

PRÁCTICA 3: GPS

MATEMÁTICA DISCRETA

Autores:

Alejandro Royo-Villanova Seguí

Eduardo Tovar Ruiz

Grupo: G08B

Madrid, noviembre 2025

Índice de Contenidos

1. INTRODUCCIÓN.....	3
2. ESTRUCTURA DEL MÓDULO “GPS.PY”	4
2.1. MAIN() → NONE.....	4
2.2. CREAM_GRAFO() → TUPLE[PD.DATAFRAME, NX.DIGRAPH]	4
2.3. BUCLE_INTERACTIVO(DF:PD.DATAFRAME, GRAFO:NX.DIGRAPH)→NONE:.....	4
2.4. ENCONTRAR_NODO(DIRECCION:STR, G:DiGRAPH, DF:DATAFRAME)→INT:.....	5
2.5. MOSTRAR_CAMINO() →:	5
3. GRAFO_PESADO.PY	7
3.1. DIJKSTRA(G, PESO, ORIGEN) → DICT	7
3.2. PRIM(G, PESO) → DICT	8
3.3. KRUSKAL(G, PESO) → LIST	8
3.4. CAMINO_MINIMO(G, PESO, ORIGEN, DESTINO) → LIST	9
3.5. MAS_CORTO → INT	10
3.6. MAS_RAPIDO → INT.....	¡ERROR! MARCADOR NO DEFINIDO.
3.7. MAS_RAPIDO_SEMAFOROS → INT	¡ERROR! MARCADOR NO DEFINIDO.
4. CALLEJERO.PY	11
4.1. COORD_TO_DECIMAL(CADENA:STR) → FLOAT:.....	11
4.2. CONSTRUIR_DIRECCION(FILA:PD.SERIES) → STR:	11
4.3. CARGA_CALLEJERO() → PD.DATAFRAME:	11
4.4. BUSCA_DIRECCION(DIRECCION:STR,CALLEJERO:PD.DATAFRAME)→TUPLE:.....	12
4.5. BUSCA_DIRECCION_FUZZY(DIRECCIÓN, CALLEJERO, UMBRAL_SIMILITUD):.....	12
4.6. CARGA_GRAFO() → NX.MULTIDIGRAPH:	13
4.7. PROCESA_GRAFO(MULTIDIGRAFO:NX.MULTIDIGRAPH) → NX.DIGRAPH:	13
4.8. EXPLORAR_DATOS(DIGRAFO: NX.DIGRAPH):	¡ERROR! MARCADOR NO DEFINIDO.
4.9. DIBUJAR_GRAFO_NX(G:NX.DIGRAPH, MIN_LONG: FLOAT):.....	13
4.10. NORMALIZA_MAXSPEED(VALOR) → TUPLE:.....	¡ERROR! MARCADOR NO DEFINIDO.
4.11. CREA_DATAFRAME_ARISTAS(DIGRAFO:NX.DIGRAPH)→PD.DATAFRAME:	¡ERROR! MARCADOR NO DEFINIDO.
5. CONCLUSIONES	14
6. BIBLIOGRAFÍA.....	16

1. INTRODUCCIÓN

Este documento expone el trabajo de Alejandro Royo-Villanova Seguí y Eduardo Tovar Ruiz, alumnos de la Escuela Técnica Superior de Ingeniería (ICAI) de la Universidad Pontificia Comillas. Dicho trabajo lleva como título “Práctica 3: GPS” y se ha realizado dentro de la asignatura “Matemática Discreta” de 2º de iMAT.

En el trabajo se crea un sistema de navegación, implementado en el módulo “gps.py”, junto con dos utilidades de apoyo “callejero.py” que sirve para crear el grafo del callejero de Madrid, así como para encontrar la posición del nodo asociado a cualquier dirección introducida por el usuario, también se apoya sobre “grafo_pesado.py”, que implementa los algoritmos de Kruskal, Prim y Dijkstra, siendo este último el utilizado para encontrar el camino mínimo entre el origen y el destino. Todo ello se basa en los conocimientos sobre teoría de grafos impartidos en clase.

El documento está estructurado en 6 apartados, siendo el primero esta introducción. El segundo explica la estructura del módulo “gps.py”. En el tercero se detalla el funcionamiento de la utilidad “grafo_pesado.py”. El cuarto apartado contiene los detalles de la utilidad “callejero.py”. En el apartado número cinco se encuentran las conclusiones, así como los problemas encontrados. Por último, el apartado seis recoge las referencias bibliográficas y recursos consultados

2. ESTRUCTURA DEL MÓDULO “GPS.PY”

El módulo principal del programa es “gps.py”. Es el que permite la interacción de entrada y de salida con el usuario.

Para ello, se han tenido que importar las utilidades complementarias “callejero.py” y “grafo_pesado.py” así como las librerías públicas “pandas”, “networkx”, y “osmx” para asegurar un correcto funcionamiento.

El módulo contiene 5 funciones que estructuran la interacción del navegador. Cada una tiene una responsabilidad específica dentro del flujo de ejecución. A continuación, se describen de forma individual:

2.1. **main()** → **None**

Esta función constituye el núcleo operativo. Sirve para iniciar el entorno de ejecución, así como para controlar el flujo de ejecución. Para ello llama a dos funciones, **crear_grafo** y **bucle_interactivo**.

2.2. **Crear_grafo()** → **Tuple[pd.DataFrame, nx.DiGraph]**

La función se encarga de crear tanto el dígrafo de Madrid, como el DataFrame de su callejero. Para ello implementa las funciones **carga_callejero**, **carga_grafo** y **procesa_grafo** de la utilidad “callejero.py”.

Por último, devuelve una tupla con el DataFrame como primer argumento y con el dígrafo como segundo.

2.3. **bucle_interactivo(df:pd.Dataframe, grafo:nx.DiGraph)→None:**

Esta función es la encargada de gestionar el ciclo de interacción con el usuario. Recibe como entrada el DataFrame del callejero y el dígrafo asociado. Además, permite solicitar la ruta entre dos direcciones introducidas por el usuario.

Cada iteración:

1. Captura las direcciones, solicitando al usuario el origen y el destino
2. Convierte las direcciones a nodos del grafo a través de la función **encontrar_nodos**
3. Selección del criterio del cálculo. Para ello presenta un menú de opciones y solicita elegir entre la ruta más corta, la más rápida y la más rápida considerando semáforos.

4. Asigna la función de peso según la opción seleccionada en el paso anterior
5. Calcula la ruta llamando a la función de "grafo_pesado.py" **camino_minimo** siguiendo el criterio elegido.
6. Visualización de la ruta gracias a la función **mostrar_camino**.

El bucle se mantiene activo mientras el usuario continúe proporcionando direcciones. La función no devuelve ningún valor ya que su único objetivo es controlar la interacción con el usuario

2.4. **encontrar_nodo(direccion:str, G:DiGraph, df:DataFrame)→int:**

Esta función tiene como finalidad transformar una dirección textual en un nodo concreto dentro del dígrafo de Madrid.

Actúa como interfaz entre el usuario y la estructura de datos utilizada en el programa, garantizando que el sistema pueda operar tolerando errores del usuario.

Para ello se llama a **busca_direccion_fuzzy** del módulo "callejero.py". Lo que devuelve la latitud y longitud de la dirección. A continuación, encuentra el nodo más cercano a esas coordenadas utilizando la función **nearest_node** de la librería "OSMnx". Por último, la función devuelve el ID asociado al nodo encontrado.

2.5. **_nombre_calle(G: nx.DiGraph, u: int, v: int) → str:**

Esta función tiene como finalidad devolver el nombre de la calle que corresponde a la arista entre "u" y "v".

Para ello accede a los datos de la arista e intenta recuperar el atributo "name", pero si no tiene o está vacío, en casos muy extraños, dice que el nombre es "vía sin nombre".

Esta es una función auxiliar de la función "*construir_instrucciones*".

2.6. **_frase_segmento(nombre:str, dist_m:float, es_primero:bool) → str**

Esta función tiene como finalidad construir la frase de la indicación.

Con ese objetivo, primero pasa la distancia de m a km, y luego selecciona el inicio de la frase, siendo este "Luego", o "Sal desde el origen y" si se trata de la primera indicación del recorrido. Finalmente, construye la frase utilizando el inicio comentado previamente, e indicando el nombre de la vía por la que va y cuanta distancia debe recorrer por ella.

Esta es otra función auxiliar de la función "*construir_instrucciones*".

2.7. **construir_instrucciones(camino:List[int], G:nx.DiGraph) → List[str]:**

Esta función toma el camino que se debe realizar, el cual es una lista de los vértices del grafo por los que se pasan para llegar al destino (return de la función “*camino_minimo*”, de la librería *grafo_pesado.py*), y el grafo, y devuelve las instrucciones a seguir para llegar al destino.

La función comienza contemplando el caso de que el camino esté vacío ya que el origen y destino coinciden, y si ocurre devuelve un mensaje diciendo esto. Si no ocurre esto, recorre las aristas del camino agrupando los tramos consecutivos que van por la misma calle, sumando la distancia que se va a recorrer por esa calle. Cuando cambia de calle, llama a “*_frase_segmento*” y crea la instrucción, y realiza el mismo proceso para la siguiente calle. Así hasta el último tramo, y escribe una frase con la distancia total recorrida.

2.8. **mostrar_camino(camino:List[int], grafo:nx.DiGraph) → None:**

Esta función escribe en orden las indicaciones que se deben seguir.

Por esto, llama a la función “*construir_instrucciones*” y hace “print” de cada una de las instrucciones, añadiendo el número de instrucción que es.

2.9. **dibujar_ruta(camino:List[int], grafo:nx.DiGraph) → None:**

Esta función tiene la finalidad de dibujar el grafo completo y resaltar la ruta que se va a realizar.

Para ello primero mira a ver si no hay camino. Si esto pasa no imprime nada y solo imprime un mensaje. Si hay camino, dibuja el grafo de todas las calles en gris y luego la ruta en rojo y las aristas un poco más anchas. Esto lo hace con el método de la librería *Networkx* llamado “*draw_networkx_edges*”. Este método toma el grafo, un diccionario con clave los nodos y valor una tupla de la longitud y latitud del nodo, el color que quieras dibujarlo, la anchura y si quieres dibujar el grafo con flechas o no. En el caso de dibujar la ruta, lo que hace es que añade un parámetro llamado “*edgelist*” al método, que es una lista de las aristas que quieres dibujar.

3. GRAFO_PESADO.PY

El módulo “grafo_pesado.py” sirve como explorador del grafo del callejero de Madrid. Implementando varios algoritmos de recorrido de grafos a la vez que una función para encontrar el camino mínimo entre dos nodos y diversas funciones para calcular los pesos.

Para asegurar el correcto funcionamiento de los algoritmos se han importado las librerías *typing*, *networkx*, *hepq* y *sys*. además, se ha creado una variable *INFTY* que tiene un valor infinito para representar la distancia entre nodos no conectados.

A continuación, se describen las diferentes funciones del módulo:

3.1. *dijkstra*(G, peso, origen) → Dict

Calcula un Árbol de Caminos Mínimos para el grafo pesado partiendo del vértice "origen" usando el algoritmo de Dijkstra. Calcula únicamente el árbol de la componente conexa que contiene a "origen".

Argumentos:

- G: *Union[nx.Graph, nx.DiGraph]*. Grafo sobre el que se quiere aplicar el algoritmo de Dijkstra
- Peso: *Union[Callable[[nx.Graph, object, object], float], Callable[[nx.DiGraph, object, object], float]]*. Función de peso que se quiere utilizar durante el algoritmo.
- Origen: *object*. Nodo desde el que se quiere aplicar el algoritmo

Returns:

- *Dict[object,object]*: Devuelve un diccionario que indica, para cada vértice alcanzable desde “origen”, que vértice es su padre en el árbol de caminos mínimos.

Raises:

- *TypeError*. Si el origen no es hashable.

La función comienza comprobando si origen es hashable y si no lo es hace raise del error indicando que el origen debe ser hashable.

A continuación, crea los diccionarios *padre*, *dist*, *visitado* y para cada nodo crea sus valores iniciales (p.e infinito para todas las distancias) y cambia el valor de la distancia a el nodo origen a 0. Además, crea la lista *Q* que sirve como cola.

El algoritmo entra en un bucle que no termina hasta que la lista *Q* se encuentre vacía en el que cambia los valores de los diccionarios de todos los nodos accesibles. Por último,

elimina el valor del padre del origen, ya que este nodo no tiene ningún padre y devuelve el diccionario de los padres de cada nodo.

3.2. `prim(G, peso) → Dict`

Calcula un Árbol Abarcador Mínimo para el grafo pesado usando el algoritmo de Prim.

Argumentos:

- G: `Union[nx.Graph, nx.DiGraph]`. Grafo sobre el que se quiere aplicar el algoritmo de Prim
- Peso: `Union[Callable[[nx.Graph, object, object], float], Callable[[nx.DiGraph, object, object], float]]`. Función de peso que se quiere utilizar durante el algoritmo.

Returns:

- `Dict[object,object]`: Devuelve un diccionario que indica, para cada vértice, que vértice es su padre en el árbol abarcador mínimo.

La función crea los diccionarios *padre*, *coste* y el heap *Q*, rellena los diccionarios con los valores iniciales de cada nodo e introduce los valores en el heap *Q*. A continuación, crea la variable *en_Q* que determina los nodos que todavía no se han añadido al árbol abarcador mínimo.

El algoritmo entra en un bucle hasta que *Q* no este vacío, y extrae el nodo con menor coste del heap *Q* a la vez que lo elimina de *en_Q*. Después actualiza los valores de sus nodos vecinos. Una vez se termina el bucle la función devuelve el diccionario de los padres de cada nodo.

3.3. `kruskal(G, peso) → List`

Calcula un Árbol Abarcador Mínimo para el grafo usando el algoritmo de Kruskal.

Argumentos:

- G: `Union[nx.Graph, nx.DiGraph]`. Grafo sobre el que se quiere aplicar el algoritmo de kruskal
- Peso: `Union[Callable[[nx.Graph, object, object], float], Callable[[nx.DiGraph, object, object], float]]`. Función de peso que se quiere utilizar durante el algoritmo.

Returns:

- `List[Tuple[object, object]]`: Devuelve una lista de los pares de vértices del grafo que forman las aristas del árbol abarcador mínimo.

Comienza creando una lista de tuplas donde cada tupla indica el peso de la arista y sus extremos. A continuación, ordena esta lista según los pesos.

Luego se crea un diccionario donde cada vértice pertenece a su propia componente conexa y crea una lista vacía llamada *aristas_aam*.

La función entra en un bucle que no termina hasta que la lista de tuplas no este vacía y extrae por cada iteración los vértices pertenecientes a la arista con menor coste. Seguido por la actualización de las componentes conexas. Una vez ya se ha recorrido todas las aristas la función devuelve la lista *aristas_aam* que contiene las aristas del árbol abarcador mínimo.

3.4. Camino_minimo(G, peso, origen, destino) → List

Argumentos:

- G: *Union[nx.Graph, nx.DiGraph]*. Grafo sobre el que se quiere aplicar el algoritmo de Dijkstra
- Peso: *Union[Callable[[nx.Graph, object, object], float], Callable[[nx.DiGraph, object, object], float]]*. Función de peso que se quiere utilizar durante el algoritmo.
- Origen: *object*. Nodo desde el que se quiere aplicar el algoritmo
- Destino: *object*. Nodo hasta el que se quiere llegar

Returns:

- *List[object]*: Devuelve la lista con los nodos por los que pasa el camino entre el origen y el destino, siendo el primer elemento el origen y el ultimo el destino.

Raises:

- *TypeError*. Si el origen o el destino no es hashable.

La función comprueba si los nodos origen y destino son hashables, en caso contrario hace raise del error.

Lo primero que hace la función es comprobar el caso trivial en el que ambos nodos sean el mismo y si es el caso devuelve únicamente el nodo de origen. Si este no es el caso, crea la lista *camino* e inserta el nodo destino. A continuación, llama a la función **Dijkstra** para obtener el padre de cada nodo. Si el nodo destino no se encuentra en este diccionario significa que no se puede alcanzar y devuelve una lista vacía.

Para añadir elementos al camino inicia una variable *actual* con el valor de *destino* y entra en un bucle que no termina hasta que no sea igual que el origen. Dentro del bucle cambia actual por su padre y lo añade al camino.

Por último, la función da la vuelta al camino y devuelve la lista.

3.5. **mas_corto(G:Union[nx.Graph, nx.DiGraph], u:object, v:object) → float:**

Función de peso que calcula la ruta más corta en metros.

La función toma el grafo y los nodos “u” (origen) y “v” (destino) y devuelve la longitud de esa arista en metros. Para ello devuelve el atributo “length” de la arista (u,v)

3.6. **_velocidad_kmh(G:nx.Graph, u:object, v:object) → float:**

Función auxiliar de “mas_rapido” que obtiene la velocidad máxima permitida en km/h para la arista (u,v).

La función primero intenta conseguir la velocidad máxima a través de la propiedad “maxspeed” de la arista, pero si no hay recurre al diccionario “MAX_SPEEDS”. De este modo, sabiendo el tipo de vía que es gracias al atributo “highway” de la arista, conseguimos la velocidad máxima para esa vía gracias al diccionario.

3.7. **mas_rapido(G:nx.Graph, u:object, v:object) → float:**

Función de peso que calcula la ruta más rápida suponiendo que se circula siempre a la velocidad máxima permitida en cada vía.

Argumentos:

G: nx.Graph. Grafo de calles.

u, v: object. Nodos que definen la arista.

Returns:

float: tiempo estimado de recorrido de la arista (u, v) en segundos.

La función toma el grafo y los nodos “u” (origen) y “v” (destino) y devuelve el tiempo en segundos que tarda en recorrer esa arista. Esto lo calcula pasando la velocidad máxima de la vía a m/s y dividiendo la longitud (atributo “length”) de la vía entre la velocidad.

3.8. **mas_rapido_semaforos(G:nx.Graph, u:object, v:object) → float:**

Función de peso que calcula la ruta más rápida teniendo en cuenta paradas esperadas en semáforos (probabilidad de 0.8 de esperar 30 segundos en cada vértice).

La función toma el grafo y los nodos “u” (origen) y “v” (destino) y devuelve el tiempo en segundos que tarda en recorrer esa arista. Esto lo calcula utilizando la función “mas_rapido” para calcular el tiempo base que se tarda en recorrerla y sumándole 24 segundos, que es el tiempo esperado de parada ($30 \times 0.8 = 24$).

4. CALLEJERO.PY

El objetivo del módulo “callejero.py” es el de representar trabajar con el callejero de Madrid para crear el grafo más completo y realista posible. Para ello se han importado las librerías *osmn*, *networkx*, *pandas*, *numpy*, *diffli*, *os*, *typin* y *matplotlib*. Además, se han creado las variables **STREET_FILE_NAME**, **PLACE_NAME**, **MAP_FILE_NAME** y **MAX_SPEEDS** para universalizar todas las funciones y evitar errores a la hora de llamar a archivos. También se han creado dos tipos de excepciones.

Las funciones implementadas son las siguientes:

4.1. **Coord_to_decimal(cadena:str) → float:**

Convierte una coordenada en el formato de grados, minutos y segundos a una en formato decimal.

Primero limpia la cadena y la separa en las variables *grados*, *minutos*, *segundos* y *orientacion*. Después suma los grados, minutos y segundos con la conversión adecuada (1 grado son 60 minutos y 1 minuto son 60 segundos). Por último, si la orientación es sur u oeste, el valor de las coordenadas se multiplica por -1.

4.2. **Construir_direccion(fila:pd.Series) → str:**

Recibe una fila del DataFrame del callejero y genera una dirección en formato textual.

Extrae de la fila del DataFrame la clase, el par y el número y lo pone todo en una cadena con el formato “*nombre de la vía , numero*”.

4.3. **carga_callejero() → pd.DataFrame:**

Carga el callejero de Madrid, lo procesa y devuelve un DataFrame de los datos procesado.

Comienza indicando las columnas que se van a utilizar. A continuación, comprueba si existe el fichero “*direcciones.csv*” y, si existe, crea el DataFrame asociado, en caso contrario devuelve un error que dice que no existe el fichero.

Si se ha conseguido leer el fichero procesa cada columna descargada mediante las funciones **coord_to_decimal** y **construir_direccion**. Una vez procesado devuelve el DataFrame.

4.4. `busca_direccion(direccion:str,callejero:pd.DataFrame)→Tuple:`

Función que busca una dirección en el formato "calle, numero" en el DataFrame del callejero de Madrid y devuelve el par (latitud, longitud) en grados.

Primero se limpia el string y se comprueba si hay coincidencia en el DataFrame del callejero, en caso de que no la haya lanza un error. Además, si hay varias coincidencias, se escoge la primera. A continuación, extrae los valores de latitud y longitud y los devuelve.

4.5. `Busca_direccion_fuzzy(dirección, callejero, umbral_similitud):`

Función que busca la dirección más parecida a la dada utilizando la comparación aproximada (fuzzy search), dada en el formato "calle, numero" y devuelve el par (latitud, longitud) en grados.

Argumentos:

- Dirección (str). Nombre completo de la calle con número.
- Callejero (DataFrame). DataFrame con la información de las calles.
- Umbral_similitud (float). Valor entre 0 y 1. Solo se aceptan coincidencias cuya similitud sea al menos este valor. Viene igualado a 0.8 por defecto

Returns:

- Tuple [float,float]: Par de (latitud, longitud) de la dirección buscada, expresados en grados.

Raises:

- AdressNotFoundError: Si no se encuentra ninguna dirección lo bastante parecida en la base de datos.

Funciona de manera muy parecida a **busca_direccion**. Limpia la cadena, comprueba si hay alguna similitud en el DataFrame, extrae las coordenadas y las devuelve. Solo que esta vez en lugar de necesitar un *perfect match*, utiliza la función **get_close_matches** de `diffib` para encontrar cadenas parecidas y de ellas escoge la más cercana.

4.6. `Carga_grafo()` → `nx.MultiDiGraph`:

Función que recupera el quiver de calles de Madrid de OpenStreetMap. Para ello primero comprueba si ya existe el archivo, y así evitar tiempos de espera innecesarios. Si no existe lo intenta descargar, puede dar error si el servidor no está disponible, y si lo ha conseguido lo guarda en un archivo llamado "Madrid.graphml" y lo devuelve.

4.7. `procesa_grafo(multidigrafo:nx.MultiDiGraph)` → `nx.DiGraph`:

Función que limpia el grafo de calles obtenido de OpenStreetMap y lo transforma a un digrafo.

En primer lugar, convierte el MultiDiGraph original en un DiGraph usando "*ox.convert.to_digraph*" de la librería *osmnx*, de forma que se simplifican las aristas paralelas. A continuación, elimina todos los bucles (aristas que empiezan y terminan en el mismo nodo) obtenidos gracias a "*nx.selfloop_edges*".

Después recorre todas las aristas del grafo y limpia sus atributos principales, que son los que se utilizarán en el resto de las funciones:

- Para el atributo "highway", si viene como lista, se selecciona el tipo de vía cuya velocidad máxima asociada en el diccionario "MAX_SPEEDS" sea mayor y se queda solo con ese tipo.
- Para el atributo "name", si es una lista, se elige el primero de la lista.
- Para el atributo "maxspeed", si no existe, está vacío o es una lista vacía, se asigna la velocidad que se corresponde al tipo de vía usando "MAX_SPEEDS". Si "maxspeed" es una lista, se convierten todos los elementos a "float" y se toma el máximo. En el caso de que sea un número, se trata de convertir y si no se puede, se vuelve a recurrir al valor del diccionario.

4.8. `dibujar_grafo_nx(G:nx.DiGraph, min_long: float)`:

Esta función sirve para mostrar el grafo dirigido del callejero de Madrid. Esto lo consigue calculando la dirección y distancia entre cada par de nodos conectados, así como la dirección y añade todo a unas listas. Mas adelante, utiliza estas listas para crear un quiverplot gracias a la función **quiver** de la librería matplotlib.

La variable min_long controla la longitud mínima de las flechas mostradas.

5. CONCLUSIONES

El desarrollo de la práctica ha permitido aplicar los conceptos de la teoría de grafos impartidos en clase a la implementación de un sistema de navegación.

La elaboración del sistema formado por el callejero, el procesamiento de grafos y las funciones de búsqueda ha permitido implementar de forma directa los conceptos de la teoría de grafo impartidos en clase, así como conceptos en análisis de datos. La librería "callejero.py" ha servido como puente entre datos reales, que pueden tener fallos tanto de formato como de contenido, y su representación computacional, incluyendo funciones de conversión de coordenadas, normalización de direcciones y búsqueda de coincidencias completas y *fuzzy* para poder tolerar errores. A su vez la descarga y limpieza del grafo de OpenStreetMap ha mostrado que, aunque parece sencillo, es de vital importancia realizar una buena limpieza de atributos, eliminar bucles y unificar información desigual en campos que pueden tener inconsistencias.

El módulo "gps.py" ha servido como componente interactivo entre el usuario y el programa. Permitiendo al usuario introducir distintas direcciones y elegir el criterio de peso a la hora de buscar la ruta optima. Esto ha sido clave para comprobar el funcionamiento del sistema, pero también aporta una utilidad real, al conseguir un uso que no necesita prácticamente explicación del sistema.

Finalmente, la librería "grafo_pesado.py" ha sido la encargada de aplicar los algoritmos de Dijkstra, Prim y Kruskal aprendidos en clase. Además, se ve como diferentes funciones de peso puede llegar a modificar por completo la ruta obtenida. Para crear las diferentes funciones de peso siempre se ha querido minimizar, para cualquier modo de navegación. En el caso de "mas_corto", se minimiza la longitud que se recorre, sin importar el tiempo que pueda tardar, por lo que se toma la longitud en metros como peso. Por otro lado, en "mas_rapido" se minimiza el tiempo que se tarda en llegar al destino, y por tanto se toma el tiempo en segundos que lleva recorrerse ese tramo como peso. Y finalmente, en "mas_rapido_semaforos" se minimiza también el tiempo en recorrer cada tramo, pero teniendo en cuenta que hay una probabilidad de 0.8 de que al pasar por un vértice se detenga 30 segundos. Para tener esto en cuenta, sumamos un tiempo extra esperado de $0.8 \times 30 = 24$ segundos a cada arista, haciendo que se penalice pasar por mayor número de aristas.

Durante la creación de las funciones se han encontrado problemas que se han tenido que ir solventando, como encontrar una forma eficiente de mostrar la dirección de las aristas cuando se trata de un grafo del tamaño de todo Madrid.

En conjunto, esta práctica ha permitido integrar teoría de grafos, manipulación de datos y técnicas de búsqueda para construir un sistema capaz de localizar direcciones y calcular sus rutas.

No obstante, este programa presenta limitaciones. Depender de los datos de OpenStreetMap implica confiar en datos incompletos o inconsistentes, que a su vez

MEMORIA PRÁCTICA 2 - G08B

puede fallar a la hora de intentar cargar los datos. Del mismo modo, el uso del *fuzzy search*, aunque permite tolerancia, puede devolver emparejamientos no esperados. Todo ello hace que el proyecto sea muy útil a nivel académico, pero no comparable con sistemas de navegación completos.

6. BIBLIOGRAFÍA

Alfaya, D. (s.f.). *Transparencias Teoría de Grafos*.

Alfaya, D. (s.f.). *Transparencias Algoritmos de recorrido de grafos*.