1 DE DICIEMBRE DE 2023

CALLEJERO Y GPS DE MADRID

PROYECTO FINAL – MATEMÁTICA DISCRETA

IGNACIO FELICES VERA
ULISES DIEZ SANTAOLALLA
2º iMAT A

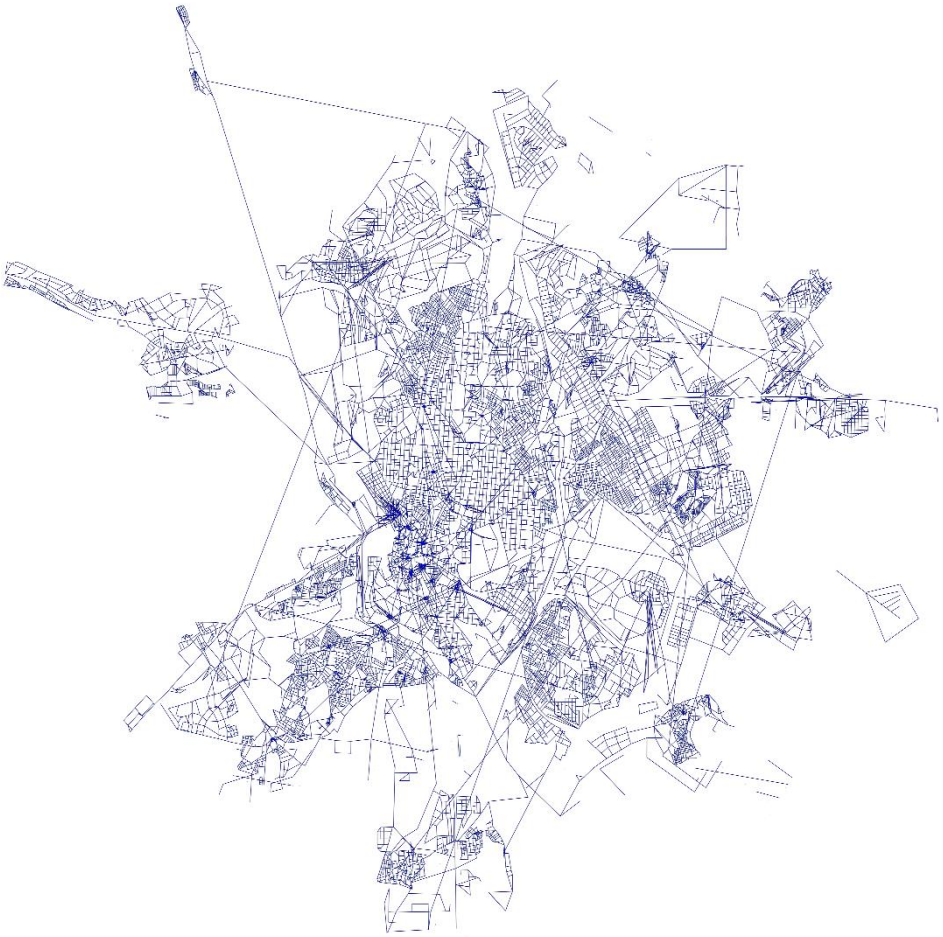
Tabla de contenido

Introducción Proyecto GPS Madrid

El objetivo de este proyecto radica en la creación de un navegador GPS personalizado diseñado específicamente para la urbe de Madrid. Este navegador será capaz de trazar rutas detalladas y proporcionar orientación precisa para los conductores, permitiéndoles navegar sin dificultades desde cualquier ubicación en Madrid hacia su destino deseado.

Para lograr esto, el navegador utilizará información detallada recopilada del callejero oficial de Madrid, accesible a través de la plataforma de datos abiertos del Gobierno en datos.gob.es. Este sistema innovador construirá un complejo mapa de intersecciones y calles, transformándolo en un grafo interactivo para proporcionar rutas claras y precisas entre las diferentes ubicaciones dentro de la ciudad.

La ambición de este proyecto es ofrecer una solución de navegación intuitiva y eficiente, aprovechando los recursos públicos disponibles para mejorar la experiencia de conducción en la densa red de calles y avenidas de Madrid. Este navegador GPS no solo simplificará los desplazamientos, sino que también se convertirá en una herramienta invaluable para quienes deseen explorar la ciudad de manera práctica y sin complicaciones.



Desarrollo del módulo “grafo.py”

Para el desarrollo de este módulo Python se partió del código proporcionado en la plantilla "grafo.py", con el objetivo de llevar a cabo una implementación del tipo abstracto de datos (TAD) grafo, permitiendo la representación tanto de grafos dirigidos como no dirigidos mediante la creación de la clase "Grafo".

La premisa básica es que un grafo se compone de dos conjuntos G=(V,A), donde V representa el conjunto de vértices y A el conjunto de aristas. En esta implementación, se ha considerado que los vértices del grafo pueden ser cualquier objeto proporcionado por el usuario de la librería, siempre y cuando cumpla con el requisito de ser "hasheable", permitiendo su uso como clave en un diccionario.

Para identificar las aristas en el grafo, se utilizarán pares de objetos de V, representando las aristas ya sea como pares ordenados o no, dependiendo del tipo de grafo. Cada arista tendrá la capacidad de almacenar un objeto de datos suministrado por el usuario y un número real (float) que denote el peso o costo de dicha arista en el grafo, lo cuál será necesario en partes más avanzadas del proyecto.

La implementación del TAD grafo comprenderá una serie de funciones, detalladas en la plantilla grafo.py, que permitirán realizar diversas operaciones:

- Inicialización: Crear un grafo dirigido o no dirigido según la indicación del usuario.
- Verificación de si el grafo es dirigido o no.
- Agregar vértices y aristas al grafo.
- Eliminar vértices y aristas específicas, junto con sus relaciones.
- Obtener información sobre una arista existente entre dos vértices dados.
- Recuperar la lista de vértices y sus adyacencias en el grafo.

Estas funciones, en general, retornarán None si la tarea solicitada no puede llevarse a cabo, sin generar excepciones, a excepción de intentar usar un objeto no "hasheable" como vértice, lo cual podría generar una excepción TypeError en Python.

Se empleará una estructura basada en el almacenamiento de listas de adyacencia del grafo, donde además de las funciones fundamentales del TAD, se implementarán métodos para calcular el grado de los vértices tanto en grafos dirigidos como no dirigidos. Estas funciones serán esenciales para analizar propiedades específicas de los vértices dentro de la estructura del grafo.

En primer lugar, se importaron las librerías necesarias para el desarrollo del código:

- from typing import List,Tuple,Dict
- import networkx as nx
- import sys
- import heapq

Funciones Principales de la clase “Grafo”

Método __init__

Función: Función básica que inicializa un nuevo grafo.

Objetivo: Crear una objeto de la clase Grafo, permitiendo indicar si el grafo es dirigido o no, e iniciando la lista de adyacencia del grafo como un diccionario vacío.

```python
def __init__(self,dirigido:bool=False):
    """ Crea un grafo dirigido o no dirigido.

    Args:
        dirigido (bool): Flag que indica si el grafo es dirigido (verdadero) o no (falso).

    Returns:
        Grafo o grafo dirigido (según lo indicado por el flag)
        inicializado sin vértices ni aristas.
    """

    # Flag que indica si el grafo es dirigido o no.
    self.dirigido=dirigido


    """
    Diccionario que almacena la lista de adyacencia del grafo.
    adyacencia[u]:  diccionario cuyas claves son la adyacencia de u
    adyacencia[u][v]:   Contenido de la arista (u,v), es decir, par (a,w) formado
                        por el objeto almacenado en la arista "a" (object) y su peso "w" (float).
    """
    self.adyacencia:Dict[object,Dict[object,Tuple[object,float]]]={}
```

Método hasheable

Función: Devolver el hash del objeto.

Objetivo: Devolver el hash del objeto, comprobando primero si este es hasheable, y de lo contrario, devolviendo la función TypeError.

```python
def hasheable(self,objeto):
    #Esta función intenta devolver el hash del objeto. Si no es posible, lanza una excepción.
    try:
        return hash(objeto)
    #Si no es hasheable, el except recoge el TypeError de la funcion hash e imprime un mensaje, siempre devolviendo un None
    except TypeError:
        print(f'El objeto {objeto} no es hasheable')
        return None
```

Método es_dirigido

Función: Permite identificar si el grafo es dirigido.

Objetivo: Verificar si el grafo es dirigido o no, devolviendo un booleano.

```python
def es_dirigido(self)->bool:
    """ Indica si el grafo es dirigido o no

    Args: None
    Returns: True si el grafo es dirigido, False si no.
    Raises: None
    """
    if self.dirigido == False:
        return False
    else:
        return True
```

Método agregar_vertice

Función: Agrega un vértice al grafo.

Objetivo: Insertar un nuevo vértice v en el grafo, verificando primero si es "hashable". En el caso de que este vértice sí sea hasheable, le añadirá a la lista de adyacencia del grafo.

```
def agregar_vertice(self,v:object)->None:
    """ Agrega el vértice v al grafo.

    Args:
        v (object): vértice que se quiere agregar. Debe ser "hashable".
    Returns: None
    Raises:
        TypeError: Si el objeto no es "hashable".
    """
    #Para cada vertice se crea un diccionario vacio
    if self.hasheable(v):
        self.adyacencia[v] = {}
```

Método agregar_arista

Función: Agrega una arista al grafo.

Objetivo: Añadir una arista entre los vértices s y t, incluyendo información adicional (data) y un peso (weight). Antes de añadir esta arista, incluyéndola en su posición de la lista de adyacencia correspondiente, comprueba si ambos vértices s y t son hasheables y si estos ya pertenecen a la lista de adyacencia. Si es un grafo no dirigido, además las aristas han de implementarse en los dos sentidos.

```
def agregar_arista(self,s:object,t:object,data:object=None,weight:float=1)->None:
    """ Si los objetos s y t son vértices del grafo, agrega
    una arista al grafo que va desde el vértice s hasta el vértice t
    y le asocia los datos "data" y el peso weight.
    En caso contrario, no hace nada.

    Args:
        s (object): vértice de origen (source).
        t (object): vértice de destino (target).
        data (object, opcional): datos de la arista. Por defecto, None.
        weight (float, opcional): peso de la arista. Por defecto, 1.
    Returns: None
    Raises:
        TypeError: Si s o t no son "hashable".
    """

    if self.hasheable(s) and self.hasheable(t):
        if s in self.adyacencia and t in self.adyacencia:
            self.adyacencia[s][t] = (data,weight)

            #Si es un grafo no dirigido, las aristas han de implementarse en los dos sentidos
            if not self.dirigido:
                self.adyacencia[t][s] = (data,weight)
```

Método eliminar_vertice

Función: Elimina un vértice del grafo.

Objetivo: Tras comprobar si el vértice que se quiere eliminar es hasheable o no, si este vértice se encuentra en la lista de adyacencia, es decir, pertenece al grafo, lo elimina de esta lista.

```
def eliminar_vertice(self,v:object)->None:
    """ Si el objeto v es un vértice del grafo lo elimiina.
    Si no, no hace nada.

    Args:
        v (object): vértices que se quiere eliminar.
    Returns: None
    Raises:
        TypeError: Si v no es "hashable".
    """

    if self.hasheable(v):
        if v in self.adyacencia:
            del self.adyacencia[v]

            for u in self.adyacencia:
                if v in self.adyacencia[u]:
                    del self.adyacencia[u][v]
```

Método eliminar_arista

Función: Elimina una arista del grafo.

Objetivo: Borrar la arista entre los vértices s y t del grafo, si ambos vértices t y s son hasheables, y pertenecen a la lista de adyacencia, es decir, pertenecen al grafo. Si el grafo no es dirigido, además habrá que quitar las aristas en los dos sentidos.

```
def eliminar_arista(self,s:object,t:object)->None:
    """ Si los objetos s y t son vértices del grafo y existe
    una arista de s a t la elimina.
    Si no, no hace nada.

    Args:
        s: vértice de origen de la arista (source).
        t: vértice de destino de la arista (target).
    Returns: None
    Raises:
        TypeError: Si s o t no son "hashable".
    """

    if self.hasheable(s) and self.hasheable(t):
        if (s in self.adyacencia) and (t in self.adyacencia) and (t in self.adyacencia[s]):
            del self.adyacencia[s][t]

            #Al ser no dirigido, hay que quitar las aristas en los dos sentidos
            if not self.dirigido:
                del self.adyacencia[t][s]
```

Método obtener_arista

Función: Obtiene información sobre una arista específica en el grafo.

Objetivo: Devuelve la información asociada a la arista que va desde el vértice s al vértice t, buscando su información en la lista de adyacencia del grafo.

```python
def obtener_arista(self,s:object,t:object)->Tuple[object,float] or None:
    """ Si los objetos s y t son vértices del grafo y existe
    una arista de u a v, devuelve sus datos y su peso en una tupla.
    Si no, devuelve None

    Args:
        s: vértice de origen de la arista (source).
        t: vértice de destino de la arista (target).
    Returns:
        Tuple[object,float]: Una tupla (a,w) con los datos "a" de la arista (s,t) y su peso
            "w" si la arista existe.
        None: Si la arista (s,t) no existe en el grafo.
    Raises:
        TypeError: Si s o t no son "hashable".
    """

    if self.hasheable(s) and self.hasheable(t):
        if (s in self.adyacencia) and (t in self.adyacencia[s]):
            return self.adyacencia[s][t]
        else:
            return None
```

Método lista_vertices

Función: Devuelve una lista de los vértices del grafo.

Objetivo: Obtiene una lista con todos los vértices presentes en el grafo a través de la lista de adyacencia, y la devuelve.

```python
def lista_vertices(self)->List[object]:
    """ Devuelve una lista con los vértices del grafo.

    Args: None
    Returns:
        List[object]: Una lista [v1,v2,...,vn] de los vértices del grafo.
    Raises: None
    """

    #list te coge los keys de forma automatica
    return list(self.adyacencia)
```

Método lista_adyacencia

Función: Devuelve la lista de adyacencia de un vértice en el grafo.

Objetivo: Devuelve una lista de vértices adyacentes al vértice u en el grafo, si existe (pertenece a la lista de adyacencia, y por lo tanto, al grafo), y si es hasheable.

```python
def lista_adyacencia(self,u:object)->List[object] or None:
    """ Si el objeto u es un vértice del grafo, devuelve
    su lista de adyacencia, es decir, una lista [v1,...,vn] con los vértices
    tales que (u,v1), (u,v2),..., (u,vn) son aristas del grafo.
    Si no, devuelve None.

    Args: u vértice del grafo
    Returns:
        List[object]: Una lista [v1,v2,...,vn] de los vértices del grafo
            adyacentes a u si u es un vértice del grafo
        None: si u no es un vértice del grafo
    Raises:
        TypeError: Si u no es "hashable".
    """

    if self.hasheable(u) and (u in self.adyacencia):
        lista_adyaciencia = []
        for vertice_adyaciente in self.adyacencia[u].keys():
            lista_adyaciencia.append(vertice_adyaciente)
        return lista_adyaciencia
    else:
        return None
```

Propiedades de los Vértices

Método grado_saliente

Función: Devuelve el grado saliente de un vértice.

Objetivo: Obtiene el número de aristas que parten del vértice v, si este vértice es hasheable y si pertenece a la lista de adyacencia del grafo.

```python
def grado_saliente(self,v:object)-> int or None:
    """ Si el objeto v es un vértice del grafo, devuelve
    su grado saliente, es decir, el número de aristas que parten de v.
    Si no, devuelve None.

    Args:
        v (object): vértice del grafo
    Returns:
        int: El grado saliente de u si el vértice existe
        None: Si el vértice no existe.
    Raises:
        TypeError: Si u no es "hashable".
    """

    if self.hasheable(v) and (v in self.adyacencia):
        return len(self.adyacencia[v].keys())
    else:
        return None
```

Método grado_entrante

Función: Devuelve el grado entrante de un vértice.

Objetivo: Calcula el número de aristas que llegan al vértice v, si este es hasheable y pertenece a la lista de adyacencia.

```python
def grado_entrante(self,v:object)->int or None:
    """ Si el objeto v es un vértice del grafo, devuelve
    su grado entrante, es decir, el número de aristas que llegan a v.
    Si no, devuelve None.

    Args:
        v (object): vértice del grafo
    Returns:
        int: El grado entrante de u si el vértice existe
        None: Si el vértice no existe.
    Raises:
        TypeError: Si v no es "hashable".
    """

    if self.hasheable(v) and (v in self.adyacencia):
        contador = 0

        for vertice in self.adyacencia.keys():
            if vertice != v:
                for vertices_adyacientes in self.adyacencia[vertice].keys():
                    if vertices_adyacientes == v:
                        contador += 1
        return contador

    else:
        return None
```

Método grado

Función: Devuelve el grado total (o saliente si es dirigido) de un vértice.

Objetivo: Devuelve el número total de aristas incidentes en el vértice v, si este es hasheable y pertenece a la lista de adyacencia, ya que para los dos tipos de grafos, el grado devuelve el equivalente al saliente.

```python
def grado(self,v:object)->int or None:
    """ Si el objeto v es un vértice del grafo, devuelve
    su grado si el grafo no es dirigido y su grado saliente si
    es dirigido.
    Si no pertenece al grafo, devuelve None.

    Args:
        v (object): vértice del grafo
    Returns:
        int: El grado grado o grado saliente de u según corresponda
            si el vértice existe
        None: Si el vértice no existe.
    Raises:
        TypeError: Si v no es "hashable".
    """
    #Ya que para los dos tipos de grafos, el grado devuelve el equivalente al saliente
    if self.hasheable(v) and (v in self.adyacencia):
        return self.grado_saliente(v)
    else:
        return None
```

Algoritmos de Recorrido de Grafos

Método dijkstra

Partiendo del Teorema de Dijkstra, el cual enuncia que “Sea G = (V, A, w) un grafo pesado conexo tal que w(a) > 0 para todo a ∈ A. Entonces para todo o ∈ V, el algoritmo de Dijkstra encuentra el árbol de caminos mínimos desde o al resto de los vértices de G.”, se elaboró el siguiente algoritmo de recorrido de Grafo, usando la librería heapq para poder usar la estructura de heap, y optimizar este recorrido de grafo, basándonos en el pseudocódigo proporcionado en las transparencias de la asignatura:

```python
def dijkstra(self, origen: object) -> Dict[object, object]:
    """ Calcula un Árbol de Caminos Mínimos para el grafo partiendo
    del vértice "origen" usando el algoritmo de Dijkstra. Calcula únicamente
    el árbol de la componente conexa que contiene a "origen".

    Args:
        origen (object): vértice del grafo de origen
    Returns:
        Dict[object,object]: Devuelve un diccionario que indica, para cada vértice alcanzable
            desde "origen", qué vértice es su padre en el árbol de caminos mínimos.
    Raises:
        TypeError: Si origen no es "hashable".
    Example:
        Si G.dijksra(1)={2:1, 3:2, 4:1} entonces 1 es padre de 2 y de 4 y 2 es padre de 3.
        En particular, un camino mínimo desde 1 hasta 3 sería 1->2->3.
    """
    if self.hasheable(origen):
        # Inicializa las distancias y padres de los vértices
        distancias = {vertice: float('inf') for vertice in self.adyacencia}
        padres = {vertice: None for vertice in self.adyacencia}
        # La distancia desde el origen hasta él mismo es 0
        distancias[origen] = 0

        # Inicializa el heap con la tupla (distancia, vértice)
        heap = [(0, origen)]
        # Conjunto de vértices no visitados
        vertices_restantes = set(self.adyacencia.keys())

        # Mientras haya vértices no visitados
        while vertices_restantes:
            # Extrae el vértice con la menor distancia actual desde el heap
            distancia_actual, vertice_actual = heapq.heappop(heap)
            if vertice_actual in vertices_restantes:
                vertices_restantes.remove(vertice_actual)

            # Itera sobre los vecinos del vértice actual
            for vecino, peso in self.adyacencia[vertice_actual].items():
                # Calcula la nueva distancia desde el origen hasta el vecino
                nueva_distancia = distancias[vertice_actual] + peso[1]
                # Si la nueva distancia es menor que la almacenada, actualiza la distancia y el padre
                if nueva_distancia < distancias[vecino]:
                    distancias[vecino] = nueva_distancia
                    padres[vecino] = vertice_actual
                    # Agrega el vecino al heap con su nueva distancia
                    heapq.heappush(heap, (nueva_distancia, vecino))

        #Como no nos interesa que salge el vertice None lo borramos
        for nodo in list(padres.keys()):
            if padres[nodo] == None:
                del padres[nodo]

        return padres
```

Método camino_minimo

Partiendo de la siguiente idea: “Sea v un vértice de un grafo pesado G = (V, A, w). Un árbol de caminos mínimos para G partiendo del vértice o ∈ V es un árbol abarcador T tal que para todo v ∈ V el único camino simple existente en T desde o hasta v es el que determina la distancia mínima entre o y v. El árbol T permite calcular la distancia de o a todos los demás vértices: dT(o, v) = dG(o, v) ∀v ∈ V” , se elaboró el siguiente código que permite el cálculo del camino mínimo desde el vértice origen hasta el vértice destino utilizando el algoritmo de Dijkstra.

```python
def camino_minimo(self,origen:object,destino:object)->List[object]:
    """ Calcula el camino mínimo desde el vértice origen hasta el vértice
    destino utilizando el algoritmo de Dijkstra.

    Args:
        origen (object): vértice del grafo de origen
        destino (object): vértice del grafo de destino
    Returns:
        List[object]: Devuelve una lista con los vértices del grafo por los que pasa
            el camino más corto entre el origen y el destino. El primer elemento de
            la lista es origen y el último destino.
    Example:
        Si G.dijksra(1,4)=[1,5,2,4] entonces el camino más corto en G entre 1 y 4 es 1->5->2->4.
    Raises:
        TypeError: Si origen o destino no son "hashable".
    """
    if self.hasheable(origen) and self.hasheable(destino):
        diccionario = self.dijkstra(origen)

        # Garantiza que el destino se encuentre en el grafo
        if diccionario[destino] is None:
            return []

        camino_correcto = []
        #Generamos el camino desde el destino hasta el origen mientras el destino no sea igual al origen
        while destino is not origen:
            camino_correcto.append(destino)
            destino = diccionario[destino]
        camino_correcto.append(destino)

        #Aplicamos un reverse ya que queremos el camino desde el origen al destino
        return list(reversed(camino_correcto))
```

Método prim

Partiendo del Teorema de Prim, el cual enuncia que “Sea G = (V, A, w) un grafo pesado conexo. Entonces el algoritmo de Prim encuentra un árbol abarcador mínimo para G”, se elaboró el siguiente algoritmo de recorrido de Grafo, basándonos en el pseudocódigo proporcionado en las transparencias de la asignatura:

```python
def prim(self):
    """ Calcula un Árbol Abarcador Mínimo para el grafo
    usando el algoritmo de Prim.

    Args: None
    Returns:
        Dict[object,object]: Devuelve un diccionario que indica, para cada vértice del
            grafo, qué vértice es su padre en el árbol abarcador mínimo.
    Raises: None
    Example:
        Si G.prim()={2:1, 3:2, 4:1} entonces en un árbol abarcador mínimo tenemos que:
            1 es padre de 2 y de 4
            2 es padre de 3
    """
    #creamos el diccionario w
    w ={}
    for vertice in self.lista_vertices():
        for v2, (_,peso) in self.adyacencia[vertice].items():
            w[(vertice,v2)] = peso

    # inicializamos las listas de padres y costes mínmos de aristas de cada vértice
    padre = {}
    costo_minimo = {}
    Q = []

    for v in self.lista_vertices():
        padre[v] = None
        costo_minimo[v] = INFTY
        heapq.heappush(Q, (costo_minimo[v], v))

    while Q:
        _, v = heapq.heappop(Q)

        #La interseccion la hago iterando por la lista de adyacentes a v y si tambien se encuentra en Q la añado a la lista
        lista_validos = []
        for vertice in self.adyacencia[v]:
            #Importante ponerlo como ti fuera una tupla ya que en Q estan guardado en tuplas
            if (costo_minimo[vertice],vertice) in Q:
                lista_validos.append(vertice)

        for x in lista_validos:
            if w[(v, x)] < costo_minimo[x]:
                costo_minimo[x] = w[(v, x)]
                padre[x] = v

                #Actualizamos Q con el nuevo coste mínimo
                heapq.heappush(Q, (costo_minimo[x], x))

    #Como no nos interesa que salge el vertice None lo borramos
    for nodo in list(padre.keys()):
        if padre[nodo] == None:
            del padre[nodo]

    return padre
```

Método kruskal

Partiendo del Teorema de Kruskal, el cual enuncia que “Sea G = (V, A, w) un grafo pesado conexo. Entonces el algoritmo de Kruskal encuentra un árbol abarcador mínimo para G.”, se elaboró el siguiente algoritmo de recorrido de Grafo, basándonos en el pseudocódigo proporcionado en las transparencias de la asignatura:

```python
def kruskal(self)-> List[Tuple[object,object]]:
    """ Calcula un Árbol Abarcador Mínimo para el grafo
    usando el algoritmo de Kruskal.

    Args: None
    Returns:
        List[Tuple[object,object]]: Devuelve una lista [(s1,t1),(s2,t2),...,(sn,tn)]
            de los pares de vértices del grafo que forman las aristas
            del arbol abarcador mínimo.
    Raises: None
    Example:
        En el ejemplo anterior en que G.kruskal()={2:1, 3:2, 4:1} podríamos tener, por ejemplo,
        G.prim=[(1,2),(1,4),(3,2)]
    """
    #Crear lista de aristas L ordenada por su peso c

    lista_aristas = []
    vertices_visitados = []
    for vertice in self.lista_vertices():
        vertices_adyacientes = self.adyacencia[vertice].keys()
        vertices_visitados.append(vertice)
        for vertice_adyaciente in vertices_adyacientes:
            if vertice_adyaciente not in vertices_visitados:
                _, peso = self.adyacencia[vertice][vertice_adyaciente]
                lista_aristas.append((peso,(vertice,vertice_adyaciente)))

    lista_aristas = sorted(lista_aristas)

    #lista: aristas aam
    aristas_aam = []

    #Diccionario
    diccionario = {}
    for vertice in self.lista_vertices():
        diccionario[vertice]={vertice}

    #Recorremos la lista de aristas hasta que este vacia
    while len(lista_aristas) > 0:
        arista = lista_aristas[0][1]
        lista_aristas.remove(lista_aristas[0])

        if diccionario[arista[0]] != diccionario[arista[1]]:
            aristas_aam.append(arista)

            #Actualizamos el diccionario y hacemos la union
            diccionario[arista[0]] = diccionario[arista[0]].union(diccionario[arista[1]])

            for vertice in diccionario[arista[0]]:
                diccionario[vertice] = diccionario[arista[0]]


    return aristas_aam
```

Reconstrucción del Grafo (NetworkX)

Método convertir_a_NetworkX

Usando la librería NetworkX, la cual nos permite el trabajo con grafos de una manera óptima y profesión, se creó esta función, la cual crea un grafo de NetworkX partiendo de la clase Grafo que se ha creado a lo largo de esta sección, diferenciando si el grafo es dirigido o no. Gracias a esta función, el grafo creado con esta clase se puede visualizar.

```python
def convertir_a_NetworkX(self)-> nx.Graph or nx.DiGraph:
    """ Construye un grafo o digrafo de Networkx según corresponda
    a partir de los datos del grafo actual.

    Args: None
    Returns:
        networkx.Graph: Objeto Graph de NetworkX si el grafo es no dirigido.
        networkx.DiGraph: Objeto DiGraph si el grafo es dirigido.
        En ambos casos, los vértices y las aristas son los contenidos en el grafo dado.
    Raises: None
    """
    #Crear un grafo de NetworkX

    if self.es_dirigido():
        grafo = nx.DiGraph()        #Grafo es dirigido
    else:
        grafo = nx.Graph()

    #Agregar vertices al grafo de NetworkX

    lista_vertices = self.lista_vertices()
    grafo.add_nodes_from(lista_vertices)
    lista_aristas = list()

    for v in lista_vertices:
        lista_vecinos = self.lista_adyacencia(v)
        for vecino in lista_vecinos:
            obj,peso = self.obtener_arista(v,vecino)
            dict = {'object':obj, 'weight':peso}
            lista_aristas.append((v,vecino,dict))
    grafo.add_edges_from(lista_aristas)

    return grafo
```

Cada función dentro de esta clase Grafo está diseñada para realizar operaciones específicas en la estructura de datos que representa al grafo, como agregar o eliminar vértices y aristas, obtener información sobre estos elementos, y explorar las conexiones entre los vértices, incluyendo recorrer el grafo con diferentes métodos, y su visualización

Esto permite un análisis exhaustivo y manipulación flexible de la estructura del grafo para su aplicación en diferentes contextos y algoritmos de grafos.

Desarrollo del módulo “callejero.py”

Para llevar a cabo la reconstrucción del grafo del callejero de Madrid, se accedió a la web de datos del Gobierno y se descargaron dos conjuntos de datos en formato CSV:

- “cruces.csv”: Contiene información sobre los cruces existentes entre las calles del callejero de Madrid.
- “direcciones.csv”: Contiene las direcciones postales vigentes con coordenadas geográficas correspondientes a dichas calles.

El objetivo es construir un grafo G=(V,A) con ciertas propiedades:

- Naturaleza del Grafo: Se considerará un grafo no dirigido, asumiendo que todas las calles son de doble sentido.
- Vértices del Grafo: Se corresponden con los cruces de las calles. Cada vértice se generará a partir de las coordenadas geográficas (x, y) donde exista un cruce.
- Aristas del Grafo: Unirán dos cruces consecutivos de cada calle.

Cada arista del grafo almacenará los datos de la calle correspondiente, mientras que cada vértice guardará la información de la intersección necesaria para la realización del navegador.

Se siguieron los siguientes pasos para la construcción del grafo:

1. Preparación de los datasets: Función que limpia y prepara los datasets "direcciones" y "cruces" para su uso.
2. Clase "Cruces": Información necesaria para reconstruir el grafo con la información de los datasets descargados.
3. Clase "Calle": Información esencial para reconstruir el grafo y facilitar la navegación.
4. Agrupación de Cruces cercanos: Función que, dado un radio R, detecta grupos de puntos cercanos entre sí y los unifica en un único punto en su centro, eliminando duplicados.
5. Recopilación de códigos de calle únicos: Se obtendrá una lista sin repetidos de todos los códigos de calle del dataset "Cruces" y "Direcciones" para crear un objeto "Calle" para cada código.
6. Creación de objetos "Cruce": Para cada coordenada (x, y) de cruces encontrada, se construirá un objeto "Cruce" con la información correspondiente.
7. Adición de Vértices al Grafo: Cada objeto "Cruce" creado se agregará como vértice a un grafo.
8. Diseño de algoritmo para conectar cruces: Se diseñará un algoritmo que decida qué cruces deben estar conectados por aristas en el grafo del callejero. Estas aristas contendrán información necesaria tanto para la reconstrucción del grafo como para la navegación.

Se busca obtener una aproximación razonable del callejero real, considerando que el conjunto de datos de cruces podría no contener la totalidad de las calles de la Comunidad de Madrid y que la geometría de algunas calles podría ser compleja de deducir exclusivamente a partir de este dataset.

En primer lugar, se importaron las librerías necesarias para el desarrollo del código:

- from dgt_main import *
- from grafo import *
- import math
- import matplotlib.pyplot as plt

En este caso el archivo Python “dgt_main.py” fue el elaborado en la asignatura Adquisición de Datos, donde se procesaron los datasets descargados de la página del gobierno, eliminando posibles errores, y preparándolos para su correcta utilización en esta práctica. Todo esto se lleva a cabo en la función “datasets()”.

Constantes y Funciones

- VELOCIDADES_CALLES: Un diccionario que almacena las velocidades máximas permitidas para diferentes tipos de vías.
- VELOCIDAD_CALLES_ESTANDAR: Constante que representa una velocidad estándar para calles estándar.

```
#Constantes con las velocidades máximas establecidas por el enunciado para cada tipo de vía.
VELOCIDADES_CALLES={"AUTOVIA":100,"AVENIDA":90,"CARRETERA":70,"CALLEJON":30,"CAMINO":30,"ESTACION DE METRO":20,"PASADIZO":20,"PLAZUELA":20,"COLONIA":20}
VELOCIDAD_CALLES_ESTANDAR=50
```

Función datasets

Función: Esta función se encarga de cargar y procesar los conjuntos de datos provenientes de los archivos "CRUCES.csv" y "DIRECCIONES.csv".

Objetivo: Devuelve dos conjuntos de datos procesados y listos para su uso posterior en la construcción del callejero. Como se ha mencionado anteriormente, esta función pertenece al archivo Python “dgt_main.py” desarrollado en la asignatura Adquisición de Datos

```
def datasets():
    path1 = 'CRUCES.csv'
    path2 = 'DIRECCIONES.csv'
    dfcruces_processed, dfdirecciones_processed = process_data(path1,path2)

    return dfcruces_processed, dfdirecciones_processed
```

Clase Cruce

Método __init__

Función: Inicializa un objeto de tipo Cruce con coordenadas coord_x y coord_y, y una lista de calles.

Objetivo: Crear un objeto cruce en el callejero, guardando en el objeto las coordenadas x, coordenadas y, y las calles asociadas al cruce.

```
def __init__(self,coord_x,coord_y,calles):
    self.coord_x=coord_x
    self.coord_y=coord_y
    self.calles = calles
```

Método __eq__ y __hash__

Función: Estos métodos permiten la comparación y el hashing de objetos Cruce basados en sus coordenadas.

Objetivo: Facilitar la identificación y la distinción de cruces en el callejero, considerando sus coordenadas como criterio de igualdad.

```
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))
```

Función distancia_entre_puntos

Función: Calcula la distancia euclidiana entre dos puntos en un plano cartesiano.

Objetivo: Proporcionar un cálculo de distancia entre dos puntos.

```
def distancia_entre_puntos(p1, p2):
    import numpy as np
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
```

Función unificar

Función: Agrupa las coordenadas de cruces cercanos basándose en un radio determinado.

Objetivo: Identificar y agrupar coordenadas de cruces que están dentro de un radio especificado, para representar puntos que pueden considerarse como una sola intersección en el callejero. Se utiliza el método apply de una función lambda para crear una columna con el nombre “Coordenadas X, Y”, donde se juntan las coordenadas que venían separadas en el dataframe procesado, y se guardan en una tupla. Después, se eliminan las coordenadas que están repetidas y se ordenan, para a continuación poder comparar las coordenadas según el radio que se ha recibido en la función. De esta manera, a través de un bucle, se va iterando por las diferentes coordenadas, comparando gracias a la función distancia_entre_puntos, si la coordenada siguiente es igual a la anterior más el radio.

Una vez se han recorrido todos los cruces, añadiendo a una lista estos cruces que están dentro del radio especificado, y se devuelve la lista.

```python
def unificar(r,dfcruces_processed):
    dfcruces_processed['Coordenadas X, Y'] = dfcruces_processed.apply(lambda row: (row['Coordenada X (Guia Urbana) cm (cruce)'], row['Coordenada Y (Guia Urbana) cm (cruce)']), axis=1)
    coordenadas = dfcruces_processed['Coordenadas X, Y'].unique()
    #clave lista ordenada para iterar rapido
    sorted_coordinates = sorted(coordenadas)
    grupos = []

    i = 0
    while i < len(sorted_coordinates) - 1:
        grupo = [sorted_coordinates[i]]
        j = i + 1
        nuevo_contador = False
        #En este bucle, al tener la lista ordenada itera las siguientes tuplas para observar si es menor que el radio
        #nuevo indice donde empezar, el primero que se salga
        #los que si que estan los quitas
        # iterar para (x-coordinate) + r
        while j < len(sorted_coordinates) and (j <= i + r):
            if nuevo_contador == False and distancia_entre_puntos(sorted_coordinates[i], sorted_coordinates[j]) > r:
                i = j
                nuevo_contador = True
            elif distancia_entre_puntos(sorted_coordinates[i], sorted_coordinates[j]) < r:
                sorted_coordinates.remove(sorted_coordinates[j])
                j -= 1

            j += 1

        grupos.append(grupo)

    return grupos
```

Función cruces

Función: Recopila información sobre los cruces del callejero a partir de un conjunto de datos procesados.

Objetivo: Construir una lista de objetos Cruce que representen los cruces del callejero, con información sobre las calles asociadas a cada cruce. Se utiliza el método apply de una función lambda para crear una columna con el nombre “Coordenadas X, Y”, donde se juntan las coordenadas que venían separadas en el dataframe procesado, y se guardan en una tupla.

Después, se eliminan las coordenadas que están repetidas a través del método unique, y se itera por toda esta lista de cruces, para obtener la lista de calles de cada cruce, evitando coger duplicadas. Esto se logrará gracias a aplicar un set a la lista de calles, y luego convirtiendo este set en una lista para evitar las calles duplicadas.

Después de esto, se crea un objeto Cruce con la coordenada x, la coordenada y, y una lista de calles del cruce. Por lo tanto, esta función nos devuelve una lista con los objetos Cruces del dataset.

```python
def cruces(dfcruces_processed):

    #Cogemos las calles principales que son cruzadas
    dfcruces_processed['Coordenadas X, Y'] = dfcruces_processed.apply(lambda row: (row['Coordenada X (Guia Urbana) cm (cruce)'], row['Coordenada Y (Guia Urbana) cm (cruce)']), axis=1)
    cruces_unicos = dfcruces_processed['Coordenadas X, Y'].unique()

    cruces_maps = []
    for cruce in cruces_unicos:

        filtered_rows = dfcruces_processed[dfcruces_processed['Coordenadas X, Y'] == cruce]
        lista_calles = filtered_rows[['Codigo de vía que cruza o enlaza', 'Codigo de vía tratado']].values.flatten()
        calles = list(set(lista_calles))
        #meto las cordenadas-x,cordenadas-y, y una lista de calles del cruce (sus codigos)
        cruce = Cruce(cruce[0], cruce[1], calles)
        cruces_maps.append(cruce)

    return cruces_maps
```

Clase Calle

Método __init__

Función: Inicializa un objeto de tipo Calle con un codigo_calle, una lista de cruces, y la información de direcciones.

Objetivo: Representar una calle en el callejero, almacenando su identificador único, los cruces asociados, el número de la calle, un diccionario de números de la calle, y el tipo de vía.

```
class Calle:
    #Completar esta clase con los datos que sea necesario almacenar de cada calle para poder reconstruir los datos del
    def __init__(self,codigo_calle,cruces,numero,diccionario_numeros,tipo_de_via) -> None:
        self.codigo_calle=codigo_calle
        self.cruces=cruces
        self.numero = numero
        self.diccionario_numeros = diccionario_numeros
        self.tipo_de_via = tipo_de_via
```

Función calles

Función: Recopila información sobre las calles del callejero a partir de los conjuntos de datos procesados.

Objetivo: Crear una lista de objetos Calle que representen las calles del callejero, incluyendo la información sobre los cruces y direcciones asociadas a cada calle. Para esto, se comenzará creando dos listas a partir del dataframe recibido, donde se guardan 'Codigo de vía tratado' y 'Codigo de via que cruza o enlaza', juntándose las dos en una única lista y aplicando el método set para eliminar los repetidos, para después volver a convertir este set a una lista. Después se itera esta lista con los códigos de las calles, filtrando las filas en dfcruces_processed donde el código de vía tratado coincide con el código de calle actual, obteniendo las coordenadas de cruces relacionados con esa calle y los almacena en una lista llamada cruces. Además, filtra las filas en dfdirecciones_processed donde el código de vía coincide con el código de calle actual, obteniendo los números de las direcciones.

Con estos datos, se crea un diccionario donde las claves son los números de las direcciones y los valores son las coordenadas (X, Y) correspondientes a esas direcciones, en formato de tupla para unificarlos.

Además de esto, se crea un objeto Calle, utilizando la información obtenida (cruces, números de dirección, diccionario de coordenadas de dirección y tipo de vía), y agregando cada objeto a una lista de calles. Por lo tanto, esta función nos devuelve una lista con los objetos Calle del dataset.

```
def calles(dfdirecciones_processed,dfcruces_processed):

    calles_maps = []

    lista_1 = list(dfcruces_processed['Codigo de vía tratado'])
    lista_2 = list(dfcruces_processed['Codigo de vía que cruza o enlaza'])
    lista_juntas = lista_1 + lista_2

    codigos_unicos = list(set(lista_juntas))


    for codigo_calle in codigos_unicos:

        filtered_rows = dfcruces_processed[dfcruces_processed['Codigo de vía tratado'] == codigo_calle]
        cruces = []
        for _, row in filtered_rows.iterrows():
            cruce = (row['Coordenada X (Guia Urbana) cm (cruce)'], row['Coordenada Y (Guia Urbana) cm (cruce)'])
            cruces.append(cruce)


        filtered_rows_1 = dfdirecciones_processed[dfdirecciones_processed['Codigo de via'] == codigo_calle]
        numeros= []
        diccionario = {}
        for _, row in filtered_rows_1.iterrows():
            numero = row['Literal de numeracion']
            numeros.append(numero)
            tipo_de_via = row['Clase de la via']
            diccionario[numero]= (row['Coordenada X (Guia Urbana) cm'],row['Coordenada Y (Guia Urbana) cm'])



        calle = Calle(codigo_calle, list(set(cruces)), numeros, diccionario, tipo_de_via)
        calles_maps.append(calle)

    return calles_maps
```

Función agregar_vertices

Función: agregar los vértices con sus atributos correspondientes al grafo.

En esta función, se recibe un objeto grafo y una lista de cruces, que será la que se ha obtenido en las funciones anteriores. Con esta lista de cruces, añade al grafo un vértice con las coordenadas de los cruces a través de su método agregar_vertice. Además, se añade a un diccionario estas coordenadas de cada cruce, y se devuelve este diccionario como resultado de la función.

```
def agregar_vertices(grafo,lista_cruces):
    dicc = {}
    for cruce in lista_cruces:
        grafo.agregar_vertice((cruce.coord_x, cruce.coord_y))
        dicc[(cruce.coord_x,cruce.coord_y)] = (cruce.coord_x,cruce.coord_y)

    return dicc
```

Función agregar_aristas

Función: agregar las aristas con sus atributos correspondientes al grafo.

En esta función, se itera sobre la lista de calles que se recibe, y para cada calle:

- Obtiene el diccionario de números de calle (dict_numeros_calle) de la calle.
- Si el diccionario de números de calle no está vacío (len(dict_numeros_calle) != 0), encuentra la coordenada mínima a partir de las claves del diccionario. Luego, ordena los cruces de la calle según la distancia a esta coordenada mínima, haciendo esto a través de una función lambda.
- Si el diccionario de números de calle está vacío o no tiene información:
    - Si la cantidad de cruces es mayor que 1 (len(calle.cruces) > 1), establece la coordenada mínima como el primer cruce y ordena los cruces según la distancia a esta coordenada mínima, otra vez a través de una función lambda.
    - Si la cantidad de cruces es igual a 1, no hace nada (pass).

Además, para cada calle, menos en el caso de no hacer nada, para cada calle procesada, crea aristas en el grafo conectando los cruces de la calle. Recorre la lista ordenada de cruces y agrega una arista entre cada par consecutivo de cruces al grafo utilizando el método grafo.agregar_arista(ini, fin).

```
def agregar_aristas(grafo,lista_calles):
    for calle in lista_calles:
        dict_numeros_calle = calle.diccionario_numeros
        claves = list(dict_numeros_calle.keys())
        if len(dict_numeros_calle) != 0:
            clave_min = min(claves)
            coordenadas_min = dict_numeros_calle[clave_min]
            coordenadas_ref = (coordenadas_min[0],coordenadas_min[1])
            cruces_ordenados = sorted(calle.cruces,key = lambda cruce: distancia_entre_puntos(coordenadas_ref,(cruce[0],cruce[1])))

            for i in range(len(cruces_ordenados) - 1):
                ini = cruces_ordenados[i]
                fin = cruces_ordenados[i+1]
                grafo.agregar_arista(ini,fin)

        else:
            if len(calle.cruces) > 1:
                coordenadas_min = calle.cruces[0]
                coordenadas_ref = (coordenadas_min[0], coordenadas_min[1])
                cruces_ordenados = sorted(calle.cruces,key = lambda cruce: distancia_entre_puntos(coordenadas_ref,(cruce[0],cruce[1])))

                for i in range(len(cruces_ordenados) - 1):
                    ini = cruces_ordenados[i]
                    fin = cruces_ordenados[i+1]
                    grafo.agregar_arista(ini,fin)
            else:
                pass
```

Función generar_Madrid

Función: crear una imagen que muestre un grafo a partir de los datasets obtenidos por el Gobierno, y que este grafo represente la ciudad de Madrid.

Esta función se basa en los anteriores métodos y funciones que han sido programados a lo largo de la práctica. Se comienza iniciando una clase Grafo, a la cual, a través de la función agregar_aristas y agregar_vertices y usando como argumentos los dataframes procesados por la función datasets(), se le añaden los cruces como aristas, y las calles como vértices.

Gracias al método convertir_a_NetworkX de la clase Grafo, se puede proceder a representar esta imagen que muestra un grafo de las calles de Madrid, usando la librería NetworkX.

```
def generar_Madrid(lista_cruces,lista_calles):

    grafo = Grafo()
    dicc = agregar_vertices(grafo,lista_cruces)
    agregar_aristas(grafo,lista_calles)

    grafo = grafo.convertir_a_NetworkX()

    plt.figure(figsize=(50,50))
    plot=plt.plot()
    nx.draw(grafo, pos=dicc, with_labels=False, node_size=0.1)
    nx.draw_networkx_edges(grafo, pos=dicc,  edge_color='b', width=0.4)
    plt.show()
```

Desarrollo del módulo “gps.py”

1. **arrancar_gps(dfdirecciones_processed, dfcruces_processed, lista_calles, lista_cruces):**
    - **Propósito:** Genera dos grafos ponderados (uno basado en la distancia euclidiana y otro en el tiempo de viaje) para representar la red vial de Madrid.
    - **Entradas:**
        - **dfdirecciones_processed**: DataFrame con datos de direcciones procesados.
        - **dfcruces_processed**: DataFrame con datos de cruces procesados.
        - **lista_calles**: Lista de objetos Calle que representan las calles en Madrid.
        - **lista_cruces**: Lista de objetos Cruce que representan los cruces en Madrid.
    - **Salidas:**
        - **grafo_1**: Grafo ponderado basado en la distancia euclidiana.
        - **grafo_2**: Grafo ponderado basado en el tiempo de viaje.

```
def arrancar_gps(dfdirecciones_processed, dfcruces_processed, lista_calles, lista_cruces):
    #Peso: distancia eucl´idea entre los nodos
    grafo_1 = generar_Madrid_1(lista_cruces,lista_calles)

    #Peso: tiempo que tarda un coche en recorrer dicha arista
    grafo_2 = generar_Madrid_2(lista_cruces,lista_calles)

    return grafo_1, grafo_2
```

2. **seleccionar_direcciones(calle, numero):**
    - **Propósito:** Selecciona una dirección específica basada en el nombre de la calle y el número.
    - **Entradas:**
        - **calle**: Nombre de la calle.
        - **numero**: Número de la dirección.
    - **Salida:**
        - Tupla con las coordenadas, código y dirección de la dirección seleccionada.

```python
def seleccionar_direcciones(calle, numero):

    for i in range(df_direcciones.shape[0]):
        if df_direcciones['Nombre de la vía'][i] == calle and df_direcciones['Nombre de la vía'][i] == numero:
            print('encontrado')

    # Read the CSV file into a DataFrame
    df_direcciones = pd.read_csv("direcciones.csv", encoding='iso-8859-1', sep=";")

    # Display the unique values in the 'Direccion completa para el numero' column
    direcciones = df_direcciones['Direccion completa para el numero']
    codigos = df_direcciones['Codigo de via']


    # Get user input for the first address
    seleccion1 = int(input("Seleccione la primera dirección (número): ")) - 1
    direccion = codigos[seleccion1]
    codigo = codigos[seleccion1]
    coordenadas = (int(df_direcciones['Coordenada X (Guia Urbana) cm'][seleccion1]), int(df_direcciones['Coordenada Y (Guia Urbana) cm'][seleccion1]))


    return coordenadas, codigo, direccion
```

3. **seleccionar_opcion_ruta(grafo_1, grafo_2):**

    - **Propósito:** Permite al usuario seleccionar entre la ruta más corta y la ruta más rápida.
    - **Entradas:**
        - **grafo_1**: Grafo ponderado basado en la distancia euclidiana.
        - **grafo_2**: Grafo ponderado basado en el tiempo de viaje.
    - **Salida:**
        - Tupla con la opción seleccionada ("corta" o "rapida") y el grafo correspondiente.

```python
def seleccionar_opcion_ruta(grafo_1, grafo_2):
    print("Opciones de ruta:")
    print("1. Ruta más corta")
    print("2. Ruta más rápida")

    try:
        seleccion = int(input("Seleccione la opción de ruta (1 o 2): "))
        if seleccion == 1:
            grafo = grafo_1
            return "corta", grafo
        elif seleccion == 2:
            grafo = grafo_2
            return "rapida" , grafo
        else:
            print("Opción no válida. Por favor, seleccione 1 o 2.")
            return None

    except ValueError:
        print("Entrada inválida. Por favor, ingrese un número.")
        return None
```

4. **encontrar_cruce_mas_cercano(direccion, cord_x, cord_y, lista_cruces):**

    - **Propósito:** Encuentra el cruce más cercano dado el nombre de la calle y las coordenadas de una dirección.
    - **Entradas:**
        - **direccion**: Nombre de la calle.
        - **cord_x**: Coordenada X de la dirección.

- **cord_y**: Coordenada Y de la dirección.
- **lista_cruces**: Lista de objetos Cruce que representan los cruces en Madrid.

- **Salida:**
  - Objeto Cruce que representa el cruce más cercano.

```python
#Funcion para encontrar el cruce mas cercano dada las cordenadas de la direccion dada
def encontrar_cruce_mas_cercano(direccion, cord_x, cord_y, lista_cruces):
    # Inicializar con una distancia grande para asegurar la actualización en la primera iteración
    distancia_minima = float('inf')
    cruce_mas_cercano = None

    df_direcciones = pd.read_csv("direcciones.csv", encoding='iso-8859-1', sep=";")
    df_cruces = pd.read_csv("cruces.csv", encoding='iso-8859-1', sep=";")

    columna = df_direcciones[df_direcciones['Direccion completa para el numero'] == direccion]
    nombre_via = columna['Nombre de la vía'].iloc[0]

    cruces_df = df_cruces[df_cruces['Nombre de la via tratado'] == nombre_via]

    for _,cruce in cruces_df.iterrows():
        # Convertir a números y manejar NaN
        coord_x_cruce = pd.to_numeric(cruce['Coordenada X (Guia Urbana) cm (cruce)'], errors='coerce')
        coord_y_cruce = pd.to_numeric(cruce['Coordenada Y (Guia Urbana) cm (cruce)'], errors='coerce')

        distancia = math.sqrt((coord_x_cruce - cord_x)**2 + (coord_y_cruce - cord_y)**2)
        if distancia < distancia_minima:
            distancia_minima = distancia
            cruce_cercano_x = coord_x_cruce
            cruce_cercano_y = coord_y_cruce

    #Una vez encontrado el cruce, cogemos sus cordeandas y buscamos el objeto cruce para ese dado
    for cruce in lista_cruces:
        if (cruce_cercano_x, cruce_cercano_y) == (cruce.coord_x, cruce.coord_y):
            cruce_mas_cercano = cruce

    return cruce_mas_cercano
```

5. **encontrar_ruta(direccion_origen, direccion_destino, tipo_ruta, grafo=Grafo()):**

   - **Propósito:** Encuentra la ruta más corta o rápida entre dos direcciones utilizando un grafo dado.
   - **Entradas:**
     - **direccion_origen**: Dirección de origen.
     - **direccion_destino**: Dirección de destino.
     - **tipo_ruta**: Tipo de ruta ("corta" o "rapida").
     - **grafo**: Grafo ponderado (por defecto, un objeto Grafo vacío).
   - **Salida:**
     - Lista de nodos representando la ruta entre las direcciones.

```python
def encontrar_ruta(direccion_origen, direccion_destino, tipo_ruta, grafo=Grafo()):

    # Elegimos el tipo de ruta que queremos tomar
    if tipo_ruta == "corta":
        #implementar la clase grafo
        ruta = grafo.camino_minimo(direccion_origen,direccion_destino)
    elif tipo_ruta == "rapida":
        #implementar la clase grafo
        ruta = grafo.camino_minimo(direccion_origen,direccion_destino)
    else:
        print("Tipo de ruta no reconocido.")
        return None

    return ruta
```

6. **gps_ruta(o**
7. **rigen, destino, grafo_1, grafo_2):**
    - **Propósito:** Encuentra la ruta y el grafo correspondiente entre dos direcciones dadas y dos grafos ponderados.
    - **Entradas:**
        - **origen**: Dirección de origen.
        - **destino**: Dirección de destino.
        - **grafo_1**: Grafo ponderado basado en la distancia euclidiana.
        - **grafo_2**: Grafo ponderado basado en el tiempo de viaje.
    - **Salida:**
        - Tupla con la lista de nodos de la ruta y el grafo seleccionado.

```python
def gps_ruta(origen, destino, grafo_1, grafo_2):

    #Cargamos el dataset de direcciones
    df_direcciones = pd.read_csv("direcciones.csv", encoding='iso-8859-1', sep=";")

    #Cargamos el dataset de cruces
    df_cruces = pd.read_csv("cruces.csv", encoding='iso-8859-1', sep=";")
    lista_cruces = cruces(df_cruces)

    #Encontramos las coordendas del origen y destino
    row_origen = df_direcciones[df_direcciones['Direccion completa para el numero'] == origen]
    coordenadas_origen = (
        float(row_origen['Coordenada X (Guia Urbana) cm'].iloc[0]),
        float(row_origen['Coordenada Y (Guia Urbana) cm'].iloc[0])
    )

    row_destino = df_direcciones[df_direcciones['Direccion completa para el numero'] == destino]
    coordenadas_destino = (
        float(row_destino['Coordenada X (Guia Urbana) cm'].iloc[0]),
        float(row_destino['Coordenada Y (Guia Urbana) cm'].iloc[0])
    )

    #Encontramos el cruce mas cercano al origen
    cruce_origen = encontrar_cruce_mas_cercano(origen, coordenadas_origen[0],coordenadas_origen[1], lista_cruces)
    cordenadas_cruce_origen = (cruce_origen.coord_x,cruce_origen.coord_y)

    #Encontramos el cruce mas cercano al destino
    cruce_destino = encontrar_cruce_mas_cercano(destino, coordenadas_destino[0],coordenadas_destino[1], lista_cruces)
    cordenadas_cruce_destino = (cruce_destino.coord_x,cruce_destino.coord_y)

    #Elegimos el tipo de ruta
    tipo_ruta, grafo = seleccionar_opcion_ruta(grafo_1, grafo_2)

    #Encontramos la ruta como una lista de cordenadas a seguir
    ruta = encontrar_ruta(cordenadas_cruce_origen, cordenadas_cruce_destino, tipo_ruta, grafo)

    return ruta , grafo
```

8. **recorrido_por_calles(grafo, camino, coordenadas_orig, destino_coord, cod_orig, cod_destino):**

    - **Propósito:** Calcula el recorrido detallado por calles dados los nodos de una ruta y sus coordenadas.
    - **Entradas:**
        - **grafo**: Grafo ponderado.
        - **camino**: Lista de nodos representando la ruta.
        - **coordenadas_orig**: Coordenadas de origen.
        - **destino_coord**: Coordenadas de destino.
        - **cod_orig**: Código de origen.
        - **cod_destino**: Código de destino.
    - **Salidas:**
        - Diccionario con información detallada sobre el recorrido por calles.
        - Lista de distancias entre las intersecciones.

```python
def recorrido_por_calles(grafo, camino, coordenadas_orig, destino_coord, cod_orig, cod_destino):
    # Lista que almacenará las aristas a recorrer
    aristas_recorrer = []

    # Iterar sobre la ruta para obtener las aristas correspondientes
    for i in range(len(camino) - 1):
        arista_actual = grafo.obtener_arista(camino[i], camino[i + 1])

        # Actualizar los vértices de la arista
        arista_actual[0]['vertice1'] = camino[i]
        arista_actual[0]['vertice2'] = camino[i + 1]
        aristas_recorrer.append(arista_actual)

    # Variables para el seguimiento de la calle actual
    calle_anterior = None
    dicc_recorrido_calles = {}
    distancias = []
    paso = 0
    calle_dist = 0

    # Primer elemento del diccionario con información del origen
    dicc_recorrido_calles[0] = (cod_orig, round(distancia_entre_puntos(coordenadas_orig, aristas_recorrer[0][0]['vertice1']) / 100))

    # Iterar sobre las aristas para calcular distancias y segmentar por calles
    for arista in aristas_recorrer:
        calle_actual = arista[0]['codigo']

        if calle_actual != calle_anterior:
            if calle_anterior is not None:
                distancias.append(round(calle_dist / 100))
            paso += 1
            calle_dist = 0
            dicc_recorrido_calles[paso] = []
            dicc_recorrido_calles[paso].append(arista)
            calle_dist = calle_dist + arista[0]['distancia']

        elif calle_actual == calle_anterior:
            dicc_recorrido_calles[paso].append(arista)
            calle_dist += arista[0]['distancia']

        calle_anterior = calle_actual
    distancias.append(calle_dist)

    # Último elemento del diccionario con información del destino
    dicc_recorrido_calles[paso + 1] = (cod_destino, distancia_entre_puntos(destino_coord, aristas_recorrer[-1][0]['vertice2']))

    return dicc_recorrido_calles, distancias
```

9. **direccion_giro(cruce_anterior, cruce_actual, cruce_siguiente):**

    - **Propósito:** Determina la dirección del giro en una intersección dada la información de los cruces.
    - **Entradas:**
        - **cruce_anterior**: Coordenadas del cruce anterior.
        - **cruce_actual**: Coordenadas del cruce actual.
        - **cruce_siguiente**: Coordenadas del cruce siguiente.
    - **Salida:**
        - Dirección del giro en la intersección.

```python
def direccion_giro(cruce_anterior, cruce_actual, cruce_siguiente):
    # Determina la dirección del giro en una intersección
    if cruce_actual[0] == cruce_anterior[0]:
        if cruce_siguiente[0] < cruce_actual[0]:
            giro = 'izquierda'
        else:
            giro = 'derecha'
    else:
        m = (cruce_actual[1] - cruce_anterior[1]) / (cruce_actual[0] - cruce_anterior[0])
        x = 1 / m * (cruce_siguiente[1] - cruce_anterior[1]) + cruce_anterior[0]
        if cruce_siguiente[0] > x:
            giro = 'derecha'
        else:
            giro = 'izquierda'

        if math.sqrt(cruce_actual[0] ** 2 + cruce_actual[1] ** 2) < math.sqrt(cruce_anterior[0] ** 2 + cruce_anterior[1] ** 2):
            if giro == 'derecha':
                return 'izquierda'
            return 'derecha'
        return giro
```

10. **obtener_nombre(codigo, df_cruces):**

    - **Propósito:** Obtiene el nombre de una calle a partir de su código utilizando el DataFrame de cruces.
    - **Entradas:**
        - **codigo**: Código de la calle.
        - **df_cruces**: DataFrame de cruces.
    - **Salida:**
        - Nombre de la calle correspondiente al código.

```python
def obtener_nombre(codigo, df_cruces):
    # Obtiene el nombre de una calle a partir de su código
    row = df_cruces[df_cruces['Codigo de vía tratado'] == codigo]

    if not row.empty:
        nombre = row.iloc[0]['Literal completo del vial tratado']
        return nombre

    else:

        return None
```

11. generar_instrucciones(diccionario_recorrido_calles: dict, distancias: list, intersecciones: pd.DataFrame):

- **Propósito:**
    - Genera instrucciones detalladas y giros basados en el diccionario de recorrido por calles, distancias y el DataFrame de intersecciones.
- **Entradas:**
    - **diccionario_recorrido_calles**: Diccionario con información detallada sobre el recorrido por calles.
    - **distancias**: Lista de distancias entre intersecciones.
    - **intersecciones**: DataFrame de intersecciones.
- **Salida:**
    - Lista de instrucciones detalladas y giros.

La función **generar_instrucciones** toma el diccionario de recorrido por calles (**diccionario_recorrido_calles**), la lista de distancias entre intersecciones (**distancias**), y el DataFrame de intersecciones (**intersecciones**). Itera sobre las calles en el recorrido, obteniendo nombres de calles y distancias. Además, determina giros en las intersecciones utilizando la función **direccion_giro**. Finalmente, construye una lista de instrucciones detalladas y giros y la devuelve.

```python
def generar_instrucciones(diccionario_recorrido_calles: dict, distancias: list, intersecciones: pd.DataFrame):
    giros = []
    instrucciones = []
    n_calle = len(diccionario_recorrido_calles.keys())
    valores = list(diccionario_recorrido_calles.values())
    instrucciones.append((diccionario_recorrido_calles[0][0], diccionario_recorrido_calles[0][1]))

    # Iterar sobre las calles para generar instrucciones y giros
    for i in range(1, len(diccionario_recorrido_calles.keys()) - 1):
        calle = obtener_nombre(valores[i][0][0]['codigo'], intersecciones)
        cm = (distancias[i - 1])
        instrucciones.append((calle, cm))
        if i + 1 < len(diccionario_recorrido_calles.keys()) - 1:
            punto_anterior = valores[i][-1][0]['origen']
            punto_interseccion = valores[i][-1][0]['destino']
            punto_siguiente = valores[i + 1][0][0]['destino']
            sentido_giro = direccion_giro(punto_anterior, punto_interseccion, punto_siguiente)
            giros.append(sentido_giro)

    # Último elemento del diccionario con información del destino
    instrucciones.append((diccionario_recorrido_calles[n_calle - 1][0], diccionario_recorrido_calles[n_calle - 1][1]))

    return instrucciones, giros
```

12. instrucciones(instrucciones, giros):

- **Propósito:**
    - Muestra por consola las instrucciones y giros para el recorrido.
- **Entradas:**
    - **instrucciones**: Lista de instrucciones detalladas.
    - **giros**: Lista de giros.
- **Salida:**
    - Impresión de las instrucciones.

La función **instrucciones** recibe la lista de instrucciones detalladas (**instrucciones**) y la lista de giros (**giros**). Imprime en la consola las instrucciones paso a paso, incluyendo la dirección de giro en las intersecciones. La información finaliza con un mensaje indicando que el destino se ha alcanzado.

```
def instrucciones(instrucciones, giros):
    final = len(instrucciones)-1
    print(f'Usted comienza {instrucciones[0][1]} metros por {obtener_nombre(instrucciones[0][0], cruces)}')
    for i in range(1,len(instrucciones)-1):

        print(f'Continue por la calle {instrucciones[i][0]} durante {instrucciones[i][1]} metros')
        if i < len(giros):
            print(f'Gire a la {giros[i]}, para la calle {instrucciones[i+1][0]}')
    print(f'Continue por la calle  {obtener_nombre(instrucciones[final][0], cruces)} durante {instrucciones[final][1]}')
    print('Usted se encuentra en su destino.')
```

13. pintar_camino_rojo(lista_cruces, lista_calles, camino):

- **Propósito:**
    - Visualiza el camino en un grafo, destacando las aristas del camino en rojo.
- **Entradas:**
    - **lista_cruces**: Lista de objetos Cruce que representan los cruces en Madrid.
    - **lista_calles**: Lista de objetos Calle que representan las calles en Madrid.
    - **camino**: Lista de nodos que representan el camino.
- **Salida:**
    - Visualización gráfica del grafo con el camino resaltado en rojo.

La función **pintar_camino_rojo** recibe la lista de cruces (**lista_cruces**), la lista de calles (**lista_calles**), y el camino representado por nodos (**camino**). Primero, convierte el camino en una lista de nodos de las aristas a pintar en rojo. Luego, crea un nuevo grafo, añade los vértices

y aristas correspondientes, y utiliza la biblioteca NetworkX para visualizar el grafo. Finalmente, resalta las aristas del camino en rojo y muestra la representación gráfica.

```python
def pintar_camino_rojo(lista_cruces, lista_calles, camino):
    camino_final = []
    #Primero pasamos el camino a una lista de nos nodos de las aristas a pintar en rojo
    for i in range(len(camino)):
        if i == 0:
            camino_final.append((camino[i], camino[i + 1]))
        elif i == (len(camino)-1):
            camino_final.append((camino[i-1], camino[i]))
        else:
            camino_final.append((camino[i - 1], camino[i]))
            camino_final.append((camino[i], camino[i + 1]))

    #creamos el grafo de nuevo como anteriormente
    grafo = Grafo()
    dicc = agregar_vertices(grafo, lista_cruces)
    agregar_aristas_peso_vel_max(grafo, lista_calles)
    grafo_nx = grafo.convertir_a_NetworkX()


    plt.figure(figsize=(50, 50))
    plot = plt.plot()
    nx.draw(grafo_nx, pos=dicc, with_labels=False, node_size=0.1)
    #especificamos las aristas a pintar son las del camino final
    nx.draw_networkx_edges(grafo_nx, pos=dicc, edgelist=camino_final, edge_color='r', width=1.5)
    plt.show()
```

Bibliografía

1. Transparencias Tema 5: Teoría de grafos - Departamento de Matemática Aplicada (Escuela Técnica Superior de Ingeniería (ICAI))
2. Transparencias Tema 6: Algoritmos de recorrido de grafos - Departamento de Matemática Aplicada (Escuela Técnica Superior de Ingeniería (ICAI))
3. Informe Práctica 3 Matemática Discreta
4. Chatgpt ChatGPT. Available at: https://openai.com/chatgpt
5. "Teoría de Grafos" - Jose Salvador Cánovas Peña (Departamento de Matemática Aplicada y Estadística (Universidad Politécnica de Cartagena) Available at: https://www.dmae.upct.es/~jose/discreta/grafos.pdf
6. NetworkX Documentation – Available at: https://networkx.org/
7. Heapq Documentation - Available at: https://docs.python.org/3/library/heapq.html
8. Pandas Documentation – Available at: https://pandas.pydata.org/