COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI
DEPARTAMENTO DE MATEMÁTICA APLICADA

TITULACIÓN

Grado en Ingeniería Matemática e Inteligencia Artificial

MATEMÁTICA DISCRETA

Curso 2023-2024

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.py" que implemente un TAD grafo y utilidades para el análisis de grafos..
- 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 a partir de los datos públicos del callejero de Madrid disponibles en la web de datos abiertos del Gobierno (datos.gob.es) y permitirá trazar rutas entre las direcciones del callejero.

PARTE 1: TAD GRAFO

Partiendo de la plantilla "grafo.py" disponible en Moodle, se realizará una implementación del tipo abstracto de datos (TAD) grafo que permita representar tanto un grafo dirigido como un grafo no dirigido. Para ello, se creará una clase "Grafo".

Siguiendo con el paradigma de que un grafo es un par de conjuntos $G = (V, A)$, sin importar qué tipo de elementos contengan dichos conjuntos, los vértices del grafo podrán ser cualquier objeto dado por el usuario de la librería (siempre que sea "hasheable", requisito para que pueda usarse como clave en un diccionario).

Las aristas se identificarán a partir de un par de objetos de V (par ordenado o no según el tipo de grafo). El grafo deberá ser capaz de almacenar, para cada arista, un objeto de datos proporcionado por el usuario y un real (float) que indicará el peso o coste de dicha arista en el grafo (y que se utilizará más adelante para los algoritmos de búsqueda de caminos y árboles abarcadores mínimos en el grafo).

El TAD incluirá las siguientes funciones (la descripción completa de los parámetros de entrada y salida de cada función se encuentran documentados en la plantilla grafo.py):

- **Inicialización:** Se creará un grafo dirigido o no dirigido en función de lo indicado por el usuario.
- **es_dirigido:** Indica si el grafo es dirigido o no.
- **agregar_vertice:** Toma un objeto proporcionado por el usuario y lo agrega como vértice del grafo.
- **agregar_arista:** Crea una arista entre los vértices asociados a dos objetos s y t dados por el usuario y le asocia un objeto de datos y un peso dados por el usuario. Si no existen los objetos, no hace nada.
- **eliminar_vertice:** Elimina el vértice asociado al objeto v proporcionado por el usuario. Si el vértice tenía aristas entrantes o salientes, también las elimina. Si no existe tal vértice, no hace nada.
- **eliminar_arista:** Elimina una arista entre dos vértices dados. Si no existe tal arista, no hace nada.
- **obtener_arista:** Devuelve una tupla con el objeto de datos y el peso asociados a la arista existente entre dos vértices dados. Si no existe tal arista, devuelve None.
- **lista_vertices:** Devuelve una lista con los vértices del grafo.
- **lista_adyacencia:** Lista de vértices adyacentes a un vértice dado. Si no existe tal objeto, devuelve None.

De forma general, estas funciones no necesitarán devolver excepciones, a menos que se intente utilizar como vértice un objeto que no pueda ser usado como clave en un diccionario (no "hasheable"), en cuyo caso, al intentarlo, Python levantará una excepción TypeError. En su lugar, las funciones podrán devolver como retorno None en caso de no realizar la tarea pedida. Se utilizará una implementación basada en el almacenamiento de las listas de adyacencia del grafo. La plantilla "grafo.py" proporcionada contiene ya los campos de la clase y su inicialización, y no deben ser modificados. La clase contiene dos campos:

- **dirigido:** Valor lógico que es verdadero si el grafo es dirigido y falso si no lo es.
- **adyacencia:** Diccionario que almacena la adyacencia del grafo de la siguiente forma:
  - Sus claves son los vértices del grafo.
  - Para cada clave u, adyacencia[u] es un diccionario cuyas claves son los vértices adyacentes a u.
  - Dado un vértice u y un adyacente $v \in N_u$, adyacencia[v][w] es el contenido de la arista $\overline{uv}$, que es un par (a, w), donde
    - a es un objeto usado para almacenar datos en la arista
    - w es un número real que representa el peso de la arista

En el caso de grafos no dirigidos, se almacenará las aristas por duplicado: las aristas no dirigidas $\{u, v\}$ se almacenarán como un par de aristas dirigidas (u, v) y (v, u) con los mismos datos.

Además de estas funciones básicas del TAD, se implementarán funciones:

- **grado_saliente**
- **grado_entrante**
- **grado**

que calculan el grado de los vértices del grafo, tanto en el caso dirigido como no dirigido. Se proporciona un script "test_grafo.py" para probar las funcionalidades básicas del TAD.

PARTE 2: RECONSTRUCCIÓN DEL GRAFO DEL CALLEJERO DE MADRID

Accede al siguiente enlace de la web de datos del Gobierno y descarga los siguientes datasets:

- Descarga el fichero "Callejero: información adicional asociada. Cruces de viales con coordenadas geográficas" en formato CSV y guárdalo como "cruces.csv".
- Descarga el fichero "Callejero: información adicional asociada. Direcciones postales vigentes con coordenadas geográficas" en formato CSV y guárdalo como "direcciones.csv".

El primer fichero contiene la información de los cruces existentes entre las calles del callejero de Madrid.
El segundo contiene las direcciones actualmente vigentes pertenecientes a dichas calles.
El campo "Código de vía" de "direcciones.csv" está en correspondencia con los campos "Código de vía tratado" y "Código de vía que cruza o enlaza" de "cruces.csv", e identifica de forma unívoca los viales del callejero.

Buscamos construir un grafo $G = (V, A)$ con las siguientes propiedades:

- Por simplicidad, consideraremos que el grafo es no dirigido y que todas las calles son de doble sentido.
- Los vértices se corresponden con los cruces de las calles. Hay un vértice por cada coordenada geográfica (x, y) donde exista un cruce, con x, y en centímetros, obtenidos de los campos "Coordenada X (Guia Urbana) cm" y "Coordenada Y (Guia Urbana) cm" del dataset. 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. Se ha proporcionado un fichero "callejero.py" en el que existen dos clases "Cruce" y "Calle". Los objetos de tipo "Cruce" será lo que se usará como vértices del grafo.
- Las aristas unen dos cruces consecutivos de cada calle.

Se pide escribir un código python que, usando la librería *grafo.py* del anterior apartado, construya un grafo con las propiedades anteriores. Cada arista almacenará además los datos de la calle a la que corresponda y cada vértice almacenará los datos de la intersección que cada equipo considere necesario para la realización del navegador descrito en la parte 4 del proyecto.

Nota: El conjunto de datos de cruces no contiene la totalidad de las calles de la Comunidad de Madrid (incluye principalmente calles del Ayuntamiento de Madrid). Además, la geometría de algunas calles podría ser compleja de deducir si se utiliza exclusivamente este dataset. Considérese, por tanto, que en este ejercicio se busca únicamente una aproximación razonable del callejero real.

Para ello, se deberá comenzar haciendo los siguientes pasos:

1. Partiendo del código desarrollado en la práctica 4 de la asignatura de Adquisición de Datos, escribid una función que limpie y prepare los datasets "direcciones" y "cruces" para su uso.

2. Completad la clase "Cruces" disponible en la plantilla "callejero.py" con la información que se necesite almacenar sobre cada cruce para poder reconstruir el grafo. En particular, se deberá, como mínimo, disponer de alguna forma de guardar dentro del objeto la lista de calles que concurren en el cruce.

3. Completad la clase "Calle" disponible en la plantilla "callejero.py" con la información que se necesite almacenar sobre cada calle para poder reconstruir el grafo y realizar la navegación. En particular, se deberá, como mínimo, disponer de alguna forma de guardar dentro del objeto la listas de cruces y de direcciones que pertenecen a dicha calle.

4. Recorred el dataset "Cruces" y recuperad una lista sin repetidos de las coordenadas (x, y) de los cruces. Algunos cruces están muy cerca unos de otros y, en realidad, corresponden a una misma intersección física, pero representada en el dataset como cruces distintos desde varios extremos (por ejemplo, esto sucede cuando hay una rotonda; las salidas están representadas como varios cruces con la calle principal). Escribid una función que, dado un radio R, permita detectar grupos de puntos que estén todos a menos de R cm de uno de ellos y que unifique dichos grupos de puntos en un único punto en su centro.

5. Recorred el dataset "Cruces" y recuperad una lista sin repetidos de todos los códigos de calle que aparecen como vía o vía que cruza en el dataset. A partir de los datos tanto de "cruces" como de "direcciones" correspondientes a cada código de calle único recuperado, cread un objeto "Calle" para cada código.

6. Para cada coordenada (x, y) de cruces hallada, construid un objeto "Cruce" para esa coordenada, con la información de las calles que se cortan en dicho cruce y cualquier otra información que se considere relevante.

7. Agregad cada objeto Cruce creado como vértice a un grafo.

8. Diseñad un algoritmo que permita decidir qué cruces deben estar conectados por aristas en el grafo del callejero y usarlo para completar la reconstrucción del grafo. En los objetos de datos asociados las aristas del grafo se agregará toda la información que se considere necesaria, tanto para la propia reconstrucción del grafo, como, posteriormente, para la navegación.

PARTE 3: PRÁCTICA PRESENCIAL DE NETWORKX

1. Abre un notebook de jupyter (networkx.ipynb) e importar 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.

11. Implementar en la clase *Grafo* de la librería *grafo.py* un método $convertir\_a\_NetworkX$ que transforme un grafo en un grafo o grafo dirigido de NetworkX según corresponda, preservando los atributos de los nodos y aristas correspondientes.

PARTE 4: FUNCIONES DE BÚSQUEDA

Agregar a la clase Grafo de la librería "grafo.py" 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 vértice 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 los pesos almacenados 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.

Nota: No se permite llamar directamente a librerías de grafos como NetworkX para realizar estos algoritmos. Se pide implementarlos a partir de los pseudocódigos del tema 6 del curso.

PARTE 5: NAVEGADOR

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

1) Al arrancar, leerá los ficheros "cruces.csv" y "direcciones.csv" y construirá el grafo de calles de la parte 2. Se crearán dos grafos: uno en el que el peso de cada arista sea la distancia euclídea entre los nodos y otro en el que el peso sea el tiempo que tarda un coche en recorrer dicha arista a la velocidad máxima permitida en dicha calle.

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.

3) Permitirá elegir al usuario si desea encontrar la ruta más corta o más rápida entre estos puntos.

4) Usando el grafo correspondiente 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.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.

Por simplicidad, se asumirá que todas las calles con el mismo tipo de vía tienen la misma velocidad máxima, según lo indicado en la siguiente tabla:

Clase de la via	**Velocidad máxima**
AUTOVIA	100
AVENIDA	90
CARRETERA	70
CALLEJON o CAMINO	30
ESTACION DE METRO, PASADIZO, PLAZUELA o COLONIA	20
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.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 "cruces.csv" ni "direcciones.csv" 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).
  - Explicación detallada de cómo se ha resuelto el problema de reconstruir el grafo de calles a partir de los datos de los dataset.
  - 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 (además de "Grafo").
  - Detalles sobre cómo se ha resuelto el problema de la recuperación de la ruta óptima y de las direcciones de navegación.
  - 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.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 (2.5 puntos):** Se puntuará que el código presentado permita recuperar una aproximación razonable del grafo del callejero de Madrid a partir de la información de los datasets.
- **Navegación (4 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 que la descripción de los altorimos y estructuras pedidos, teniendo especialmente importancia la descripción de los algoritmos de reconstrucción del grafo y de la obtención de direcciones de navegación.