

**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](#_Toc123643791)

[**2.** **GPS** 3](#_Toc123643792)

[2.1. INTERFAZ 3](#_Toc123643793)

[2.2. EXTRACT 3](#_Toc123643794)

[2.3. HANDLER\_SIGNAL 4](#_Toc123643795)

[2.4. XX 4](#_Toc123643796)

[2.5. X 4](#_Toc123643797)

[2.6. X 4](#_Toc123643798)

[2.7. X 4](#_Toc123643799)

[2.8. X 4](#_Toc123643800)

[2.9. XX 4](#_Toc123643801)

[2.10. X 4](#_Toc123643802)

[2.11. X 4](#_Toc123643803)

[2.12. X 4](#_Toc123643804)

[2.13. X 4](#_Toc123643805)

[2.14. X 4](#_Toc123643806)

[**3.** **MODULAR.PY** 5](#_Toc123643807)

[**4.** **RSA.PY** 6](#_Toc123643808)

[4.5. ATAQUE\_TEXTO\_PLANO 7](#_Toc123643809)

[**5.** **TEXTO PLANO RECUPERADO** 8](#_Toc123643810)

[**6.** **BIBLIOGRAFÍA** 9](#_Toc123643811)

# **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.

## COMPROBAR\_DIRECCION

GHK

## 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…”*.

# **MODULAR.PY**

En la librería modular.py hemos realizado cambios para asegurarnos que ninguna función da error.

En primer lugar, hemos modificado la función factorizar para que solo ejecutase el algoritmo de Rho Pollard en caso de que el número a factorizar fuese mayor que 10000. De esta forma, nos hemos asegurado de que ejemplos como “factorizar(4)” no nos daba error.

También hemos arreglado la función “es\_primo”. Antes era determinista con el algoritmo de Miller-Rabin hasta 64 bits. Como para esta práctica hemos necesitado calcular primos más grandes, hemos asumido como cierta la hipótesis de Riemann de manera que pudiésemos ejecutar una versión determinista para cualquier número , y tomando como base los primos hasta comprobamos la primalidad de un número.

# **RSA.PY**

* 1. GENERAR\_CLAVES

Inicialmente este algoritmo tomaba los límites entre los que los primos podían estar comprendidos y ejecutaba el comando de la librería modular “lista\_primos”, de la que cogíamos 2 números primos para calcular la clave pública y privada. Rápidamente vimos que no era óptima y decidimos cambiar la manera en la que cogíamos 2 primos.

Creamos una nueva función en rsa.py llamada “generar\_primos” que no devuelve una lista de todos los primos, ya que es una operación que necesita mucha memoria. Devuelve solamente 2 primos dentro del intervalo para calcular con ellos claves. Coge un número aleatorio dentro del intervalo y comprueba si es primo o no. Si no lo es le suma 1 y vuelve a comprobarlo. Así hasta encontrarlo. Si llega al máximo del intervalo, empieza desde el límite inferior. Si llega hasta el número aleatorio inicial, devuelve que no existen primos dentro del intervalo. Hacemos el mismo proceso para el segundo primo, comprobando también que no sea el mismo número que el primer primo.

Últimamente, comprobamos que el seleccionado es comprimo con phi(n). De no ser así, aumentamos en dos, hasta que sea coprimo.

Con la nueva función de generar primos, no solo nos genera claves de 20 dígitos rápidamente, si no que nos genera hasta claves de 200 dígitos ( bits aproximadamente) en menos de 20 segundos.

* 1. APLICAR\_PADDING Y ELIMINAR\_PADDING

Estas funciones permiten añadir dígitos padding para cifrar el mensaje y eliminar dichos dígitos a la hora de descifrarlo. Esto se hace para evitar que ataques como el ataque a texto plano descifren fácilmente el mensaje.

Para aplicar el padding, la función recibe un número entero que determina la cantidad de cifras a añadir detrás de cada mensaje. De esta forma, ejecutamos un bucle for de duración , en el que añadimos al final de cada mensaje un número aleatorio entre 0 y 9.

A la hora de eliminar el padding, una vez aplicada la clave privada, eliminamos el número de dígitos establecido y obtenemos el mensaje original.

* 1. CIFRAR\_RSA Y DESCIFRAR\_RSA

Estas funciones se encargan de recibir un mensaje y tranformarlo realizando operaciones matemáticas. Para dichas operaciones necesitamos las claves públicas y privadas correspondientes.

A la hora de cifrar un mensaje requerimos de la clave pública (, ). Tomamos el mensaje y habiendo aplicado el padding correspondiente, realizamos la siguiente operación matemática para cifrar:

Donde es el mensaje cifrado.

Para descifrar, la operación es parecida. Requerimos de la clave pública (, ) y la clave privada . Esta última es la inversa de módulo . De esta forma, la operación que realizamos para descifrar es la opuesta a la que realizamos para cifrar:

Después de ejecutar esto, eliminamos el padding correspondiente y obtenemos el mensaje original.

* 1. ROMPER\_CLAVE

La función romper clave trata de encontrar dicha dada la clave pública (, ). De esta forma, si conseguimos encontrarla, podríamos descifrar cualquier mensaje enviado con dicha clave.

Para ello, debemos factorizar para poder calcular fácilmente . Una vez encontrado esto, simplemente debemos aplicar la propiedad siguiente:

Por lo tanto, debemos calcular la inversa de módulo . La única limitación que tiene este método para romper mensajes es que la función factorizar es muy costosa para números muy grandes, por lo que no es eficiente.

## ATAQUE\_TEXTO\_PLANO

Esta función se encarga de descifrar un texto cifrado sin conocer la clave privada, es decir, es un método alternativo al anterior. Dicho texto cifrado debe no tener padding. El procedimiento es el siguiente:

* Almacenamos en un diccionario todos los caracteres de Unicode junto con su respectivo cifrado.
* Para conseguir dicho cifrado, se pasa cada caracter a su código asociado y se cifra con la clave pública con la que se ha cifrado previamente el mensaje a descifrar.
* Una vez hecho esto, se recorre el mensaje cifrado buscando las coincidencias con los caracteres cifrados del diccionario y se sustituyen.

De esta manera, evitamos tener que factorizar para calcular , y por lo tanto, de no haber padding es una función bastante eficiente.

La única limitación como ya hemos mencionado anteriormente es que el mensaje no debe tener padding, ya que de ser así no habría coincidencias con los cifrados calculados.

# **TEXTO PLANO RECUPERADO**

El texto plano descifrado de la práctica presencial es:

“*En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda.*”

Este descifrado tarda aproximadamente segundos.

# **BIBLIOGRAFÍA**

“Prueba De Primalidad De Miller-Rabin.” Frwiki.wiki, <https://es.frwiki.wiki/wiki/Test_de_primalit%C3%A9_de_Miller-Rabin>.

“Ataque De Texto Plano Elegido Introducción y Diferentes Formas.” Hmong, <https://hmong.es/wiki/Chosen_plaintext_attack>.