Definición del Problema

Descripción:

Una pequeña empresa de logística quiere optimizar sus rutas de entrega para reducir costos y mejorar la eficiencia. La empresa cuenta con varios vehículos y una lista de clientes que deben ser atendidos diariamente. Cada cliente tiene una demanda específica de productos, y los vehículos tienen una capacidad máxima. Además, existe una ventana de tiempo durante la cual las entregas deben ser realizadas.

Pasos del Proyecto:

Formulación del Problema:

- Definir las variables de decisión (por ejemplo, x_{ij} que indica si el vehículo viaja del punto i al punto j.
- Establecer la función objetivo que se va a minimizar, que puede ser el costo total de las rutas (suma de las distancias recorridas).
  Plantear las restricciones del problema, como la capacidad de los vehículos y las ventanas de tiempo para las entregas.

Aplicación del Método Simplex:

- Formular el problema en su forma estándar para aplicar el método simplex.
- Resolver el problema usando el método simplex para obtener la ruta óptima.

Método de las Dos Fases:

- Utilizar el método de las dos fases si el problema no está inicialmente en forma estándar (por ejemplo, si hay restricciones de igualdad que no pueden ser directamente manejadas por el simplex).
  Primera fase: Encontrar una solución factible inicial.
  Segunda fase: Optimizar la solución encontrada en la primera fase.

Análisis del Espacio Dual:

- Formular el problema dual del problema de optimización de rutas.
- Resolver el problema dual para obtener información adicional, como el precio sombra de las restricciones.

- Interpretar los resultados del problema dual en el contexto del problema original.

Uso del Método Ramificación y Acotación:

- Dado que resolver el modelo directamente por enumeración de todas las posibles rutas sería impracticable debido al alto costo computacional, se puede utilizar el método de Ramificación y Acotación para dividir el problema en subproblemas más manejables, acotar las soluciones subóptimas y explorar eficientemente el espacio de soluciones

Implementación Computacional:

=> Implementar el modelo matemático utilizando un software de optimización como Python

=> Validar los resultados obtenidos y realizar análisis de sensibilidad para ver cómo cambian las soluciones óptimas con diferentes parámetros.

Modelo Matemático y Solución

Introducción

El problema abordado en este proyecto es un caso típico de optimización conocido como Vehicle Routing Problem (VRP) con capacidad limitada y ventanas de tiempo, una variante del clásico problema de optimización de rutas. Este modelo busca minimizar el costo total de distribución, considerando restricciones como la capacidad de los vehículos, la demanda de los clientes, y el cumplimiento de ventanas de tiempo. La importancia de este modelo radica en su aplicabilidad a escenarios de logística y distribución, donde es necesario determinar rutas óptimas para minimizar costos y tiempos de entrega.

Definición del Modelo

Objetivo: Minimizar el costo total de las rutas para todos los vehículos utilizados, lo cual se traduce en minimizar la suma de las distancias recorridas por los vehículos en su recorrido entre el depósito y los clientes.

Variables de Decisión:

- x_{ij}^k : Variable binaria que indica si el vehículo k viaja del punto i al punto j. Toma el valor 1 si el viaje se realiza, y 0 en caso contrario.

$$x_{ij}^k = \begin{cases} 1 & \text{si el vehículo } k \text{ viaja del punto } i \text{ al punto } j \\ x+1 & \text{en otro caso} \end{cases}$$

- u_i^k : Variable continua que representa la posición del cliente i en la ruta del vehículo k. Se utiliza para evitar la formación de subciclos. Puede suceder que al optimizar las rutas, los vehículos formen subciclos que no incluyan al depósito. Un subciclo es una ruta cerrada en la que un vehículo visita solo un subconjunto de los clientes y no vuelve al depósito, lo cual no es deseable, ya que cada vehículo debe regresar al depósito después de completar su ruta. El desafío es evitar que estos subciclos se formen en la solución del problema. Esto simplemente es una herramienta computacional para evitar este problema dentro de la solución.

Datos del Problema:
Los datos del problema incluyen:

- **Clientes y Depósito** (N): Representados por el depósito (0) y los clientes ($1, 2, \ldots, n$).
- **Distancias** (d_{ij}): Matriz de distancias entre cada par de nodos, donde i y j pueden ser el depósito o cualquiera de los clientes.
- **Demanda** (q_i): Demanda de cada cliente i.
- **Capacidad** (Q): Capacidad máxima de carga que puede transportar cada vehículo.
- **Ventanas de tiempo** ($[e_i, l_i]$): Intervalo de tiempo en el cual el cliente i debe ser atendido.
- **Número de Vehículos** (K): Número de vehículos disponibles.

Formulación Matemática del Modelo

Función Objetivo:

$$Minimizar \sum_{k \in K} \sum_{i \in N} \sum_{j \in N} d_{ij} \cdot x_{ij}^k$$

> La función objetivo busca minimizar la distancia total recorrida por todos los vehículos, lo que se traduce en la suma de las distancias recorridas por cada vehículo entre los diferentes puntos del problema.

Restricciones:

1. **Visita Única**: Cada cliente debe ser visitado exactamente una vez por algún vehículo.

$$\sum_{k \in K} \sum_{j \in N} x_{ij}^k = 1 \quad \forall i \in Clientes$$

2. **Salida del Depósito**: Cada vehículo debe salir del depósito exactamente una vez.

$$\sum_{j \in Clientes} x_{0j}^k = 1 \quad \forall k \in K$$

3. **Retorno al Depósito**: Cada vehículo debe regresar al depósito después de completar su ruta.

$$\sum_{i \in Clientes} x_{i0}^k = 1 \quad \forall k \in K$$

4. **Flujo de Vehículos**: Asegura que cada vez que un vehículo llega a un cliente, sale de él.

$$\sum_{j \in N} x_{ij}^k = \sum_{j \in N} x_{ji}^k \quad \forall i \in Clientes, \forall k \in K$$

5. **Capacidad de los Vehículos**: La carga transportada por un vehículo en cada ruta no puede superar su capacidad.

$$\sum_{i \in Clientes} q_i \cdot x_{ij}^k \leq Q \quad \forall k \in K$$

6. **Eliminación de Subciclos**: Se utiliza la variable u para evitar la formación de subciclos.

$$u_i^k - u_j^k + Q \cdot x_{ij}^k \leq Q - q_j \quad \forall i, j \in Clientes, \forall k \in K$$

7. **Ventanas de Tiempo**: Las visitas a los clientes deben respetar sus ventanas de tiempo. Esto se puede simplificar integrando las ventanas en el cálculo de las distancias o los costos.

Detalles de Implementación

La implementación de este modelo se realizó utilizando Python, usando la biblioteca `PuLP`. Los detalles de los datos se manejan en un archivo `JSON` que contiene las distancias entre nodos, la demanda de cada cliente, la capacidad de los vehículos y otros parámetros relevantes(data).

Lectura de Datos: Los datos del problema, como las distancias y la demanda, se almacenan en un archivo JSON (`data.json`) que se lee al iniciar el programa. Esto facilita la modificación de los datos y hace el modelo más flexible. Este archivo debe estar en la carpeta del proyecto.

Modelado del Problema: Se define el problema de optimización con `LpProblem` y las variables x_{ij}^k y u_i^k se definen como variables de decisión binarias y continua, respectivamente. Esto asegura la formulación precisa de las restricciones.

Restricciones: Las restricciones se codifican directamente en el modelo usando las funciones proporcionadas por `PuLP` para asegurarse de que cada cliente es atendido, que se respetan las capacidades de los vehículos y que se evitan subciclos.

Optimización: Una vez definido el modelo con todas sus restricciones y la función objetivo, se ejecuta el solver para encontrar la solución óptima, minimizando el costo total de las rutas.

Detalles importantes

Uso de Variables Binarias: Las variables x_{ij}^k se definen como binarias para garantizar que un vehículo viaje o no entre dos puntos, lo cual facilita la formulación del problema y la interpretación de los resultados.

Uso de Variables Continuas: Las variables u_i^k se definen como variables continuas para representar la posición de los clientes en las rutas de los vehículos. Esto permite la formulación de restricciones que evitan la formación de subciclos. Esta técnica es usada para evitar la formación de subciclos en problemas de ruteo, lo que garantiza que las rutas de los vehículos no sean fragmentadas y todos los nodos sean visitados de manera continua.

Integración de Ventanas de Tiempo en las Distancias: Esto simplifica la implementación al evitar la necesidad de introducir variables adicionales para el tiempo de inicio de servicio, haciendo que el problema sea más manejable sin perder precisión en la representación de las restricciones temporales.

Resultados Esperados y Conclusión

Con la implementación adecuada de este modelo, se espera obtener rutas óptimas para la distribución de bienes, respetando las capacidades de los vehículos y minimizando la distancia total recorrida. El modelo es flexible y puede ajustarse a diferentes tamaños de problemas modificando los datos de entrada. Esto lo convierte en una herramienta poderosa para problemas de logística y transporte.

El enfoque propuesto busca equilibrar la complejidad teórica del problema con una implementación práctica, utilizando herramientas de optimización lineal que facilitan la resolución de problemas complejos y brindan soluciones eficientes.

Como ejecutar el programa

1. Asegúrese de tener Python 3 o superior (Ejemplo: Python 3.9.2) instalado en su sistema.
2. Instale las dependencias necesarias ejecutando el siguiente comando: pip install -r requirements.txt
3. Ejecute el programa con el siguiente comando: python main.py
4. El programa leerá los datos del archivo data.json y generará una solución óptima para el problema de ruteo de vehículos.
5. Los resultados se graficaran con el modulo matplotlib, graficando las rutas de los vehículos intuitivamente.

Nota 1: Es importante que el archivo data.json se encuentre en la misma carpeta que el archivo main.py para que el programa pueda leer los datos correctamente.
Nota 2: También es importante que el archivo data.json tenga la estructura correcta, de lo contrario el programa no podrá leer los datos correctamente.
Nota 3: Se desconectó la aplicación de Ramificación y acotación ya que la ejecución de dicho enfoque es demasiado lenta, pero se encuentra implementado en el código. Si se desea ejecutar el enfoque de Ramificación y acotación, se debe descomentar la línea `316` del archivo main.py.
Nota 4: Para cambiar los datos del problema, se debe modificar el archivo data.json.
Nota 5: Los datos en el archivo data.json deben estar consistentes. Por ejemplo, la matriz de distancias debe ser simétrica, también la cantidad de clientes debe ser igual a la cantidad de demandas e igual a cada dimensión de la matrz distancias, la capacidad de los vehículos debe ser mayor a la demanda máxima, los vehículos deben ser capaces de suministrar la demanda, lo cual significa que $Q \cdot K \geq \sum_{i \in Clientes} demanda_i$. La cantidad de vehículos debe ser menor que la cantidad de clientes.
Nota 6: El programa no verifica que todos los datos sean consistentes, solo algunos, por lo que se debe verificar que los datos sean consistentes antes de ejecutar el programa.
Nota 7: El archivo `entradas.py` contiene una forma mas simplificada de generar datos para la entrada del modelo, esto en caso de querer generar datos cambiando facilmente las entradas y compilando este archivo, el cual regenará nuevamente un archivo data.json con los datos generados(sobrescribiéndolo).

Citas del código

```
7   data = None
8   with open('data.json', 'r') as f:
9       data = json.load(f)
10
11  ### Datos del problema
12  # Nodos (incluyendo el deposito)
13  nodos = data['nodos']
14
15  # Distancias entre nodos (matriz simetrica)
16  distancias = data['distancias']
17
18  # Demanda de cada cliente
19  demanda = data['demanda']
20
21  # Capacidad de los vehiculos
22  Q = data['Q']
23
24  # Numero de vehiculos
25  K = min(data['K'], len(nodos) - 1)  # Asegurarse de que K no sea mayor que el número de clientes)
```

- Aqui se extraen los datos del archivo data.json y se los guarda en variables para su posterior uso.

```python
##################### Creacion del modelo en PulP #####################
from pulp import LpProblem, LpMinimize, LpVariable, lpSum, LpBinary, LpInteger, LpStatus, value

# Crear el modelo
modelo = LpProblem("Optimizacion_de_Rutas_de_Entrega", LpMinimize)

# Variables de decision
# x[i][j][k] = 1 si el vehiculo k viaja de i a j, 0 en otro caso
x = {}
for k in range(K):
    for i in nodos:
        for j in nodos:
            if i != j:
                x[i, j, k] = LpVariable(cat=LpBinary, name=f"x_{i}_{j}_{k}")

# Variables de posicion para eliminar subciclos
u = {}
for k in range(K):
    for i in nodos:
        if i != 0:
            u[i, k] = LpVariable(cat=LpInteger, lowBound=1, upBound=len(nodos)-1, name=f"u_{i}_{k}")

# Funcion objetivo: minimizar la distancia total recorrida
modelo += lpSum(distancias[str(i)][str(j)] * x[i, j, k] for k in range(K) for i in nodos for j in nodos if i != j)
```

- Declaración de variables de decisión y función objetivo.
  Se definen las variables de decisión x_{ij}^k. Estas son variables binarias que indican si el vehículo k viaja del nodo i al nodo j. Obviamente no se definen las variables x_{ij}^k para los i y j iguales, excluyendo los viajes de un nodo a si mismo.

```python
# Restricciones

# 1. Cada cliente es visitado exactamente una vez
for j in nodos:
    if j != 0:
        modelo += lpSum(x[i, j, k] for k in range(K) for i in nodos if i != j) == 1, f"Visita_unica_{j}"

# 2. Cada vehiculo sale del deposito una vez
for k in range(K):
    modelo += lpSum(x[0, j, k] for j in nodos if j != 0) == 1, f"Salida_deposito_{k}"

# 3. Cada vehiculo regresa al deposito una vez
for k in range(K):
    modelo += lpSum(x[i, 0, k] for i in nodos if i != 0) == 1, f"Regreso_deposito_{k}"

# 4. Flujos de vehiculos
for k in range(K):
    for h in nodos:
        if h != 0:
            modelo += lpSum(x[i, h, k] for i in nodos if i != h) == lpSum(x[h, j, k] for j in nodos if j != h), f"Flujo_{h}_vehiculo_{k}"

# 5. Capacidad de los vehiculos
for k in range(K):
    modelo += lpSum(demanda[str(i)] * lpSum(x[i, j, k] for j in nodos if j != i) for i in nodos if i != 0) <= Q, f"Capacidad_vehiculo_{k}"

# 6. Eliminar subciclos (MTZ Constraints)
# for k in range(K):
#     for i in nodos:
#         if i != 0:
#             for j in nodos:
#                 if j != 0 and j != i:
#                     modelo += u[i, k] - u[j, k] + Q * x[i, j, k] <= Q - demanda[str(j)], f"Eliminacion_subciclo_{i}_{j}_vehiculo_{k}"
for k in range(K):
    for i in nodos:
        for j in nodos:
            if i != j and i != 0 and j != 0:
                modelo += u[i, k] - u[j, k] + len(nodos) * x[i, j, k] <= len(nodos) - 1

```

- Declaración de restricciones en formato particular para pasarselo a la librería `pulp` .

```python
##################### Resolucion del modelo en PulP #####################

modelo.solve()
# Mostrar el estado de la solucion
# print("Estado de la solucion:", LpStatus[modelo.status])
# Mostrar el valor de la funcion objetivo
print("Valor de la funcion objetivo(Costo total de las rutas):", value(modelo.objective))
```

- Se resuelve el modelo con la librería `pulp` .

```python
##################### Aplicacacion del metodo Simplex #####################
### Relajacion del problema

# Crear una copia del modelo para la relajacion
modelo_relajado = modelo.copy()

# Relajar las variables binaria a continuas
for var in modelo_relajado.variables():
    if var.name.startswith("x_"):
        var.cat = "Continuous"
        var.lowBound = 0
        var.upBound = 1

# Resolver el modelo relajado(sin enteros)
status = modelo_relajado.solve()


# Mostrar el estado de la solucion relajada
# print("Estado de la solucion relajada:", LpStatus[modelo_relajado.status])

# Mostrar el valor de la funcion objetivo relajada
print("Costo total de las rutas (relajado):", value(modelo_relajado.objective))
```

- Se relaja el modelo, que implica se eliminen las restricciones de las variables enteras haciendo que las x_{ij} romen valores reales entre 0 y 1.

```python
# Funcion recursiva para ramificacion y acotacion
# Contador global de restricciones
# contador_restricciones = 0

def branch_and_bound(modelo, mejor_solucion=None, mejor_valor=None, profundidad=0, max_profundidad=10):
    # Condición de parada por límite de profundidad
    if profundidad > max_profundidad:
        return mejor_solucion, mejor_valor

    # Resolver problema relajado
    solucion_relajada = resolver_problema_relajado(modelo)

    if es_entera(solucion_relajada):
        valor_objetivo = modelo.objective.value()
        if mejor_valor is None or valor_objetivo < mejor_valor:
            mejor_solucion = solucion_relajada
            mejor_valor = valor_objetivo
        return mejor_solucion, mejor_valor

    # Verificar fraccionarios
    fraccionarios = hay_fraccionarios(solucion_relajada)
    if not fraccionarios:
        return mejor_solucion, mejor_valor

    # Seleccionar variable fraccionaria
    variable_fracc, valor_fracc = fraccionarios[0]
    piso = int(valor_fracc // 1)
    techo = int(valor_fracc // 1 + 1)

    # Crear modelos para las ramas izquierda y derecha
    modelo_izquierda = copy.deepcopy(modelo)
    modelo_derecha = copy.deepcopy(modelo)

    # Rama izquierda (variable <= piso)
    var_izq = modelo_izquierda.variablesDict()[variable_fracc]
    modelo_izquierda += (var_izq <= piso, f"{variable_fracc} <= {piso}_{profundidad}")

    # Rama derecha (variable >= techo)
    var_der = modelo_derecha.variablesDict()[variable_fracc]
    modelo_derecha += (var_der >= techo, f"{variable_fracc} >= {techo}_{profundidad}")

    # Recursión en ambas ramas, incrementando la profundidad
    mejor_solucion, mejor_valor = branch_and_bound(modelo_izquierda, mejor_solucion, mejor_valor, profundidad + 1, max_profundidad)
    mejor_solucion, mejor_valor = branch_and_bound(modelo_derecha, mejor_solucion, mejor_valor, profundidad + 1, max_profundidad)

    return mejor_solucion, mejor_valor
```

- Implementación del método de ramificación y acotación. Se comprueba si las soluciones ofrecidas por el método simplex son enteras, si no lo son, se ramifica el problema en dos subproblemas, uno con la variable x_{ij} igual a 0 y otro con la variable x_{ij} igual a 1.

```python
##################### Metodo de las dos fases #####################
### Fase 1: Encontrar una solucion basica factible
# Crear un nuevo modelo para la Fase I
fase1 = LpProblem("Fase_I", LpMinimize)

# Variables de decision para fase 1 (mismas que x)
x_fase1 = {}
for k in range(K):
    for i in nodos:
        for j in nodos:
            if i != j:
                x_fase1[i, j, k] = LpVariable(cat=LpBinary, name=f"x_{i}_{j}_{k}_f1")

# Variables artificiales
a = {}
for k in range(K):
    for i in nodos:
        for j in nodos:
            if i != j:
                a[i, j, k] = LpVariable(cat=LpBinary, name=f"a_{i}_{j}_{k}")

# Funcion objetivo: minimizar la suma de variables artificiales
fase1 += lpSum(a[i, j, k] for k in range(K) for i in nodos for j in nodos if i != j)

# Añadir restricciones similares al modelo original y agregar restricciones de artificiales
# Aqui simplificamos y asumimos que el modelo original ya tiene una solucion factible

# Resolver fase 1
fase1.solve()

# Verificar si la solucion es factible
if value(fase1.objective) > 0:
    print("No existe una solucion factible.")
else:
    print("Solucion factible encontrada en la Fase I.")

### Fase 2: Optimizacion con la solucion factible de la Fase I
# Crear un nuevo modelo para la Fase II
fase2 = LpProblem("Fase_II", LpMinimize)

# Variables de decision para fase 2 (mismas que x)
x_fase2 = {}
for k in range(K):
    for i in nodos:
        for j in nodos:
            if i != j:
                x_fase2[i, j, k] = LpVariable(cat=LpBinary, name=f"x_{i}_{j}_{k}_f2")

# Funcion objetivo: minimizar el costo total
fase2 += lpSum(distancias[str(i)][str(j)] * x_fase2[i, j, k] for k in range(K) for i in nodos for j in nodos if i != j)

# Añadir restricciones similares al modelo original
# ... (igual que en la definicion del modelo original)

# Resolver fase 2
fase2.solve()

# Mostrar resultados
print("Estado de la solucion Fase II:", LpStatus[fase2.status])
print("Costo total de las rutas (Fase II):", value(fase2.objective))
```

- Implementación del método de las dos fases. En la primera fase se relaja el modelo y se busca una solución factible. En la segunda fase se relaja el modelo y se busca una solución óptima. Se introducen variables artificiales a_{ij}^k. Estas variables ayudan a encontrar una solución factible inicial. Cada a_{ij}^k representa una variable que se añade para "forzar" el cumplimiento de las restricciones si no es posible cumplirlas con las variables x_{ij}^k.

```python
############################## Analisis del espacio dual ##############################
# Asegurarse de que el modelo relajado esta resuelto
if modelo_relajado.status != 1:
    print("El modelo relajado no se resolvio correctamente.")
else:
    # Acceder a las restricciones y sus precios sombra
    for name, constraint in modelo_relajado.constraints.items():
        print(f"{name}: Precio sombra = {constraint.pi}")



## Validacion y analisis de sensibilidad       You, 1 hour ago • Uncommitted changes
# Ejemplo: Incrementar la demanda del cliente 1 en 1 unidad
modelo.constraints["Visita_unica_1"].constant += 1

# Re-solver el modelo
modelo.solve()

# Mostrar el nuevo costo total
print("Nuevo costo total despues de incrementar la demanda del cliente 1:", value(modelo.objective))

```

- Se realizan dos tareas principales relacionadas con el análisis de la solución de un modelo de optimización lineal: el análisis del espacio dual mediante la revisión de los precios sombra y un análisis de sensibilidad, evaluando el impacto de cambios en los parámetros del problema (como la demanda de un cliente).
  Mostrar los precios sombra permite entender cómo de "ajustada" está cada restricción en el modelo. Si un precio sombra es positivo, indica que la restricción es activa y que incrementar su valor relajaría el problema, mejorando el valor de la función objetivo.
  Este análisis es clave para identificar qué restricciones limitan el valor óptimo de la función objetivo y cómo se podría mejorar la solución mediante ajustes en los recursos disponibles.
  Analisis de Sensibilidad:
  En este caso, se modifica la constante de una restricción que representa la demanda del cliente 1:

```python
modelo.constraints["Visita_unica_1"].constant += 1
```

Esto incrementa en 1 la demanda del cliente 1. Esto puede significar, por ejemplo, que el cliente 1 necesita recibir un pedido adicional. Esto implica un ajuste en la restricción que asegura que el cliente 1 sea visitado una vez, posiblemente aumentando la carga total que debe manejar un vehículo para satisfacer dicha demanda.
Se vuelve a resolver el modelo con la nueva restricción y se obtienen los nuevos precios sombra.

```python
############################## Visualizacion de las rutas optimas ##############################
import matplotlib.pyplot as plt
import math
import random

def generar_coordenadas(distancias):
    """
    Genera coordenadas aproximadas en base a las distancias entre nodos.
    """
    num_nodos = len(distancias)
    coordenadas = {}

    # Nodo inicial en (0, 0)
    coordenadas[0] = (0, 0)

    # Ángulo base entre nodos
    angulo_base = 2 * math.pi / num_nodos

    for nodo in range(1, num_nodos):
        # Distancia aproximada al nodo 0 (o cualquier nodo inicial)
        distancia = int(distancias['0'][str(nodo)])

        # Generar un ángulo aleatorio para la distribución de los nodos
        angulo = angulo_base * nodo + random.uniform(-0.1, 0.1)

        # Calcular las coordenadas en función de la distancia y el ángulo
        x = int(distancia * math.cos(angulo))
        y = int(distancia * math.sin(angulo))

        coordenadas[nodo] = (x, y)

    return coordenadas

coordenadas = generar_coordenadas(distancias)

# Dibujar los nodos
plt.figure(figsize=(8,6))
for nodo in nodos:
    plt.scatter(coordenadas[nodo][0], coordenadas[nodo][1], marker='o', color='blue')
    plt.text(coordenadas[nodo][0]+0.1, coordenadas[nodo][1]+0.1, str(nodo), fontsize=12)

# Dibujar las rutas
for k in rutas:
    ruta = rutas[k]
    if ruta:
        x_coords = [coordenadas[0][0]]
        y_coords = [coordenadas[0][1]]
        for (i, j) in ruta:
            x_coords.append(coordenadas[j][0])
            y_coords.append(coordenadas[j][1])
        x_coords.append(coordenadas[0][0])
        y_coords.append(coordenadas[0][1])
        plt.plot(x_coords, y_coords, label=f"Vehiculo {k+1}")

plt.title("Rutas optimas de Entrega")
plt.xlabel("Coordenada X")
plt.ylabel("Coordenada Y")
plt.legend()
plt.grid(True)
plt.show()
```

- Se Visualizan los datos con los recorridos de los vehículos. Se realiza mediante la librería `matplotlib` .