con las distancias en minutos en vez de kilómetros y poner la restricción de que cada ruta debe durar 60 o menos de 60 minutos. La función fitness_function ahora trabaja con distancias en minutos para posteriormente de la función genetic_algorithm_multiple_runs se comparé entre distancias en minutos.

Explicado los pequeños retoques de nuestro algoritmo el resultado obtenido se muestra en las siguientes imágenes:

```
Vehículo 6:
  Ruta: Almacén -> Cliente 4 -> Cliente 10 -> Cliente 8 -> Almacén
  Duración: 55 min
  Distancia recorrida: 40,5 km
  Demanda total: 2864.0/[3129]kg
  Coste: 5,67€
Vehículo 4:
  Ruta: Almacén -> Cliente 19 -> Cliente 14 -> Almacén
  Duración: 55 min
  Distancia recorrida: 46,01 km
  Demanda total: 1833.0/[3321]kg
  Coste: 8,74€
Vehículo 1:
  Ruta: Almacén -> Cliente 18 -> Cliente 16 -> Almacén
  Duración: 42 min
  Distancia recorrida: 26,38 km
  Demanda total: 1863.0/[2026]kg
  Coste: 5,28€
Vehículo 3:
  Ruta: Almacén -> Cliente 20 -> Almacén
  Duración: 34 min
  Distancia recorrida: 29,66 km
  Demanda total: 881.0/[4881]kg
  Coste: 5,93€
```



Se observa que al poner la restricción de menos de 60 min hay más rutas de lo normal, en este caso siete por lo que el vehículo dos al volver del primer trayecto tiene que repostar y volver a salir.

Aunque haya habido más rutas y se haya recorrido más distancia en kilómetros, el precio no se ha visto demasiado afectado y esto se ha debido a que los trayectos más largos se los han llevado los vehículos con costo por kilómetro más barato.

Para finalizar, vemos la eficacia y el cumplimiento de las restricciones de minutos, capacidad y autonomía que no han sido rebasadas en ningún momento.

En conclusión, se ha visto que el número de rutas ha aumentado en relación con el aumento de restricciones pero el resultado ha sido mejor de lo esperado.