Memoria Práctica
Inteligencia Artificial

Metro de Atenas con Algoritmo A*

Participantes (Grupo 5):
-Geovanna Aguilar Avilés ~ 200023
-Miguel Luis Cerdá Palmer ~ 200096
-Miguel Ciurana Tomás ~ 200244
-Jessica Zhu Zhu ~ 200132
-Daniel Sánchez Ferrari ~ 200024

Índice:

Introducción……3

Explicación Interfaz Gráfica……4

Explicación Código Algoritmo Estrella……... 5

Dificultades…… 8

Conclusiones……..8

Introducción:

La práctica que se nos presentó tenía por objetivo que mediante el uso del Algoritmo A* encontrásemos un camino mínimo dentro de la línea de metro de Atenas.

En un principio, decidimos utilizar Python como lenguaje para diseñar el algoritmo, porque nos fue recomendado por el profesor utilizar éste. Pero también porque nos dimos cuenta que en Java (el lenguaje en el que todo el grupo se encontraba más cómodo) sería mucho más complicado hacer una interfaz visual.

Decidimos que nuestro algoritmo se regiría por dos parámetros: el tiempo que tarda el metro en realizar el viaje y la distancia que recorre. Estos parámetros determinan el peso de las aristas.

Nuestro grupo utilizó archivos de la extensión .json con el fin de almacenar la información de las diferentes estaciones. El primer archivo, aristas.json, contiene la información de las aristas. Dentro está Origen y Destino, que representan una arista. Debajo de estos dos está criterio: en el que está puesto la cantidad de transbordos y la distancia del origen al destino.

Las distancias las obtuvimos utilizando una página web: https://www.metrolinemap.com/station/athens/sepolia/, en la que al poner una estación, te mostraba la distancia a sus estaciones adyacentes. Los transbordos, fueron deducidos, simplemente mirando el mapa.

El otro archivo, coordenadas.json contiene la estación y sus coordenadas (Latitud y Longitud). La información fue obtenida utilizando Google Maps, introducimos la estación en el buscador y en la URL estaba puesta la latitud y la longitud.

Explicación Interfaz Gráfica:

Mediante la librería PyQt5, implementamos una interfaz gráfica en la que podemos encontrar un mapa del metro y diferentes opciones como el cambio de Criterio en una QBoxCombo o una caja de múltiple elección.

Para seleccionar destino se haría por Push Buttons con dos posibles elecciones. La primera sería el push button de Origen, que marcaría el comienzo. Por otro lado, estaría el destino, que marcaría a donde queremos llegar mediante el criterio aplicado, el cual se elegiría en la QBoxCombo explicada anteriormente. Los Push buttons se localizan mediante coordenadas de una imagen 900x900, y éstas se encuentran en el archivo metro.json.

La interfaz gráfica se divide en 2 programas. El archivo qui.py se ocupa de la distribución de la interfaz, que como podemos observar (*Figura 1*) reserva sitio para la imagen y para los layouts donde devuelve los resultados del algoritmo.

El segundo programa, es el QImageLabel.py, el cual localiza la clase estación ubicada en estacion.py, además de crear y dar valores a los Push Buttons. Estos, tomarán los valores *x* (posición en el eje x de la imagen) e *y* (posición en el eje y de la imagen) y el nombre de la estación. Después, se van actualizando el resto de valores.

El diseño también cuenta con otro botón de búsqueda que se ocupa de ejecutar el algoritmo estrella cuando se presiona.

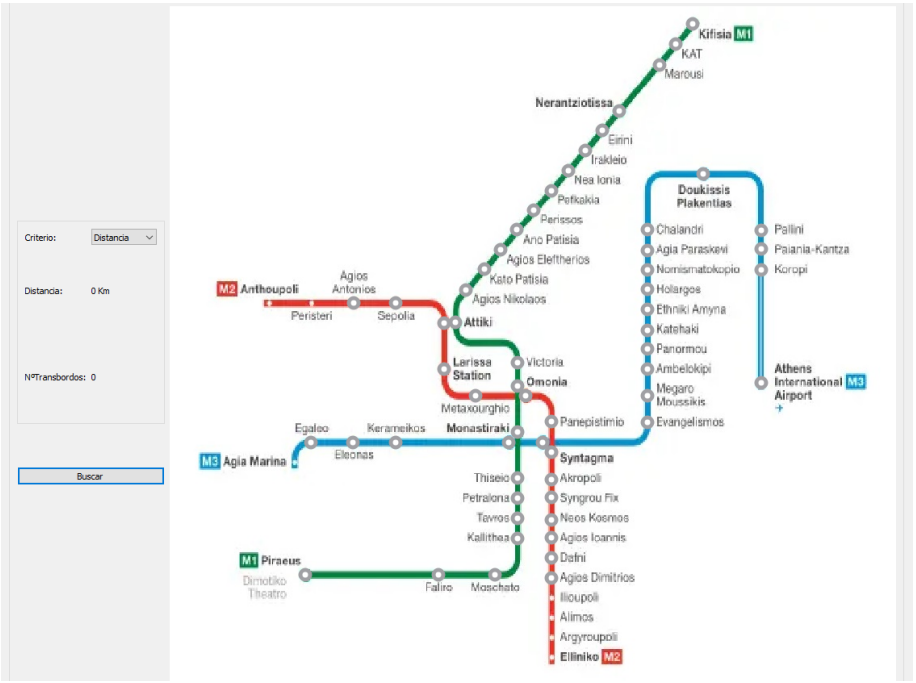

Figura 1

Explicación Código Algoritmo Estrella:

En la primera parte del código, el *init*, inicializamos las variables:
-***Origen*** : Marca el nodo Origen seleccionado en la interfaz gráfica.
-***Destino***: Marca el nodo Destino seleccionado en la interfaz gráfica.
-***Criterio***: Marca el criterio utilizado en el algoritmo (Distancia o Transbordo).
-***net***: es el Grafo de pesos que iremos utilizando.
-***aristas***: array que almacena el valor de las aristas del grafo net.
-***estaciones***: array que almacena el nombre de la estación.
-***opnlst***: Lista abierta del algoritmo estrella.
-***closedlst***: Lista Cerrada del algoritmo estrella.
-***deflst***: marcará el camino a seguir a la hora de devolver la solución total.

```
15
16      def __init__(self, origen, destino, criterio):
17          super().__init__()
18          self.origen = origen
19          self.destino = destino
20          self.criterio = criterio
21          self.net = net.Graph()
22          self.actual = None
23
24          estacionesParse = parse('coordenadas.json')  # en coordenadas ponemos la info del archivo coordenadas
25
26          self.estaciones = []
27          for n in estacionesParse:
28              self.estaciones.append(n['Estacion'])  # aqui se ponen solo las estaciones
29
30          aristasParse = parse('aristas.json')  # en aristas ponemos la info de las aristas
31
32          self.aristas = []
33          for n in aristasParse:
34              self.aristas.append((n['Origen'], n['Destino'], n['Criterio'][criterio]))
35
36          self.openlst = []
37          self.closedlst = []
38          self.deflst = []
39
```

En la función *inicialización* se introducen las estaciones como nodos del grafo y las aristas como las aristas que unen los nodos del grafo. También se inicializan las variables G y F de los nodos para cuando haya que realizar los cálculos. Por último en el array coords se introducen las coordenadas: latitud y longitud. Tras eso, damos paso al algoritmo

```
def inicializacion(self):
    self.net.add_nodes_from(self.estaciones)
    self.net.add_weighted_edges_from(self.aristas)  # añadimos las aristas como aristas que unen las estaciones
    self.net.nodes[self.origen]['G'] = 0
    self.net.nodes[self.origen]['F'] = 0  # inicializo

    coords = parse('coordenadas.json')
    for n in coords:
        self.net.nodes[n['Estacion']]['Coord'] = [n['Latitud'], n['Longitud']]

    return self.algoritmo()
```

La función *getTodo* realiza los cálculos para obtener la G, H y F. La primera parte calcula la H. Se calcula utilizando las fórmulas presentadas en las variables n y d. Los demás pasos previos son obtener la latitud y longitud del nodo padre e hijo.

```
def getTodo(self, actual, hijo):  # esta funcion hace la H, la G y la F

        #esta parte te calcula la H
    coordAct = self.net.nodes[self.destino]['Coord']  # pillo las cordenadas del actual
    coordHijo = self.net.nodes[hijo]['Coord']  # same version hijo

    latAct = radians(coordAct[0])
    lonAct = radians(coordAct[1])
    lonHj = radians(coordHijo[1])
    latHj = radians(coordHijo[0])

    lonDif = lonHj - lonAct
    latDif = latHj - latAct

    n = sin(latDif / 2) ** 2 + cos(latHj) * cos(latAct) * sin(lonDif / 2) ** 2
    d = 2 * atan2(sqrt(n), sqrt(1 - n))
    H = 6371 * d
```

Tras calcular la H, calculamos la G y la F. Para conseguir la G simplemente se saca del peso de la arista. Tras eso, se establece la G y la H del nodo con los números que hemos obtenido previamente y la F se saca de la suma de la G y la H. Tras esto, la función termina.

```
G = self.net[actual][hijo]['weight']

self.net.nodes[hijo]['G'] = self.net.nodes[actual]['G'] + G
self.net.nodes[hijo]['H'] = self.net.nodes[actual]['G'] + H
self.net.nodes[hijo]['F'] = self.net.nodes[hijo]['G'] + self.net.nodes[hijo]['H']
```

La función *vuelta* se ocupa de formar el camino definitivo que usará el algoritmo para construir almacenando en deflst(Explicado en la función init)

```
def vuelta(self, actual, origen): # metodo recursivo, vamos yendo del destino al origen a traves de padres
    self.deflst.append(actual)
    if actual != origen:
        self.vuelta(self.net.nodes[actual]['Padre'], origen)
```

Dificultades:

Una de nuestras principales dificultades fue usar un lenguaje prácticamente nuevo para nosotros. Sin embargo, después de informarnos a través de varias fuentes, encontramos varias herramientas que nos sirvieron de ayuda.

Otra de las dificultades que nos encontramos por el camino, fue la creación de una interfaz gráfica, ya que no habíamos hecho ninguna a lo largo de la carrera. Encontramos dos librerías de Python para hacerlo, PyQt5 y Tkinter, pero acabamos decantándonos por PyQt5. Al final conseguimos hacer la interfaz gráfica gracias a diferentes manuales de la librería en internet.

Conclusiones:

Pese a no tener mucha base a la hora de usar Python, ya que no es el lenguaje al que estamos habituados. Ha sido bastante satisfactorio conseguir realizar tanto el algoritmo como la interfaz gráfica.

Además, hemos podido comprender de una manera más práctica el funcionamiento de algunas herramientas de búsqueda de rutas, tales como Moovit o Google Maps, las cuales además de utilizar el algoritmo de Dijkstra, emplean el algoritmo A Estrella al ser más eficiente para rutas de mayor escala.