Memoria Práctica Inteligencia Artificial

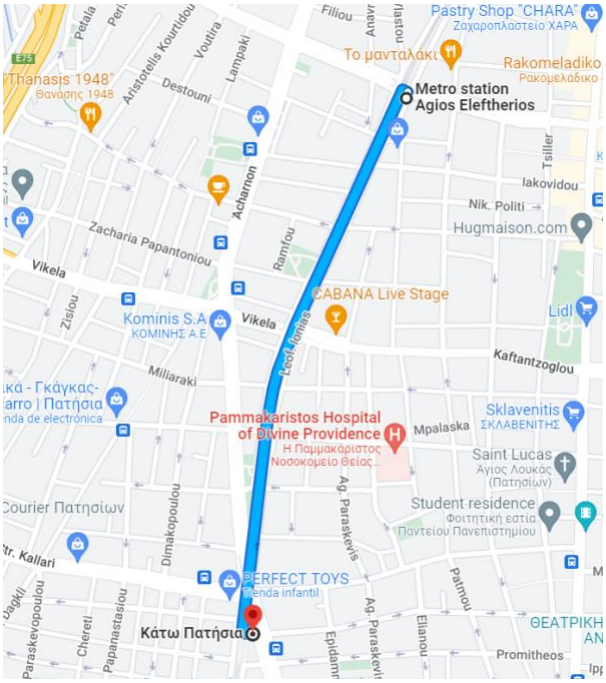
Camino óptimo entre dos estaciones:
Metro de Atenas

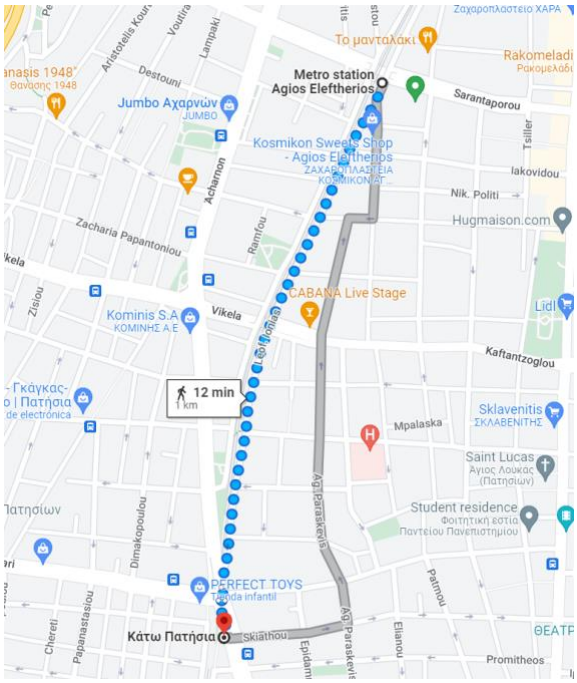
Grupo 4:
Pedro Amaya Moreno
Juan López González
Pablo Rodríguez Beceiro
Alejandro Jose Piguave Illescas

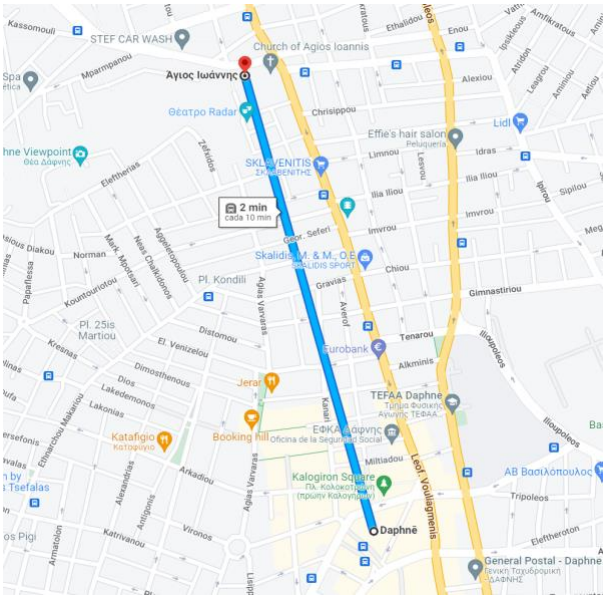
Búsqueda de la información:

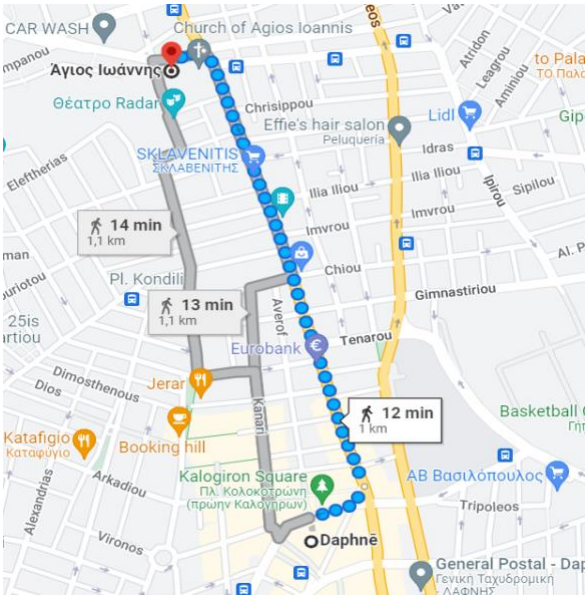
Antes del inicio del proyecto era necesaria la recaudación de diversa información sobre el metro de Atenas: la distancia entre las diferentes estaciones y sus coordenadas, los horarios de apertura y cierre, la frecuencia de pasada de los trenes, el tiempo de trasbordos…

La recaudación de la mayoría de esta información se ha hecho a través de diferentes páginas webs como la página oficial del metro de Atenas, Wikipedia o Google Maps. Los únicos datos que no se han sacado de manera directa de alguna de las páginas anteriores han sido las distancias que hay entre las estaciones. Estas se han calculado de manera aproximada a través de la herramienta de creación de rutas entre dos puntos de Google Maps, aproximando las rutas a pie con la forma de la vía para intentar acercarnos de la manera más fiel posible a las distancias reales entre estaciones. A continuación, se muestran unos ejemplos:









Las que no tenían cerca ninguna ruta a pie, se aproximaron usando la herramienta de distancias que también provee Google Maps. Por ejemplo, el final de la línea 3, entre Ambelokipi y el aeropuerto, no hay ninguna ruta a pie que se aproxime bien a las distancias entre las estaciones, así que las obtenemos con la herramienta de distancias:

Diseño del programa:

Antes de describir el algoritmo A* que hemos implementado, vamos a describir brevemente la estructura del código.

Para acceder a los datos de las diferentes líneas de metro, hemos creado un documento JSON con el siguiente formato:

```
{
    "__lineasMetro__": true,
    "lineas": [
      [
            [estacion1_ln1, 0],
            [estacion2_ln1, dist_a_estacion_1 + 0],
            [estacion3_ln1, dist_a_estacion_2 + (dist_a_estacion_1 +
            0)]
            …
      ]
       [lin_2],
       [lin_3]
      ],
    "stNodes": [nodo_1, nodo_2, …]
}
```

__lineasMetro__ únicamente indica que los datos suministrados que estamos leyendo son líneas de metro.

En *líneas* guardamos los nombres de las estaciones junto con la distancia que hay entre la estación que consideramos el inicio de la línea hasta esta estación. Hacemos esto para simplificar los cálculos de la distancia en la ejecución del programa.

stNodes son las estaciones que vamos a usar en el cálculo del camino mínimo. Estas son:

Al resto de estaciones de las líneas 1, 2 y 3 sólo se pueden acceder a través de Monastiraki, Syntagma y Attiki. Por lo tanto, para cualquier camino mínimo que calculemos, cuyas estaciones de inicio o fin no sean nodos, primero habrá que moverse a una de estas estaciones en los extremos (la estación a la que nos movemos dependerá de donde está el inicio o el fin).

Utilizando estos datos generamos un objeto de clase *MetAtenas* donde almacenamos estos datos en atributos de la clase:

- *st_lin* es un diccionario que asocia los nombres de las estaciones con su línea y su posición dentro de la línea (las estaciones de transbordo están asociadas a varias líneas y posiciones).

- En *st_names* y *st_dist* se asocia el número de línea con sus nombres y distancias respectivamente.

- En *train_speed* y *start_travel_time* se almacena la velocidad media del tren que usamos para pasar de distancias a minutos y la hora de comienzo de trayecto. También tenemos *min_node_dist*, la distancia mínima entre los nodos que usaremos para cálculos posteriores, en un principio es la distancia que recorre el tren en 1 minuto.

- En *st_nodes* asociamos los nombres de las estaciones que consideramos nodos con sus adyacencias a otros nodos. También sirve para identificar si una estación es un nodo, ya que si no lo es devuelve *None*.

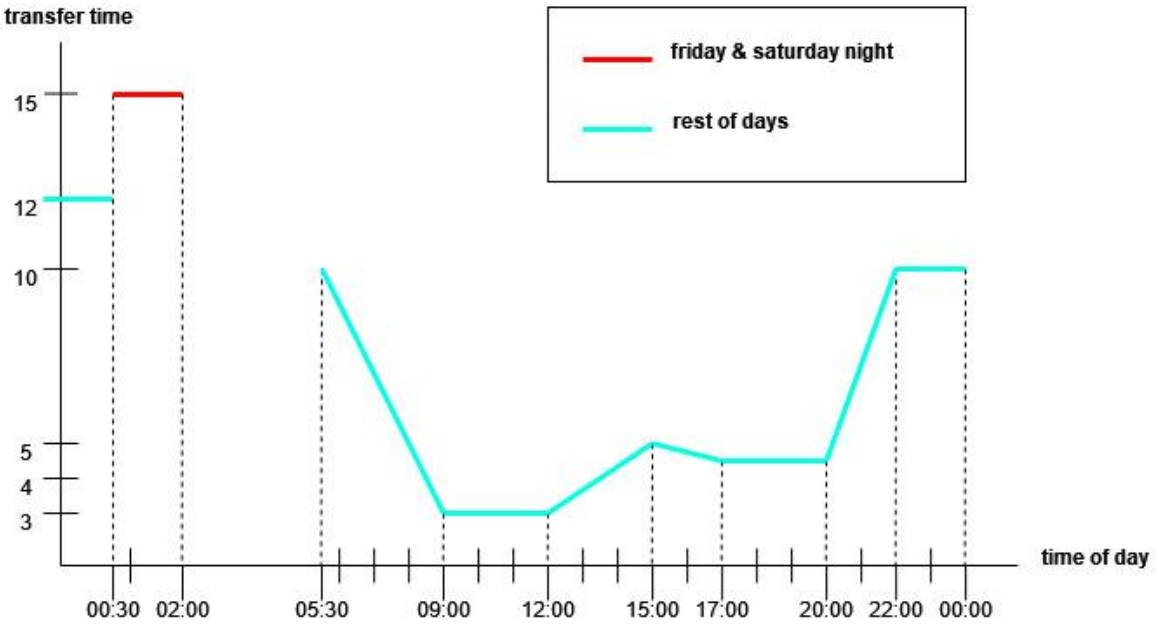
Las adyacencias se calculan con el método de la clase get_adyacencies. En get_adyacencies también calcularemos la distancia mínima entre nodos:

```
=====================================================================
get_adyacencies (self, st_node_names):
       adyacencies = {}
       min_dist = train_speed
       for st_node_name in st_node_names:
              st_actual = st_lin[st_node_name]
              ady = []
              distance = 0
              for lin, pos in pos_st_node:
                     st_names_lin = st_names[lin]
                            if st_actual not First and prev_st is Node:
                                   distance = dist(prev_st, st_actual)
ady.append((prev_st_name, lin, distance)
min_dist = min(min_dist, distance)
if st_actual not End and next_st is Node:
       distance = dist(next_st, st_actual)
       ady.append((next_st_name, lin,distance)
       min_dist = min(distance, min_dist)
st_nodes = adyacencies;
min_node_dist = min_dist
=====================================================================
```

- En *st_intervals* asociamos las líneas con los intervalos que contienen. Un intervalo consiste en una posición de inicio y de final del intervalo junto a un *step*. El *step* indica si el intervalo es de inicio (1), de fin (-1) o intermedio (0). Como hemos establecido los nodos en el metro de Atenas, no hay ningún intervalo intermedio. get_intervals nos calcula estos intervalos.

```
=====================================================================
def get_intervals(self) -> dict:
        st_intervals = {}
        for num_ln in st_names.keys():
            line_names, line_intervals = self.st_names[num_ln], []
            start, step= -1, 0
            read_a_node, len_lin = False, len(line_names)
            for pos, st_name in enumerate(line_names):
                is_a_node = st_name is a Node
                if start == -1 and not is_a_node:
                    start = pos
                    step = 1 if not read_a_node else -1
                elif start != -1 and (is_a_node or pos == len_lin - 1):
                    step = 0 if read_a_node and is_a_node else step
                    last_st = step == -1? pos: pos - 1
                    line_intervals.append((start, last_st, step))
                    read_a_node, start = False, -1
                else:
                    read_a_node = is_a_node or read_a_node
            st_intervals[num_ln] = tuple(line_intervals)
        return st_intervals
=====================================================================
```

Además, aplicamos una función para obtener los tiempos de transbordo dependiendo de la hora que es y el tiempo que ha pasado desde esa hora. La función la hemos interpolado de información sobre los tiempos de espera que hemos encontrado en internet, por ejemplo Helenizarte.



Donde el tiempo no está definido, consideramos que el metro está cerrado a esa hora y devolvemos -1.

Ya definidas todas las variables que vamos a usar, ya podemos describir como hemos implementado el algoritmo A*. El algoritmo sólo lo aplicaremos entre las estaciones que hemos descrito como nodos. Las demás después describiremos como las hemos obtenido.

Primero vamos a definir cómo calculamos el coste real (g(x)) y el coste heurístico (h(x)).

El coste real g(x) será el tiempo que se tarda de ir de la estación de inicio a la estación x, este se divide en dos costes: g1(x) y g2(x):

- g1(x) es el tiempo que tardamos en recorrer la distancia de nuestro nodo origen hasta la estación x. Como sabemos las distancias entre estaciones y la velocidad media de los trenes, obtener los minutos necesarios es un cálculo sencillo.

Por ejemplo, si nuestro nodo de inicio es Attiki y queremos ver cuánto es:
g1(Monastiraki) =
(D1 + D2 + D3) / train_speed

- g2(x) es el tiempo total empleado en los transbordos. Es la suma del tiempo que se ha empleado antes de llegar a la estación x en transbordos, más si realizamos un transbordo para llegar a x, el tiempo de transbordo que obtenemos. Lo obtenemos con la función descrita antes, pasándola la hora de comienzo y el tiempo que hemos necesitado para llegar a este nodo.

  Hacemos esto ya que hemos definido un tiempo de trasbordo que varía dependiendo de la hora del día, y para comprobar si el metro está cerrado a esa hora. Debido a que no

tenemos los horarios exactos de pasada de cada metro por cada estación, en caso de trasbordo siempre se tiene en cuenta el peor de los casos, es decir, siempre se va a tener en cuenta el tiempo máximo que transcurre entre dos trenes contiguos.

Por ejemplo, si queremos ir de Syntagma a Metaxourgio y no se pudiera pasar por Panepistimio. Si estamos en Omonia, para el g2(Metaxourgio) = T1 + T2 + T3
Con T3 siendo igual a:
T3 = tiempo_transbordo(g1(Metaxourgio) + T1 + T2)

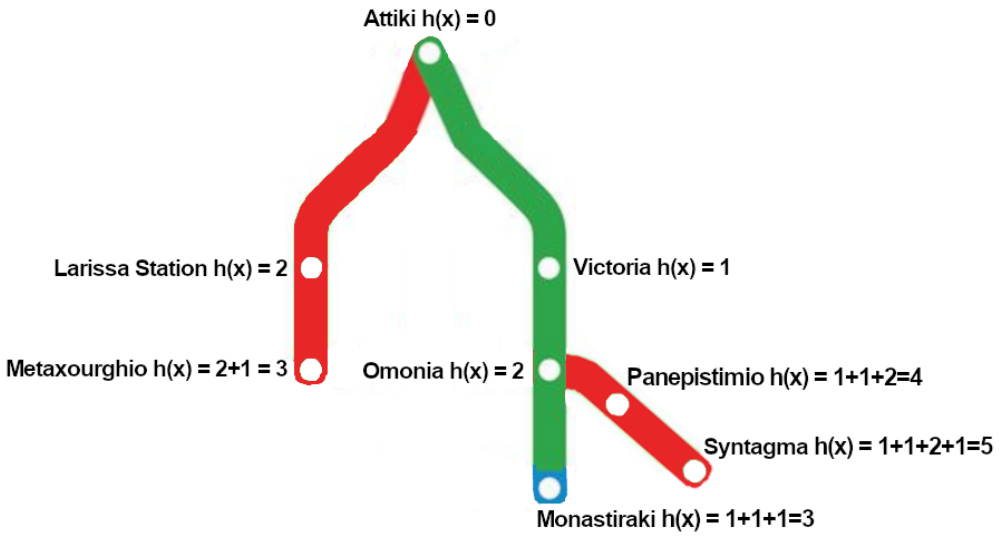
Finalmente, g(x) se calculará como la suma de g1(x) y g2(x).

El coste heurístico h(x) es el peso de ir de la estación del inicio a la estación x. El peso es una medida simulada calculada asumiendo que los movimientos por la misma línea tienen un coste homogéneo, 1 por ejemplo, y los trasbordos otro, por ejemplo 2. Con estos costes calculamos un árbol de expansión de peso mínimo con el algoritmo de Dijkstra. Para mejorar el funcionamiento y evitar cálculos innecesarios, empezamos el árbol desde el nodo destino y vamos poco a poco recorriendo el mapa hasta llegar a la estación de origen.

El cálculo del tiempo se realiza de la estación de destino a estación de origen (sentido opuesto). El proceso es el siguiente:

1) Como el destino no es una estación considerada como nodo, primero nos movemos al nodo más cercano calculando el tiempo a este nodo. En este caso el nodo más cercano es Attiki, y hemos llegado a él por la línea 1.

2) Una vez llegamos a este nodo y teniendo en cuenta que estamos en la línea 1 (los trasbordos suponen un coste en forma de tiempo por lo tanto se intentarán minimizar), cómo se quiere calcular el camino de peso (tiempo) mínimo desde Attiki hasta Omonia se calcula el árbol de expansión de peso mínimo con el algoritmo de Dijkstra con raíz en Attiki considerando todos los nodos mediante la función heurística h(x). Para este ejemplo supondremos que el peso de todas aristas es 1 y que si se pasa de una línea a otra en una arista el peso será 2. Utilizando lo anterior y que estamos en la línea 1 (pasar a la línea dos supondrá un peso de valor 2) llegamos al siguiente árbol:



El camino mínimo desde Attiki a Omonia es:
[Attiki,Victoria,Omonia]
con un coste de h(Omonia)=2

- En el cálculo hay un caso especial, por el que puedes llegar a dos estaciones con el mismo coste por líneas distintas. Añadimos ambas al algoritmo.
- En los cálculos nuestros, al principio h(x) no valía, porque no se cumplía la condición del A*. Lo solventamos obteniendo la distancia mínima entre los nodos (*min_node_dist)* y dividiéndola entre la velocidad, obtenemos el nuevo coste para moverse por la misma línea. El del transbordo es el coste anterior por 3.

Diseño del algoritmo:

1) Moverse de una estación a un nodo.

Como hemos descrito antes, no consideramos que todas las estaciones sean nodos. Por tanto, si el inicio o fin del camino no lo es, tenemos que desplazarnos hasta el nodo más cercano, es decir, recorrer el intervalo hasta llegar al nodo.

Podemos encontrarnos varias excepciones a la regla:

- Si la estación de inicio y fin están en la misma línea y mismo intervalo: No pasamos por ningún nodo, devolvemos directamente el camino y el tiempo entre ambos sin ningún trasbordo (por ejemplo, ir desde Faliro hasta Thissio en la línea 1).

- Si la estación de inicio y fin están en distintas líneas, pero el nodo al que se desplazan es el mismo: Devolvemos el tiempo y el camino directamente, añadiendo el tiempo de trasbordo tras recorrer la primera línea hasta el nodo.

- Si la estación es un nodo, no hace falta desplazarse (por ejemplo, si la estación de origen es Atikki).

- Si no ocurren ninguno de los casos anteriores, y o la estación de inicio o la de fin no son nodos, nos desplazamos al nodo más cercano pasándole a *min_cam* la línea que hemos usado para desplazarnos (el caso del segundo ejemplo explicado anteriormente). Guardamos las estaciones recorridas en estos movimientos al igual que las distancias. A la función de min_cam le pasamos la linea del inicio (0 si nodo), del fin, el tiempo que ya ha transcurrido (si el inicio no es un nodo, el tiempo que se tarda hasta llegar al nodo).

2) Cálculo del camino mínimo:

Aplicamos el algoritmo A*. La información de los nodos la almacenamos en tuplas que contienen el nombre de la estación actual (*st_at*), de la estación de la que han venido (*st_prev*), la distancia recorrida (*dist_trav*), la línea actual (*through_line*), el tiempo de trasbordo (*tm_trans*) y el coste (f(x) = g1(x) + g2(x) + h(x)).

Tenemos una lista abierta donde vamos añadiendo los nodos que podemos visitar, y un diccionario de nodos visitados, donde asociamos el nombre de la estación con la estación previa, también nos sirve para comprobar si hemos visitado ya una estación y así no visitarla dos veces.

La lista la vamos ordenando según el coste de los nodos de mayor a menor, y extraemos siempre el último, que será el de coste menor. En nuestra lista puede haber dos instancias del mismo nodo o incluso una instancia de un nodo ya visitado, ya que no nos aseguramos de que solo haya una instancia en la lista.

Lo que hacemos es extraer siempre el último nodo, y si ya lo hemos visitado, lo descartamos, si no lo añadimos a visitados y añadimos sus adyacencias a nodos.

El pseudocódigo del algoritmo es el siguiente:

```
================================================================
def min_cam(self, st_from, st_to, lin_from=0, lin_to=0, tm_used=0):
        h_vals = heuristic_costs(st_to, st_lin=lin_to)
        open_stck =
      [(st_from, st_from, dist=0, lin_from, tmTrans=0,  cost=0)]
        visited = {}
        comp = f_cost_1 >= f_cost_2
        while len(open_stck) != 0 and open_stck[-1][0] != st_to:
            st_at, st_from, st_dist, at_ln, tm_trans =
                                          open_stck.pop ()[:-1]
            if st_at was visited:
                continue
            visited[st_at] = st_from
            adyacencies = self.st_nodes[st_at]
            for nxt_st_nm, through_ln, nxt_st_dist in adyacencies:
                if nxt_st_nm was visited:
                    continue
                tt_tm = (st_dist + nxt_st_dist)/ train_speed
                nxt_trans_tm = 0
                 # Needs a transfer
                if at_ln not in {0, through_ln}:
                    time = transfer_line_time(tt_tm + tm_trans
                                                    + tm_used)
                    if time > 0:
                        nxt_trans_tm += time
                    else: # metro is closed
                        continue
                 # Last station needs a transfer
                if nxt_st_nm = st_to and lin_to not in {0,
                                                through_ln}:
                    time = transfer_line_time(tt_tm + tm_trans
                                                    + tm_used)
                   if time > 0:
                        nxt_trans_tm += time
                    else: # metro is closed
                        continue
                # f(x) = g(x) + h(x) (g(x) = g1(x) + g2(x))
```

```
                # g2(x) = nxt_trans_tm + tm_trans
                node_cost = nxt_trans_tm + tm_trans + tt_tm +
                                                    h_vals[nxt_st_nm]
                insert_sorted(open_stck,
                 (nxt_st_nm, st_at, st_dist + nxt_st_dist,through_ln,
                      nxt_trans_tm + tm_trans, node_cost), comp)
        if len(open_stck) == 0:
            return None
        node_info = open_stck.pop()
        path = [node_info[0]]
        st_prev = node_info[1]
        while visited[st_prev] != st_prev:
            path.insert(0, st_prev)
            st_prev = visited[st_prev]
        path.insert(0, st_prev)
        return {'path': tuple(path), 'dist': node_info[2],
                'tmTrans': node_info[4]}
================================================================
```

Interfaz gráfica:

Para la interfaz gráfica del programa se ha hecho uso de la librería *Tkinter* de Python con la que se ha organizado dicha interfaz en dos partes:

El panel de la izquierda contiene elementos que permiten seleccionar los parámetros de entrada para el algoritmo, los cuales son:

- *Seleccionar origen*: al hacer click en el botón, se activa la selección de la estación de origen haciendo click en cualquiera de las estaciones del mapa del panel de la derecha.

- *Seleccionar destino*: al hacer click en el botón, se activa la selección de la estación de destino haciendo click en cualquiera de las estaciones del mapa del panel de la derecha.

- *Simular interrupción*: al hacer click en el botón, se permite seleccionar una arista entre las que se muestran disponibles para simular un corte de vías entre las 2 estaciones que une.

- *Seleccionar fecha de salida*: permite seleccionar el día (de lunes a domingo), hora y minutos de salida.

- *Seleccionar velocidad del tren*: permite seleccionar la velocidad media del tren (por defecto es 80, de nuestras fuentes de información, pero los tiempos que se obtienen con ella son muy bajos, debido a que no se cuenta el tiempo entre parada y arranque de un tren, se recomienda usar 60 o 70 km/h.

El botón de "*calcular camino*" solo se activará cuando se hayan introducido todos los parámetros necesarios para el cálculo de la ruta.

El panel de la derecha contiene la imagen del mapa con las estaciones a las que es posible dar

click para seleccionarlas desde el panel de la izquierda. Una vez calculada la ruta, las estaciones que forman parte de esta aparecerán en azul en este panel.

A continuación, adjuntamos una imagen de la interfaz junto con un ejemplo del uso de la misma yendo desde Aghios Antonios hasta Evangelismos:

Bibliografía y Fuentes:

En esta sección recopilamos las fuentes que hemos usado:

- **Google Maps:** Para la obtención de las distancias a través de sus herramientas.
- Helenizarte: Para información general y aproximación de los tiempos de espera.
- Wikipedia Metro Atenas: Información general