ENTREGA MENSUAL - GRAFO METRO DE MADRID

En este documento hemos redactado la explicación y el funcionamiento de todo el código que ha sido necesario para completar la tarea de creación de un grafo sobre el metro de madrid. Vamos a detallar cómo se crean cada estación, las uniones de estas, así como el flujo del código.

El objetivo de este documento es dejar claro todo el funcionamiento y como se ha optimizado para que se pueda realizar cualquiera de las tareas deseadas y como así se nos solicitó.

Inicialización del código:

Inicialmente leemos el archivo que se nos proporciona con todos los datos necesarios gracias a la librería Pandas de python, de este modo resulta con mayor sencillez el trato de los datos provenientes del archivo. Esta lectura la asignamos a una variable denominada “data” que será la que utilicemos cuando tengamos que hacer referencia al archivo.

Además procedemos con la creación de dos variables, “vertices” y “aristas”, que inicialmente son asignadas a diccionarios vacíos y en estas guardaremos los vértices (estaciones del metro) y aristas (conexiones entre las estaciones) según las vayamos creando.

Función “distancia_euclidiana(origen_x, origen_y, destino_x, destino_y)”:

Esta función denominada distancia_euclidiana nos permite calcular la distancia que hay entre la estación de origen o salida y la estación de destino o llegada.
Recibe como parámetros los valores de “x” e “y” de cada estación provenientes del archivo anteriormente citado.

De esta manera podemos ir reduciendo las posibilidades de estaciones que serán añadidas cuando busquemos el camino de forma que podamos optimizarlo para la creación y recorrido del camino más óptimo en cuanto a tiempo y distancia.

Función “crear_vertice(nombre, vecinos, coordenadas_origen)”: (Terminada)

La función ‘crear_vertice’ nos permite crear un vértice en base a una estación del metro de Madrid, esta función recibe los siguientes argumentos:

- “nombre”: Recibe el nombre de la estación que deseamos convertir en un vértice
- “vecinos”: Recibe una lista de las estaciones adyacentes a esa estación en las líneas colindantes
- “coordenadas_origen”: Recibe las coordenadas x e y de la estación a convertir en vértice

De este modo la estructura del vértice tras su creación es la siguiente:

nombre = {"vecinos": vecinos, "x": x, "y": y}

Gracias a esta función en el bucle de la función main() podemos pasarle los datos de cada una de las filas del Dataset del metro de Madrid e ir creando en cada loop del bucle principal el vértice de esa estación con los datos necesarios y que utilizaremos posteriormente.

Función “crear_arista(origen_nombre, destino_nombre, coordenadas_origen, coordenadas_destino)”: (Terminada)

La función de ‘crear_arista’ nos permite crear las conexiones entre vértices, es decir podemos crear la unión entre estaciones y añadirle atributos, es decir, le añadimos información que será necesaria en un futuro, en nuestro caso concreto hemos optado por introducir como atributo la distancia/tiempo entre los vértices de este modo nos será más fácil el cálculo de buscar camino en base a la distancia euclidiana.

La función de ‘crear_arista’ recibe como argumentos:

- origen_nombre: Nombre de la estación de salida.
- destino_nombre: Nombre de la estación de llegada.
- coordenadas_origen: Las coordenadas x e y de la estación de salida.
- coordenadas_destino: Las coordenadas x e y de la estación de llegada.

De este modo la estructura que nos queda una vez la arista ya ha sido creada es la siguiente:

(origen_nombre, destino_nombre)] = {"distancia": distancia}

Siendo distancia el cálculo de la distancia euclidiana anteriormente mencionada

Función “eliminar_vertice(vertice)”: (Terminada)

La función `eliminar_vertice` se encarga de eliminar un vértice del grafo que representa las estaciones y sus conexiones. Recibe como parámetro el nombre del vértice a eliminar, la función realiza varias acciones:

1. Eliminación de Aristas y Actualización de Vecinos:
   - ➢ Se recorren los vecinos del vértice a eliminar y se eliminan las aristas asociadas a él.
   - ➢ Se actualizan los vecinos de las estaciones restantes, eliminando la estación eliminada de sus listas de vecinos.

2. Reestructuración de Vecinos:
   - ➢ Se ajustan los vecinos restantes para mantener las conexiones válidas en el grafo.
   - ➢ Se añaden nuevos vecinos entre las estaciones adyacentes a la estación eliminada para mantener la conectividad.

3. Eliminación del Vértice:
   - ➢ Finalmente, se elimina el vértice deseado del grafo, asegurando que no quede rastro de él en la representación del sistema de transporte.

Esta función es esencial para permitir cambios dinámicos en el grafo al eliminar estaciones específicas y ajustar las conexiones restantes, garantizando que el grafo se mantenga coherente tras la eliminación de un vértice.

Función “buscar_vertice(nombre, vecinos, coordenadas)”:

La función ‘buscar_vertice’ nos permite dentro del grafo del metro de Madrid buscar una estación y que nos devuelva la información al respecto de ese vértice / estación. Esta búsqueda se puede hacer de 3 formas distintas, ya que el código lo hemos creado para que el usuario pueda buscar información de una estación / vértice en base a cualquier parámetro de los siguientes:

1. Nombre Vértice: Nos permite hacer la búsqueda de una estación/vértice en base al nombre de la estación que deseamos buscar y nos devolverá todos los datos que contiene esta.
2. Vecinos: Nos permite que introduciendo las estaciones que son adyacentes respecto a la estación que deseamos buscar, nos devuelve el nombre e información del vértice/estación que tenga como vecinos los valores que hemos introducido.

3. Posición / Coordenadas: Nos permite que introduciendo unas coordenadas x e y de la estación que deseamos, nos devuelva todos los datos e información asociados al vértice / estación que tiene asociadas esas coordenadas.

La función 'buscar_vertice' recibe varios parámetros dependiendo del modo de búsqueda de estación que el usuario ha seleccionado, serían los siguientes:

- ➢ Búsqueda por nombre: Recibe los valores de (nombre, None, None), de este modo únicamente nos lo buscaría por el nombre e indicamos None en los demás parámetros para que no sean utilizados.
- ➢ Búsqueda por vecinos: Recibe los valores de (None, vecinos, None), de este modo únicamente nos lo buscaría por una lista de vecinos e indicamos None en los demás parámetros para que no sean utilizados.
- ➢ Búsqueda por coordenadas: Recibe los valores de (None, None, coordenadas), de este modo únicamente nos lo buscaría por las coordenadas e indicamos None en los demás parámetros para que no sean utilizados.

Función "calcular_size()":

La función 'calcular_size' nos permite calcular la longitud del grafo del metro de Madrid por completo. Realiza mediante la función len() de python, la cual recibe como parámetro el grafo ya creado, la operación de cálculo para la longitud del grafo completo.

Función "buscar_camino(origen_nombre, destino_nombre)":

La función 'buscar_camino', 'construir_ruta_recursive', 'construir_ruta' nos permite buscar y crear el camino desde la estación de salida a la estación de llegada o destino, para posteriormente conseguir el camino más óptimo teniendo en cuenta el tiempo que tarda en recorrer y la distancia recorrida por cada uno de los caminos.

Esta función tiene como objetivo encontrar el camino más corto entre dos nodos en el grafo del metro. Para hacer esto se utiliza un variante del algoritmo de Dijkstra llamado algoritmo de búsqueda A*. Es el algoritmo de Dijksta con la adición de una función heurística para añadir precisión y no tener que explorar nodos/caminos de forma innecesaria, pero manteniendo la búsqueda del camino más corto.
En el algoritmo definimos el camino más corto como el camino con menos metros desplazados, por lo tanto el peso de las aristas será la distancia entre su nodo de origen y

su nodo de destino. Utilizaremos las coordenadas proporcionadas de cada estación para calcular la distancia entre cada nodo.

$$f(n) = g(n) + h(n)$$

Donde n es el nodo siguiente, $g(n)$ es el cálculo del coste (en metros) desde el origen hasta n, $h(n)$ es la función heurística y $f(n)$ es la función estimatoria de la distancia entre el nodo siguiente (n) y el destino.

Para calcular $f(n)$ se requiere la función heurística $h(n)$. En este proyecto decidimos utilizar las distancia euclidiana entre el origen y el destino como función heurística.

Distancia euclidiana entre dos puntos:

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - q_i)^2}$$

Simplificamos en este caso ya que el grafo es de dos dimensiones.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Donde d es la distancia entre los puntos con coordenadas (x y . 1 , y 1) (x 2 , y 2).

Para visualizar este proceso, tomemos la estación “Lago" como origen y” Gran Vía" como destino. Primero se exploran los nodos vecinos a Lago, Batán y Principe Pio. La línea roja que conecta la estación origen con su vecino es el cálculo de la función $g(n)$ el coste entre el origen y el nodo siguiente. La línea morada que conecta los nodos siguiente con el destino, son la distancia euclidiana entre las estaciones, la función heurística $h(n)$. La suma de ambas expresiones produce el cálculo estimatorio. De los dos nodos, el nodo con el valor $f(n)$ menor será el que se continuará explorando y el proceso se repetirá con ese nodo.

En el código este funcionamiento se implementa con el uso de pilas, ***open_set*** es la pila que contiene los nombres y el coste desde el origen de cada nodo que será explorado. En cambio ***closed_set*** contiene el nombre de los nodos ya explorados. El bucle se repetirá siempre y cuando exista un elemento dentro de ***open_set***. Una vez se comienza el bucle, se selecciona ***current_node*** el nodo que será explorado; se selecciona este nodo ordenando la pila de open_set con una función anónima para extraer el elemento con el menor $f(n)$. En otras palabras, dentro de la función anónima, para todos los nodos n dentro de ***open_set*** se calcula su distancia euclidiana desde n hasta el destino ($h(n)$) y se le suma el coste desde el origen hasta **n** ($g(n)$). Basándose en esa suma el que tenga el menor valor es igualado a current_node.

Dentro del bucle, se elimina *current_node* de la pila ***open_set*** y se añade a ***closed_set***.

Por cada vecino a ese nodo, se calcula el coste de ese vecino $g(vecino)$ (en el caso que ese vecino esté dentro de ***closed_set***, osea se ha procesado ya, se salta ese vecino) se suma $g(n)$ más la distancia euclidiana entre n y $vecino$. En el caso que el vecino no esté en ***open_set*** se añade a la lista para ser procesado, en el caso que ya está en la pila, se compara si el valor potencial de $g(vecino)$ es menor que el que ya está guardado. En el caso que si es menor, significa que se ha encontrado una forma más rápida de llegar a ese nodo y por lo tanto se sobreescribe su información en la pila.

El bucle se finaliza cuando current_node = destino.

Función “main()”:

La función ‘main’ nos permite leer el archivo y poder extraer los datos necesarios mediante la utilización de un bucle y así poder ir inicializando las funciones en base sea necesario para el funcionamiento del código.

Al hacer run del código, inicializamos el bucle while de la función de este modo podemos hacer que el usuario elija aquella opción que desea realizar entre las disponibles en el menú. De este modo podemos hacer una mejor interacción entre el usuario y el código, teniendo el usuario las siguientes opciones:

1. Buscar Estación: En caso de que el usuario elija la opción 1, le mostraremos un segundo menú en el cual el usuario pueda elegir la forma o método por el que quiere buscar un vértice / estación:
    - Búsqueda por nombre de la estación
    - Búsqueda por estaciones adyacentes
    - Búsqueda por coordenadas
2. Eliminar Vértice: En caso de que el usuario elija la opción 2, le preguntaremos cuál es la estación que desea eliminar e inicializamos la función ‘eliminar_vertice()’
3. Buscar Camino: En caso de que el usuario elija la opción 3, le preguntamos cuál es la estación de salida y cual es la estación de llegada / destino e inicializamos la función buscar_camino, construir_ruta, construir_ruta_recursive.
4. Break: En caso de que el usuario elija la opción 4, el programa se detendrá y se saldrá del metro de Madrid.