MEMORIA

Grupo 1

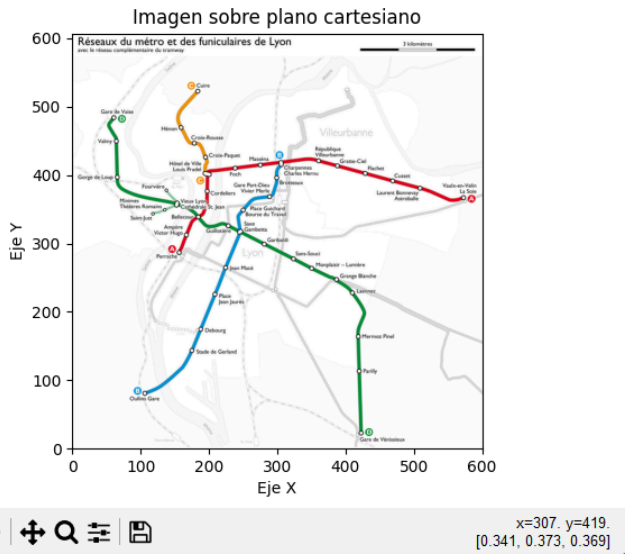
Gabriel Izquierdo de Cabo
Manuel Alejandro Arias Juárez
Rodrigo Menéndez Trejo
Alejandro Becerra Tapia
Alfonso Marín Mite
Luis Pérez López

1. Objetivo

Crear un programa capaz de obtener la ruta de metro más óptima entre dos estaciones del metro de lyon mediante el empleo del algoritmo A* y usando el lenguaje de programación Python. También implementaremos una interfaz gráfica para el uso de nuestro programa.

2. Algoritmo A* e implementación

El algoritmo usado para esta práctica es el algoritmo A* visto en clase. Para medir distancias, usamos un programa diseñado por nosotros (El programa Ejes.py). Con ese programa, hemos sacado las coordenadas en píxeles de cada estación. En la imagen adjunta aparece la interfaz del programa, donde las coordenadas que aparecen abajo a la derecha son las coordenadas sobre las que se encuentra el cursor. Con esos datos, hemos conseguido sacar tanto las distancias entre estaciones conectadas entre sí (es decir, las aristas) como la tabla "h" que necesitaremos para la implementación del algoritmo. Hemos sacado esas distancias a base de usar pitágoras, usando la diferencia de las coordenadas x e y de las estaciones que estábamos mirando como catetos, siendo la distancia real la hipotenusa del "triángulo". Para hallar cuáles estaciones están conexas entre sí (pertenecen a la misma línea), añadimos los nodos que hemos creado (las estaciones) a sus respectivas listas. Después, hemos usado nuestra función *sacadistreal* para hallar esas distancias entre estaciones de la misma línea.



Hemos usado nuestra función *heurística* para hallar la heurística entre el nodo que pongas (origen) y el destino del camino que queremos realizar. Ya metiéndonos en la función *a_estrella*, que es donde hacemos el algoritmo en sí, primeramente inicializamos los parámetros correspondientes al camino, a las pilas de activos y consumidos y a la distancia que llevamos. Creamos un bucle while que actuará mientras tengamos nodos en la lista activos, habiendo añadido primeramente el nodo origen. Dentro del bucle, primeramente sacaremos el nodo que esté más arriba en la pila de activos, y con ese sacaremos la distancia acumulada. Lo que hay justo después, es la resolución de si hemos encontrado el camino, cosa que explicaremos posteriormente. El *continue* que tenemos justo

después de ese bucle implica que si tenemos ese nodo en la pila de consumidos, no lo trataremos y continuaremos con el bucle.

Ahora nos metemos con el cálculo en sí de las distancias. Miraremos todas las aristas del nodo actual con un bucle for y sacaremos el nodo del otro extremo de la arista. Miraremos si hay transbordo para sumar una distancia equivalente a unos tiempos que hemos sacado con unos cálculos explicados posteriormente.

En cuanto encontremos el nodo destino en la pila de activos, inicializamos la variable camino con la estación actual (el destino) e iremos insertando las estaciones a base de encontrar el vecino de menor índice perteneciente a la pila. Iremos siguiendo ese criterio para cada nodo hasta llegar al nodo origen. Iremos añadiendo cada estación al principio de la lista para tener la lista con el camino ordenado.

En nuestro algoritmo hemos tenido en cuenta el coste de los transbordos entre líneas. Para sacar estos costes hemos realizado los siguientes cálculos:

1 km → 57 pixeles en el mapa (programa Ejes.py)

Línea A.
Longitud total: 9.2 km.
Tiempo en recorrerla entera: 25 min.

9.2 km * 57 px/km = 524 píxeles
524 píxeles / 25 min = 21 píxeles/min

21 píxeles/min en línea A
3 min (caminando) + 2 min (frecuencia de trenes) = 5 min
5 min x 21 píxeles/min = 105 píxeles de coste en transbordos

Línea B.
Longitud total: 10.1 km.
Tiempo en recorrerla entera: 20 mins

10.1 km(longitud de la línea) * 57 px/km = 576 píxeles;
576 píxeles/20 min = 29 píxeles/min

29 píxeles/min en línea B
3 min (caminando) + 2 min (frecuencia de trenes) = 5 min
5 min x 29 píxeles/min = 145 píxeles de coste en transbordos

Línea C.
Longitud total: 2.5 km.
Tiempo en recorrerla entera: 10 mins.

2.5 km(longitud de la línea) * 57 px/km = 143 píxeles
143 píxeles/10 min = 14 píxeles/min

14 píxeles/min en línea C
3 min (caminando) + 3.5 min (frecuencia de trenes) = 6.5 min
6.5 min x 14 píxeles/min = 91 píxeles de coste en transbordos

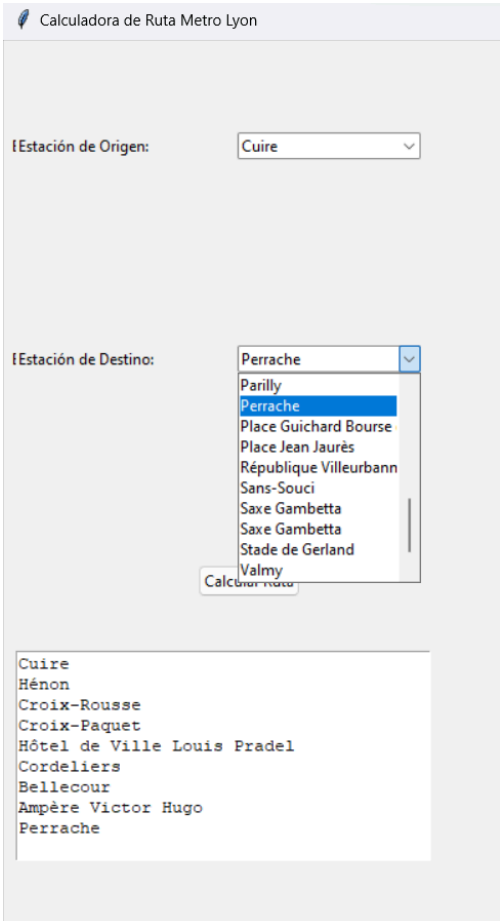
Línea D.
Longitud total: 12.6 km.
Tiempo en recorrerla entera: 30 mins.

12.6 km * 57 px/km = 718 píxeles
718 píxeles/30 min = 24 píxeles/min

24 píxeles/min en línea D
3 min (caminando) + 1 min (frecuencia de trenes) = 4 min
4 min x 24 píxeles/min = 96 píxeles de coste en transbordos

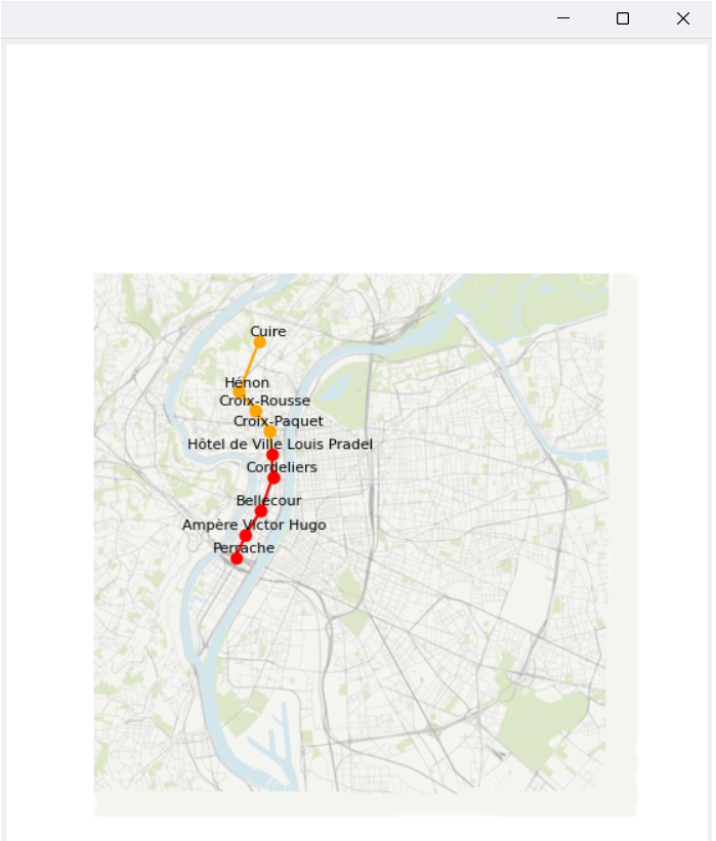
3. Interfaz Gráfica

Para el desarrollo de esta aplicación era necesaria una implementación gráfica para que el usuario tenga la posibilidad de una mayor interacción con la búsqueda de la ruta óptima entre dos estaciones, es por ello que la distribución se ha dividido de la siguiente forma:



1. **Sección con campos rellenables**
    - Consiste en dos campos para rellenar los campos de la estación origen y la estación destino, ambos campos cuentan con un menú desplegable con todas las estaciones de las 4 lineas (A, B, C, D) ordenadas en orden alfabético
    - El botón “Calcular Ruta” ejecuta el cálculo de la ruta deseada con el algoritmo A* (funcion “a_estrella” en main.py)
    - Un cuadro de texto en el que aparecen las estaciones por donde se debe seguir la ruta (estaciones origen y destino incluidas)

2. **Sección con un gráfico con la ruta deseada**
    - Posee un mapa de la ciudad de Lyon, el cual al ejecutar el programa con el botón “Calcular Ruta” cambia a un mapa con la ruta deseada dibujada.
    - Cuando se muestra la ruta deseada, el color de la ruta dibujada cambia de color según las líneas por las que pase (La línea A es roja, la línea B es azul, la línea C



es naranja y la línea D es verde)

- Además, el gráfico posee los nombres de cada ruta encima de cada estación

4. Problemas encontrados y decisiones de implementación

Cómo valorar el aumento de coste a la hora de hacer un transbordo está muy poco claro y es bastante dudoso, pues hay muchos factores que dependen del azar y muchos datos que son difíciles de obtener.

No estábamos familiarizados con el lenguaje python y eso nos ralentizó mucho al empezar.

Obtener los costes entre estaciones fue muy costoso al principio pues había poca información y no todas las formas de representar los datos se podían aplicar de forma óptima al programar el algoritmo A*.

Decidimos implementar las estaciones indicando las coordenadas de cada una e indicando que aristas existen en el plano, y a cada estación le asignamos una lista de estaciones “vecinas”.

Para comprobar si hay transbordo, miramos desde qué línea venimos y hacia cual vamos, y si son distintas sumamos el coste de transbordo correspondiente.