COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI
DEPARTAMENTO DE MATEMÁTICA APLICADA

TITULACIÓN

Grado en Ingeniería Matemática e Inteligencia Artificial

MATEMÁTICA DISCRETA

Curso 2024-2025

PRÁCTICA 3: GPS

OBJETIVOS GENERALES

- Comprender la aplicación de la teoría de grafos a la modelización y resolución de problemas del mundo real, en este caso, al problema de determinar una ruta en un GPS.
- Construir una librería "grafo_pesado.py" en la que se implementen distintos algoritmos para el análisis de grafos pesados.
- Usar dicha librería para construir un sistema de navegación "gps.py" capaz de buscar dos direcciones en el callejero de Madrid y recuperar la ruta más corta o más rápida entre ellos.

DESCRIPCIÓN DEL PROYECTO

El producto final que se desarrollará con el segundo proyecto es un navegador que permita trazar rutas e indicar direcciones a un conductor desde una dirección de Madrid a otra. El navegador construirá un grafo de intersecciones de calles utilizando OpenStreetMap y permitirá trazar rutas entre las direcciones del callejero de Madrid disponibles en la web de datos abiertos del Ayuntamiento de Madrid (datos.madrid.es).

PARTE 1: INSTALACIÓN DE LIBRERÍAS Y PLANTILLAS

En esta práctica vamos a utilizar las siguientes dos librerías de Python:

- NetworkX: Librería para la gestión, análisis y visualización de grafos (enlace al manual).
- OSMnx: Librería para la descarga, análisis y visualización de redes de calles y otros datos geoespaciales de OpenStreetMap en formato NetworkX (enlace al manual).

En Anaconda, crea un entorno e instala con pip los *requirements* dados en el fichero "requirements_gps.txt" que encontrarás en Moodle, haciendo:

pip install -r requirements_gps.txt

Para poder probar la práctica en los laboratorios (o en el aula virtual), puedes hacer lo siguiente una vez conectado a un laboratorio:

1. Ejecuta: "conda init" en cualquier terminal para habilitar conda activate.
2. Cierra y abre de nuevo la terminal.
3. Ejecuta: "conda activate icai".
4. Descarga el fichero "requirements_gps.txt" y colócalo en tu carpeta de trabajo.
5. Abre una terminal en dicha carpeta y ejecuta: "pip install -r requirements_gps.txt".

Es obligatorio probar correctamente todo el código y verificar que puede ejecutarse en los laboratorios antes de entregarlo.

Descarga también las plantillas "callejero.py" y "grafo_pesado.py" que encontrarás en Moodle.

PARTE 2: RECOPILACIÓN DE DATOS DEL CALLEJERO OFICIAL DE MADRID

Para el GPS, vamos a descargar y utilizar los datos oficiales del callejero vigente de Madrid del Ayuntamiento de Madrid. Comenzaremos la práctica descargando estos datos y procesándolos de forma apropiada para poder utilizarlos.

1. Accede al siguiente enlace de la web de datos del Ayuntamiento de Madrid.

2. Descarga el fichero "Relación de direcciones vigentes, con coordenadas" en formato CSV y guárdalo como "direcciones.csv".

3. Carga en un dataframe las columnas "VIA_CLASE", "VIA_PAR", "VIA_NOMBRE", "NUMERO", "LATITUD" y "LONGITUD" del fichero "direcciones.csv". Éstas contienen el tipo de clase (ej. Calle, Paseo, etc.), la partícula (ej. de, de la, del, etc.), el nombre (sin acentos y en mayúsculas), el número, la latitud y la longitud respectivamente de una dirección vigente en el callejero oficial de Madrid.

4. En el fichero, la LATITUD y LONGITUD están expresadas como cadenas de texto que representan la posición geográfica de cada dirección en grados minutos y segundos, seguidos de una orientación (N, S, E y W según el caso). Convierte estas cadenas en dos floats que representen la latitud y longitud en grados con decimales, siguiendo la convención de que las latitudes N y las longitudes E son positivas y que las S y W son negativas. Por ejemplo, esta función convertirá "3º42'24.69" W" en el float -3.7058583. Para ello, recuerda que 1 grado son 60 minutos y que 1 minuto son 60 segundos.

5. Escribe una función de python que realice todo el proceso anterior (y cualquier otro procesado adicional que consideres necesario realizar en el dataframe para desarrollar los siguientes apartados de la práctica) y devuelva como resultado el dataframe procesado.

6. Escribe una función que reciba como entrada una cadena de texto que representa una dirección en el siguiente formato:

"Calle de Alberto Aguilera, 1"

y el dataframe procesado anteriormente y devuelva la latitud y longitud de la dirección pedida si ésta existe en el dataset, o una excepción ".

7. **(Opcional)** Haz que la función anterior realice la búsqueda utilizando una comparación de cadenas aproximada ("fuzzy search") en lugar de una búsqueda exacta.

PARTE 3: PRÁCTICA PRESENCIAL DE NETWORKX

1. Abre un notebook de jupyter (networkx.ipynb) e importa las librerías networkx y matplotlib.pyplot.

2. Crea un grafo no dirigido de NetworkX $G = (V, A)$, con

$$V = \{1, 2, 3, 4, 5\}$$

$$A = \{\{1, 2\}, \{1, 4\}, \{3, 4\}, \{2, 4\}, \{3, 2\}, \{3, 5\}\}$$

3. Elimina el vértice 5 del grafo.

4. Utilizando *networkx+pyplot*, representa gráficamente el grafo G.

5. Crea un grafo decorado de NetworkX $G_2 = (V_2, A_2)$, con

$$V = \{a, b, 10, 11, 3.14\}$$

$$A = \{\{a, 10\}, \{a, 3.14\}, \{b, 11\}\}$$

en el que a las aristas se les ha dado la siguiente información

- La arista $\{a, 10\}$ tiene como datos el texto "Una cadena" y peso 1.1.
- La arista $\{a, 3.14\}$ tiene como datos el texto "Otra cadena" y peso 1.2.
- La arista $\{b, 11\}$ tiene como datos el texto "Un objeto" y peso 5.7.

y representarlo usando *networkx+pyplot*.

6. Construye un grafo dirigido de NetworkX $G_d = (V, A)$ con

$$V = \{1, 2, 3, 4\}$$

$$A = \{(1, 2), (2, 1), (1, 4), (3, 4), (2, 4), (4, 2), (3, 2)\}$$

y represéntalo usando *networkx+pyplot*.

7. Calcula los grados de todos los vértices de G y G_2, así como los grados entrantes y salientes de G_d.

8. Calcula las componentes conexas del grafo G_2 y calcula las componentes conexas y fuertemente conexas del grafo G_d.

9. Calcula el árbol abarcador mínimo de G y represéntalo gráficamente sobre el grafo.

10. Calcula el camino más corto en G entre los vértices 1 y 3 y represéntalo gráficamente sobre el grafo.

PARTE 4: RECONSTRUCCIÓN DEL GRAFO DE CALLES Vamos a utilizar la librería OSMnx para crear un grafo NetworkX con las calles de Madrid a partir de los datos de OpenStreetMap. Buscamos construir un grafo $G = (V, A)$ con las siguientes propiedades:

- Los vértices se corresponden con los cruces de las calles. Hay un vértice por cada coordenada geográfica $(latitud, longitud)$ donde exista un cruce, donde la latitud y longitud están expresadas en grados en coma flotante. Nótese que es posible que varias calles se crucen en el mismo punto (por ejemplo, en una rotonda) y cada vértice representa, por tanto, el conjunto de todos los cruces de calles que suceden en ese punto geográfico.
- Las aristas unen dos cruces consecutivos de cada calle.
- El sentido de cada arista será el sentido de circulación en dicha calle. Si hay dos sentidos posibles, habrá una arista en cada dirección.
- Los vértices y las aristas contendrán, además, información adicional sobre las calles, como su longitud o la velocidad máxima por la que se puede circular en ellas.

Los caminos en este grafo serán, por lo tanto, las rutas entre distintas intersecciones de las Calles de Madrid.

Se pide escribir un código Python que, usando las librerías OSMnx y NetworkX, construya y represente gráficamente un grafo con las propiedades anteriores.

Para ello, se deberá comenzar haciendo los siguientes pasos:

1. Escribe una función que, usando la función `graph_from_place` de OSMnx, descargue en un grafo NetworkX la información de OpenStreetMap de "Madrid, Spain" de la red de carreteras ("network_type='drive'"). Para evitar descargas innecesarias (y posiblemente lentas) de los servidores de OpenStreetMap, utiliza las funciones `save_graphml` y `load_graphml` de OSMnx para guardar el grafo descargado en un fichero "madrid.graphml". La función deberá comprobar si este fichero existe. Si no existe, deberá descargar la información, crear el grafo y guardarlo. Si existe, deberá leer la información del grafo del fichero.

   Nota: Tras descargarlo, puedes pintar el grafo obtenido usando la función `plot_graph` de OSMnx, ajustando adecuadamente el tamaño de los nodos (se recomienda 0) y el grosor de las aristas (se recomienda 0.5).

2. El grafo descargado por defecto de OSMnx es un multidigrafo, en el que pueden existir varias aristas "paralelas" entre dos intersecciones consecutivas (por ejemplo, vías laterales de la misma calle) y bucles (por ejemplo, calles que vuelven sobre sí misma). Para simplificar el problema de navegación, trabajaremos, en su lugar, en el grafo dirigido sin bucles subyacente. Escribe una función que convierta el grafo devuelto por OSMnx en un grafo dirigido sin bucles de NetworkX (Digraph).

   Para ello, puedes utilizar (entre otras) las siguientes funciones de OSMnx y NetworkX:

   - `convert.to_digraph`: Función de OSMnx que convierte un multidigrafo en un grafo dirigido con bucles
   - `selfloop_edges`: Función de NetworkX que devuelve la lista de bucles de un grafo dirigido.

3. Usando NetworkX y el manual de referencia de OSMnx, explora los datos almacenados como contenido de los nodos y las aristas del grafo que has construido. Usando la información geográfica de la posición de cada intersección contenida en el grafo (atributos "x" e "y" de cada nodo) y las funciones de dibujo de NetworkX, dibuja el grafo dirigido obtenido de la función anterior.

PARTE 5: FUNCIONES DE BÚSQUEDA

Completar en la librería "grafo_pesado.py" el código de las siguientes funciones:

1. `dijkstra`: Cálculo de un árbol de caminos mínimos a partir de un vértice dado usando el algoritmo de Dijkstra. La función "dijkstra" recibirá un grafo (posiblemente dirigido), un vértice y una función de pesos para el grafo y devolverá un diccionario que indicará el padre de cada vértice del árbol de distancias mínimas de la componente conexa a la que pertenece el vértice.

2. `camino_minimo`: Cálculo del camino más corto entre dos vértices en función de la función de pesos indicada para las aristas del grafo. El resultado de la función "camino_minimo" se dará como una lista de los vértices por los que pasa el camino en orden desde el origen al destino.

3. `prim`: Cálculo de un árbol abarcador mínimo (de hecho, en este caso, podría ser bosque abarcador si hay varias componentes conexas) usando el algoritmo de Prim. Al igual que con "dijkstra", la función "prim" devolverá un diccionario que indicará el padre de cada vértice del bosque abarcador.

4. `kruskal`: Cálculo de un árbol abarcador mínimo de un grafo usando el algoritmo de Kruskal. Por simplicidad, en este caso la función "kruskal" devuelve una lista $[(u_1, v_1), (u_2, v_2), (u_3, v_3), \ldots, (u_k, v_k)]$ con las aristas (u_i, v_i) que pertenecen a dicho árbol.

En todas estas funciones, se pasa como argumento un grafo o grafo dirigido (objetos `Graph` o `DiGraph` de NetworkX respectivamente) y una función de coste que indica los pesos de cada arista de estos grafos o grafos dirigidos. Estás funciones de pesos son de tipo

```
Callable[[nx.Graph,object,object],float]
```
o
```
Callable[[nx.DiGraph,object,object],float]
```

Esto significa que, al llamarlas, se les pasa como argumento una función que

- Recibe tres parámetros: un grafo o grafo dirigido de NetworkX y dos objetos (vértices)
- Devuelven un `float` correspondiente al peso de la arista

Por ejemplo, una posible definición para una de estas funciones de coste sería:

```
def mi_peso(G:nx.Graph,u:object, v:object):
        return G[u][v]['valor']
```

donde 'valor' es un cierto campo de los datos almacenados en la arista en el grafo G.

Nota: No se permite llamar directamente a las funciones de cálculo de caminos o árboles abarcadores del propio NetworkX para realizar estos algoritmos. Se pide implementarlos a partir de los pseudocódigos del tema 6 del curso.

PARTE 6: NAVEGADOR

Usando el código desarrollado anteriormente, se pide desarrollar una aplicación "gps.py" que haga lo siguiente:

1) Al arrancar, leerá el fichero "direcciones.csv" y construirá el grafo de calles de la parte 4.

2) Permitirá al usuario seleccionar dos direcciones (origen y destino) de la base de datos de direcciones ("direcciones.csv"). Se valorará positivamente que la forma de seleccionar estas direcciones sea lo más natural posible para el usuario. Por simplicidad, se tomarán como puntos de origen y destino en el grafo los vértices más cercanos a la ubicación de las direcciones pedidas.

3) Permitirá elegir al usuario uno de los siguientes modos de cálculo de ruta:

   a) Ruta más corta entre esas dos direcciones: Se considera la longitud total en metros del camino recorrido.

   b) Ruta más rápida entre esas dos direcciones: Se considera el tiempo de viaje del camino recorrido, asumiendo que se recorre cada calle a la velocidad máxima permitida por la vía.

   c) Ruta más rápida, optimizando semáforos: Se considera un modelo en el que cada vez que se atraviesa un cruce de calles existe una probabilidad $p = 0.8$ de que el usuario deba detenerse durante 30s por un semáforo o algún otro tipo de señal vial y en el que circula el resto del tiempo a la velocidad máxima de la vía. Se dará la ruta que minimice el tiempo esperado de viaje según este modelo.

4) Usando una función de peso adecuada, en función de lo elegido en el punto (3), se calculará el camino mínimo desde el origen al destino. Para ello, se deberán usar las funciones programadas en grafo_pesado.py.

5) La aplicación analizará dicho camino y construirá una lista de instrucciones detalladas que permitan a un automóvil navegar desde el origen hasta el destino.

6) Finalmente, usando NetworkX, se mostrará la ruta elegida resaltada sobre el grafo del callejero.

7) Tras mostrar la ruta, se volverá al punto 2 para seleccionar una nueva ruta hasta que se introduzca un origen o destino vacíos.

Las velocidades máximas de cada vía se obtendrán a partir de la información contenida en las aristas del grafo de OpenStreetMap. En caso de que alguna vía no cuente con dicha información, se asumirá que se siguen las siguientes velocidades máximas, en función del tipo de calle indicado en OpenStreetMap:

Tipo de vía en OSM (highway)	**Velocidad máxima**
living_street	20
residential	30
primary_link, unclassified, secondary_link o trunk_link	40
trunk, primary, secondary, tertiary, tertiary_link o busway	50
motorway_link	70
motorway	100
Cualquier otro tipo	50

Esta tabla está resumida en un diccionario y una constante disponibles en "callejero.py".

A la hora de dar las instrucciones, se valorará que sean lo más precisas posibles. Como mínimo, se deberá indicar:

- Cuánta distancia (en metros) se debe continuar por cada vía antes de tomar un giro hacia otra calle.
- Al tomar un desvío, cuál será el nombre de la siguiente calle por la que se deberá circular.
- A la hora de girar, si se debe girar a la izquierda o a la derecha. Opcionalmente, si hay un cruce múltiple, se precisará por qué salida debe continuarse.
- El navegador no debería dar instrucciones redundantes mientras se continúe por la misma calle (más allá de continuar por dicha calle X metros).

ENTREGAS

Como resultado final de la práctica se entregará

- El código desarrollado, apropiadamente comentado y documentado (ver guía de estilo PEP8 en Moodle):
  - La librería "grafo_pesado.py".
  - La librería "callejero.py".
  - El programa "gps.py".
  - Cualquier otra librería, script o fichero de configuración adicional que haya sido programada por los alumnos para el desarrollo de la práctica y que sea necesario para ejecutar los scripts. **Debido a su peso, no incluir los datasets originales "direcciones.csv" ni el grafo "madrid.graphml" en la entrega**
- Una memoria en PDF con el nombre P3GPxxx.pdf (donde GPxxx es el número de grupo de la pareja) en el que se especificarán, como mínimo, los siguientes elementos:
  - Nombre y apellidos de los integrantes del grupo y código del grupo de prácticas (GPxxx).
  - Descripción de la estructura general del programa "gps.py", indicándose si se han implementado módulos adicionales y cómo se ha organizado el código en las mismas, además de las estructuras de datos creadas para almacenar la información del problema, si las hubiera.
  - Detalles sobre cómo se ha resuelto el problema de la recuperación de la ruta óptima y de las direcciones de navegación. En concreto, se deberá explicar detalladamente cómo se han elegido las funciones de coste para que el programa pueda calcular cada uno de los 3 tipos de ruta solicitados y cómo se decide la forma de dar las instrucciones de navegación a lo largo del camino.
  - Cualquier comentario que se considere relevante sobre el diseño de la implementación.
  - Bibliografía, incluyendo referencias debidamente citadas a cualquier fuente utilizada para desarrollar la práctica.

Todos los ficheros anteriores se cargarán en la tarea correspondiente de Moodle. Salvo que sea imprescindible, **los ficheros deben subirse individualmente a Moodle, no en una carpeta comprimida**.

CRITERIOS DE EVALUACIÓN

- **Librería "grafo_pesado.py" (2 puntos):** Se puntuará que todos los métodos solicitados estén implementados y funcionen correctamente. Es obligatorio que el script "test_grafo.py" se ejecute sin errores para la librería entregada.
- **Reconstrucción del grafo del callejero y recuperación de direcciones (2 puntos):** Se puntuará que el código presentado permita recuperar correctamente el grafo del callejero de Madrid y la información de dirección vigentes a partir de la información de los datasets, y se valorará positivamente que el sistema permita encontrar direcciones a partir de una entrada de texto del usuario de forma eficiente y cómoda para el usuario.
- **Navegación (4.5 puntos):** Se valorará que el programa pueda encontrar adecuadamente las calles y las rutas óptimas a partir del grafo construido y la calidad y precisión de las instrucciones de navegación y su representación.
- **Organización y calidad del código (0.5 puntos):** Se valorará el uso de una programación estructurada, siguiendo un buen estilo de programación, la modularidad y reutilización del código, así como que el código esté correctamente documentado.
- **Memoria (1 punto):** Se valorará la descripción de los algoritmos y estructuras pedidos, teniendo especialmente importancia la justificación de la elección de las funciones de coste y la descripción de los algoritmos de obtención de direcciones de navegación.