# PRÁCTICA 3 – GP02A

Enrique Fernández-Baíllo Rodríguez de Tembleque

Javier Escobar Serrano

## CONSTRUCCIÓN DEL GRAFO DE CALLES

Para construir el grafo de calles, fue muy útil añadir a ambos DataFrames una columna llamada “coordenadas”, donde aparecen las coordenadas del eje X y del eje Y como elementos de una lista, facilitando el uso de las coordenadas tanto de los cruces como de las direcciones, y haciendo más rápida su lectura al solo tener que leer un dato en vez de dos. Esto se llevó a cabo mediante la creación de la función “coordenadas_generar”, añadida a dgt.py y añadida a la función de “process_data”.

```python
def coordenadas_generar(df_cruces: pd.DataFrame, df_direcc: pd.DataFrame) -> pd.DataFrame:
    df_cruces["coordenadas"] = list(zip(df_cruces["Coordenada X (Guia Urbana) cm (cruce)"], df_cruces["Coordenada Y (Guia Urbana) cm (cruce)"]))
    df_direcc["coordenadas"] = list(zip(df_direcc["Coordenada X (Guia Urbana) cm"], df_direcc["Coordenada Y (Guia Urbana) cm"]))
    return df_cruces, df_direcc
```

El archivo de callejero.py define primero las constantes VELOCIDAD_CALLES y VELOCIDAD_CALLES_ESTANDAR, importa las librerías necesarias y después procesa los DataFrames de cruces y de direcciones. Después, define los objetos de Calle y de Cruce.

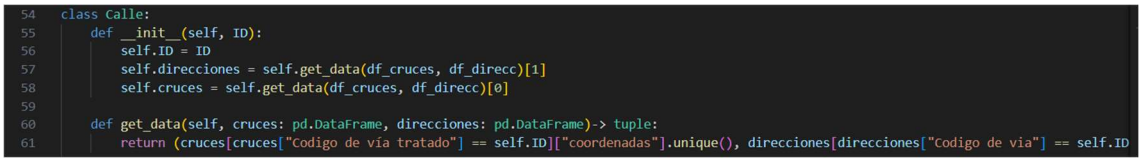
- OBJETO “CRUCE”

El objeto “Cruce” se inicia definiendo la coordenada X, la coordenada Y del cruce y las calles asignadas al cruce, esto es, una lista con los códigos de vía de las calles que forman parte del cruce, generada mediante la función “get_calles”.

```python
class Cruce:
    def __init__(self,coord_x,coord_y):
        self.coord_x=coord_x
        self.coord_y=coord_y
        self.calles = self.get_calles(df_cruces)

    """Se hace que la clase Cruce sea "hashable" mediante la implementación de los métodos
    __eq__ y __hash__, haciendo que dos objetos de tipo Cruce se consideren iguales cuando
    sus coordenadas coincidan (es decir, C1==C2 si y sólo si C1 y C2 tienen las mismas coordenadas),
    independientemente de los otros campos que puedan estar almacenados en los objetos.
    La función __hash__ se adapta en consecuencia para que sólo dependa del par (coord_x, coord_y).
    """
    def __eq__(self,other) -> int:
        if type(other) is type(self):
            return ((self.coord_x==other.coord_x) and (self.coord_y==other.coord_y))
        else:
            return False

    def __hash__(self) -> int:
        return hash((self.coord_x,self.coord_y))

    def get_calles(self, cruces: pd.DataFrame):
        return cruces[cruces["coordenadas"] == (self.coord_x, self.coord_y)]["Codigo de vía tratado"].unique()

```

- OBJETO “CALLE”

El objeto “Calle” se inicia asignando a la calle su ID, definiendo un DataFrame de sus direcciones asociadas (es decir, el DataFrame de direcciones filtrado de tal forma que solo muestre direcciones pertenecientes a esa calle) y una lista de las coordenadas de todos los cruces que forman parte de la calle. Para obtener estos últimos datos, se hizo uso de la función “get_data”.



```
class Calle:
    def __init__(self, ID):
        self.ID = ID
        self.direcciones = self.get_data(df_cruces, df_direcc)[1]
        self.cruces = self.get_data(df_cruces, df_direcc)[0]

    def get_data(self, cruces: pd.DataFrame, direcciones: pd.DataFrame)-> tuple:
        return (cruces[cruces["Codigo de via tratado"] == self.ID]["coordenadas"].unique(), direcciones[direcciones["Codigo de via"] == self.ID
```

Después, se define la función get_velocidad, la cual busca cuál es la clase de la vía y mediante ello, haciendo uso de las constantes definidas al inicio, devuelve cuál es la velocidad máxima permitida en esa calle. Por último, se define la función que ordena los cruces.

- ordenar_cruces

La idea principal del algoritmo es la siguiente: para cada cruce perteneciente a la calle, se calcula la distancia euclídea (mediante una función definida como “dist” mostrada posteriormente que calcula la distancia euclídea entre dos puntos) de las coordenadas del cruce con las coordenadas de todas las direcciones de la calle. Esto se guarda en un diccionario llamado “distancias” que toma como llave la distancia entre el cruce y la casa en cuestión en centímetros, y como valor el número de la casa. Por ejemplo: distancias = {2001.52: 1, 360.2: 2, 1005: 3, 2670: 4}. Después, se queda con el número de la casa que esté más cerca del cruce y comprueba si es impar, si no es impar, se toma la siguiente distancia más cercana hasta que lo sea, y asocia el número de la casa y el cruce en un diccionario llamado cruces_ordenados, que tiene formato de {numero_casa : cruce}. El por qué de que solo se busquen números impares es para asegurar que los cruces tienen asociados números de casas que estén en la misma acera y, por ende, se pueda asegurar que, por ejemplo, el número 11 está más avanzado en la calle que el número 5.

Una vez está completo del diccionario de cruces ordenados, con todos los cruces de la calle asociados a un número de casa, se ordena este diccionario de menor a mayor, y se guarda la lista de los valores, es decir, la lista de los cruces. Esta lista estará ordenada ya que cada cruce de la lista aparece en orden según qué edificio de la calle sea el más cercano al cruce, por tanto, los cruces están ordenados correctamente.

La programación de este algoritmo se ha complicado por dos motivos:

- EMPATES: es decir, cuando dos o más cruces tienen como edificio más cercano el mismo edificio, y hay que definir un orden entre estos cruces.
  Para el caso de los empates dobles, esto se ha conseguido comparando la distancia de ambos cruces con el edificio que se encuentra dos números por atrás (de número son dos números por detrás y no uno para que este esté en la misma acera). Al compararse la distancia con un edificio anterior de la calle, uno estará más cerca y el otro más lejos, por lo que se supone que el cruce más cercano al número anterior va primero, y el cruce más lejano va después. Este caso se ha programado exactamente igual, pero a la inversa (comparando con el edificio de dos números después) para el caso de que el empate suceda en el número 1 de la calle, ya que no existe el número -1 en una calle y por tanto daría error. Así, el que esté más lejos será el primero y el que esté más cerca será el siguiente. Si da la casualidad de que el edificio situado dos números después no existe, o no está registrado, se compara con el edificio que está otros dos números después, y así sucesivamente. Si esto entra en un bucle infinito, se capa la recurrencia cuando el número supera 9999.
  Para los casos de empates triples este algoritmo no es 100% preciso ya que se comparan los dos primeros y luego el tercero con el segundo, pero no el tercero con el primero. De todas formas, los empates triples son estadísticamente insignificantes por

lo que se decidió no complicar el código todo lo que lo complicaría para un caso prácticamente inexistente.

- DIRECCIONES “KM”: Las direcciones con el prefijo KM tienen un sistema de numeración completamente diferente, que hace que este algoritmo no ordene bien los cruces. Como este algoritmo solo ordena las calles que tienen cruces, todas las calles cono cruces que tienen direcciones de KM también tienen direcciones con números corrientes, y la mayoría de los cruces pertenecen a esta sección de la calle, menos la Av. De La Paz, que tiene cruces y una única dirección, en KM.
  La solución que se ha tomado ha sido simplemente ignorar las direcciones KM, y comparar las distancias de cada cruce con los números de la calle que no siguen este formato de dirección. Para la Av. De La Paz, al solo existir una dirección, se han dejado los cruces tal y como estaban, ya que no existen datos de referencia para comparar el orden que deben seguir estos cruces y solo supone una calle de potencial error dentro de todo el callejero de Madrid.

- ### filtrar_por_radios

La función de unificar las rotondas en un único cruce, llamada “filtrar_por_radios” se apoya en otras dos funciones llamadas “dist” y “closest”.

```python
def filtrar_por_radios(R: int):
    coordenadas_a_tratar = sorted(df_cruces["coordenadas"].unique())
    coordenadas_limpias = []
    for coordenada in coordenadas_a_tratar:
        for x, y in coordenadas_limpias:
            if dist(coordenada, (x, y)) <= R:
                break
        else:
            coordenadas_limpias.append(coordenada)

    df_cruces["coordenadas"] = df_cruces["coordenadas"].apply(lambda x: x if x in coordenadas_limpias else closest(x, coordenadas_limpias, R))
    return coordenadas_limpias

def dist(coordenada1, coordenada2):
    return sqrt((coordenada1[0] - coordenada2[0])**2 + (coordenada1[1] - coordenada2[1])**2)

def closest(coordenada, coordenadas_limpias, R):
    coor_x, coor_y = coordenada[0], coordenada[1]
    for x, y in coordenadas_limpias:
        if (coor_x - R <= x <= coor_x + R) and (coor_y - R <= y <= coor_y + R):
            return (x, y)
```

Esta función genera una lista de coordenadas a tratar, con todas las coordenadas de todos los cruces, y una lista vacía de coordenadas limpias. Para cada coordenada a tratar, comprueba si la coordenada está dentro de un radio R de alguna coordenada ya existente mediante la función “dist”, que calcula la distancia euclídea entre dos puntos. Si es así, no la añade a la lista de coordenadas limpias, y si no es así, la añade como una nueva coordenada.

Una vez generada la lista de coordenadas limpias, para cada fila del DataFrame de cruces se comprueba si las coordenadas de cada cruce están en la lista de coordenadas limpias o no. Si están, se dejan intactas, y si no están, se busca mediante la función “closest” cuáles son las coordenadas de la lista de coordenadas limpias que tienen a menos de R distancia euclídea, y se le asigna esas nuevas coordenadas al cruce. Debido a que dos cruces se consideran iguales (__eq__) si tienen las mismas coordenadas, estos cruces se convierten por consiguiente en el mismo cruce.

- ### generar_grafos

Función que toma todas las funciones definidas anteriormente para generar ambos grafos, uno el cual el peso de sus aristas sea la distancia entre los cruces, y otro el cual el peso sea el tiempo que se tarda en recorrer esa distancia a la velocidad máxima. Esta última parte (la del peso de las aristas) corresponde al apartado siguiente de la práctica, pero se ha unificado con

esta función para compactar toda la creación de los grafos en una sola función, siendo además que los grafos producidos son iguales teniendo o no teniendo peso las aristas.

1. Se filtra el DataFrame de cruces por radio, como ha sido explicado anteriormente. El radio escogido finalmente para unificar las rotondas ha sido de 20 metros, ya que unifica la mayor parte de las rotondas, y no unifica cruces que no debería unificar.

```
def generar_grafos():
    # 2000 centímetros = 20 metros, se considera que un cruce está dentro del radio de otro si está a
    # menos de 20 metros de distancia según las observaciones
    coordenadas_limpias = filtrar_por_radios(2000)
```

2. Se crea un diccionario de cruces de formato {coordenada: Cruce} para poder acceder rápidamente a la información de los cruces mediante sus coordenadas. Se crea también una lista con todos los objetos cruce.

```
    # Creamos los cruces
    cruces = {}
    for coordenada in coordenadas_limpias:
        cruces[coordenada] = Cruce(coordenada[0], coordenada[1])
    list_cruces:list[Cruce] = list(cruces.values())
```

3. Se crea un diccionario de calles de formato {código de vía: Calle}, y una lista de todos los objetos Calle. Solo se crean objetos de Calle si hay algún cruce que las contenga, ya que no pueden existir aristas si no existen vértices que se unan entre sí.

```
# Creamos las calles
calles_dict = {}
calles = []
for cruce in list_cruces: # Escogemos los cruces
    for calle in cruce.calles: # Escogemos las calles de cada cruce
        if calle not in calles: # Si la calle no está en la lista de calles, la añadimos
            calles.append(calle)
            calles_dict[calle] = Calle(calle)
calles = [Calle(calle) for calle in calles] # Creamos los objetos calle
```

4. Se crean ambos grafos no dirigidos (grafo_d para el grafo cuyos pesos son las distancias y grafo_t para el grafo de tiempos) y se agregan los objetos Cruce como vértice.

```
# Creamos el grafo
grafo_d = Grafo(False)
grafo_t = Grafo(False)

# Añadir vértices al grafo
for cruce in list_cruces:
    grafo_d.agregar_vertice(cruce)
    grafo_t.agregar_vertice(cruce)
```

5. Para cada calle creada, se ordenan los cruces de la calle mediante la función “ordenar_cruces” y se agregan las aristas a cada grafo, con su peso correspondiente, uniendo el primer cruce ordenado con el segundo, el segundo con el tercero, y así sucesivamente.

```python
# Para las aristas, se añaden las calles que conectan dos cruces como aristas del grafo
for calle in calles:
    calle.ordenar_cruces()
    cruces_calle = calle.cruces
    for i in range(len(cruces_calle)):
        if i != len(cruces_calle) - 1:
            coords_actual, coords_siguiente = cruces_calle[i], cruces_calle[i+1]
            cruce_actual, cruce_siguiente = cruces[coords_actual], cruces[coords_siguiente]

            distancia = dist(coords_actual, coords_siguiente)
            tiempo = ((distancia / 100000) / calle.get_velocidad()) * 60 # En minutos

            grafo_d.agregar_arista(cruce_actual, cruce_siguiente, None, distancia)
            grafo_t.agregar_arista(cruce_actual, cruce_siguiente, None, tiempo)
```

6. Se retornan los dos grafos y, para uso futuro, los diccionarios de cruces y calles.

## GPS.PY

Gps.py comienza importando las librerías necesarias y las constantes de velocidades, después toma los dos DataFrames, de cruces y de direcciones, y los procesa. Para este apartado fue necesario añadir una modificación adicional a dgt.py, el cual fue la función “clean_names_dir”, que quita los espacios iniciales y finales sobrantes en las columnas “Clase de la via”, “Nombre de la vía” y “Partícula de la vía” del DataFrame de direcciones. Esta función está añadida también a “process_data”, de tal forma que es un proceso que se lleva a cabo al procesar los datos al inicio.

```python
columnas_a_corregir = ["Clase de la via", "Nombre de la vía", "Partícula de la vía"]

for i in columnas_a_corregir:
    df_direcc[i] = df_direcc[i].apply(lambda x: x.strip()) # Eliminamos espacios en blanco
return df_direcc
```

Después, se define una expresión regular llamada regex_direcciones, la cual interpretará las direcciones introducidas por la terminal y separará sus partes. Esta expresión regular está hecha de tal forma que acepte varios formatos de introducir la dirección. Algunos ejemplos de direcciones que separaría correctamente: “cALLe de LOS ARCOS, NUM4”, “calle arcos 4”, “AUTOVÍA A-1 KM1100”, “avenida DE AMERICA 50B”

La expresión regular es:

```
([A-ZÁÉÍÓÚÜ]+)\s*(DEL|DE LA|DE LOS|DE LAS|DE)?\s+([A-ZÁÉÍÓÚÜ]-[0-9]+|[A-ZÁÉÍÓÚÜ\s]+),?\s+(NUM|KM)?\s?([0-9]+)([A-Z]{0,2})
```

Las direcciones introducidas son todas convertidas a mayúscula (ya que todas las direcciones están registradas en mayúscula) por lo que solo se buscan matches en mayúscula. Esta expresión regular te permite utilizar tildes introduciendo las direcciones, pese a que en el DataFrame ninguna dirección utiliza tildes. Esto se arregla más tarde. La expresión regular te obliga a introducir la clase de la vía, algún espacio (pudiendo ser varios), opcionalmente te permite introducir la partícula de la vía, algún espacio (si se ha introducido partícula), el nombre de la vía (permitiendo este tanto el formato común de nombre como el formato de carreteras, aceptando, por ejemplo, LÓPEZ DE HOYOS y M-30). Después, se introducirá opcionalmente una coma, algún espacio y, opcionalmente, las palabras NUM o KM, seguidas de algún espacio opcional y un número obligatorio de dirección, seguido opcionalmente de un sufijo.

Esta expresión regular será capaz de separar la clase de la vía, la partícula de la vía, su nombre, el prefijo de la dirección, el número y el sufijo.

Se importan las funciones necesarias de callejero.py y se crean las funciones necesarias para este apartado:

- procesar_direcciones

1. Toma como argumento una dirección y, mediante el método “replace”, elimina todas las tildes de la dirección.
2. Haciendo uso de regex_direcciones, toma la clase, el nombre, el número y el sufijo de la calle.
3. Filtra el DataFrame para encontrar la dirección introducida y toma el código de vía de la calle y las coordenadas de la dirección.

```
def procesar_direccciones(direccion: str):
    direccion = direccion.replace("Á", "A").replace("É", "E").replace("Í", "I").replace("Ó", "O").replace("Ú", "U").replace("Ü", "U")
    clase = re.search(regex_direcciones, direccion).group(1)
    nombre = re.search(regex_direcciones, direccion).group(3)
    numero = re.search(regex_direcciones, direccion).group(5)
    sufijo = re.search(regex_direcciones, direccion).group(6)

    df_direcc_filtrado = df_direcc[(df_direcc["Clase de la via"] == clase)]
    df_direcc_filtrado = df_direcc_filtrado[(df_direcc_filtrado["Nombre de la vía"] == nombre)]
    df_direcc_filtrado = df_direcc_filtrado[(df_direcc_filtrado["Número"] == int(numero))]
    if sufijo:
        df_direcc_filtrado = df_direcc_filtrado[(df_direcc_filtrado["Sufijo de numeración"] == sufijo)]

    coordenadas = df_direcc_filtrado["coordenadas"].unique()[0]
    ID_calle = df_direcc_filtrado["Codigo de via"].unique()[0]
```

4. Para cada cruce de la calle, hace la distancia de ese cruce con la dirección, y se queda con la más pequeña, de esta forma consigue el cruce más cercano a la dirección introducida, ya que los caminos solo se pueden encontrar a partir de un vértice.
5. Retorna el cruce más cerano, las coordenadas del vértice, el código de vía de la calle y las coordenadas de la dirección introducida.

```
cruces_calle = calles_dict[ID_calle].cruces

distancias = {}
for cruce in cruces_calle:
    distancias[cruce] = dist(coordenadas, cruce)
distancias = dict(sorted(distancias.items(), key=lambda x: x[1]))
distancias = list(distancias.keys())
cruce_mas_cercano = cruces[distancias[0]]
return cruce_mas_cercano, distancias[0], ID_calle, coordenadas
```

- Funciones auxiliares para la función “instrucciones”

La función de instrucciones requiere de las funciones llamadas “hay_giro”, “calle_actual” y “tipo_de_giro”, “hay_giro_origen_destino” y “calle_origen_destino”.

- hay_giro: toma el cruce actual y el cruce siguiente del cruce siguiente. Si ambos no tienen ninguna calle asociada en común, significa que hay giro después de la arista formada por el cruce actual y el cruce siguiente, ya que, mientras se siga por una misma calle, todos los cruces que se pasen tendrán entre ellos una calle en común (por la que se está pasando). La función retorna True si hay giro, y False si no lo hay.
- hay_giro_origen_destino: la misma función que la anterior, hecha para el caso en que las primeras coordenadas son las de la dirección de origen o las últimas son las de la dirección de destino.

- calle_actual: toma el cruce actual y el siguiente y comprueba cuál es la calle que tienen en común, esa será la calle que actualmente se está recorriendo. Se retorna la clase, la partícula y el nombre de la calle actual.
- calle_origen_destino: devuelve la clase, la partícula y el nombre de la calle de origen o de destino mediante el código de vía.
- tipo_de_giro: toma 3 puntos (el cruce actual, el siguiente y el siguiente del siguiente) y genera dos vectores, uno equivalente a la arista (trozo de calle) actual y otro equivalente a la arista hacia la que se va a girar. Calcula el producto vectorial de estos dos vectores, y, si el resultado es positivo, el giro es a la izquierda. De lo contrario es negativo. Esta técnica es, en esencia, igual que la "regla de la mano derecha".

- instrucciones

Función que imprime en la terminal una lista detallada de instrucciones con el siguiente formato:

Avanza 25 metros por ROS DE OLANO

Gira a la izquierda hacia LOPEZ DE HOYOS

Avanza 500 metros por LOPEZ DE HOYOS

Gira a la derecha por CALLE DEL PRINCIPE DE VERGARA

…

Ha llegado a su destino.

Para ello, itera el camino recibido hasta la penúltima arista (la última la genera aparte).

1. Toma el cruce actual (inicialmente el cruce más cercano al origen), el cruce siguiente y el cruce siguiente del siguiente.
2. Se comprueba si hay giro al final de la arista que se está recorriendo actualmente.
3. Se suma la distancia de la arista.
4. Se repite hasta que haya giro, acumulando la distancia recorrida.
5. Si hay giro, se imprimen las ordenes de recorrer la distancia total acumulada por la calle actual y de giro hacia la calle siguiente. Se reinicia la distancia a 0 para calcular la nueva distancia recorrida en la nueva calle.

```
def hay_giro(cruce_actual:Cruce, cruce_siguiente2:Cruce) -> bool:
    for calle in cruce_actual.calles:
        if calle in cruce_siguiente2.calles:
            return False
    return True

def calle_actual(cruce_actual:Cruce, cruce_siguiente:Cruce, df_cruces=df_cruces):
    for calle in cruce_actual.calles:
        if calle in cruce_siguiente.calles:
            clase = df_cruces[df_cruces["Codigo de vía tratado"] == calle].iloc[0]["Clase de la via tratado"]
            particula = df_cruces[df_cruces["Codigo de vía tratado"] == calle].iloc[0]["Particula de la via tratado"]
            calle = df_cruces[df_cruces["Codigo de vía tratado"] == calle].iloc[0]["Nombre de la via tratado"]
            return clase, particula, calle

def tipo_de_giro(p1, p2, p3):
    v1 = (p2[0]-p1[0], p2[1]-p1[1])
    v2 = (p3[0]-p2[0], p3[1]-p2[1])
    lenv1 = math.sqrt((v1[0]**2 + v1[1]**2))
    lenv2 = math.sqrt((v2[0]**2 + v2[1]**2))

    angulo = math.acos((v1[0]*v2[0] + v1[1]*v2[1])/(lenv1*lenv2))
    if v1[0]*v2[1] - v1[1]*v2[0] < 0:
        angulo = 2 * math.pi - angulo

    if angulo < math.pi:
        giro = "derecha"
    else:
        giro = "izquierda"
    return giro
```

La última arista se genera aparte ya que no existe el cruce siguiente del siguiente, por lo que solo se calcula la distancia entre el cruce penúltimo y el cruce final, el más cercano a la dirección de destino, y se imprime la instrucción de avanzar esa distancia por la calle propia de la arista.

```
cruce_penultimo = camino[-2]
cruce_ultimo = camino[-1]
coords_penultimo = (cruce_penultimo.coord_x, cruce_penultimo.coord_y)
coords_ultimo = (cruce_ultimo.coord_x, cruce_ultimo.coord_y)
distancia += dist(coords_penultimo, coords_ultimo)
clase, particula, calle = calle_actual(cruce_penultimo, cruce_ultimo)

if particula:
    print(f"Avanza {round(distancia)/100} metros por {clase} {particula} {calle}")
else:
    print(f"Avanza {round(distancia)/100} metros por {clase} {calle}")
print("\nHa llegado a su destino.\n")
```

Añadir las instrucciones desde el origen hasta el cruce más cercano al origen, y desde el cruce más cercano al destino al destino se resuelve mediante una iteración aparte al principio del bucle y otra al final, haciendo uso de las funciones especialmente definidas para la ocasión. Por ejemplo, para el caso desde el origen al destino:

```
def instrucciones(camino, coord_o, coord_d, ID_origen, ID_destino):
    cruce_actual = camino[0]
    cruce_siguiente = camino[1]
    coords_actual = (cruce_actual.coord_x, cruce_actual.coord_y)
    coords_siguiente = (cruce_siguiente.coord_x, cruce_siguiente.coord_y)
    distancia = dist(coord_o, coords_actual)
    clase, particula, calle = calle_origen_destino(ID_origen)
    clase_2, particula_2, calle_2 = calle_actual(cruce_actual, cruce_siguiente)

    if hay_giro_origen_destino(cruce_siguiente, ID_origen):
        giro = tipo_de_giro(coord_o, coords_actual, coords_siguiente)
        if particula:
            print(f"Avanza {round(distancia)/100} metros por {clase} {particula} {calle}")
            print(f"Gira a la {giro} hacia {clase_2} {particula_2} {calle_2}")
        else:
            print(f"Avanza {round(distancia)/100} metros por {clase} {calle}")
            print(f"Gira a la {giro} hacia {clase_2} {calle_2}")
```

- mostrar_ruta

Función que dibuja el gráfico con la ruta encontrada pintada de rojo por encima. Toma como argumentos el grafo y el camino mínimo en formato [(cruce1, cruce2), (cruce2, cruce3), …].

1. Convierte el grafo a NetworkX con la función “convertir_a_networkx” de la librería grafo.py
2. Genera el diccionario de posiciones necesario para dibujar el grafo correctamente, donde las llaves son los vértices (objetos Cruce) y los valores son sus coordenadas en el espacio.
3. Dibuja el grafo.
4. Dibuja el camino por encima haciendo uso del camino mínimo.
5. Muestra el grafo.

```python
def mostrar_ruta(grafo:Grafo, camino_min_aristas:list):
    G = grafo.convertir_a_NetworkX()

    pos = {}
    for cruce in grafo.lista_vertices():
        pos[cruce] = (cruce.coord_x, cruce.coord_y)

    nx.draw_networkx_nodes(G, pos=pos, node_size=1)
    nx.draw_networkx_edges(G, pos=pos, width=0.5, edge_color="black")
    nx.draw_networkx_edges(G, pos=pos, arrows=True, edgelist=camino_min_aristas, edge_color='r', width=2.0)
    plt.show()
```

Para generar el camino mínimo de aristas se utiliza la función generar_edgelist, que toma un camino mínimo en el formato generado por la función “camino_minimo” de grafo.py y lo convierte al formato necesario para poder utilizarlo en el argumento edgelist.

```python
def generar_edgelist(camino):
    aristas = []
    for i in range(len(camino)-1):
        aristas.append((camino[i], camino[i+1]))
    return aristas
```

- main

El proceso principal sigue los siguientes pasos:

1. Como los DataFrames y la regex ya se han generado previamente para evitar errores, directamente se generan los grafos mediante la función “generar_grafos”.
2. Se piden las direcciones de origen y de destino. Si una de ambas está vacía, el programa terminará.
3. Se intentan procesar ambas direcciones. Si una de ellas, o ambas, son incorrectas, se imprimirá un mensaje de error y se volverán a pedir.
4. Se comprueba si ambas direcciones son iguales, en caso de que lo sean, se imprime un mensaje de error.
5. Se pide si se busca encontrar la ruta más corta o la ruta más rápida en coche. Si el modo (1 o 2, respectivamente) se introduce incorrectamente, se imprime un mensaje de error y se vuelve a pedir.
6. Se genera el camino mínimo entre el cruce más cercano al origen y el cruce más cercano al destino. Para ello, se utiliza la función de “camino_minimo” definida en grafo.py que hace uso del algoritmo de Dijkstra para generar un árbol abarcador de caminos mínimos a partir de un nodo, y después muestra el camino mínimo entre el nodo inicial y el nodo que se busca en formato [nodo_0, nodo_1, nodo_2, …, nodo_n].
7. Se hace uso de “generar_edgelist” para convertir el camino mínimo generado al formato correcto para pintarlo en el grafo.
8. Se imprimen en la terminal todas las instrucciones del trayecto.
9. Se muestra el grafo indicando la ruta a seguir.
10. Se vuelve al paso 2 para calcular una nueva ruta entre dos direcciones, o para salir del programa si se introduce una o dos direcciones vacías.

# BIBLIOGRAFÍA


- Documentación de pandas: [pandas documentation — pandas 2.1.4 documentation (pydata.org)]
- Transparencias de Matemática Discreta de los Temas 5 y 6.