Desarrollo e implementación del Algoritmo A*

Íñigo Araluce, Víctor Berrozpe, Daniel Díaz,
Juan Hernández y José Manuel Vega

Diciembre 2020

1 Introducción

El objetivo es desarrollar una aplicación que encuentre el camino mínimo entre dos vértices de un grafo con el algoritmo de búsqueda A*. La implementación se ha adaptado concretamente para que los vértices representen estaciones del metro de las líneas 1, 2 y 3 de Atenas. La distancia entre dos vértices es el tiempo, en minutos, que tarda un tren en ir de una estación a otra, penalizando los trasbordos. Además, como el recorrido se puede hacer en ambas direcciones, el tiempo de ida y de vuelta es el mismo.

Nuestra aplicación está desarrollada en Python y, a pesar de estar orientada al metro de Atenas, permite hallar el camino mínimo para un grafo genérico, que se puede pasar a través de un fichero JSON.

El algoritmo de A* necesita una función heurística, h(n), para la que hemos utilizado la distancia euclídea entre dos estaciones, basándonos en el mapa del metro de Atenas.

2 Algoritmo de búsqueda

2.1 Primera aproximación

Inicialmente empezamos implementando el algoritmo de Dijkstra. A* no es más que una mejora de este, por lo que nos pareció una buena idea empezar por la versión sencilla y luego mejorarla. A continuación, escribimos un pequeño programa para pasar un grafo a JSON. El formato consiste en asignar a cada estación sus coordenadas y conexiones. Una vez implementado esto, disponíamos de un algoritmo de Dijkstra funcional que podía operar sobre cualquier grafo.

2.2 Desarrollando A*

Una vez implementado Dijkstra solo había que hacer ligeras modificaciones. La principal diferencia de A* es que utiliza la función estimadora (f(n) = g(n) + h(n)) para evaluar los nodos del grafo. En está función, g(n) representa el coste real del desplazamiento desde el origen hasta el nodo actual, mientras que h(n) es una función heurística que aproxima el coste del camino mínimo entre el nodo actual y el nodo destino.

Consideramos una penalización de 6 minutos por trasbordo, la media de su duración. Así, las estaciones que tienen intersecciones de dos líneas son representadas en nuestro grafo como dos nodos, uno por cada línea. Por ejemplo, como la estación Syntagma es la intersección entre las líneas 1 y 2, en nuestro grafo tendremos dos nodos en el mismo lugar: Syntagma(red) y Syntagma(blue), conectados por un peso de 6 minutos, la duración media del trasbordo.

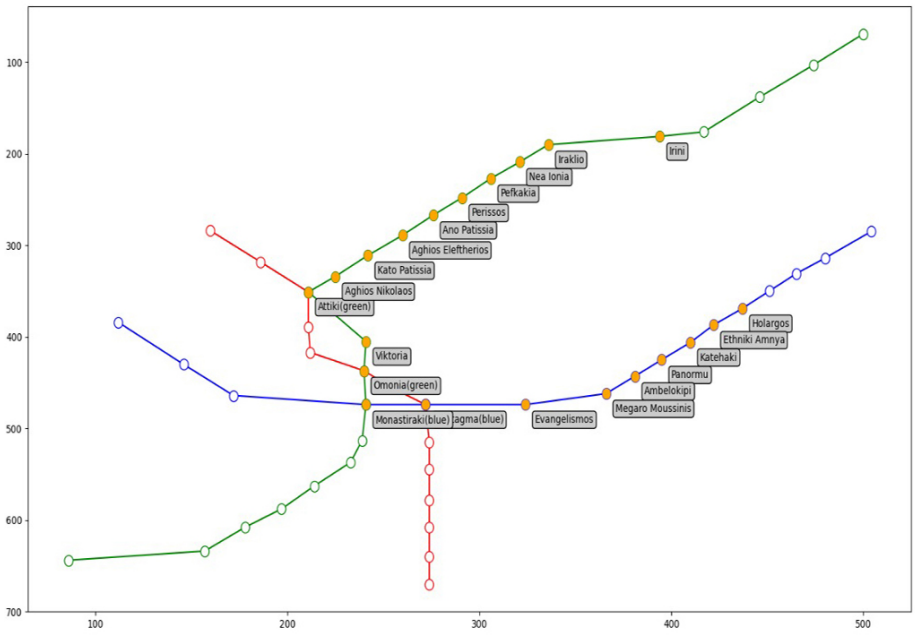
2.3 Función heurística

Una vez implementada la función g(n), con Dijkstra, el siguiente paso era implementar la función h(n), para poder calcular f(n) y completar A*.

Una vez más, la función heurística que hemos utilizado es la de la distancia euclídea entre estaciones, que hemos calculado con la ayuda de Photoshop. A la hora de sumar las funciones g(n) y h(n) encontrábamos valores que no nos encajaban y, tras mucho buscar, caímos en la cuenta de que se nos había olvidado pasar de píxeles de distancia a minutos. La solución fue fácil: calcular la distancia en píxeles entre dos estaciones y el tiempo que las separa para estimar el factor de conversión. Así, calculamos la velocidad media del metro para que al dividir la distancia entre la velocidad nos dé el tiempo por píxel; 16.6 píxeles por minuto.

3 Interfaz gráfica

En un principio, queríamos hacer uso de alguna librería que permitiese dibujar grafos para representar las líneas del metro. Sin embargo, acabamos decantándonos por MatPlotLib, que hace uso de las coordenadas cartesianas, puesto que ya las habíamos calculado para nuestra función heurística. El camino mínimo entre dos estaciones queda marcado coloreando las estaciones que lo conforman de naranja y sus vértices reciben una etiqueta con su nombre.



4 Código

Para finalizar, comentamos las partes más esenciales del código de nuestra práctica, que está en Github: A_Star

4.1 Main

La función main es la base del programa; lee los argumentos del usuario, como el grafo y las estaciones de origen y de destino, y ejecuta el algoritmo de búsqueda.

```
def main():
    graph = parse_graph_json(read_commands())

    a_star = A_Star(graph, "Holargos", "Irini")
    a_star.run()
```

Además contamos con las funciones: read_commands, una función genérica que lee los comandos de la consola; y parse_graph_json, que nos permite abrir un grafo que se encuentre en un fichero (el cual para nosotros será el JSON ya mencionado) para poder operar con él desde Python.

```
def read_commands():
    parser = OptionParser("%prog -g <graph_json>")
    parser.add_option("-g", dest="graph", help="JSON file with graph")
    parser.add_option("-o", dest="origin", help="Station at the origin")
    parser.add_option("-t", dest="target", help="Station at the target")

    (options, args) = parser.parse_args()

    # Show help if no input graph is given
    if not options.graph:
        parser.print_help()
        exit(0)

    return options
```

4.2 Parse

Usando la biblioteca pandas formateamos la información que tenemos en ficheros CSV sobre nuestro grafo (conexiones y coordenadas) a un único archivo: graphs.json.

En parse_coordinates creamos un diccionario para cada vértice del grafo, guardando sus coordenadas, su color y sus conexiones (peso y vecino).. Representamos las estaciones con una sola línea por su nombre, y las que tienen más de una tendrán, además, el color de la línea; por ejemplo: Holargos y Syntagma(red).

```
def parse_coordinates():
    coordinates = pd.read_csv(DATA_DIR + COORDINATES_FILENAME)

    for stop in coordinates["Stops"]:
        stop_row = coordinates[(coordinates["Stops"] == stop)]

        colors = stop_row["Color"].iloc[0].split()
        is_intersection = len(colors) > 1

        # We make a different node for every line that goes through a station
        # This simplifies dealing with transshipments
        for color in colors:
            name = stop

            if is_intersection:
                name += "(" + color + ")"

            graph[name] = {
                "x": int(stop_row["x"].iloc[0]),
                "y": int(stop_row["y"].iloc[0]),
                "color": color,
                "edges": []
            }
```

Parse_connections se ocupa de añadir las aristas del grafo, teniendo en cuenta las variaciones de nombre para las aristas intersección y que las conexiones existen en ambas direcciones.

```
def parse_connections():
    connections = pd.read_csv(DATA_DIR + CONNECTIONS_FILENAME)

    for _, route in connections.iterrows():
        # A station will have more than a name if it's an intersection
        stops1 = get_stop_names_for_station(route["Stop 1"])
        stops2 = get_stop_names_for_station(route["Stop 2"])

        for stop1 in stops1:
            for stop2 in stops2:
                # Connections need to be replicated in both directions
                graph[stop1]["edges"].append([int(route["Duration"]), stop2])
                graph[stop2]["edges"].append([int(route["Duration"]), stop1])
```

4.3 A_Star

Contamos con una clase que a partir de un grafo y dos estaciones de origen y de destino encuentra el camino mínimo entre ellas, con A*.

Empieza iniciando dos listas: una lista con los nodos visitados (visited) y una cola con los hijos de los nodos visitados (queue). Inicialmente solo está el nodo origen en la cola, con todas sus varaciones.

```
class A_Star:
    def __init__(self, graph, origin, target):
        self.graph = graph
        self.gui = Gui(graph)

        # We find one of the names of the station to get the coordinates
        target_name = self.get_names_for_station(target)[0]
        self.target = Node(target, graph[target_name]["x"], graph[target_name]["y"])

        self.queue = []
        self.visited = []

        # Start the queue with the origin station
        # As it may have different names, if it's an intersection,
        # it can get added multiple times, with initial cost 0
        for name in self.get_names_for_station(origin):
            self.queue.append(Node(name, graph[name]["x"], graph[name]["y"]))
```

Veamos qué hace el run de esta clase. Comenzamos iniciando un bucle que se ejecutará siempre que haya nodos en la lista queue y que el destino no sea el nodo con menor peso. En cada iteración mostramos en la interfaz gráfica la iteración actual del algoritmo (es decir, el camino que se está evaluando) y evaluamos cada uno de los vecinos del nodo actual. Si está en la lista de visitados y su peso es mayor que el actual, lo sacamos, actualizamos su peso y lo añadimos de vuelta a la cola; si está en la cola y su peso es mayor que el actual, lo actualizamos; y si no está en ningún sitio, se lo añadimos a la cola.

Para acabar comprobamos que se ha encontrado un camino mínimo hasta el destino y, en caso contrario, tiramos un error.

```
def run(self):
    node = None

    while self.queue:
        node = min(self.queue, key = lambda node: node.f)

        self.queue.remove(node)
        self.visited.append(node)

        # If the current node is the target node, stop looking
        if node.name.split('(')[0] == self.target.name:
            break

        self.gui.draw_graph(path(node))

        for weight, name in self.graph[node.name]["edges"]:

            child = Node(name, self.graph[name]["x"], self.graph[name]["y"])
            cost = self.calculate_cost(node, child, weight)

            if child in self.visited:
                # Update the weight of a visited node
                # The children will get updated when the new node pops out of the queue
                child = find_node(self.visited, child.name)

               if child.g > cost:
                    self.update_node_in_queue(child, node, cost, is_new=True)
                    self.visited.remove(child)
            elif child in self.queue:
                # Update the weight of a queued node
                child = find_node(self.queue, child.name)

                if child.g > cost: # Being the same node, h is preserved
                    self.update_node_in_queue(child, node, cost)
            else:
                # Add the new node to the queue
                self.update_node_in_queue(child, node, cost, is_new=True)


   if node.x != self.target.x or node.y != self.target.y:
       raise ValueError("The target is inaccessible")
```

La función run utiliza las siguientes funciones auxiliares:

```
def calculate_cost(self, node, child, weight):
    cost = node.g + weight

    if self.graph[node.name]["color"] != self.graph[child.name]["color"]:
        cost += TRANSSHIPMENT_PENALTY

    return cost
```

Esta función calcula el coste para llegar a un nodo desde su padre; coste de llegar al padre, más el peso de la arista que conecta al padre con el hijo, más una penalización de 6 minutos para los trasbordos.

```
def update_node_in_queue(self, node, parent, cost, is_new=False):
    node.g = cost
    node.estimate_path_length(self.target)

    node.parent = parent

    if is_new:
        self.queue.append(node)
```

Esta función actualiza el peso estimado de un nodo en la cola, y si es un nodo nuevo lo añadirá.

```
def show(self, node):
    node_path = path(node)

    print("Minimum weight: " + str(node.g))
    print("Minimum path: " + ", ".join(n.name for n in node_path))

    self.gui.draw_graph(node_path, permanent=True)
```

Esta función imprime el peso y el camino mínimo de un nodo y lo muestra por la interfaz gráfica.

```
def path(node):
    """ Go up the tree to get the best path of a node """
    nodes = [node]

    while node.parent:
        node = node.parent
        nodes.append(node)

    nodes.reverse()

    return nodes
```

Esta función calcula el camino que se ha tomado para llegar a un nodo, iterando de padre en padre hasta llegar al origen.

```
def find_node(nodes, name):
    """ Find a node in a list of nodes """
    for node in nodes:
        if node.name == name:
            return node

    raise ValueError("The node isn't in the list!")
```

Esta función busca un nodo dentro de una lista, por el nombre, y lo devuelve, dando un error si no lo encuentra.

4.4 Gui

La interfaz gráfica de nuestra aplicación, como explicamos en su apartado, está implementada con MatPlotLib. Usamos dos funciones: la principal, draw_graph, y una auxiliar, get_rect.

La función draw_graph representa un camino sobre un grafo dado. Dibujamos el grafo con los nodos en blanco y las aristas del color de la línea que conecta las estaciones. Además, marcamos el camino, coloreando sus vértices con naranja, y redimensionamos la interfaz gráfica a 1280x720 píxeles (720p) para mayor claridad.

```
def draw_graph(self, path, permanent=False):
    lines = {
        "red": ['Aghios Antonios', 'Sepolia', 'Attiki(red)', 'Larissa Station', 'Metaxourghio', 'Omonia(red)', 'Pan
        "blue": ['Egaleo', 'Eleonas', 'Kerameikos', 'Monastiraki(blue)', 'Syntagma(blue)', 'Evangelismos', 'Megaro M
        "green": ['Piraeus', 'Faliro', 'Moschato', 'Kallithea', 'Tavros', 'Petralona', 'Thissio', 'Monastiraki(green
    }

    for color in lines.keys():
        plt.plot(*self.get_rect(lines[color]), color=color, marker="o", markersize=10, markerfacecolor="white")

    for node in path:
        plt.plot(node.x, node.y, color="orange", marker="o", markersize=9)


    plt.gca().invert_yaxis()
    plt.tight_layout()
    plt.xticks([])
    plt.yticks([])
    manager = plt.get_current_fig_manager()
    manager.resize(1280,720)

    if permanent:
        for node in path:
           plt.annotate(node.name,[node.x +5,node.y+20],bbox = dict(boxstyle ="round", fc ="0.8"),fontsize= 9)
        plt.ioff()
        plt.show()
    else:
        plt.draw()
        plt.pause(0.001)
        plt.clf()
```

get_rect calcula la línea (x e ys) que une a una lista de estaciones.

```
def get_rect(self, stations):
    x = []
    y = []

    for station in stations:
        x.append(self.data[station]["x"])
        y.append(self.data[station]["y"])

    return [x, y]
```

4.5 Node

La clase Node es una clase auxiliar que representa los nodos del grafo. Cada nodo tendrá como atributos su nombre, su nodo de procedencia, o padre, sus coordenadas x e y y valores para f y g.

También tenemos un método auxiliar, estimate_path_length, que, dado un nodo objetivo, calcula el valor de la función heurística h(n) según las directrices antes explicadas. Además hemos sobrecargado el operador ==, siendo dos nodos iguales si sus nombres lo son, y la conversión a string para pruebas.

```
class Node:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y

        self.parent = None

        self.g = 0
        self.f = 0

    def __str__(self):
        if self.parent:
            return "Name: {0}, Parent: {1}".format(self.name, self.parent.name)
        else:
            return "Name: " + self.name

    def __eq__(self, node):
        return self.name == node.name

    def estimate_path_length(self, target):
        h = math.sqrt((self.x - target.x)**2 + (self.y - target.y)**2) / PIXELS_TO_SECONDS
        self.f = self.g + h
```

References

[1] *Frecuencia de paso de los trenes. Athens Metro - Lines, map, fares and timetable. (2020). Accedido el 5 diciembre de 2020, en https://www.introducingathens.com/metro*

[2] *Tiempos entre estaciones. Atikko Metro : Mapa del metro de Atenas, Grecia. (2020). Accedido el 5 de diciembre de 2020, en https://mapa-metro.com/es/Grecia/Atenas/Atenas-Metro-mapa.htm*

[3] *Algoritmo A*. A* search algorithm. (2020). Accedido el 6 de diciembre 2020, en https://en.wikipedia.org/wiki/A*_search_algorithm*

[4] *Documentación de Pandas. Pandas documentation — pandas 1.1.5 documentation. (2020). Accedido el 6 de diciembre de 2020, en https://pandas.pydata.org/docs/*

[5] *Documentación de MatPlotLib. User's Guide — Matplotlib 3.3.3 documentation. (2020). Accedido el 5 de diciembre de 2020, en https://matplotlib.org/users/index.html*

[6] *Documentación de MatPlotLib (Ejemplos y Sumario). Matplotlib.pyplot - Matplotlib 3.3.3 documentation. (2020). Accedido el 5 de diciembre de 2020, en https://matplotlib.org/api/pyplot_summary.html*