INSTITUTO UNIVERSITÁRIO DE LISBOA

**1º ano da Licenciatura de Ciência de Dados 22/23**

**UC de Estrutura de Dados e Algoritmos**

Docente: Maria Cabral Diogo Pinto Albuquerque

Lisboa, 31 de maio de 2023

# Rede do Metro de Londres: visualização e estudo da rede usando um grafo

Trabalho 2

Realizado por:

Bernardo Dinis, 111401
Matilde Costa, 110995

# Indíce



# Introdução

O presente trabalho tem como objetivo responder ao Trabalho 2 da Unidade Curricular de Estrutura de Dados e Algoritmos. Pretende-se visualizar, representar e analisar informações sobre a rede do Metro de Londres utilizando um grafo. Os grafos são estruturas fundamentais em ciência da computação e têm uma grande importância em várias áreas. Neste âmbito servirão para a análise de uma rede complexa, exploração da estrutura da rede e otimização das rotas.

A otimização do sistema de transporte público do metro é um desafio complexo e crucial para a cidade de Londres. Neste projeto, o nosso objetivo concreto é identificar maneiras eficientes de aprimorar o seu funcionamento, tendo em consideração fatores como tempo de viagem. Procura-se oferecer *insights* e recomendações que contribuam para uma experiência de viagem mais eficiente. Através de uma análise detalhada da rede existente e da aplicação de técnicas e estudo do grafo, pretende-se melhorar a qualidade de vida dos usuários, tornando o sistema de transporte público do metrô de Londres mais confiável, acessível e eficiente.

## Código do estudo da rede do Metro de Londres

## Aquisição de Dados

Em relação à obtenção dos dados, os mesmos foram adquiridos no formato CSV por meio dos arquivos de conexões (*connections*) e estações (*stations*), fornecidos pela Docente. Utilizando ferramentas de análise de dados, como o Excel, foi possível verificar que não existiam linhas duplicadas nos conjuntos de dados fornecidos. Apesar disso, é importante ressaltar que a análise realizada foi baseada nos dados disponíveis e não foram adquiridos conjuntos de dados adicionais para aprofundar o estudo.

A decisão de não adquirir mais *datasets* para o estudo foi tomada levando em consideração as restrições e objetivos estabelecidos para este trabalho. Embora a obtenção de mais dados possa fornecer informações adicionais e possibilitar análises mais abrangentes, optou-se por explorar os conjuntos de dados disponíveis de forma a cumprir com os objetivos principais.

## Parte I

A classe LondonNetworkGraph é responsável por importar as estações e as ligações para o grafo, além de fornecer diversas funcionalidades para análise da rede do metro. Entre as funcionalidades implementadas estão a contagem do número total de estações, o número de estações por zona, o número total de conexões, o número de conexões por linha, o grau médio das estações e o peso médio das conexões. Além disso, a classe permite a visualização do grafo com auxílio das bibliotecas indicadas, NetworkX, Matplotlib e Folium.

## Implementação da classe LondonNetworkGraph

Inicialmente, foram importadas as bibliotecas necessárias para a execução do projeto.

```
''''Importação das bibliotecas''''
import networkx as nx
import matplotlib.pyplot as plt
import folium
import random   #Geração de números aleatórios
import datetime #Trabalhar com valores relacionados com tempo
import math     #Calcular a distância euclidiana
```

De seguida, encontra-se o código que foi utilizado para a criação da classe, possibilitando criar e manipular a rede do metro de londres na forma de um grafo.

**Implementação dos métodos da classe**

As funções stations e connections são responsáveis por construir a rede de estações do Metro de Londres. A função stations lê um arquivo com informações sobre as estações, como identificação, localização geográfica e nome. Esses dados são adicionados como “nodes” no grafo, representando cada estação. A função connections, por sua vez, lê um arquivo com informações sobre as conexões entre as estações, como linha, distância e horários de pico. Essas

```
class LondonNetworkGraph:
    def __init__(self):
        self.graph = nx.DiGraph()

''''O método stations, lê as estações do ficheiro''''
    def stations(self, file_path):
        with open(file_path, 'r') as file:
            next(file)                          #Salta a 1 linha com os atributos,do ficheiro
            next(file)                          #salta a 2 linha do ficheiro vazia, do ficheiro
            for line in file:
                data = line.strip().split(',')
                if len(data) == 8:
                    station_id = int(data[0])  #Extrai o id da estação
                    latitude = float(data[1])  #Extrai a latitude da estação
                    longitude = float(data[2]) #Extrai a longitude da estação
                    name = data[3]             #Extrai o nome da estação
                    #display_name = data[4]    #Não é necessário e no csv está desformatado
                    zone = data[5]             #Extrai a zona da estação
                    total_lines = int(data[6]) #Extrai o total de linhas da estação
                    rail = int(data[7])        #Extrai o trilho da estação

                    #Adicionar estação (node) ao grafo
                    self.graph.add_node(station_id, latitude=latitude, longitude=longitude, name=name,
                                        zone=zone, total_lines=total_lines, rail=rail)

''''O método connections lê as conexões das estações do ficheiro''''
    def connections(self, file_path):
        with open(file_path, 'r') as file:
            next(file)                          #Salta a 1 linha com os atributos,do ficheiro
            for line in file:
                data = line.strip().split(',')
                if len(data) == 7:
                    line = int(data[0])              #Extrai a linha da estação
                    from_station_id = int(data[1])   #Extrai o id da estação de origem
                    to_station_id = int(data[2])     #Extrai o id da estação final
                    distance = float(data[3])        #Extrai a distância
                    off_peak = float(data[4])        #Extrai o tempo de viagem em "off peak"
                    am_peak = float(data[5])         #Extrai o tempo de viagem em "am peak"
                    inter_peak = float(data[6])      #Extrai o tempo de viagem em "inter peak"
```

informações são adicionadas como “edges” no grafo, estabelecendo as conexões entre as estações.

**Nota:** Na função stations, o campo "display_name" não é utilizado devido à presença frequente de valores nulos ("NULL") e à presença da tag HTML "<br/ >". Essas ocorrências podem estar a indicar a falta de um nome válido para as estações e a formatação dos dados para exibição em HTML. Neste trabalho, o foco está na construção e análise da rede de estações do Metro de Londres com base nos outros atributos disponíveis. Portanto, a exclusão do campo "display_name" não afetará negativamente as análises e visualizações pretendidas.

De seguida, foram adicionados métodos adicionais à classe, com o objetivo de obter informações estatísticas sobre o grafo da rede de transporte de Londres.

```
 ''''O método n_stations devolve o número total de estações (nodes)''''
    def n_stations(self):
        return self.graph.number_of_nodes()

''''O método n_stations_zones devolve o número de estações de cada zona''''
     def n_stations_zone(self):
    zone_count = {}  #Dicionário para contar o número de estações em cada zona

            for node in self.graph.nodes:  #Itera sobre todos as estações (nodes) do grafo
                if 'zone' in self.graph.nodes[node]:   #Verifica se a estação possui a chave 'zone'
                zone = self.graph.nodes[node]['zone']  #Obtém o valor da zona da estação
                #Incrementa o contador da zona ou adiciona a zona com valor 1 caso ainda não exista
                zone_count[zone] = zone_count.get(zone, 0) + 1
            return zone_count  #Devolve o dicionário com o número de estações em cada zona

''''O método n_edges devolve o número de todas as conexões (edges)''''
    def n_edges(self):
        return self.graph.number_of_edges()

''''O método n_edges_line devolve o número de conexões (edges) por linhas''''
    def n_edges_line(self):
            line_count = {}
            for connection in self.graph.edges.values():
                line = connection['line']                          # Obtém o número da linha da conexão
                line_count[line] = line_count.get(line, 0) + 1     # Incrementa o contador de conexões para a linha atual
            return line_count

''''O método mean_degree devolve o grau médio das estações (nodes)''''
    def mean_degree(self):
        degrees = [degree for _, degree in self.graph.degree()]  # Obtém os graus (número de edges) de cada node no grafo
        return sum(degrees) / len(degrees)  # Devolve a média dos graus dos nodes

''''O método mean_weight devolve o peso médio das conexões (edges)''''
    def mean_weight(self, weight):
        weights = [connection[weight] for connection in self.graph.edges.values()]
        return sum(weights) / len(weights) #Devolve a média do peso (distância) das conexões
```

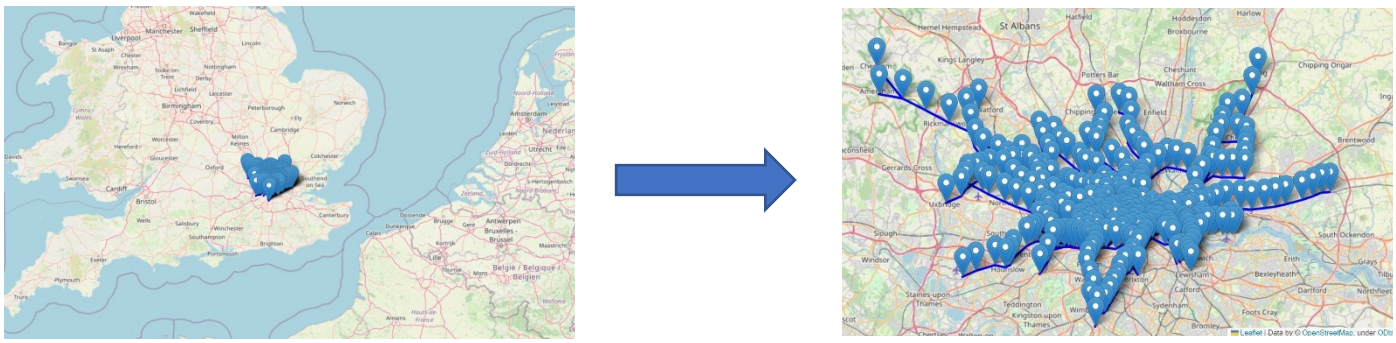
## Implementação do visualize

O método *visualize* cria um mapa com as estações da rede de transporte de Londres. Inicialmente, é criado um mapa vazio utilizando a biblioteca *Folium*. Esta biblioteca é uma ferramenta em Python para a visualização de dados geoespaciais em mapas interativos. Oferece recursos para adicionar marcadores, linhas e polígonos aos mapas, permitindo criar visualizações claras e interativas. É amplamente utilizada para visualizar dados geográficos de forma intuitiva.

O método, itera sobre cada estação (*node)* no grafo e adiciona marcadores para cada estação. Para cada estação, são obtidos os valores da latitude, longitude e o seu nome. Em seguida, itera sobre cada conexão (*edge*) no grafo e adiciona linhas para representar as conexões entre as estações. Para cada conexão, são obtidos os dados das estações de origem e destino. Caso contrário, uma linha é desenhada no mapa, conectando as coordenadas de latitude e longitude das estações de origem e destino.

```
'''''O método visualize cria um mapa com as estações''''
def visualize(self):
        #Cria um mapa vazio
        map = folium.Map(location=[51.5074, -0.1278], zoom_start=11)
        #Adiciona marcas às estações
        for node, data in self.graph.nodes(data=True):
            latitude = data.get('latitude')
            longitude = data.get('longitude')
            name = data.get('name')
            #Salta estações sem valores na latitude e na longitude
            if latitude is None or longitude is None:
                continue
            folium.Marker([latitude, longitude], popup=name).add_to(map)
        #Adiciona conexões em linhas
        for from_station, to_station, data in self.graph.edges(data=True):
            from_data = self.graph.nodes[from_station]
            to_data = self.graph.nodes[to_station]
            #Salta estações sem valores na latitude e na longitude guardados
            if 'latitude' not in from_data or 'longitude' not in from_data or 'latitude' not in to_data or 'longitude' not in
            to_data:
                continue
            from_latitude = from_data['latitude']
            from_longitude = from_data['longitude']
            to_latitude = to_data['latitude']
            to_longitude = to_data['longitude']
            folium.PolyLine([(from_latitude, from_longitude), (to_latitude, to_longitude)], color='blue').add_to(map)
        #Demonstração do mapa
        display(map)
```

Por fim, o mapa é exibido usando a função display para demonstração:



## Parte II

Na segunda fase do trabalho, desenvolveu-se uma simulação em Python que calcula e visualiza o caminho mais curto entre duas estações utilizando o algoritmo de Dijkstra. O Google Maps utiliza, por exemplo, algoritmos este para determinar as melhores rotas, considerando fatores como distância, tempo de viagem e preferências do usuário, proporcionando uma navegação eficiente e precisa. Nesta parte do trabalho, a simulação consiste no mesmo, em gerar aleatoriamente dois pontos de partida e destino dentro de uma determinada área, assim como uma hora do dia. Em seguida, é possível obter a estação mais próxima para cada um dos pontos, determinando o caminho mais rápido entre essas duas estações com base na hora do dia.

Ambas as funções são úteis para criar informações aleatórias que podem ser utilizadas em simulações ou análises no contexto de um sistema de transporte, como a geração de trajetos aleatórios ou a análise de fluxos de passageiros em diferentes períodos do dia.

A função *randomize_locations* é responsável por gerar aleatoriamente dois pontos, representados pelas coordenadas de latitude e longitude, dentro de uma determinada área delimitada pelos valores de x1, x2, y1 e y2. Esses pontos são gerados para representar o início e o fim de uma rota ou trajeto. A função *randomize_time* tem a finalidade de gerar aleatoriamente uma hora do dia. A hora é representada por um valor inteiro entre 0 e 23. Com base nesse valor, a função determina em qual período do dia é que essa hora se encaixa, sendo 1 para o *am peak* (7h às 10h), 2 para o *inter peak* (10h às 16h) e 3 para o *off peak*.

```
''''Gera aleatoriamente coordenadas de localização para um ponto inicial e final dentro'''''
   def randomize_locations(self, x1, x2, y1, y2):
       start_latitude = random.uniform(x1, x2)
       start_longitude = random.uniform(y1, y2)
       end_latitude = random.uniform(x1, x2)
       end_longitude = random.uniform(y1, y2)
       start_point = (start_latitude, start_longitude) #tuplo das coordenadas
       end_point = (end_latitude, end_longitude)
       while start_point == end_point:
           self.randomize_locations(x1, x2, y1, y2)
       print('Start point is: ', start_point, 'End point is: ', end_point) #check values
       return start_point, end_point  #Devolve um tuplo com as coordenadas (latitude, longitude) do ponto inicial e final

'''Gera aleatoriamente uma hora do dia e determina o período de pico correspondente''''
   def randomize_time(self):
       hour = random.randint(0, 23)
       if 7 <= hour < 10:    #am peak
           start_time = 1
       elif 10 <= hour < 16: #inter peak
           start_time = 2
       else:
           start_time = 3    #off peak
       print('Peak is:', start_time, '. Hour is:', hour)
       return start_time     #Devolve um tuplo com o período de pico (1- AM Peak, 2- Inter Peak, 3- Off Peak) e a hora do dia
```

A função *find_nearest_station* tem como objetivo encontrar a estação mais próxima com base num ponto de referência. É percorrida todas as estações presentes do grafo e é obtido as coordenadas de latitude e longitude de cada estação. Caso as coordenadas estejam disponíveis, é calculada a distância euclidiana entre o ponto de referência e a estação utilizando a função *calculate_distance*. A menor distância encontrada até o momento é atualizada a cada iteração,

juntamente com o *id* da estação mais próxima. No final do processo, a função retorna o *id* da estação mais próxima encontrada.

A função *calculate_distance* é responsável por calcular a distância euclidiana entre dois pontos. Os pontos são representados pelas coordenadas x1, y1 e x2, y2.

```
'''''Encontra a estação mais próxima com base em um ponto de referência''''
def find_nearest_station(self, point):
    #Inicializa a menor distância como infinito
    min_distance = float('inf')
    #Inicializa a estação mais próxima como None
    nearest_station = None
    for station_id, station_data in self.graph.nodes(data=True):
        # Obtém as coordenadas da estação
        latitude = station_data.get('latitude')
        longitude = station_data.get('longitude')
        #Salta para a próxima estação se as coordenadas não estiverem presentes
        if latitude is None or longitude is None:
            continue
        # Cria um objeto de ponto para a estação
        station_point = (latitude, longitude)
        # Calcula a distância entre o ponto de referência e a estação
        distance = self.calculate_distance(point, station_point)
        # Atualiza a menor distância e a estação mais próxima, se necessário
        if distance < min_distance:
            min_distance = distance
            nearest_station = station_id
    print('Nearest station is:', nearest_station)
    return nearest_station  #Devolve o ID da estação mais próxima

''''' Calcula a distância euclidiana entre dois pontos''''
def calculate_distance(self, point1, point2):
    # Extrai as coordenadas dos pontos
    x1, y1 = point1
    x2, y2 = point2
    # Calcula a distância euclidiana entre os pontos
    distance = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
    # Retorna a distância calculada
    return distance  #Devolve a distância euclidiana entre os dois pontos
```

A função *shortest_path* tem como objetivo encontrar o caminho mais curto entre duas localizações aleatórias. Primeiro, um tempo de partida aleatório é definido utilizando a função *randomize_time*. De seguida, são geradas localizações aleatórias para o ponto de partida e o ponto de destino utilizando a função *randomize_locations*. Com o uso da função *find_nearest_station* são encontradas as estações mais próximas do ponto de partida e do ponto de destino.

Para garantir que as estações de partida e destino são diferentes, utiliza-se um loop. Caso as estações sejam iguais, novas localizações aleatórias são geradas e as estações mais próximas são encontradas novamente.

Por fim, é calculado o caminho mais curto utilizando o algoritmo de Dijkstra implementado na biblioteca NetworkX. Este algoritmo para encontra o caminho mais curto num grafo com *edges* positivas. Este recebe como entrada um *node* de origem e calcula o caminho mais curto até todos os outros *nodes* do grafo. O cálculo do caminho mais curto leva em consideração o tempo de partida e o peso das *edges* do grafo, que representam os períodos de pico. Dependendo do tempo de partida, é selecionado o peso adequado para o cálculo do caminho mais curto.

```
''''Encontra o caminho mais curto entre duas localizações aleatórias''''
def shortest_path(self, x1, x2, y1, y2):
    #Define o tempo de partida aleatório
    start_time = self.randomize_time()
    #Gera localizações aleatórias para o ponto de partida e o ponto de destino
    start_point, end_point = self.randomize_locations(x1, x2, y1, y2)
    #Encontra a estação mais próxima do ponto de partida e do de destino
    start_station = self.find_nearest_station(start_point)
    end_station = self.find_nearest_station(end_point)
    #Garante que as estações de partida e destino são diferentes
    while start_station == end_station:
        start_point, end_point = self.randomize_locations(x1, x2, y1, y2)
        start_station = self.find_nearest_station(start_point)
        end_station = self.find_nearest_station(end_point)
    #Calcula o caminho mais curto com base no tempo de partida e peso (pico de tempo)
    if start_time == 1:
        shortest_path = nx.dijkstra_path(self.graph, start_station, end_station, weight='am_peak')
    elif start_time == 2:
        shortest_path = nx.dijkstra_path(self.graph, start_station, end_station, weight='inter_peak')
    else:
        shortest_path = nx.dijkstra_path(self.graph, start_station, end_station, weight='off_peak')
    return shortest_path
        return shortest_path  #Devolve o caminho mais curto encontrado
```

## Conclusão

Neste trabalho, foi desenvolvido um código em Python 3 para analisar o sistema do Metro de Londres por meio da utilização de grafos e programação. O objetivo principal foi estudar como os grafos possibilitam o estudo de grandes redes e aprofundar o nosso conhecimento e prática face a este conteúdo. No entanto, é importante destacar que o estudo realizado é limitado devido a algumas restrições. Primeiramente, não foi possível obter acesso ao horário de funcionamento das linhas, o que poderia fornecer uma análise mais precisa do sistema. Além disso, não há informações sobre o ano em que os dados foram adquiridos, o que pode impactar na atualidade e relevância das informações obtidas.

Apesar dessas limitações, o trabalho permitiu explorar conceitos fundamentais de análise de dados e visualização de informações geográficas. O código desenvolvido utilizando a biblioteca NetworkX possibilitou a construção do grafo representando as estações e as suas conexões, permitindo calcular métricas como o número de estações, o número de conexões por linha, a média de grau das estações e o peso médio das conexões. Essas métricas fornecem *insights* interessantes sobre a estrutura e a conectividade do sistema do metro.

É importante realçar que existem estudos mais avançados e complexos, como o sistema do *Google Maps*, que incorporam diversas outras variáveis e considerações, como informações em tempo real, planejamento de rotas e integração com outros meios de transporte. No entanto, este trabalho buscou explorar de forma básica e introdutória o potencial de análise de dados em relação ao sistema do Metro de Londres.