



Licenciatura em Ciência de Dados - 1º ano

Relatório: "Rede do Metro de Londres: visualização e estudo usando um grafo"

Unidade Curricular de Estruturas de Dados e Algoritmos

31 de maio de 2023

Discentes: João Dias nº 110305 / David Franco nº 110733

2022/2023

A. INTRODUÇÃO

O presente relatório tem como objetivo representar, visualizar e analisar a rede do metro de Londres, através da construção de um grafo. Esta representação permitirá explorar diversos cenários, análises e simulações com o cálculo do caminho mais curto pelo algoritmo de Dijkstra, que permitirá não só definir rotas ótimas entre dois pontos, como melhorar significativamente o planeamento de viagens pela capital inglesa.

Os gráficos interativos aqui apresentados permitirão compreender melhor os diferentes componentes deste sistema, identificar as estações centrais e avaliar as principais ligações da cidade, contribuindo assim para uma melhor percepção das potencialidades deste meio de transporte.

A par e passo, poderá ver-se de forma detalhada a implementação dos diversos métodos e algoritmos utilizados na análise do grafo, bem como as respectivas justificações que foram guiando os processos de tomada de decisão nas secções ao longo do trabalho.

B. DESENVOLVIMENTO: MÉTODOS E RESULTADOS

B.1. AQUISIÇÃO E PRÉ-PROCESSAMENTO DOS DADOS

Datasets

Para efeitos de análise, a rede do metro de Londres é aqui representada pela classe LondonNetworkGraph, que contém um grafo direcionado, cujos vértices representam as estações, obtidas do dataset stations.csv, e cujas arestas representam as ligações entre as diversas estações, obtidas do dataset connections.csv.

O dataset stations.csv apresenta a informação relativa ao id único de cada estação, a sua localização (coordenadas de latitude e longitude), o nome, a zona onde se insere, o número total de linhas que passam pela estação e a indicação se a estação é servida por uma linha ferroviária (considerando-se 0 para não e 1 para sim). Já o dataset connections.csv apresenta a informação relativa à linha, o id único de cada estação, separando as estações de partida e de chegada, a distância em quilómetros entre as duas estações e os respectivos tempos em minutos para percorrer a ligação fora do horário de ponta, em horas de ponta no período da manhã (onde se assumiu que a expressão “0700-1000” no nome da variável representa o intervalo entre as 7h00 e as 10h00) e entre horas de pontas (onde se assumiu que a expressão “1000-1600” no nome da variável representa o intervalo entre as 10h00 e as 16h00). Como parte da expressão - “1000” - aparece em ambos os nomes das variáveis, seguiu-se a lógica dos intervalos matemáticos, onde o primeiro elemento é fechado e o segundo elemento é aberto, considerando-se assim o período da manhã entre as 7h00 e as 9h00 e o período entre picos entre as 10h00 e as 16h00.

Classe LondonNetworkGraph()

Na classe LondonNetworkGraph, a inicialização da instância do grafo (Figura 1) é feita utilizando um MultiDiGraph da biblioteca NetworkX, um tipo de grafo direcional que permite ter múltiplas arestas paralelas, com os mesmos pares de vértices, o que é útil quando há diferentes linhas de metro que conectam as mesmas estações. A título de exemplo considerem-se as viagens da estação 49 à 87, que tanto podem ser realizadas através da linha 1 como da linha 9, o que ilustra a necessidade de consideração de arestas paralelas na construção do grafo.

```
class LondonNetworkGraph:

    def __init__(self):
        self.graph = nx.MultiDiGraph()
        self.weights = {}
```

Fig.1 - Implementação do método construtor

O dicionário weights guarda informações sobre as conexões e os seus pesos e será posteriormente utilizado essencialmente para a implementação do algoritmo de Dijkstra.

Funções stations e connections

As funções stations e connections vão ler os dados das estações e conexões do metro de Londres a partir dos ficheiros CSV disponibilizados e guardar estes mesmos valores na classe LondonNetworkGraph.

```
def stations(self, file_path):
    stations_dataframe = pd.read_csv(file_path)
    for index, row in stations_dataframe.iterrows():
        station_id = row['id']
        latitude = row['latitude']
        longitude = row['longitude']
        name = row['name']
        display_name = row['display_name']
        zone = row['zone']
        total_lines = row['total_lines']
        rail = row['rail']
        self.graph.add_node(station_id, latitude=latitude,
                            longitude=longitude, name=name,
                            display_name=display_name,
                            zone=zone, total_lines=total_lines,
                            rail=rail)
```

Fig.2 - Implementação do stations

Na Figura 2, o método stations será utilizado para ler o ficheiro stations.csv e adicionar cada uma das estações como um vértice no grafo, juntamente com as suas variáveis como atributos de vértice.

```
def connections(self, file_path):
    connections_dataframe = pd.read_csv(file_path)
    for index, row in connections_dataframe.iterrows():
        line = row['Line']
        from_station = row['From Station Id']
        to_station = row['To Station Id']
        distance = row['Distance (Kms)']
        off_peak_time = row['Off Peak Running Time (mins)']
        am_peak_time = row['AM peak (0700-1000) Running Time (Mins)']
        inter_peak_time = row['Inter peak (1000 - 1600) Running time (mins)']
        self.graph.add_edge(from_station, to_station, line=line,
                            distance=distance, off_peak_time=off_peak_time,
                            am_peak_time=am_peak_time,
                            inter_peak_time=inter_peak_time)
        self.weights[(from_station, to_station, line)] = (distance, off_peak_time,
                                                          am_peak_time,
                                                          inter_peak_time)
```

Fig.3 - Implementação do connections

Por sua vez, na Figura 3, a função connections irá receber o ficheiro connections.csv e adicionar cada ligação como uma aresta no grafo, em conjunto com as suas informações como atributos de aresta. Paralelamente, a informação relativa às estações de partida, de chegada e linhas e os seus respetivos pesos serão guardados no dicionário weights para utilizar posteriormente para a construção do algoritmo de Dijkstra.

B.2. MÉTODOS DESENVOLVIDOS PARA A ANÁLISE DOS DADOS

Métodos de cálculo e contagem de estações e conexões

```
def n_stations(self):
    return self.graph.number_of_nodes()
```

Fig. 4 - Implementação do n_stations

Na Figura 4, a função n_stations devolve o número total de vértices (estações) que se encontram no grafo, permitindo ter uma noção do panorama geral em termos da dimensão do metro de Londres.

```
def n_stations_zone(self):
    stations_by_zone = {}
    for station_id in self.graph.nodes():
        zone = self.graph.nodes[station_id]['zone']
        lower_zone = int(zone)
        upper_zone = lower_zone + 1
        if lower_zone in stations_by_zone:
            stations_by_zone[lower_zone] += 1
        else:
            stations_by_zone[lower_zone] = 1
        if zone % 1 != 0:
            if upper_zone in stations_by_zone:
                stations_by_zone[upper_zone] += 1
            else:
                stations_by_zone[upper_zone] = 1
    return stations_by_zone
```

Fig. 5 - Implementação do n_stations_zone

Já a Figura 5 representa a função n_stations_zone, que devolve um dicionário que contém nas chaves as zonas do metro de Londres e nos valores o número de estações. O método percorre todas as estações, à medida que conta o número de estações existentes em cada zona, guardando essas informações no dicionário stations_by_zone.

O método obtém a zona de cada estação e arredonda para o valor inteiro inferior. De seguida, confirma se a zona já existe no dicionário. Caso já exista, incrementa uma unidade ao contador de estações para essa zona. Caso contrário, cria a entrada da zona no dicionário com um valor inicial de 1.

Se a zona possuir uma parte fracionária, nomeadamente .5, também é considerada a zona inteira imediatamente a seguir (upper_zone), realizando-se o mesmo método de contagem no dicionário.

Esta função ilustra a densidade e amplitude da rede de metro em cada zona em concreto, permitindo assim identificar áreas com maior ou menor concentração de estações.

```
def n_edges(self):
    return self.graph.number_of_edges()
```

Fig. 6 - Implementação do n_edges

A Figura 6 ilustra a função n_edges que tem como objetivo devolver o número total de arestas (ligações) do grafo, obtidas utilizando a função da biblioteca networkx number_of_edges. Esta função disponibiliza informações sobre a conectividade e complexidade da rede.

```
def n_edges_line(self):
    edges_by_line = {}
    for node_A, node_B, key in self.graph.edges(keys=True):
        line = self.graph[node_A][node_B][key]['line']
        if line in edges_by_line:
            edges_by_line[line] += 1
        else:
            edges_by_line[line] = 1
    return edges_by_line
```

Fig. 7 - Implementação do n_edges_line

O método n_edges_line, ilustrado na Figura 7, devolve o número de ligações por linha, começando por percorrer todas as arestas do grafo e armazenando na variável line o valor da linha associado a cada aresta. De seguida, verifica-se se essa linha já está contida no dicionário edges_by_lines e, em caso afirmativo, incrementa-se em uma unidade o contador da respetiva linha. Caso contrário, acrescenta-se essa linha ao dicionário, com o seu valor a 1.

```
def mean_degree(self):
    return self.n_edges() / self.n_stations()
```

Fig. 8 - Implementação do mean_degree

Para calcular o grau médio das estações, o método da Figura 8 divide o número total de arestas, dado pela função n_edges() pelo número total de vértices do grafo, dado pela função n_stations(). Como resultado obtém-se quantas arestas, em média, estão conectadas a cada vértice do grafo.

```
def mean_weight(self, weight):
    total_weight = 0
    count = 0

    for node_A, node_B, data in self.graph.edges(data=True):
        if weight in data:
            total_weight += data[weight]
            count += 1

    if count > 0:
        return total_weight / count
    else:
        return 0
```

Fig. 9 - Implementação do mean_weight

A função mean_weight, na Figura 9, calcula a média de um determinado peso ('distance', 'off_peak_time', 'am_peak_time' ou 'inter_peak_time') das arestas do grafo. Para tal, percorre todas as arestas do grafo e, se o peso passado no parâmetro existir nos atributos das arestas, o valor desse peso é adicionado à variável total_weight e a variável count é incrementada em uma unidade para registar o número de arestas, para cálculo posterior da média. Depois disso, a função verifica se o contador em

count é maior que zero para evitar a divisão por zero e se houver pelo menos uma aresta com o peso especificado, será devolvida a média dos pesos, dividindo o total_weight pelo count. Caso contrário, o valor 0 será devolvido.

Visualização do grafo

O método responsável pelo desenho do grafo é o `visualize(self, visualization='folium')`, onde no parâmetro visualization se podem escolher as opções 'folium' (padrão), 'matplotlib' e 'networkx' para a apresentação do grafo utilizando a biblioteca escolhida.

```
plt.figure(figsize=(14, 10))
node_size = 250
if visualization == 'networkx':
    nx.draw(self.graph, with_labels=True, node_size=node_size)
    plt.show()
```

Fig. 10 - Implementação parcial do visualize (utilizando networkx)

A Figura 10 ilustra parte da implementação do visualize, com recurso à biblioteca networkx, onde o código utiliza a função draw da biblioteca em questão para desenhar o grafo. Os vértices do grafo são exibidos com rótulos, representando os nomes das estações. O tamanho dos vértices é definido como node_size, que é configurado para 250. Por fim, a função plt.show() é utilizada para exibir a visualização do grafo desenhado.

```
elif visualization == 'matplotlib':
    node_positions = {node_id: (self.graph.nodes[node_id]['longitude'],
                                self.graph.nodes[node_id]['latitude'])
                      for node_id in self.graph.nodes}

    fig, ax = plt.subplots(figsize=(14, 10))

    for edge in self.graph.edges:
        node_a = edge[0]
        node_b = edge[1]
        x = [node_positions[node_a][0], node_positions[node_b][0]]
        y = [node_positions[node_a][1], node_positions[node_b][1]]
        ax.plot(x, y, color='gray', linewidth=0.5, alpha=0.5)

    for node_id, pos in node_positions.items():
        ax.plot(pos[0], pos[1], 'o', color='darkblue', markersize=5)

    for node_id, pos in node_positions.items():
        ax.text(pos[0], pos[1], self.graph.nodes[node_id]['name'], fontsize=10)

    ax.set_xlabel('Longitude')
    ax.set_ylabel('Latitude')
    ax.set_title('London Metro Network Graph')

    ax.set_xticks([])
    ax.set_yticks([])

    ax.set_xlim(min(node_positions.values(), key=lambda x: x[0])[0] - 0.01,
                max(node_positions.values(), key=lambda x: x[0])[0] + 0.01)
```

```

        ax.set_ylim(min(node_positions.values(), key=lambda x: x[1])[1] - 0.01,
                    max(node_positions.values(), key=lambda x: x[1])[1] + 0.01)

        plt.tight_layout()
        plt.show()

```

Fig.11 - Implementação parcial do visualize (utilizando matplotlib)

Na Figura 11 pode ver-se a implementação parcial do visualize, com recurso à biblioteca matplotlib, onde as posições dos vértices do grafo são determinadas pelas coordenadas de longitude e latitude armazenadas nos dados do grafo, que serão guardadas em node_positions. De seguida, é criada uma figura e um eixo para desenhar o grafo. As arestas são desenhadas como linhas cinzas e os vértices são representados por círculos azuis. Os nomes das estações são adicionados como rótulos de texto aos vértices. As configurações como rótulos dos eixos e limites são também definidas e no final o gráfico é exibido.

```

    elif visualization == 'folium':
        center_lat = sum(node['latitude'] for node in self.graph.nodes.values()) / len(self.graph.nodes)
        center_lon = sum(node['longitude'] for node in self.graph.nodes.values()) / len(self.graph.nodes)

        map_graph = folium.Map(location=[center_lat, center_lon], zoom_start=12)

        for edge in self.graph.edges:
            node_a = edge[0]
            node_b = edge[1]
            coords = [(self.graph.nodes[node_a]['latitude'],
                       self.graph.nodes[node_a]['longitude']),
                      (self.graph.nodes[node_b]['latitude'],
                       self.graph.nodes[node_b]['longitude'])]

            popup_text = f"Line: {self.graph.edges[edge]['line']}"

            folium.PolyLine(coords, color='darkblue', weight=2, opacity=0.5,
                             popup=popup_text).add_to(map_graph)

        for node_id in self.graph.nodes:
            node = self.graph.nodes[node_id]
            folium.Marker(location=[node['latitude'], node['longitude']],
                          popup=node['name']).add_to(map_graph)

        display(map_graph)

```

Fig.12 - Implementação parcial do visualize (utilizando Folium)

Nesta parte do código, representada na Figura 12, a visualização é feita através da biblioteca Folium. O mapa é inicializado com uma localização central calculada a partir das coordenadas médias de latitude e longitude das estações presentes no grafo, seguindo-se de um loop que percorre todas as arestas do grafo, obtendo os vértices de origem e destino e as suas respetivas coordenadas de latitude e longitude. Essas informações são usadas para criar uma linha poligonal no mapa, representando a conexão entre as estações. É também adicionado um pop-up para exibir o número da linha associada à conexão, que aparece ao clicar nela no mapa interativo. Além disso, o outro for loop percorre todas as estações do

grafo, adicionando marcadores no mapa para cada estação, exibindo o nome da estação como pop-up. No final, o mapa é exibido usando a função `display()` para a visualização no ambiente de execução.

Algoritmo de Dijkstra (implementação em Python)

O método `shortest_path(self, source, target, peso)` é responsável por calcular o caminho mais curto entre duas estações do grafo do metro de Londres, tendo em conta o peso associado às conexões. O método procura aplicar o algoritmo de Dijkstra, cuja implementação será detalhada abaixo.

```
start_nodes = [i[0] for i in self.weights]
end_nodes = [i[1] for i in self.weights]
all_nodes = set(start_nodes + end_nodes)
distances = {}
predecessors = {}
for station in all_nodes:
    distances[station] = float('inf')
distances[source] = 0
```

Fig. 13 - Implementação parcial do `shortest_path` (inicializar distâncias)

A Figura 13 mostra a parte da implementação do método responsável por inicializar as distâncias entre as estações do grafo. As variáveis `start_nodes` e `end_nodes` recorrem a uma list comprehension para percorrer o dicionário dos pesos¹ e obter as estações de partida e de chegada, respetivamente. De seguida, a `all_nodes` é utilizada para criar um conjunto de todas as estações existentes, sem que se repitam as que existem tanto como estações de partida como de destino. Este conjunto é percorrido e cada estação é inicializada com uma distância infinita (`float('inf')`), à exceção da `source`, que representa a estação de partida e como tal é definida como 0. No início, todas as distâncias são desconhecidas e atribuir o valor infinito é uma maneira conveniente de representar essa falta de conhecimento inicial, já que permite ao algoritmo substituir estas distâncias à medida que progride e encontra caminhos mais curtos.

```
visited = set()
while len(visited) < len(all_nodes):
    min_distance = float('inf')
    min_station = None
    for station in all_nodes:
        if station not in visited and distances[station] < min_distance:
            min_distance = distances[station]
            min_station = station
    visited.add(min_station)
```

Fig. 14 - Implementação parcial do `shortest_path` (parte central do algoritmo de Dijkstra)

A Figura 14 ilustra a implementação da parte central do algoritmo de Dijkstra. A variável `visited` contém um conjunto vazio, que será utilizado para armazenar as estações que já foram visitadas durante o processo de encontrar o caminho mais curto. O loop `while` assegura que todas as estações são visitadas antes do término do algoritmo. E, de seguida, a variável `min_distance` é utilizada para acompanhar a menor distância encontrada a cada iteração do loop e a `min_station` será utilizada para armazenar a estação correspondente que tem a menor distância encontrada. O `for` loop percorre todas as estações existentes e a condição `if` verifica se a estação não foi visitada e se a distância atual da estação,

¹ O dicionário `self.weights` armazena na chave o tuplo (estação de partida, estação de destino, linha) e no valor o tuplo (distância, minutos fora do pico, minutos no pico da manhã, minutos entre picos).

armazenada em `distances[station]` é menor do que a menor distância atual (`min_distance`). Caso isto seja verdade, a `min_distance` é atualizada com a distância da estação atual e a `min_station` é atualizada com a estação atual, contendo assim a estação com menor distância encontrada até ao momento, que será colocada na lista das visitadas (`visited`) para que não volte a ser visitada.

```

for neighbor in all_nodes:
    if neighbor not in visited:
        for key in self.weights.keys():
            if key[:2] == (min_station, neighbor):
                edge_weight = self.weights[key][peso]
                distance = distances[min_station] + edge_weight

                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    predecessors[neighbor] = min_station

```

Fig. 15 - Implementação parcial do shortest_path (conexões da `min_station`)

As conexões da estação, representada por `min_station`, são exploradas na implementação ilustrada na Figura 15. As duas primeiras linhas da Fig. 15 garantem que todas as estações são percorridas, sem repetir visitas às estações já visitadas. O for loop que se segue itera sobre todas as chaves do dicionário, que contêm os pesos e, a partir daí, se existir uma conexão direta entre a estação (`min_station`) e o vizinho (`neighbor`), no dicionário que armazena os pesos das conexões (`self.weights`), a variável `edge_weight` irá armazenar o valor do peso² específico dessa conexão e a variável `distance` irá conter a distância total até à estação `neighbor`, considerando a distância atual da `min_station` + o peso de conexão entre as estações (`edge_weight`). De seguida, compara-se a distância total calculada (`distance`) com distância atual registada do `neighbor` e, caso seja menor, encontrou-se um caminho mais curto para chegar até ele, procedendo-se assim à atualização da distância da estação `neighbor` (`distances[neighbor]`) para o valor da distância total encontrada (`distance`), ao mesmo tempo que se guarda o predecessor da estação `neighbor` para ser a `min_station`. Estes valores guardados no dicionário `predecessors` serão fundamentais para reconstruir o caminho mais curto posteriormente.

```

path = []
current_station = target
while current_station != source:
    path.append(current_station)
    current_station = predecessors[current_station]
path.append(source)
path.reverse()
return path

```

Fig. 16 - Implementação parcial do shortest_path (reconstrução do caminho mais curto)

A última parte da implementação do algoritmo de Dijkstra em Python, responsável por reconstruir o caminho mais curto encontrado desde a estação de destino (`target`) até à estação de origem (`source`), pode ver-se na Figura 16. A lista `path` será utilizada para armazenar as estações ao longo do caminho mais curto, ao passo que a `current_station` indica o ponto de partida para percorrer o caminho de regresso à estação de origem. O loop utilizado tem como objetivo garantir a execução até que a estação de origem seja alcançada, enquanto se vai adicionando à lista (`path`) as estações ao longo do caminho, retrocedendo desde o destino até à origem. A `current_station` vai sendo atualizada com a estação

² Parâmetro que determina qual o valor do peso a ser utilizado entre a `distance`, `off_peak_time`, `am_peak_time` e `inter_peak_time`.

anterior, obtida através do dicionário predecessors, que mapeia cada estação ao seu predecessor pelo caminho mais curto. No final do loop, acrescenta-se a estação de partida à lista e inverte-se a ordem da lista, para que se obtenha o caminho mais curto desde a estação de partida até à de chegada.

Cálculo da estação mais próxima

O método `get_nearest_station(self, start, end)` é responsável por encontrar as estações mais próximas a partir de duas estações de referência: a estação de partida (start) e a estação de destino (end).

```
def get_nearest_station(self, start, end):
    min_start_distance = float('inf')
    min_end_distance = float('inf')
    nearest_start_station = None
    nearest_end_station = None

    for connection, weights in self.weights.items():
        from_station, to_station, line = connection
        if from_station == start:
            start_distance = weights[0]
            if start_distance < min_start_distance:
                min_start_distance = start_distance
                nearest_start_station = self.get_station_name(to_station)
        if from_station == end:
            end_distance = weights[0]
            if end_distance < min_end_distance:
                min_end_distance = end_distance
                nearest_end_station = self.get_station_name(to_station)

    return nearest_start_station, nearest_end_station
```

Fig.17 - Implementação do `get_nearest_station`

A Figura 17 ilustra a implementação do método `get_nearest_station`, que começa por inicializar as menores distâncias com o valor infinito para a estação de partida e de chegada de referência, nas variáveis `min_start_distance` e `min_end_distance`, respetivamente. Enquanto isso, são criadas as variáveis `nearest_start_station` e `nearest_end_station` que irão armazenar os nomes das estações mais próximas.

O `for` loop percorre o dicionário de pesos (`self.weights`) e extrai do tuplo `connection`, a estação de partida (`from_station`) e a estação de chegada (`to_station`). De seguida, é feita uma verificação para encontrar a estação de partida passada no parâmetro `start`, e caso isso se verifique, a distância da conexão, que pode ser encontrada no elemento com o índice 0 do dicionário de pesos (`self.weights`) é atribuída à variável `start_distance`, que é depois comparada com a menor distância encontrada até ao momento para a estação de partida (`min_start_distance`). Caso a `start_distance` seja menor, o valor da `min_start_distance` é atualizado para o valor da distância da conexão atual (`start_distance`) e a variável `nearest_start_station` vai ser atualizada com o valor obtido com o método `get_station_name`, que devolve o nome da estação com base no seu ID. Esta lógica é depois aplicada de forma idêntica para encontrar a estação mais próxima à estação de chegada, passada no parâmetro `end`.

O output da função corresponde ao nome da estação mais próxima da de partida e ao nome da estação mais próxima da de chegada, respetivamente.

Visualização do caminho mais curto

O método `visualize_graph_shortest_path(self, start, end, shortest_path)` é utilizado para visualizar o caminho parte entre duas estações, com recurso à biblioteca Folium para a criação de um mapa interativo.

```

def visualize_graph_shortest_path(self, start, end, shortest_path):
    center_lat = sum(node['latitude'] for node in self.graph.nodes.values()) / len(self.graph.nodes)
    center_lon = sum(node['longitude'] for node in self.graph.nodes.values()) / len(self.graph.nodes)

    map_graph = folium.Map(location=[center_lat, center_lon], zoom_start=12)

    for edge in self.graph.edges():
        start_node = edge[0]
        end_node = edge[1]
        start_coords = [self.graph.nodes[start_node]['latitude'],
                        self.graph.nodes[start_node]['longitude']]
        end_coords = [self.graph.nodes[end_node]['latitude'],
                      self.graph.nodes[end_node]['longitude']]
        folium.PolyLine([start_coords, end_coords], color='grey',
                        weight=2, opacity=0.5).add_to(map_graph)

    for node in self.graph.nodes():
        station_name = self.get_station_name(node)
        folium.Marker([self.graph.nodes[node]['latitude'],
                      self.graph.nodes[node]['longitude']],
                      popup=station_name).add_to(map_graph)

    folium.Marker([self.graph.nodes[start]['latitude'],
                  self.graph.nodes[start]['longitude']],
                  popup='Start Point',
                  icon=folium.Icon(color='green')).add_to(map_graph)
    folium.Marker([self.graph.nodes[end]['latitude'],
                  self.graph.nodes[end]['longitude']],
                  popup='End Point',
                  icon=folium.Icon(color='red')).add_to(map_graph)

    path_coords = [(self.graph.nodes[node]['latitude'],
                    self.graph.nodes[node]['longitude']) for node in shortest_path]
    folium.PolyLine(path_coords, color='black', weight=4).add_to(map_graph)

    display(map_graph)

```

Fig.18 - Implementação do `visualize_graph_shortest_path`

A implementação deste método, conforme se pode ver na Figura 18, começa por calcular o ponto médio para as latitudes e longitudes com a respetiva informação dos vértices (ou estações) que constam em `self.graph.nodes`, para que se possa centrar a visualização do mapa nesse ponto. Após isso, a variável `map_graph` é utilizada para guardar o mapa Folium criado com os pontos calculados anteriormente no centro. O código procura iterar sobre as arestas (conexões entre estações) definidas em `self.graph.edges()`, obtendo-se para cada uma, os vértices de origem e destino, bem como as respectivas coordenadas de latitude e longitude, que irão ser ligadas através de uma linha (Polylines) a cinza no mapa. Depois disso, o código procurará iterar sobre os vértices do grafo, de forma a guardar o nome da estação na variável `station_name` e a colocar um marcador no mapa na posição

correspondente à latitude e longitude de cada vértice, destacando a verde a estação de partida e a vermelho a estação de chegada. O caminho mais curto é guardado numa lista de coordenadas (path_coords), mapeando as coordenadas de latitude e longitude de cada vértice no caminho mais curto, sendo depois adicionada uma linha (Polylines) a preto para realçar visualmente o caminho mais curto entre os dois pontos. Em suma, este método recebe o grafo, as coordenadas da estação inicial e da final, bem como o caminho mais curto entre esses dois vértices e cria um mapa interativo para que seja mais fácil visualizar esse caminho.

Simulação em Python do caminho mais curto com o algoritmo de Dijkstra

A simulação em Python da aplicação do algoritmo desenvolvido entre um ponto de partida e um ponto de chegada aleatórios, numa dada hora aleatória, pode obter-se através do método desenvolvido para o efeito: generate_random_path(self, metodo='python').

```
from_stations = [i[0] for i in self.graph.edges()]
to_stations = [i[1] for i in self.graph.edges()]

x1, x2 = min(from_stations), max(from_stations)
y1, y2 = min(to_stations), max(to_stations)

start = random.randint(x1, x2)
end = random.randint(y1, y2)

while start == end:
    end = random.randint(y1, y2)
```

Fig. 19 - Implementação parcial do generate_random_path (gerar aleatoriamente dois pontos (start, end))

A Figura 19 ilustra a parte inicial da implementação do método generate_random_path ao gerar dois pontos aleatórios (start, end), que irão ser utilizados para definir o caminho mais curto entre a estação start e a estação end. A variável from_stations guarda através de uma list comprehension todas as estações de partida que se encontram na posição 0 do self.graph.edges(), enquanto que a variável to_stations procede de forma semelhante ao guardar todas as estações de destino. Note-se que os valores guardados correspondem aos valores numéricos (IDs) da estação, que variam entre 1 e 303. As variáveis x1 e x2 irão armazenar a informação relativa aos valores mínimos e máximos das estações de partida, respetivamente. E, de forma semelhante, as variáveis y1 e y2 irão armazenar os valores mínimos e máximos respectivos para as estações de chegada. A variável start irá conter um número inteiro aleatório gerado através da função randint da biblioteca random entre os valores dados por x1 e x2, enquanto que a variável end irá conter um número inteiro aleatório obtido da mesma forma entre os valores dados por y1 e y2. Esta construção garante que mesmo que sejam acrescentadas estações ao dataset, continuem a ser gerados números aleatórios compreendidos entre os mínimos e máximos das estações presentes no conjunto. O while loop é utilizado para garantir que o ponto de partida e o de chegada são diferentes.

```
hour = random.randint(1, 24)
print("Random Hour:", hour)
```

Fig. 20 - Implementação parcial do generate_random_path (gerar hora aleatória)

De seguida, a Figura 20 mostra-nos como se obtém a hora aleatória. Com recurso à biblioteca random, utiliza-se novamente a função randint para gerar um número inteiro entre 1 e 24 que irá representar a hora aleatória em que será corrida a simulação.

```
nearest_start_station, nearest_end_station = self.get_nearest_station(start, end)
```

Fig. 21 - Implementação parcial do generate_random_path (obter as estações mais próximas do start e do end)

A Figura 21 mostra como as variáveis nearest_start_station e nearest_end_station irão armazenar a estação mais próxima da estação de partida e do destino, respectivamente.

```
if metodo == 'python':
    if hour >= 7 and hour <= 9:
        shortest_path = self.shortest_path(float(start), float(end), 2)
    elif hour >= 10 and hour <= 16:
        shortest_path = self.shortest_path(float(start), float(end), 3)
    else:
        shortest_path = self.shortest_path(float(start), float(end), 1)

elif metodo == 'networkx':
    if hour >= 7 and hour <= 9:
        weight_attribute = 'am_peak_time'
    elif hour >= 10 and hour <= 16:
        weight_attribute = 'inter_peak_time'
    else:
        weight_attribute = 'off_peak_time'
    shortest_path = nx.shortest_path(self.graph, source=start,
                                      target=end, weight=weight_attribute)
```

Fig. 22 - Implementação parcial do generate_random_path (escolha do método e dos pesos do algoritmo)

A Figura 22 ilustra as condições estabelecidas, em primeiro lugar, para o método a aplicar, que tanto pode utilizar a implementação do algoritmo de Dijkstra em Python desenvolvida e apresentada nas secções anteriores, como pode utilizar o algoritmo de Dijkstra da biblioteca NetworkX (por padrão, caso não seja passado esse parâmetro, o método utilizado é a implementação em Python). Além disso, a Figura 22 mostra também um conjunto de ifs que, com base na hora aleatória gerada, irão escolher o peso das arestas que mais se adequa a esse horário para o cálculo do caminho mais rápido. Para tal, considerou-se que o pico da manhã (am_peak_time) decorre entre as 7h e as 9h. O tempo entre picos (inter_peak_time) decorre entre as 10h e as 16h. E todos os outros horários que não estes, são considerados fora do horário de ponta (off_peak_time). Consoante a hora gerada aleatoriamente, o shortest_path irá adequar o seu último parâmetro ao peso das arestas correspondente.

```
print("Start:", start)
print("End:", end)
print("Shortest Path:", shortest_path)
print("Nearest start station:", nearest_start_station)
print("Nearest end station:", nearest_end_station)
self.visualize_graph_shortest_path(start, end, shortest_path)
```

Fig. 23 - Implementação parcial do generate_random_path (apresentação e visualização da simulação)

O método termina ao apresentar os resultados da simulação, conforme mostra a Figura 23. Serão imprimidas as estações aleatórias geradas (start, end), a lista com o caminho mais curto/rápido entre essas estações, bem como o nome da estação mais próxima da estação de partida e do destino. Ao mesmo tempo, chama-se o método visualize_graph_shortest_path que irá apresentar um mapa interativo da biblioteca Folium, com o caminho mais curto/rápido e os respectivos pontos de partida e de destino definidos.

B.3. RECOLHA E ANÁLISE DOS RESULTADOS

Com base nos datasets analisados na classe LondonNetworkGraph e através da aplicação do método `n_stations`, é possível ver que a rede do metro de Londres é atualmente constituída por 302 estações (vértices no grafo) e através da aplicação do método `n_edges` pode ver-se que existem atualmente 743 ligações entre as estações (arestas no grafo).

Se olharmos à distribuição das estações por zona, com a aplicação do método `n_stations_zone`, percebe-se que a zona 1 abrange 64 estações, a zona 2 abrange 96 estações, a zona 3 abrange 70 estações, a zona 4 abrange 44 estações, a zona 5 abrange 29 estações, a zona 6 abrange 20 estações, a zona 7 abrange 3 estações, a zona 8 abrange 2 estações, a zona 9 abrange apenas 1 estação e a zona 10 abrange 2 estações.

Considerando-se agora as ligações por linha, com a aplicação do método `n_edges_line`, concluiu-se que a linha 1 apresenta 48 ligações, a linha 2 apresenta 98 ligações, a linha 3 apresenta 54 ligações, a linha 4 apresenta 118 ligações, a linha 5 apresenta 16 ligações, a linha 6 apresenta 54 ligações, a linha 7 apresenta 52 ligações, a linha 8 apresenta 70 ligações, a linha 9 apresenta 102 ligações, a linha 10 apresenta 99 ligações, a linha 11 apresenta 30 ligações e a linha 12 apresenta 2 ligações.

O grau médio de arestas incidentes nas estações (vértices) do grafo é obtido através do método `mean_degree` e é de aproximadamente 2.46. Isso significa que, em média, cada estação está conectada a cerca de 2.46 outras estações no grafo.

O método `mean_weight` permite calcular a média dos pesos, tendo-se obtido assim uma distância média de 1.31 kms entre estações, um tempo médio de percurso entre estações de 2.08 minutos (fora das horas de ponta), de 2.44 minutos (durante o pico da manhã) e de 2.39 minutos (entre picos).

O método `visualize` permite-nos ver a representação do grafo com diferentes bibliotecas, sendo a do `networkx` dada pela representação da Figura 24, que nos ajuda desde logo a identificar que existem várias estações que não estão ligadas à estrutura da rede principal do metro, a Figura 25 representa o desenho do grafo com o `Matplotlib`, que nos ajuda a ter uma melhor percepção das estações periféricas e das suas distâncias até ao núcleo da rede, que é muito denso, e a Figura 26 representa o desenho do grafo com o `Folium`, que por ser um mapa interativo nos permite aprofundar as análises e o nível das visualizações.

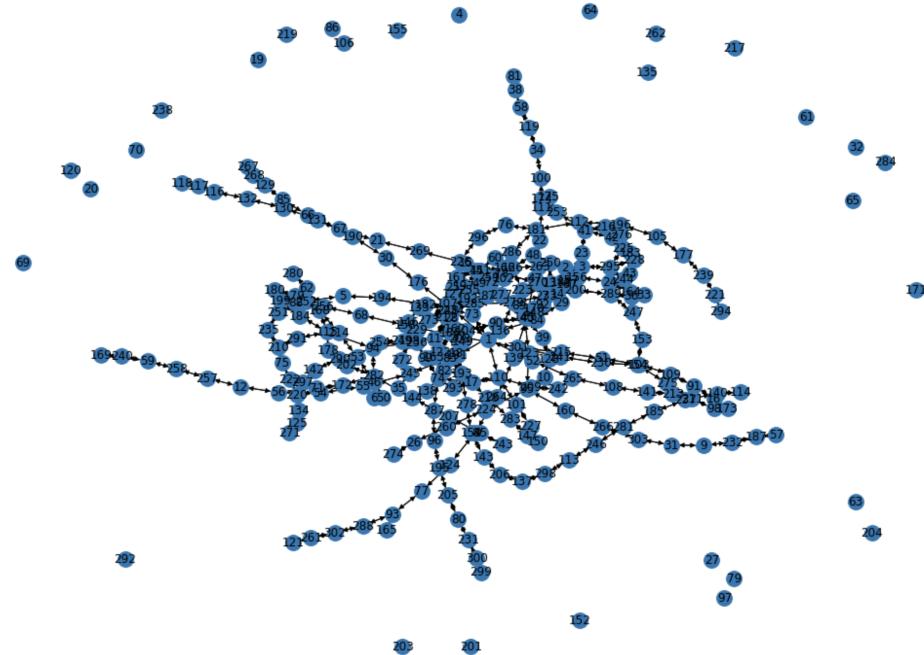


Fig. 24 - Visualização do grafo através da biblioteca NetworkX



Fig. 25 - Visualização do grafo através da biblioteca Matplotlib

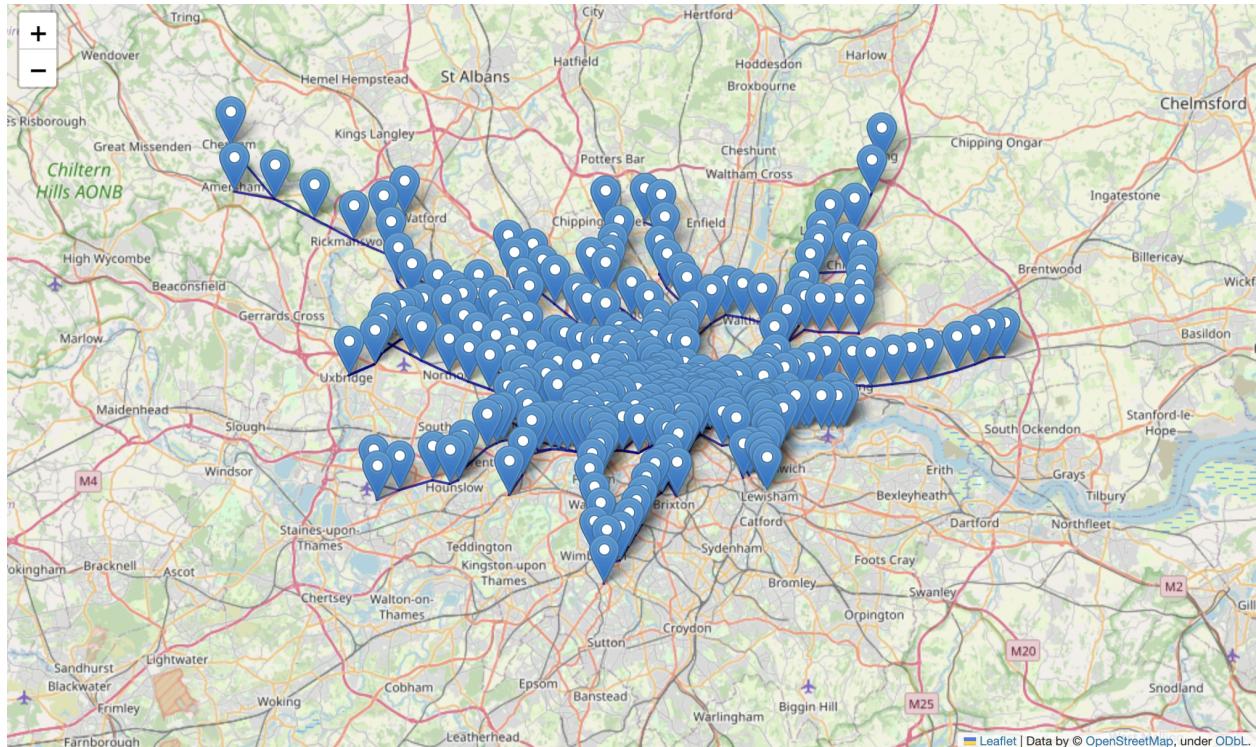


Fig. 26 - Visualização do grafo através da biblioteca Folium

A Figura 27 ilustra uma simulação final, com recurso à biblioteca Folium, entre dois pontos aleatórios, numa determinada hora aleatória, com o percurso mais rápido, assinalado a preto no mapa, entre a estação de partida a verde e a estação de chegada a vermelho, após a aplicação do algoritmo de Dijkstra, implementado em Python.

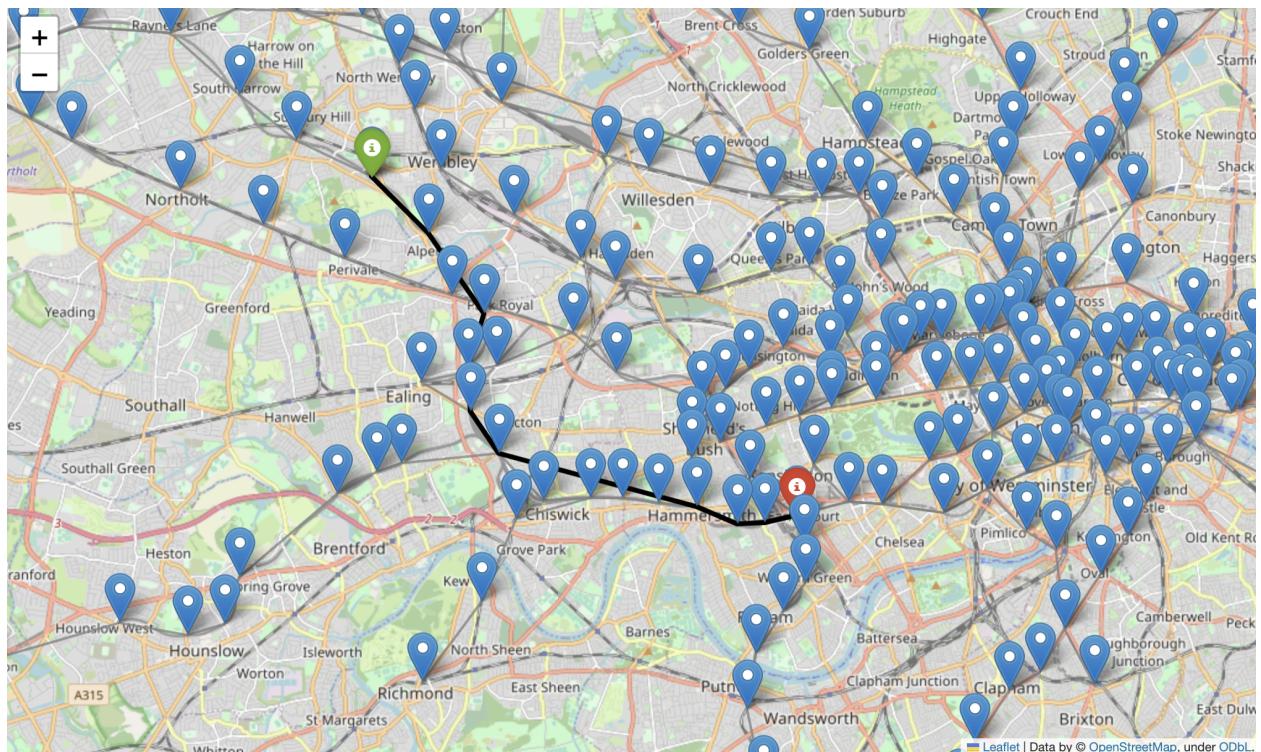


Fig. 27 - Simulação final do caminho mais rápido/curto entre dois pontos aleatórios numa hora aleatória

C. CONCLUSÃO

No presente relatório foi efetuada uma análise aprofundada acerca da rede do metro de Londres por meio da construção de um grafo, permitindo explorar diversos cenários, simulações e análises. A obtenção e pré-processamento dos dados foram realizados com base nos datasets disponibilizados que continham informações sobre as estações e as suas conexões. A classe LondonNetworkGraph foi projetada em prol de representar o grafo e fornecer os métodos necessários para a visualização e cálculo de rotas. No que diz respeito ao algoritmo de Dijkstra, este foi implementado para calcular o caminho mais curto/rápido entre duas estações, de forma a melhorar o planeamento de viagens. Os resultados revelaram que a rede do metro da capital inglesa possui 302 estações e 743 ligações entre elas. Perante a análise, foram observadas a distribuição das estações por zonas e as ligações entre elas, percebendo-se que a zona 2 é a melhor servida no que toca ao número de estações e a linha 4 é a que apresenta o maior número de ligações. O grau médio das estações mostrou a conectividade da rede, onde a distância média entre as estações e os tempos médios de percurso foram calculados para fornecer uma visão global deste sistema de transporte. A visualização do grafo revelou a estrutura da rede, destacando tanto as estações centrais quanto as estações periféricas, revelando também algumas estações que se encontram sem ligações atribuídas. Estes resultados proporcionam uma boa compreensão da rede de transporte e das suas capacidades, auxiliando na identificação das principais ligações da cidade e contribuindo para uma visão mais completa do metro da capital inglesa.