INTELIGENCIA ARTIFICIAL

PRÁCTICA: TRAYECTO ÓPTIMO EN METRO DE LYON

Grupo 8:

Sebastian Kay Conde Lorenzo
Lucía Qing Díaz de Sarralde Ramírez
Iván Hernández Pérez
Francisco Manuel López López
Ge Sheng
Katherine Aide Urbano Tacuri

ÍNDICE

Breve explicación del algoritmo

El **algoritmo A*** es un algoritmo de búsqueda informada. Esto quiere decir que no solo tiene información sobre la meta que debe alcanzar navegando por el espacio de estados, sino que también posee información sobre el propio espacio de estados.

En el caso del algoritmo A*, esta información viene dada por una función h: $V \rightarrow \mathbb{R}$, donde V es el conjunto de estados del espacio de estados. A esta función la llamamos la **función heurística**.

Por otro lado, el algoritmo también debe disponer de los propios datos del camino. Estos son naturales al espacio de estados y no suponen una información extra. Podríamos decir que son la *ground truth*.
Estos datos vienen dados por los pesos de las conexiones entre vértices adyacentes del grafo y son aprovechados por una función g: $V \rightarrow \mathbb{R}$ que podríamos llamar **función de coste del camino**, donde $g(n)$ devuelve el coste del camino mínimo desde el nodo inicial hasta el nodo n.

Luego, el algoritmo buscará el camino que minimice la función de coste $f(n) = g(n) + h(n)$, que funciona como un estimador de "qué tan bueno es el camino desde el nodo inicial hasta el nodo final que pasa por el nodo n". Consideramos entonces que, a menor f mejor es dicho "camino".

Por último, cada nodo v tendrá asociada una **arista de retroceso**, que es la arista que lo conecta con un nodo inmediatamente anterior en el camino mínimo desde el nodo inicial hasta v.

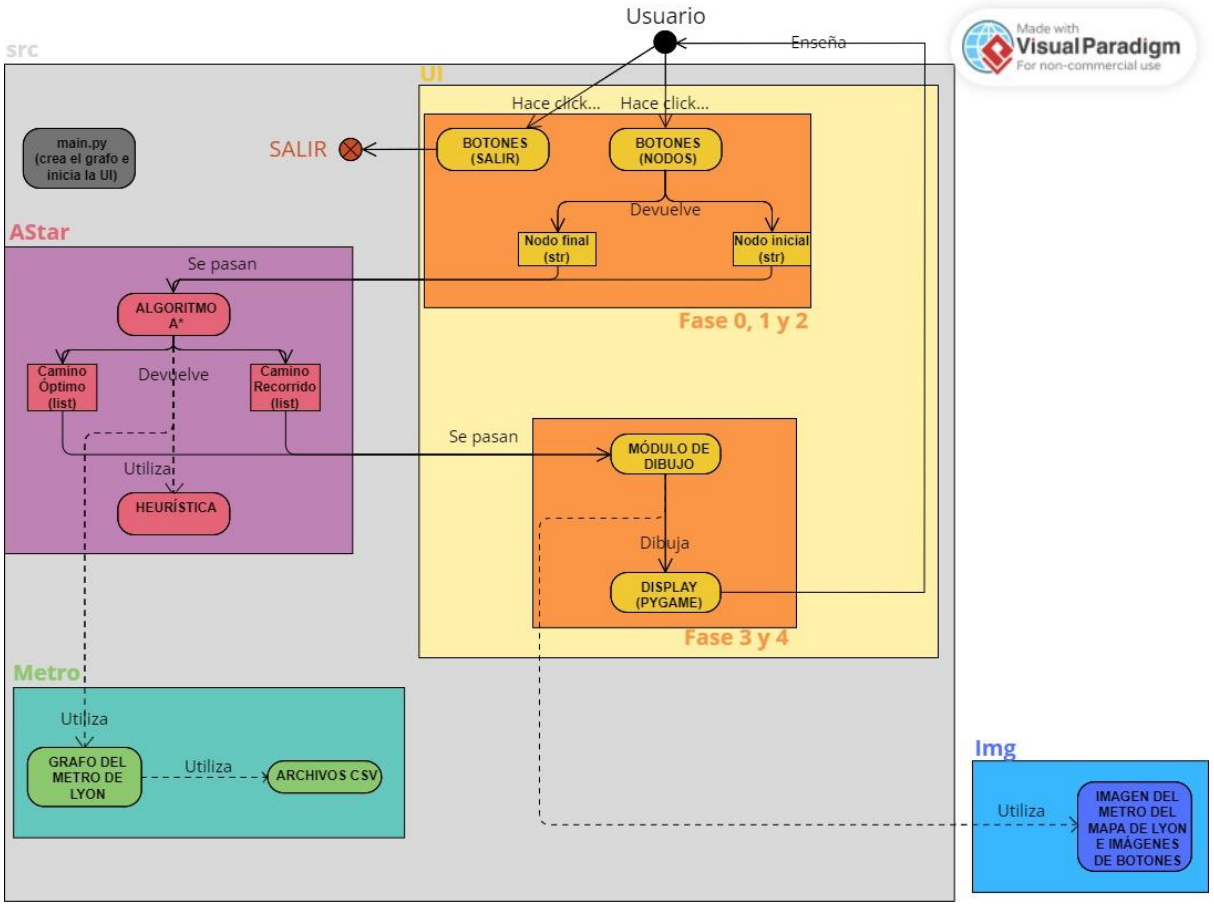
Entonces, dados dos nodos, uno inicial y uno final, el algoritmo seguirá (*a grosso modo*) los siguientes pasos:

1. Inicializar dos listas: Una a la que llamaremos **lista abierta** (OS por *Open Set*) y una a la que llamaremos **lista cerrada** (CS por *Closed Set*). La lista abierta contendrá los nodos que todavía pueden ser expandidos, mientras que la lista cerrada contendrá los nodos ya expandidos a los que no volveremos.

2. Añadimos el nodo inicial a la lista abierta. Se tendrá que $g(inicial) = 0$ y $f(inicial) = h(inicial)$. Luego, $g(n) = f(n) = \infty \, \forall n \in V$
3. Mientras OS no sea vacía:
   3.1. $actual := min_{n \in V \cap OS}\{f(n)\}$. Es decir, seleccionamos el nodo de OS con el f mínimo. Expandiremos este nodo.
   3.2. Si $actual = final$:
      3.2.1. **TERMINAR** y devolver ruta usando las aristas de retroceso
   3.3. Añadir $actual$ a la lista cerrada (ya lo hemos desarrollado)
   3.4. Para todo $v \in vecinos(actual)$:
      3.4.1. $g_{nueva}(v) = g(actual) + w(actual, v)$, donde $w(actual, v)$ es el peso de la conexión entre $actual$ y v. Con este dato buscamos estudiar si el camino que estamos tomando desde el nodo inicial hasta v actualmente es mejor que otros tomados anteriormente.
      3.4.2. Si $g_{nueva}(v) < g(v)$ y $v \in OS$:
         3.4.2.1. $g(v) = g_{nueva}(v)$, y habremos encontrado un mejor camino hasta v
         3.4.2.2. $f(v) = g_{nueva}(v) + h(v)$
         3.4.2.3. $aristaRetroceso(v) = actual$
      3.4.3. Si $v \notin OS$ y $v \notin CS$:
         3.4.3.1. Añadimos v a OS
         3.4.3.2. $g(v) = g_{nueva}(v)$, y habremos encontrado un mejor camino hasta v
         3.4.3.3. $f(v) = g_{nueva}(v) + h(v)$
         3.4.3.4. $aristaRetroceso(v) = actual$
   3.5. Volver a **3**
4. **ERROR**. No se ha podido hallar un camino desde el nodo inicial hasta el final.

Estructura del código

A continuación presentamos un diagrama explicando la estructura del código:



En el fichero *AStar.py* dentro del directorio *src/AStar*, se ha buscado definir la función del algoritmo de manera genérica. Es decir, tomando como argumento un grafo de la librería *iGraph*, una función heurística (que es un objeto genérico tipo *callable* de python) y los nombres de un nodo inicial y final.

Hemos buscado también generalizar las funciones heurísticas, haciendo que sean funciones dependientes del grafo, el vértice inicial, el vértice final, el vértice actual, la lista abierta y un diccionario de aristas de retroceso. Esto lo hicimos así con el fin de poder implementar heurísticas sin tener que modificar el algoritmo, de manera que sólo sea cuestión de pasarlas al algoritmo como parámetro.

Cálculo de las funciones h y g

La función heurística h que hemos decidido usar ha sido la que está dentro del fichero *AStar.py* llamada **euclideanDistanceHeuristic**. Esta heurística calcula la distancia euclídea entre dos puntos geográficos de la tierra (que, en este caso, serán nuestras estaciones de metro). Esto lo hace a través de una librería llamada ***geopy*** (ver anexos), usando su módulo *distance*, el cual contiene una función llamada *geodesic*, la cual calcula dicha distancia teniendo en cuenta la curvatura de la tierra (calcula la distancia geodésica).

El cálculo de g se lleva a cabo en la propia función del algoritmo A*. Donde g, para cada nodo vecino n al nodo que estamos estudiando ($actual$), se calcula como la suma de $g(actual)$ más el peso de la arista que conecta $actual$ con n. Este valor se actualiza sólo si es menor al valor que ya tenía, en el caso de que ya se haya visitado n. En caso de no haberse visitado (no está ni en la lista abierta ni en la cerrada) consideramos $g(n) = \infty$ y se actualiza el valor, logrando así tomar siempre el camino de menor peso hacia el objetivo.

Para obtener los pesos entre las conexiones de los nodos del grafo hemos usado las distancias (reales) en kilómetros entre estaciones de metro contiguas.

Por último, nos gustaría responder a la pregunta de **por qué** hemos escogido la heurística de la distancia euclídea: Se debe a que ha demostrado ser la mejor heurística entre las tres que hemos implementado. Dichas heurísticas adicionales son:

- **Heurística del árbol de mínimo alcance y máxima expansión (*Minimum Spanning Tree):***

  El cómputo de esta heurística se lleva a cabo en la función definida como ***MSTHeuristic*** dentro de *AStar.py*, que tiene en cuenta el nodo de inicio, el nodo $actual$ y el grafo.
  Para el valor de retorno se hace uso del árbol de mínimo alcance y máxima expansión, el cual usa el grafo original sin tener en cuenta el camino que se ha llevado a cabo desde el origen hasta el nodo $actual$; obteniéndose entonces un árbol que conecta a todos los nodos y que tiene

el menor coste de la suma de pesos de las aristas. Es esta suma la que es retornada.

Esta heurística ha demostrado ser peor frente a la heurística de distancia euclídea porque toma en consideración todos los pesos del árbol, salvo los del camino recorrido, por lo que realmente no es una heurística demasiado "focalizada". En la demostración gráfica, se puede ver como el algoritmo "divaga" mucho antes de llegar a la meta.

- **Heurística de Djikstra:**

Esta heurística se calcula en la función ***djikstraHeuristic*** dentro de *AStar.py*, y para cada nodo n devuelve la suma de los pesos de las conexiones entre vértices adyacentes en el camino mínimo entre n y $final$ utilizando el algoritmo de Djikstra. Este algoritmo es aplicado sobre el grafo sin el camino mínimo recorrido hasta el momento entre $inicial$ y n.

Nuevamente, esta heurística ha demostrado ser mucho peor frente a la heurística de la distancia euclídea porque, dado que estamos eliminando el camino recorrido, muchas veces el algoritmo de Djikstra no puede encontrar el camino mínimo, por lo que la función retorna 0 y, nuevamente, nuestro agente se queda "divagando" antes de llegar a la meta.

Por último, vale la pena destacar que estas tres heurísticas (la distancia euclídea, el árbol de alcance mínimo y la heurística de Djikstra) retornan valores aproximando la distancia al nodo $final$ desde $actual$ por **subestimación**, por lo que tenemos garantizado que el agente (el algoritmo A*), encontrará el camino mínimo.

Detalles sobre la implementación del algoritmo

Si se inspecciona el archivo *AStar.py* dentro del directorio *src/AStar*, puede uno darse cuenta de que hay dos implementaciones del algoritmo A*.

Una, la que está comentada, es la versión más "tradicional/clásica" del algoritmo, que incluye una lista abierta, una cerrada, un *map* de aristas de retroceso, una función de coste g, una función heurística h y una función $f = g + h$.

La otra versión, la que no está comentada, es una implementación que contiene los mismos elementos salvo la lista cerrada. Añade también una lista donde se añaden los nodos visitados ordenadamente para que la interfaz gráfica pueda luego pintar los pasos intermedios del algoritmo.

Esta segunda versión no utiliza la lista cerrada porque aprovecha el hecho de que los nodos en esta lista no van a ser visitados nuevamente. Se vale entonces únicamente de la lista abierta.

Hemos decidido incluir esta versión porque nos ha parecido un poco más "elegante" que la implementación clásica. Nos hemos topado con la idea en la página de *Wikipedia* del algoritmo (ver anexos) y la hemos encontrado interesante, llevándonos eso a basar esta segunda implementación en dicha idea (o, mejor dicho, pseudocódigo).

Es relevante mencionar que, obviamente, ambas implementaciones funcionan exactamente igual. Es decir, generan los mismos pasos intermedios y generan el mismo camino.

Dificultades en la realización del proyecto

A la hora de plantear el proyecto uno de los tópicos más tratados fue la decisión de la heurística. En un principio se pensó en usar el árbol de alcance mínimo y máxima expansión (MST, por sus siglas en inglés), pero finalmente nos terminamos decantando por la heurística de la distancia euclídea, después de contrastar ambas. Luego, también se propuso el uso del algoritmo de Dijkstra para obtener un camino mínimo desde el nodo actual hasta el objetivo, pero no era más eficiente que la heurística de la distancia euclídea por los motivos expuestos en apartados anteriores.

Otra cuestión que se discutió en un principio fue el hecho de si optimizar la distancia o el tiempo, en el sentido de si el algoritmo debería buscar el camino que minimice la distancia entre un nodo origen y un nodo extremo, o si debería buscar el camino que minimice el tiempo de recorrido entre dichos nodos. Claramente la segunda opción presenta más dificultades a la hora de obtener los pesos de las conexiones entre nodos (estaciones) adyacentes, pues estos pesos deberían representar el tiempo que tarda el metro en ir de una estación a otra, y conseguir este dato resulta complicado.

Se consideró la opción de minimizar el tiempo en una primera reunión de planeación del proyecto porque nos pareció interesante poder tomar en cuenta el tiempo de transbordo que hay entre estaciones (cosa que, por ejemplo, el algoritmo de *Google Maps* no considera y, al menos en el metro de Madrid, resulta en una cantidad de tiempo considerable en la que el viajero no está avanzando hacia su destino en el metro). Sin embargo, terminamos descartando esta idea por los motivos ya expuestos. Adicionalmente, modelizar los transbordos con el formalismo de grafos suponía una dificultad que consideramos innecesaria. Es por estas razones que se acordó de manera unánime minimizar la distancia recorrida en vez del tiempo.

A la hora de construir el grafo que el algoritmo debe recorrer, consideramos varias opciones, entre ellas programarlo desde cero. Sin embargo, nos topamos con una librería llamada ***igraph*** (la cual ya hemos mencionado antes en el documento). Esta librería de python y R proporciona un objeto de tipo *Graph* que representa un grafo (dirigido, no dirigido, con pesos, sin pesos…) y nos da una interfaz sencilla para interactuar con él. Utilizando esta

librería hemos sido capaces de centrarnos en hacer un buen algoritmo, interfaz y funciones heurísticas, en vez de tener que lidiar con problemas de software relativos a la implementación de la estructura de datos que es el grafo.

Referente también a la construcción del grafo, una **decisión de diseño** que hemos tomado ha sido referirnos a los vértices por sus nombres textuales (es decir, los nombres de las estaciones de metro). Para ello, hemos añadido dichos nombres como atributos de los vértices. Hacer esto nos ha facilitado mucho el hacer referencias a los vértices y, en consecuencia, nos ha ahorrado errores relativos a la implementación del algoritmo y la interfaz gráfica.

Supuso también una dificultad reunir los datos para construir el grafo antes mencionado; en particular sus pesos. Aunque la distancia entre estaciones contiguas era algo factible de obtener (contrario al tiempo de recorrido), no fue sencillo y supuso un trabajo manual tedioso de extracción de una página web (ver anexos). Estos datos, que al final vendrían a ser los pesos del grafo, se reunieron en un excel de forma manual. Este excel es el que utiliza el código que crea el objeto de tipo *igraph.Graph*, utilizado por el algoritmo.

Luego, para la implementación de la heurística de distancia euclídea hubo que hacer un proceso similar, sólo que esta vez se extrajeron coordenadas en longitud y latitud de cada estación utilizando *Google Maps*. Estas fueron también reunidas en una tabla de excel a la cuál accede el código que crea el grafo, haciendo de dichas coordenadas atributos de los vértices.

Por último, a la hora de enfrentarnos a la implementación de la interfaz gráfica, hemos decidido utilizar la librería de python ***pygame***, la cual también proporciona una interfaz sencilla para hacer elementos gráficos interactivos. Para llevar a cabo la interfaz gráfica hemos intentado utilizar una arquitectura tipo *Model View Controller (MVC)*, y hemos dividido las distintas etapas de interacción en fases.

Anexos

A continuación presentamos algunos enlaces de interés que fueron utilizados para el proyecto:

- **Documentación de la librería *igraph*:**
  https://python.igraph.org/en/stable/

- **Datos sobre las distancias reales entre estaciones en el metro de Lyon:**
  https://www.metrolinemap.com/station/lyon/place-guichard/

- **Documentación de la librería *pygame:***
  https://www.pygame.org/docs/

- **Página usada para conseguir las coordenadas de las estaciones:**
  https://www.google.com/maps

- **Página de *Wikipedia* del algoritmo:**
  https://en.wikipedia.org/wiki/A*_search_algorithm

- **Documentación de la librería *geopy*:**
  https://geopy.readthedocs.io/en/stable/