

**Ingeniería Matemática e Inteligencia Artificial**

**Memoria – Práctica 3**

GPS

Asignatura de Matemática Discreta

GP04

Alejandro Martínez de Guinea García - 202113492

Javier Prieto Domínguez - 202103783

**Índice**

[**1.** **Introducción** 2](#_Toc123728193)

[**2.** **GPS** 3](#_Toc123728194)

[2.1. INTERFAZ 3](#_Toc123728195)

[2.2. EXTRACT 3](#_Toc123728196)

[2.3. DISTANCE 3](#_Toc123728197)

[2.4. CLEAN 4](#_Toc123728198)

[2.5. UNIFY\_VERTICES 4](#_Toc123728199)

[2.6. SELECT\_RELEVANT\_INFO 4](#_Toc123728200)

[2.7. ASSIGN VERTICES 4](#_Toc123728201)

[2.8. GET\_WEIGHT 4](#_Toc123728202)

[2.9. CREATE\_GRAPH 4](#_Toc123728203)

[2.10. LEVENSHTEIN 5](#_Toc123728204)

[2.11. COMPROBAR\_DIRECCION 5](#_Toc123728205)

[2.12. COMPROBAR\_NUMERO 5](#_Toc123728206)

[2.13. DRAW\_STREET 5](#_Toc123728207)

[2.14. INSTRUCCIONES 5](#_Toc123728208)

[2.15. HANDLER\_SIGNAL 5](#_Toc123728209)

[**3.** **GRAFO.PY** 6](#_Toc123728210)

[3.1. CLASES 6](#_Toc123728211)

[3.2. FUNCIONES DE LA CLASE GRAFO 6](#_Toc123728212)

# **Introducción**

En nuestra tercera y última práctica de Matemática Discreta se nos pedía realizar un programa que simulase un GPS de la ciudad de Madrid. Para ello, emplearíamos nuestros conocimientos de teoría de grafos para desarrollar una librería grafo.py y más adelante el programa principal.

Al comenzar a programar cada función primero hacíamos un algoritmo básico, intentando ceñirnos a los conocimientos aprendidos en clase. Posteriormente a eso, tratamos de optimizar dichas funciones de cara a poder ejecutar el GPS en menor tiempo y por tanto, acercarnos a la realidad todo lo posible (los GPSs actuales son casi inmediatos en sus búsquedas).

# **GPS**

## INTERFAZ

Al correr gps.py se muestra un menú que permite realizar las diferentes funciones:

* Encontrar la ruta más rápida/corta entre dos puntos: esta es la funcionalidad principal, en la cual se introducen un origen y un destino y se encuentra el camino mínimo entre ellos.
* Visualizar el mapa de Madrid: se permite al usuario
* Visualizar una calle del mapa de Madrid: el usuario debe introducir la clave pública del destinatario al que va a enviar el mensaje para poder cifrarlo.
* Salir del programa

Una vez el usuario elige la opción que desea realizar, esta se ejecuta y posteriormente se vuelve a mostrar el menú para elegir de nuevo otra opción.

Ahora pasamos a explicar cómo se ejecuta cada opción.

Encontrar la ruta entre dos puntos

Esta opción es la principal del programa, que simula tal cual la función de un GPS: buscar el camino mínimo entre un origen y un destino. Por ello, lo primero que se le pide al usuario es que introduzca ese origen y ese destino de manera correcta. De no ser así, se mostrará de nuevo el menú para evitar un error. Una vez introducidos ambos puntos, se le pedirá al usuario si quiere encontrar la ruta más corta (distancia) o la más rápida (tiempo). Una vez hecho esto, el programa generará una imagen con el camino encontrado.

Visualizar el mapa de Madrid

Esta opción simplemente genera una imagen con el mapa de Madrid sin resaltar ninguna calle o camino. Esto se nos ocurrió ya que en la aplicación de Google Maps, es posible ver el mapa sin necesidad de buscar ninguna dirección.

Visualizar una calle específica

De nuevo, por inspiración de Google Maps, decidimos hacer que si se busca una calle específica, generar una imagen con dicha calle resaltada.

## EXTRACT

Esta función se encarga de extraer los datos de los CSVs con la información del callejero de Madrid. Además, aprovechamos para quitar errores del dataset de direcciones y cambiamos el tipo de un par de columnas.

## DISTANCE

Dadas dos coordenadas de la forma , devuelve la distancia euclídea entre ambas, es decir:

Esto lo empleamos para el cálculo del peso de las aristas.

## CLEAN

En esta función, limpiamos el dataset para que todos los datos estén tal y como nosotros queremos, para su posterior tratamiento. En ella, quitamos los espacios innecesarios de todas las columnas, corregimos los caracteres raros, creamos nuevas columnas en el dataset de direcciones: “Direccion” y “Nombre completo de la calle”. La primera de ellas, almacenamos para cada dirección su código de calle y su número de la siguiente manera: <código>-<número>. De esta forma, todas las direcciones que tuviesen un número con letra, los eliminábamos y nos quedábamos solo con una de las direcciones (cuando hay letras se refieren a bloques de una misma dirección). La segunda la usamos para detectar la calle al buscarla en el código principal.

## UNIFY\_VERTICES

Para las glorietas, decidimos unificar todos sus vértices en uno solo cuyas coordenadas serían la media del resto de coordenadas. Por lo tanto, para cada glorieta del dataset, realizábamos dicha operación y corregíamos las coordenadas de todo el dataset.

## SELECT\_RELEVANT\_INFO

Para poder definir las aristas, decidimos crearnos unos dataset auxiliares con información específica. Para el dataset auxiliar de cruces, seleccionamos: código de calle, coordenada X y coordenada Y. Para el de direcciones, además de las anteriores, cogimos también la columna “Direccion”. Nos creamos una columna en el dataset auxiliar de direcciones que determinase si el número de dicha dirección era par o no.

## ASSIGN VERTICES

Una vez definidos los dataset auxiliares, nuestro enfoque para crear las aristas era asignar a cada cruce el número de calle más cercano. Para ello, decidimos coger de la misma calle solo los números pares o solo los impares (la opción con más números). Esto lo hicimos debido a que en muchas calles no están alineados los pares y los impares.

Por lo tanto, calculábamos que número de calle era el más cercano y se lo asignábamos al cruce en el dataset auxiliar. Finalmente, ordenamos dicho dataset por código y número de calle.

## GET\_WEIGHT

Esta función se encarga de atribuirle un peso a cada arista. Las entradas son: tipo de vía, coordenadas de origen, coordenadas de destino y tipo de grafo. El tipo de vía nos permite determinar la velocidad máxima a la que se puede circular. Después, con el origen y el destino, determinábamos la distancia en kilómetros. Finalmente, según el tipo de grafo, devolvíamos el peso en kilómetros o determinábamos el tiempo dividiendo entre la velocidad máxima.

## CREATE\_GRAPH

En esta función, creamos el grafo principal del programa. Para ello, vamos definiendo las aristas a la vez que las introducimos en el grafo. Para empezar, iteramos sobre el dataset auxiliar de cruces. Primero, comprobamos si las coordenadas de dicho cruce están ya registradas en el grafo. De ser así, simplemente añadimos la calle en la información del vértice. Si no, creamos el vértice con un id asignado y lo agregamos al grafo. Una vez hecho esto, comprobamos si dicho vértice comparte calle con el vértice anteriormente registrado. De esta forma, enlazamos dos a dos los vértices del dataset siempre que estén en la misma calle (como están ordenados, las aristas estarán bien definidas).

## LEVENSHTEIN

Esta función se encarga de encontrar la mayor coincidencia entre el nombre de calle introducido por el usuario y las posibilidades a elegir. El algoritmo funciona calculando el número de inserciones, sustituciones o cambios mínimos que hay que hacer sobre un string para convertirlo en el otro.

## COMPROBAR\_DIRECCION

La función comprobar dirección toma como argumento el input de la calle que le metemos al programa, ya sea para buscar una calle o para encontrar un camino, y recorre todo el dataset ejecutando la distancia levenshtein con cada calle para encontrar la más parecida y devolverla.

## COMPROBAR\_NUMERO

Al escribir la calle en la interfaz, comprobamos si el número introducido existe para la calle seleccionada. De no ser así, se imprime un mensaje de error y no se devuelve nada.

## DRAW\_STREET

Esta función nos permite dibujar una calle específica. Para ello, recibe como entradas la calle a dibujar y los dos grafos (nuestro grafo y el correspondiente de networkX). De esta forma, para dibujar la calle, nos creamos un grafo auxiliar y lo pintamos de otro color sobre nuestro grafo original.

## INSTRUCCIONES

Esta función comprueba si a lo largo del camino mínimo hay que seguir recto, girar a derecha o girar a izquierda. Para ello, almacena la calle en la que se encuentra la primera arista y en todos los cruces que se encuentre a partir de ese momento y que estén en dicha calle, los marca con una “R”. A partir de aquí, todos los cruces no marcados, los comprueba y realiza una operación algebraica para determinar si el giro es a izquierda o derecha.

Para ello, cogemos tres cruces: el cruce de giro (denominémoslo ), el anterior ( ) y el siguiente ( ). Creamos con esos cruces, dos vectores: y (los vectores se calculan con la diferencia de coordenadas, es decir tienen dos dimensiones). Les añadimos una coordenada más a ambos, que va a ser 0 en los dos casos. De esta forma, es como si los dos vectores perteneciesen a , y estuviesen contenidos en el plano . Con esto, ya es fácil determinar si el giro es a derecha o a izquierda. Realizamos el siguiente producto vectorial:

Si la coordenada resultante es positiva, significa que el giro es a la izquierda (marcamos con una “I”), y si es negativa, es a la derecha (marcamos con una “D”). Finalmente, el destino lo marcamos con una “T”.

Por lo tanto, damos una instrucción en cada cruce según la letra marcada. Si es una “R”, mostramos el mensaje “Siga recto por la <calle> durante <peso hasta siguiente giro> metros.”. Si es una “I” o una “D”, se muestra “Gire a la izquierda/derecha por la <calle>”. Finalmente, si es una “T”, se muestra “Ha llegado a su destino”.

## HANDLER\_SIGNAL

Esta función se encarga de controlar la interrupción intencionada por teclado *“Ctrl+C”*. Al ejecutar dicha acción, el programa se interrumpe mostrando un error por pantalla. En este caso, evitamos que se muestre dicho error, y en su lugar, mostramos el mensaje *“Saliendo por interrupción…”*.

# **GRAFO.PY**

## CLASES

En la librería de grafo se definen dos clases. La clase ‘Vertice’ y la clase ‘Grafo’. El objeto Vertice tendrá simplemente 3 atributos: su id, las calles que llegan a ese vértice (en el caso del gps nuestros vértices serán nuestros cruces), y las coordenadas en la que se encuentra. El objeto Grafo tiene otros atributos como: si es dirigido o no; diccionarios con el id, las coordenadas y los nodos adyacentes asociados a un vértice; información sobre las aristas y una lista de vértices. Todos estos atributos nos servirán para que nuestro gps.py se ejecute de manera más rápida

## FUNCIONES DE LA CLASE GRAFO

Estas funciones son bastante simples, se pueden resumir en 3 o 4 líneas de código y con lo único que hemos debido tener cuidado es de agregar y eliminar los objetos de todos los atributos del grafo.

* + 1. FUNCIONES

#### es\_dirigido

Indica si el grafo es dirigido o no. No recibe argumentos y devuelve un valor booleano.

#### agregar\_vertice

Agrega un vértice al grafo tomado de los argumentos recibidos.

#### agregar\_arista

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. Toma como argumentos el vértice de origen, el de destino, un diccionario data y un weight que será un float.

#### eliminar\_vertice

Si el objeto v es un vértice del grafo lo elimina. Si no, no hace nada.

#### eliminar\_arista

Si los vértices s y t están en el grafo y existe una arista entre ellos la elimina. Si no, no hace nada.

#### obtener\_arista

Esta función obtiene los datos y el peso de una arista si sus vértices y la arista existen. Toma como argumentos los dos vértices y devuelve una tupla con un diccionario y un float.

#### lista\_adyacencia

Si el objeto u es un vértice del grafo devuelve su lista de adyacencia. Simplemente devuelve la lista que ya está almacenada en el atributo matriz\_adyacencia de la clase grafo.

#### grado\_saliente/ entrante

Si el vértice de los argumentos es un vértice del grafo devuelve su grado saliente/entrante

#### grado

Si el objeto u 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.

* + 1. ALGORITMOS

En todos los algoritmos utilizamos la librería headq para almacenar la lista de prioridad.

#### Dijkstra

Para este algoritmo hemos creado otra función llamada dijkstra\_parada que introduce una condición de parada al algoritmo de Dijkstra y así no buscar el árbol de caminos mínimos entero. Esa condición de parada es que el árbol se detiene al extraer el nodo de destino. Para obtener el árbol de caminos mínimos de todo el grafo (es decir, hacer el algoritmo de dijkstra), se llama a la función dijkstra\_parada sin condición de parada.

#### Camino\_minimo

Obtiene dos vértices (origen y destino). Esta función simplemente llama a la función dijkstra\_parada desde el origen y con la condición de parada en el vértice destino.

#### Prim/Kruskal

Calcula el árbol abarcador mínimo del grafo siguiendo el algoritmo de Prim/Kruskal. La diferencia de estos dos algoritmos además de su método es lo que devuelven las funciones. El algoritmo de Prim devuelve la lista de padres de los vértices mientras que el algoritmo de Kruskal devuelve la lista de aristas del árbol abarcador mínimo.

* + 1. CONVERTIR A NETWORKX

Esta última función convierte el grafo a un grafo de la librería de networkx de Python. Esta nos permitirá pintar los grafos y en el caso de la librería gps.py, el callejero de Madrid.