TRABAJO PRÁCTICO 3: MATEMÁTICA DISCRETA

CURSO 2022-2023

**Memoria**

**GPS**

Autores:

**Lucía Prado Fernández-Vega**

**Lara Ocón Madrid**

**GP02**

Madrid, enero, 2023

# Introducción

Este documento expone la memoria de la práctica final del GPS, en la que se aplica la teoría de grafos a la modelización y resolución de la generación de rutas optimas mediante el diseño de un GPS. Para implementar este programa, el GPS cuenta con una librería ‘*grafo.py’* que implementa un TAD grafo y utilidades para el análisis de este, y un sistema de navegación ‘*gps.py’* que se emplea como interfaz principal para permitir al usuario la búsqueda de la ruta más corta o más rápida entre dos direcciones del callejero de Madrid.

.

# Implementación del TAD en grafo.py

A la hora de implementar el TAD generado en la librería grafo.py hemos creado dos grafos. Ambos guardan las mismas aristas y vértices con la misma información pero con una pequeña diferencia, en uno consideramos el peso de las aristas como la distancia de un vértice a otro, y en el segundo, consideramos que el peso es el tiempo que se tarda en recorrer dicha arista (longitud de la arista partido de la velocidad máxima de circulación permitida en dicha arista).

De esta forma a la hora de comenzar a recorrer el csv de cruces, hemos creado dos grafos empleando la librería de *grafo.py*, que se basa en el uso de lasi siguientes clases y métodos:

## Clase Arista

Primero, definimos la clase arista donde almacenaremos la información acerca de estas. Sus atributos serán: origen (vértice origen), destino (vértice destino), data (información acerca de la arista, puede ser un diccionario con información acerca de la distancia, velocidad… de la arista) y peso (peso de la arista).

## Clase grafo

En segundo lugar, se define la clase grafo que consta de dos atributos: dirigido, que define si este se trata de un grafo dirigido, en el que las aristas tienen asociadas una dirección especifica, o no dirigido, en el que cada arista se puede recorrer en ambas direcciones indistintamente; el atributo vértices contiene un diccionario de vértices en el que las claves son objetos (los vértices), y los valores son listas de aristas donde cada arista es un objeto de la clase Arista, definida en el anterior punto.

## Métodos básicos del TAD

Dentro de la clase grafo, hemos definido los siguientes métodos.

### Es\_dirigido

Esta función devuelve un booleano: True si es un grafo dirigido y False si no lo es.

### Agregar vértice

Función que agrega el vértice como una nueva clave en el diccionario de vértices asociado al grafo. Dado que es un vértice nuevo, los valores para dicha clave los inicializa como una lista vacía, para más adelante ir añadiendo las aristas.

### Agregar arista

Si ambos vértices introducidos como argumentos de la función se encuentran dentro del grafo que se está construyendo (es decir, los dos vértices son claves del diccionario de vértice), se agrega dicha arista a la lista de aristas asociada al vértice.

En caso de que el grafo no sea dirigido, además de agregar la arista desde u hasta v, agrega la arista por duplicado, pero en el otro sentido, es decir, de v hasta u.

Si alguno de los vértices no pertenece al grafo, la función no hace nada.

### Eliminar vértice

Como argumento se introduce el vértice que se desea eliminar, y dicha función elimina el vértice del diccionario y todas sus aristas adyacentes asociadas al resto de vértices del grafo.

Dado que en el grafo guardamos los vértices en un diccionario donde la clave es el vértice y el valor, todas las aristas que salen de este, simplemente eliminamos dicha clave. Para ocuparnos de las aristas en las que el vértice no es origen si no destino, recorremos el resto de vértices del grafo y eliminamos dichas aristas en las que el destino es el vértice que tratamos de eliminar.

### Eliminar arista

Buscamos los vértices origen y destino en las claves del diccionario de vértices, en caso de que no encontrar uno o los dos, no hacemos nada. En caso de sí encontrarlos, accedemos a las aristas de dichos vértices y eliminamos la arista en cuestión. Si el grafo es no dirigido, también eliminamos la arista que va en el sentido contrario.

### Obtener arista

Su funcionamiento es parecido al de eliminar arista, la gran diferencia es que en este caso, en lugar de eliminar la arista una vez la encuentra, devuelve en una tupla su información (arista.data y arista.peso).

# Función iniciar gps.py (fase i)

Este apartado expone el modo de reconstrucción del grafo del callejero de Madrid empleado en el programa de *gps.py* a partir de los datos aportados por los ficheros del callejero (*direcciones.csv y cruces.csv: la dirección de acceso a dichas fuentes de datos se incluye en la bibliografía*).

Antes de nada, inicializamos los grafos que vamos a emplear: grafo1 (objeto grafo del TAD grafo donde consideraremos el peso de las aristas como la distancia euclídea entre sus vértices origen y destino) y grafo2 (similar al anterior, con la diferencia de que el peso de la arista es el tiempo que se tarda en llegar de un vértice a otro). Además, inicializamos el grafo G, que será el que usaremos para pintar el mapa y las rutas sugeridas con networkx.

## Clase Vértice

Antes de nada, definimos la clase vértice que vamos a emplear en grafo1 y grafo2.

La clase vértice tiene como atributos: un diccionario de calles, donde las claves son las calles (‘Código de vía tratado’) y los valores son el número de calle al que pertenece el vértice; coords, que guarda las coordenadas del vértice en cm en una tupla; y num, que usaremos como identificador único del vértice, esto nos será útil para después implementarlo con networkx.

Además, definimos una serie de métodos como \_\_hash\_\_ y \_\_eq\_\_, en las cuales definiremos que dos objetos vértice son el mismo si tienen las mismas coordenadas. Esto es importante, dado que, al usar los vértices como claves al diccionario de vértices, es importante que no diferencie dos vértices que realmente son el mismo.

## Iniciar( )

Nada más lanzar el programa gps.py, se llama a la función iniciar, que a partir de los csv de cruces y direcciones, genera los 3 grafos sobre los cuales generaremos las distintas rutas: grafo1 y grafo2 (grafos de la librería TAD grafo) y G (grafo de networkx).

Además, cronometra el tiempo que se tarda en generar dichos grafos y una vez ha finalizado imprime un mensaje por pantalla avisando al usuario de que se han generado los grafos y el tiempo que ha tardado en hacerlo.

Para generar los grafos, la función encargada de ello es generar\_grafos() que definiremos a continuación:

### Generar\_grafos()

Esta es una de las funciones más importantes del programa y una de las piezas clave.

Primero, lee los ficheros ‘cruces.csv’ y ‘direcciones.csv’ y los carga en un pandas DataFrame, para poder trabajar con ellos. En primer lugar, ordena cruces por código de vía tratado y código de vía que cruza o enlaza; y direcciones por código de vía. Esto lo hacemos para que a medida que vamos añadiendo vértices, podamos añadir todos los vértices de una calle en una tanda, y para generar las aristas simplemente tengamos que ordenar los vértices de la calle por su número dentro de la calle y enlazarlos dos a dos.

Hecho esto, limpiamos algunos valores del df de direcciones y añadimos dos columnas a dicho df. Una que es el numero de la calle en formato int, y otra que indica si el numero es par o impar. Esto lo hacemos dado que el propósito de usar el df de direcciones es para ordenar la sucesión de vértices dentro de una misma calle en función del número de calle. Sin embargo, dado que consideramos todas las aristas de doble dirección, y en la vida real puede haber dos números pares a un lado de la calle entre dos números impares del otro lado de la calle, al ordenar los vértices, puede dar lugar a calles sin sentido. Por ello, cuando vayamos a asignar el numero de calle a un vértice, consideraremos únicamente las filas dentro del de direcciones donde tengamos más números (pares o impares), puesto que sí que estamos seguros que tanto un lado de una calle como el otro, si están correctamente ordenados los números de mayor a menor.

Tratados los dfs, iniciamos las siguientes variables: vértices (diccionario donde la clave será la tupla de coordenadas de vértices y el valor, el objeto vértice), calle\_act (‘código de vía’ de la calle que estamos tratando actualmente), vel (velocidad de la calle actual), nombre\_calle (nombre completo de la calle actual), nombre\_calle\_acortado (nombre acortado de la calle actual), dict\_vertices (diccionario donde guardamos únicamente la tanda de vértices que pertenecen a la calle que estamos tratando, la clave será el número del vértice dentro de la calle y el valor será el objeto vértice), num\_vertice (numero que asignaremos a cada vértice como identificador único) y rotondas\_procesadas (lista donde guardaremos los nombres de las rotondas y plazas que iremos unificando en el df de cruces).

Una vez hecho esto, comenzamos a recorrer el csv de cruces, considerando cada fila de este df como un vértice. Antes de hacer nada, vemos si el cruce se trata de una plaza o rotonda, en ese caso, y si aún no la hemos añadido a la lista de rotonda\_procesadas, llamamos a la función unificar\_rotondas( ) que unificará todas las coordenadas de dicha rotonda/plaza.

En caso de que no se trate de una rotonda, o ya la hayamos procesado, vemos si las coordenadas del cruce ya las tenemos en el diccionario de vértices. En caso de no tenerlas

guardamos las coordenadas del cruce y el código de la primera vía del cruce, y buscamos dentro del csv de direcciones para esa calle y esas coordenadas, cual es el número de calle mas cercano a dicho cruce.

Para generar los grafos, empezamos procesando el fichero de cruces por filas procesando cada calle: si las coordenadas de dicho punto de la calle no se encuentran en el diccionario de vértices, se genera uno nuevo asociado a dichas coordenadas y se añade el vértice generado a ambos grafos y al generado por Network X. En el caso de que la coordenada procesada sea la de una glorieta, se deben unificar las coordenadas de la rotonda en la que nos encontramos. Para esto, empleamos las funciones de unificar rotondas detalladas más adelante en este mismo apartado.

Una vez las coordenadas han sido procesadas, se recorren las filas del fichero de direcciones para asociar a cada vértice las calles más cercanas seleccionando aquellas que están a la menor distancia.

Al pasar a la siguiente calle dentro del fichero de cruces, se deben ordenar los vértices de dicha calle por clave (numero) para poder generar las aristas de manera eficiente y correcta. Una vez ordenados, generamos las aristas para todos los vértices la calle actual asociando a cada una la información necesaria según el grafo generado: en el grafo 1, introducimos como peso la distancia entre el origen y destino de la arista, mientras que en el grafo 2, la distancia se divide por la velocidad asociada al tipo de vía para obtener los minutos necesarios para recorrer dicha arista. En ambos casos, las aristas se añaden como duplicados, pero variando el origen y destino ya que los grafos se van a tomar como grafos no dirigidos. Las aristas se añaden además al grafo de Network X que será empleado más adelante.

### Unificar rotondas

Esta función recibe la calle en la que nos encontramos y el fichero de cruces y devuelve las coordenadas unificadas y el fichero de cruces procesado según la glorieta en la que nos encontramos. Toma las coordenadas que componen la rotonda, y las une en una única, que será la media de todas.

### Velocidad

Finalmente, esta función simplemente asocia la velocidad al tipo de vía mediante un diccionario que tiene como claves los tipos de vía y como valores sus velocidades.

# Estructura gps.py (resto de funciones)

En este apartado se detalla la codificación del resto de las funciones empleadas en el programa de *gps.py* para generar la ruta que haya sido solicitada por el usuario, donde este introduce un origen, un destino, y un modo de búsqueda (ruta mas rápida o ruta más corta).

## Convertir a Network X

Función para generar el grafo con la librería de Network X, que permite visualizar las rutas que se obtengan en el callejero. Se añaden los vértices y aristas del grafo junto con su información (distancia, calle y numero de calle).

## Encontrar coordenadas cm grafo

La función encontrar coordenadas cm grafo recibe el grafo y las coordenadas y busca las más cercanas a las introducidas dentro del mapa procesando todos los vértices del grafo.

## Encontrar número más cercano

Recibe como argumentos el grafo siendo procesado y la calle y numero en los que nos encontramos. Busca entre las aristas del grafo las aristas con coincidencia en el nombre de la calle y las procesa para seleccionar aquella con el número más cercano.

## Encontrar coordenadas dirección grafo

Recibe el grafo y la dirección solicitada, cuyo formato debería ser el nombre de la vía y el número separados por una coma. Tras procesar la dirección introducida, se emplea la función de encontrar número más cercano para obtener el vértice más cercano a dicha dirección.

# Búsqueda de la ruta óptima

Para llevar a cabo la búsqueda de la ruta óptima y de las direcciones de navegación entre dos direcciones hemos empleado varias funciones, tanto de *grafo.py* como de *gps.py*.

## Algoritmos de búsqueda: *grafo.py*

En la librería de grafo.py, se implementan los distintos algoritmos de búsqueda que se utilizan para buscar la ruta más corta o rápida, según la solicitud del usuario.

### Algoritmo de Dijkstra

Dicho algoritmo calcula el árbol abarcador mínimo para el grafo generado, que en este caso se trata del grafo del callejero de Madrid, partiendo de la dirección de origen. Para empezar, se inicializan los diccionarios asociados a cada vértice del callejero que contienen la distancia al origen, su padre en dicho árbol abarcador mínimo, y su estado, es decir, si dicho vértice ha sido visitado o no.

A continuación, se selecciona el vértice con menor distancia al origen, y se repite esta operación en relación con cada vértice extraído en el paso anterior hasta que no queden vértices por conectar en el grafo. Mediante se extraen los vértices, se actualizan las distancias y los padres de aquellos que todavía no hayan sido extraídos del mapa.

### Camino mínimo

Esta función es muy similar a la descrita anteriormente (*Dijkstra*), con la distinción de la condición de parada, ya que el algoritmo del camino mínimo continúa generando el árbol abarcador mínimo hasta que alcanza un determinado destino. A partir del camino generado, reconstruye la ruta más corta para llegar a dicho punto desde el origen.

### Algoritmo de Prim

De nuevo, esta función permite generar el árbol abarcador mínimo del grafo pero no se basa en ningún origen introducido, sino que simplemente extrae el nodo con menor peso asociado en cada iteración y selecciona su arista asociada con menor peso dentro de la lista de aristas que se obtiene del diccionario de la clase Grafo. Los nodos se continúan recorriendo hasta que todos hayan sido extraídos, es decir, que se genera un árbol abarcador mínimo para todo el grafo.

### Algoritmo de Kruskal

Finalmente, la librería grafo.py ofrece también el algoritmo de búsqueda de Kruskal, que se basa en el del Prim para, de nuevo, generar el árbol abarcador mínimo del grafo extrayendo las aristas con menor peso asociado que sean indispensables en cada paso para la conexión de todos los vértices del grafo. Es decir, las aristas con menor peso pero que no aportan nuevos enlaces entre distintos puntos del mapa son se mantendrán inexploradas. Para ejecutar esta operación, se genera una lista que contiene todas las aristas del grafo junto con su peso y un índice para resolver bloqueos en el caso de que dos aristas tengan el mismo peso (se selecciona aquella que se haya introducido antes). A continuación, se extraen las aristas por orden de peso ascendente, asegurando que los vértices de estas no se encuentren en la misma componente.

## Navegación entre dos direcciones: *gps.py*

A partir de los vértices y aristas obtenidos mediante los algoritmos de búsqueda de la ruta optima, el programa de *gps.py* es capaz de reconstruir el camino y las direcciones para recorrer dicho trayecto apoyándose en las funciones de Network X.

### Menú

Función que muestra el menú principal del programa. Permite al usuario seleccionar el modo de introducción de la dirección (con el nombre de la calle o las coordenadas asociadas al punto en el que se encuentra) e introducir el origen y destino por pantalla.

En caso de introducir el origen y destino empleando su dirección, el formato debe ser el nombre de la calle y el número, separados por una coma. Mediante la llamada a la función encontrar coordenadas dirección grafo se obtienen los vértices más cercanos a dicha dirección. En caso de error, se notifica por pantalla y el programa vuelve a solicitar la dirección a menos que se solicite interrumpir el programa.

La solicitud de la ruta mediante coordenadas es análoga a la anterior, pero emplea la función encontrar coordenadas cm grado para obtener los vértices mas cercanos a las coordenadas introducidas.

Una vez introducidos el origen y destino de manera correcta, el usuario elige si desea obtener la ruta más corta o rápida hasta su destino. Si elige la ruta más corta, se genera el primer grafo, en el que las aristas tienen como peso la distancia entre sus vértices. En caso contrario, se genera el grafo 2, empleando en este caso el tiempo empleado para recorrer la arista como peso asociado.

A continuación, se genera la ruta solicitada empleando la función de camino mínimo de los algoritmos de búsqueda introduciendo como argumentos el origen y destino. Si no se obtiene ninguna ruta, se notifica al usuario. Si hay éxito, se cargan las instrucciones de la ruta (generadas por la función *cargar instrucciones ruta*) y el usuario puede elegir si mostrarlas por pantalla o almacenarlas en un archivo. En cualquier caso, también se generará un mapa en el que se marca la ruta generada (*imprimir mapa ruta*).

### Cargar instrucciones ruta

Dada una sucesión de vértices que componen la ruta, imprime las instrucciones para ir de cada vértice a otro. Para ello, recorre la ruta y obtiene la arista que conecta a cada par de vértices consecutivos que deberá seguir el usuario para completar la ruta. Para no ser redundantes, indica cuantos metros

### Imprimir mapa ruta

# Bibliografía

* <https://datos.gob.es/es/catalogo/l01280796-callejero-informacion-adicional-asociada-codigos-postales-zonas-ser-categoria-fiscal-parcela-catastral-etc1>
* <https://networkx.org>