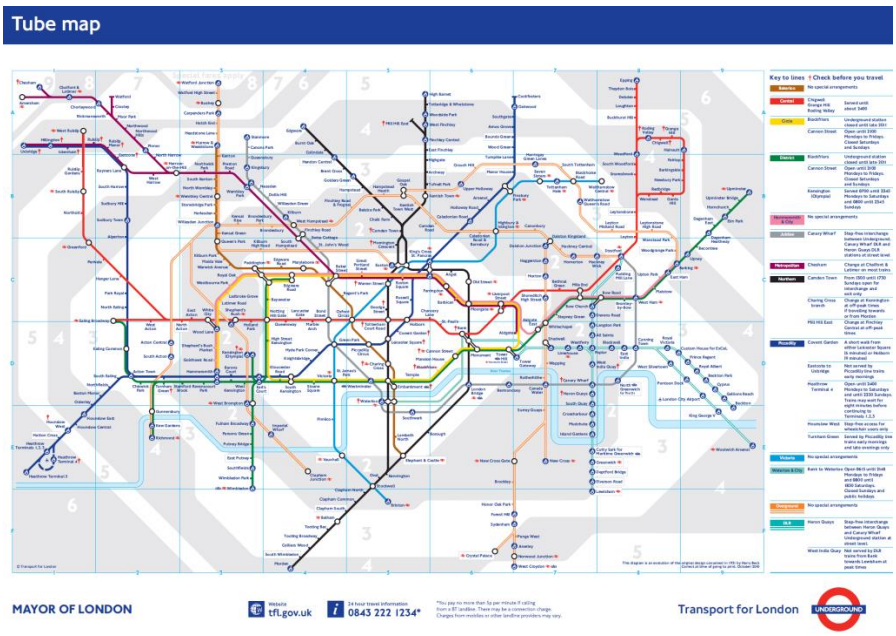
INSTITUTO UNIVERSITÁRIO DE LISBOA

Unidade Curricular de Estruturas de Dados e Algoritmos

1º ano da Licenciatura de Ciência de Dados | 2º Semestre

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



Afonso Lourenço | Nº111487 | CDA2

Afonso Santos | Nº111431 | CDA2

Docente: Maria Pinto Albuquerque

maio, 2023

Introdução:

A rede do Metro de Londres é uma das mais complexas e extensas do mundo, conectando diferentes partes da cidade e facilitando o deslocamento diário de milhões de pessoas. Neste relatório, propomos representar, visualizar e analisar informações sobre a rede do Metro de Londres utilizando um grafo. Para isso, vão ser utilizados datasets fornecidos em Stations.csv e Connections.csv. A implementação do grafo vai ser feita em Python e vai possuir 3 funcionalidades de visualização do grafo, utilizando as bibliotecas NetworkX, Matploblib e Folium. Esta implementação do grafo que representa a rede do Metro de Londres será utilizada para fazer uma simulação em Python que permita estudar caminhos mais curtos entre duas estações através do cálculo destes caminhos e sua visualização.

Desenvolvimento:

Aquisição de Dados

A primeira etapa deste trabalho envolveu a aquisição de dados sobre a rede do Metro de Londres. Felizmente, os dados necessários foram disponibilizados em arquivos CSV, que continham informações essenciais para a representação do grafo da rede do Metro, como as estações, linhas e conexões. Devido à estrutura e qualidade confiáveis dos dados fornecidos, não houve necessidade de recolher informações adicionais em outros conjuntos de dados.

Limpeza dos Dados

Após a aquisição dos dados, é necessário realizar a etapa de limpeza, manipulação e transformação dos mesmos. Neste trabalho, a limpeza dos dados envolveu a verificação da consistência dos registos, a validação dos formatos das informações fornecidas e a seleção das colunas relevantes para a análise. No entanto, devido à qualidade dos dados originais, não foi necessário realizar alterações no conjunto de dados.

Embora não tenhamos encontrado problemas relevantes nos dados fornecidos, notámos que a estação número 189 não existe e que existem linhas paralelas, logo o número de conexões não é igual ao número de linhas. Isto deve-se ao facto de um grafo apenas apresentar 1 aresta para um ponto de chegada e partida simultaneamente.

Análise dos dados e recolha de resultados

Nesta fase, realizamos uma análise dos dados obtidos sobre a rede do Metro de Londres, começando por implementar a classe LondonNetworkGraph. Pressupôs a utilização da biblioteca NetworkX da qual se utilizou o MultiDiGraph, dado que a rede do Metro de Londres deve ser representada por um multigrafo direcionado (existem linhas paralelas e não é arbitrário existirem ambos os sentidos de certa linha). A criação desta classe serviu para calcular diversas métricas relacionadas à rede do Metro de Londres. Entre as métricas calculadas, destacamos as seguintes:

Número de estações: n_stations() – devolve o número de total de estações

Número de estações por zona: n_stations_zone() – devolve o número de estações de cada zona

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

def n_stations_zone(self):
    zones = {}
    for node, data in self.graph.nodes(data=True):
        zone = float(data['zone'])
        int_zone = int(zone)  #Arredondar para o número inteiro abaixo
        upper_zone = int_zone + 1 #Considerar o número acima

        if int_zone in zones:
            zones[int_zone] += 1
        else:
            zones[int_zone] = 1

        if zone > int_zone:  #Verificar se a zona é um valor decimal
            if upper_zone in zones:
                zones[upper_zone] += 1
            else:
                zones[upper_zone] = 1

    return zones
```

Número de arestas: n_edges() – devolve o número total de arestas

Número de arestas por linha: n_edges_line() – devolve o número de arestas por linha

```
def n_edges(self):
    return self.graph.number_of_edges() #Equivale ao numero de troços sem contar os repetidos

def n_edges_line(self):
    lines = {}
    for connection in self.connectionslist:
        line = connection[0]
        if line in lines:
            lines[line] += 1
        else:
            lines[line] = 1
    return lines
```

Grau médio: mean_degree() – devolve o grau médio das estações

Peso médio: mean_weigth(weight) - devolve o peso médio das conexões (dado o tipo de peso)

```
def mean_degree(self):
    degrees = [degree for _, degree in self.graph.degree()]
    return sum(degrees) / len(degrees)

def mean_weight(self, weight):
    weights = [data[weight] for _, _, data in self.graph.edges(data=True) if weight in data]
    return sum(weights) / len(weights)
```

Para criar visualizações do Metro de Londres utilizamos o NetworkX, o Folium e o Matplotlib. Por meio de um código, gerámos figuras/gráficos que representam as localizações e as conexões das estações na rede do Metro.

NetworkX:

```
def visualizeNet(self):
    # Visualização usando NetworkX
    nx.draw(self.graph, with_labels=True, node_color='lightblue', edge_color='gray', node_size=500)
    plt.show()
```

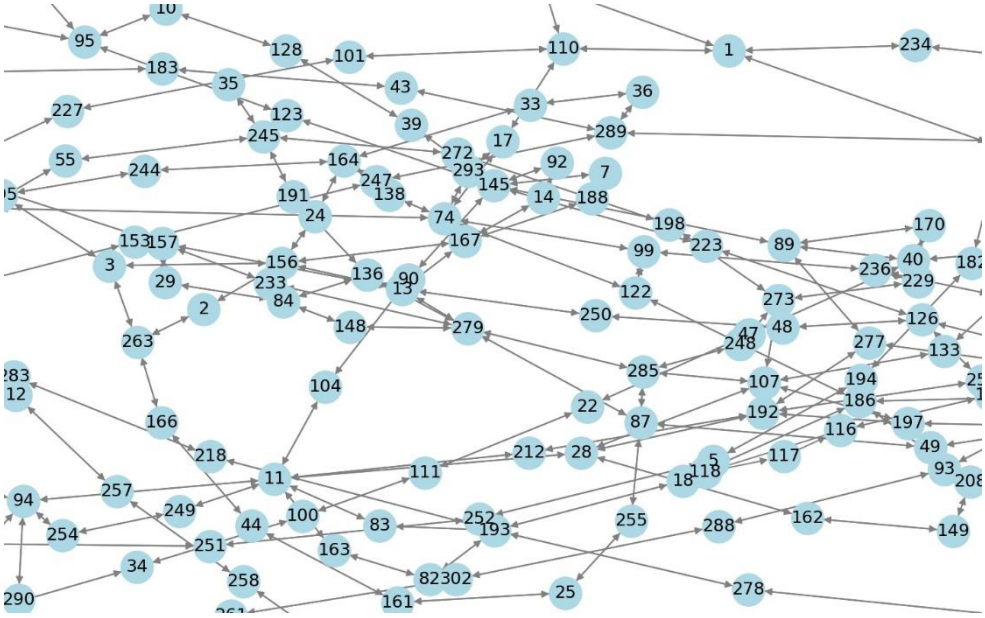


Figura 1: Estrato do grafo usando NetworkX

Folium:

```
def visualizeFol(self):
    # Visualização usando Folium
    m = folium.Map(location=[0, 0], zoom_start=2)
    for node, data in self.graph.nodes(data=True):
        folium.Marker(location=[data['latitude'], data['longitude']], popup=node).add_to(m)
    for source, target, data in self.graph.edges(data=True):
        folium.PolyLine(locations=[(self.graph.nodes[source]['latitude'], self.graph.nodes[source]['longitude']),
                                   (self.graph.nodes[target]['latitude'], self.graph.nodes[target]['longitude'])],
                        color='red', weight=2, popup=data['line']).add_to(m)

    m.save('graph.html')
```

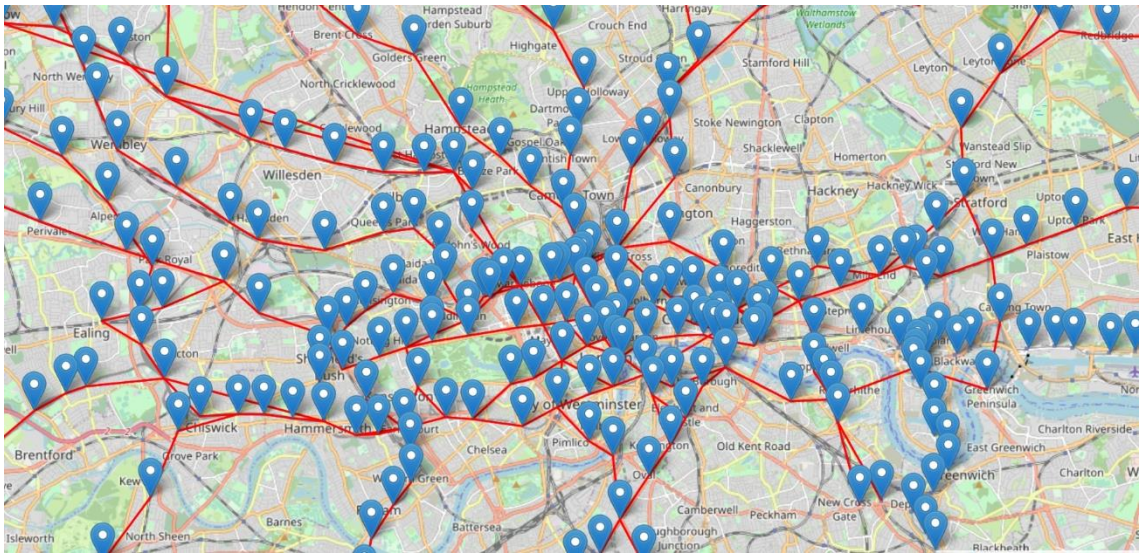


Figura 2: Estrato do grafo usando Folium

Matplotlib:

```
def visualizeMat(self):
    # Visualização usando Matplotlib
    station_coords = {}
    for row in self.stationslist:
        latitude = float(row[1])
        longitude = float(row[2])
        station_name = row[3]
        station_coords[station_name] = (longitude, latitude)

    fig, ax = plt.subplots()


    for station, (longitude, latitude) in station_coords.items():
        ax.plot(longitude, latitude, 'ko')
        ax.text(longitude + 0.001, latitude + 0.001, station)


    for row in self.connectionslist:
        from_station_id = int(row[0])
        to_station_id = int(row[1])
        if from_station_id <= 0 or from_station_id > len(station_coords.keys()) or to_station_id <= 0 or
        to_station_id > len(station_coords.keys()):
            continue
        from_station = list(station_coords.keys())[from_station_id - 1]
        to_station = list(station_coords.keys())[to_station_id - 1]
        x_values = [station_coords[from_station][0], station_coords[to_station][0]]
        y_values = [station_coords[from_station][1], station_coords[to_station][1]]
        ax.plot(x_values, y_values, 'k-')

    ax.set_aspect('auto')

    plt.title('London Underground')
    plt.show()
```

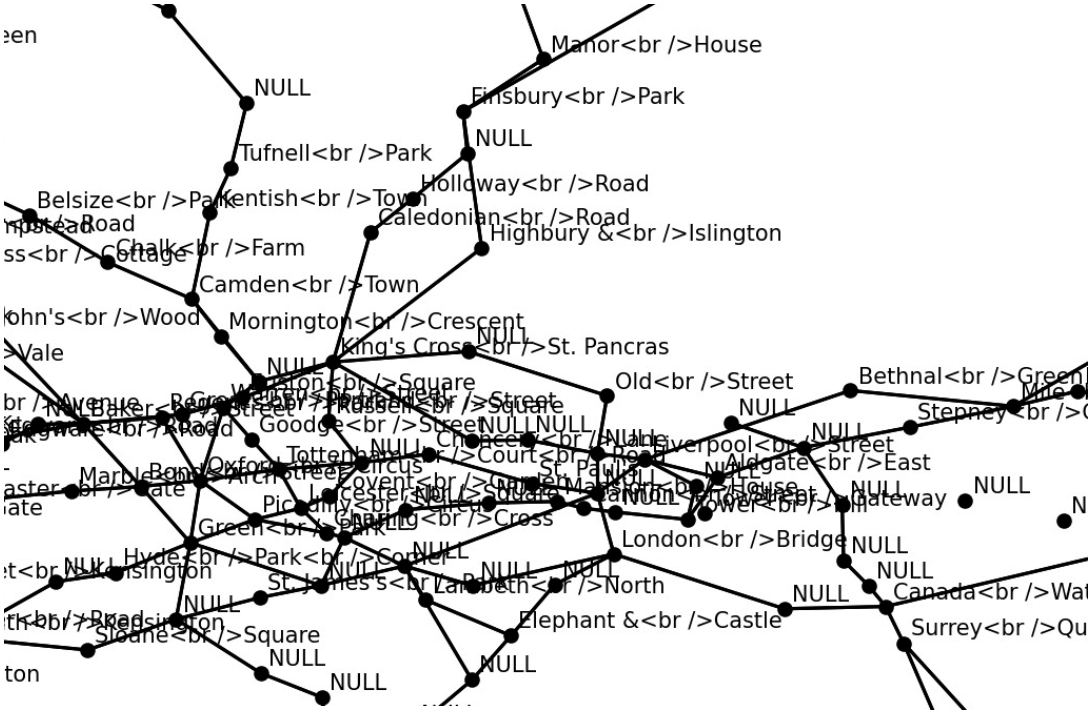


Figura 3: Estrato do grafo usando Matplotlib

Além das métricas quantitativas, realizamos uma análise dos dados para identificar possíveis relações qualitativas na rede do Metro de Londres. Depois de implementar estes métodos de visualização conseguimos ter uma melhor perceção da distribuição geográfica das estações, a presença de estações com inúmeras conexões e a proximidade com pontos

de interesse na cidade, como o Museu Britânico, a Torre de Londres e o Palácio de Buckingham. Resumindo, a visualização gráfica permitiu-nos ter uma compreensão mais intuitiva da disposição das estações e das linhas na cidade.

Depois de criada a classe LondonNetworkGraph com visualização usando as bibliotecas já referidas, foi desenvolvida uma simulação para calcular e visualizar o caminho mais rápido entre duas estações, usando o algoritmo de Dijkstra.

Para gerar aleatoriamente dois pontos (start, end) entre x1 e x2, y1 e y2 e gerar aleatoriamente uma hora do dia foram usadas as seguintes funções:

```
def random_points(x1=51.4022, x2=51.7052, y1=-0.6110, y2=0.2510): # Define a função random_points que gera dois p
    start = (random.uniform(x1, x2), random.uniform(y1, y2)) # Gera um ponto aleatório, dentro dos limites, para
    end = (random.uniform(x1, x2), random.uniform(y1, y2)) # Gera um ponto aleatório dentro, dos limite,s para a
    return start, end # Retorna um tuplo contendo os pontos de partida e chegada gerados.

def random_time(): # Define a função random_time que gera uma hora aleatória.
    hour = random.randint(0, 23) # Gera um número inteiro aleatório entre 0 e 23 para representar a hora.
    minute = random.randint(0, 59) # Gera um número inteiro aleatório entre 0 e 59 para representar os minutos.
    second = random.randint(0, 59) # Gera um número inteiro aleatório entre 0 e 59 para representar os segundos.
    return hour, minute, second # Retorna um tuplo contendo a hora, os minutos e os segundos gerados.
```

De seguida, o pontos que obtivemos são usados noutra função que vai receber esses pontos e calcular qual a estação mais próxima de cada ponto através da distância euclidiana, outra função que foi implementada.

```
def calculate_nearest_station(self, point): #Dado um ponto(x,y), procura a estação mais proxima
    nearest_station = None #Iniciar a estação em None
    min_distance = float('inf') #iniciar a distancia minima em infinito
    for node, data in self.graph.nodes(data=True): #Do grafo em NetworkX, vai guardar os valores 'node' e data'
        station_point = (data['latitude'], data['longitude']) #Criará um ponto com as respectivas coordenadas da estação em
        distance = self.calculate_distance(point, station_point) #Invoca a função calculate_distance() para calcular a dist
        if distance < min_distance: #Verifica se esse valor (distancia entre o ponto dado e a estação) é o minimo
            nearest_station = node #Caso se verifique, guarda qual a estação
            min_distance = distance #Caso se verifique, guarda qual a distância
    return nearest_station

def calculate_distance(self, point1, point2):#Dados 2 pontos distintos (x,y), calcula a respectiva distancia euclidiana ent
    x1, y1 = point1
    x2, y2 = point2
    distance = ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
    return distance
```

Depois de saber os locais, a hora do dia e as estações é preciso calcular qual é o caminho mais rápido entre essas duas estações. Para isto, foram usados dois métodos:

O método do NetworkX que implementa o algoritmo de Dijkstra

```
def shortest_path(self, start_station, end_station,time=None):# Aplicar o algoritmo de Dijkstra para encontrar o caminho mai
    if time==None:
        shortest_path = nx.dijkstra_path(self.graph, start_station, end_station, weight='lenght')
        return shortest_path
    elif 7<int(time[0])<=10:
        shortest_path = nx.dijkstra_path(self.graph, start_station, end_station, weight='ampeak')
        return shortest_path
    elif 10<int(time[0])<=16:
        shortest_path = nx.dijkstra_path(self.graph, start_station, end_station, weight='interpeak')
        return shortest_path
    else:
        shortest_path = nx.dijkstra_path(self.graph, start_station, end_station, weight='weight')
        return shortest_path
```

Implementação do algoritmo de Dijkstra

```
def dijkstra(self, start_node): #Aplicação do algoritmo Dijkstra, que recebe o nó inicial
    self.weight = {} #Cria um dicionário vazio chamado weight para armazenar os pesos dos nós.
    self.visited = set() #Cria um conjunto vazio  para armazenar os nós visitados.
    self.previous = {} #Cria um dicionário vazio  para armazenar os nós anteriores no caminho mais curto.
    self.weight = {node: float('inf') for node in self.graphedges} # Inicializa todos os pesos dos nós como infinito.
    self.weight[start_node] = 0 # Define o peso do nó inicial como 0.
    self.previous = {node: None for node in self.graphedges} #Define todos os nós anteriores como None.

    while self.visited != set(self.graphedges): #Enquanto houver nós não visitados no grafo:
        min_weight = float('inf') # Define o menor peso como infinito.
        min_node = None # Define o nó com o menor peso como None.
        for node in self.graphedges:
            if node not in self.visited and self.weight[node] < min_weight: # Se o nó não foi visitado e tem um peso menor d
                min_weight = self.weight[node] # Atualiza o menor peso com o peso do nó.
                min_node = node # Atualiza o nó com o menor peso.

        self.visited.add(min_node)# Guardar os nós  visitados com menor peso



        for neighbor, weights in self.graphedges[min_node].items():
            for weight in weights: #Como existem linhas paralelas, corre utilizando peso a peso (quando existem)
                new_weight = self.weight[min_node] + weight# Calcula o novo peso somando o peso do nó atual com o peso do vi
                if new_weight < self.weight[neighbor]: # Se o novo peso for menor que o peso anterior do vizinho:
                    self.weight[neighbor] = new_weight # Atualiza o peso do vizinho com o novo peso.
                    self.previous[neighbor] = min_node # Define o nó atual como o nó anterior do vizinho.
```

```
def shortest_path_dijkstra(self, start_station, end_station,time=None):#Criar qual o caminho mais rapido ou perto, dadas as
    self.graphedges={} #Cria um dicionario vazio
    if time is None:#Se não for dado um tempo, é calculado o percurso que contem a distancia minima
        #Criar-se-á um dicionário apenas com os dados relevantes para o caso
        for line in self.connectionslist:
            if line[1] not in self.graphedges:
                self.graphedges[line[1]] = {}  #Neste caso, contendo apenas o nó de partida, de chegada e o valor respectiv
            if line[2] not in self.graphedges[line[1]]:
                self.graphedges[line[1]][line[2]] = []  #Neste caso, existindo (ou ainda não) o nó de partida, de chegada (
            self.graphedges[line[1]][line[2]].append(float(line[3]))

        self.dijkstra(start_station) #Corre o Algoritmo Dijkstra acima definido

        path = [] #Cria uma lista vazia (inicialmente) que guarda o percurso mais curto
        current_node = end_station #Define como nó atual o da estação de chegada
        while current_node is not None: #Corre até não existirem nós disponiveis
            path.insert(0, current_node) #Coloca no início da lista o nó atual
            current_node = self.previous[current_node]  #Coloca o nó atual como sendo o anterior ao mesmo (dado que ele est
        return path

    elif 7 < int(time[0]) <= 10:#Dado um tempo, é calculado o percurso que contem o percurso mais rapido em horário AM PEAK
        for line in self.connectionslist:
            if line[1] not in self.graphedges:
                self.graphedges[line[1]] = {}
            if line[2] not in self.graphedges[line[1]]:
                self.graphedges[line[1]][line[2]] = []
            self.graphedges[line[1]][line[2]].append(float(line[5]))

        self.dijkstra(start_station)

        path = []
        current_node = end_station
        while current_node is not None:
            path.insert(0, current_node)
            current_node = self.previous[current_node]
        return path

    elif 10 < int(time[0]) <= 16:#Dado um tempo, é calculado o percurso que contem o percurso mais rapido em horário INTER
        for line in self.connectionslist:
            if line[1] not in self.graphedges:
                self.graphedges[line[1]] = {}
            if line[2] not in self.graphedges[line[1]]:
                self.graphedges[line[1]][line[2]] = []
            self.graphedges[line[1]][line[2]].append(float(line[6]))

        self.dijkstra(start_station)

        path = []
        current_node = end_station
        while current_node is not None:
            path.insert(0, current_node)
            current_node = self.previous[current_node]
        return path

    else:#Dado um tempo, que não corresponde nem ao INTER PEAK, nem ao AM PEAK é calculado o percurso que contem o percurso
        for line in self.connectionslist:
            if line[1] not in self.graphedges:
                self.graphedges[line[1]] = {}
            if line[2] not in self.graphedges[line[1]]:
                self.graphedges[line[1]][line[2]] = []
            self.graphedges[line[1]][line[2]].append(float(line[4]))

        self.dijkstra(start_station)

        path = []
        current_node = end_station
        while current_node is not None:
            path.insert(0, current_node)
            current_node = self.previous[current_node]
        return path
```

Por fim, foi desenvolvido um método de visualização, utilizando o Folium, que mostra o grafo por completo, os pontos gerados e pinta o caminho mais rápido entre o início e o fim. Em baixo, está apresentado o código para este efeito e um exemplo de como este fica utilizando dois pontos aleatórios e a uma certa hora do dia.

```python
def draw_path(self, random_points1,time=None): #Recebe os dois pontos ((x1,y1),(x2,y2)) e possivelmente um tema e desenha um
    start_station = self.calculate_nearest_station(random_points1[0]) #Corre a função calculate_nearest_station() utilizando
    end_station = self.calculate_nearest_station(random_points1[1]) #Corre a função calculate_nearest_station() utilizando o
    path = self.shortest_path(start_station, end_station, time) #Corre a função shortest_path() utilizando as estações acima

    m = folium.Map(location=[0, 0], zoom_start=2) #Cria um mapa grafo vazio em Folium

    folium.Marker(location=[random_points1[0][0],random_points1[0][1]], icon=folium.Icon(color='red')).add_to(m) #Marca as c
    folium.Marker(location=[random_points1[1][0],random_points1[1][1]], icon=folium.Icon(color='red')).add_to(m) #Marca as c

    for node, data in self.graph.nodes(data=True): #Adiciona ao mapa grafo todas as estações: para isso, vai buscar todos os
        folium.Marker(location=[data['latitude'], data['longitude']], popup=node).add_to(m)

    for source, target, data in self.graph.edges(data=True): #Adiciona ao mapa grafo todas as conecções: para isso, vai busc
        folium.PolyLine(locations=[(self.graph.nodes[source]['latitude'], self.graph.nodes[source]['longitude']),
                                   (self.graph.nodes[target]['latitude'], self.graph.nodes[target]['longitude'])],
                        color='red', weight=2, popup=data['line']).add_to(m)

    for i in range(len(path) - 1): #Dada a lista path, marca todas as arestas do caminho otimo
        source = path[i]
        target = path[i + 1]
        data = self.graph.edges[source, target] # Obtém os dados da aresta no grafo original.

        folium.PolyLine(locations=[(self.graph.nodes[source]['latitude'], self.graph.nodes[source]['longitude']),
                                   (self.graph.nodes[target]['latitude'], self.graph.nodes[target]['longitude'])],
                        color='green', weight=5, popup=data['line']).add_to(m)

    m.save('graph_with_path.html') #Salva o mapa em um arquivo HTML
```

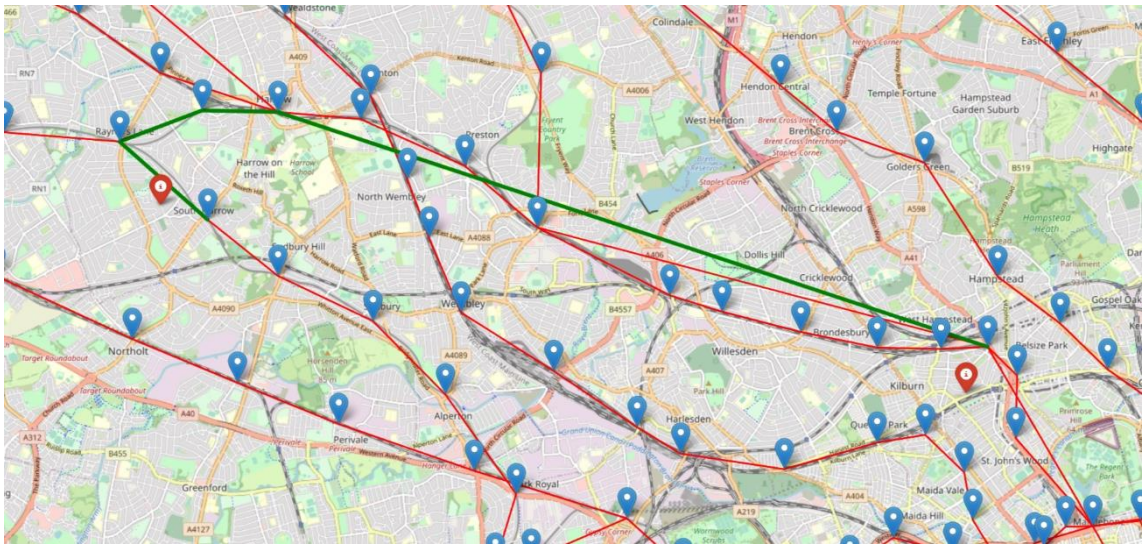


Figura 4: Estrato do grafo usando Folium com caminho mais rápido marcado

Legenda: Ponto aleatório; Estações; Linhas; Caminho mais rápido

Como podemos observar, foram criados dois pontos aleatórios e, com base na localização dos mesmos, foram escolhidas as estações mais próximas. Depois de escolhidas as estações, o algoritmo calculou o caminho mais rápido entre elas.

Descrição da análise de resultados e visualização

Com base nos resultados obtidos da análise dos dados da rede do Metro de Londres, destacámos os seguintes pontos:

Distribuição geográfica das estações: Observámos que há uma distribuição abrangente das estações por toda a cidade de Londres, com maior densidade nas áreas centrais. Isto

possivelmente acontece porque nestas regiões se concentram mais pessoas e, por isso, há mais movimento.

Estações com alta centralidade de grau: Identificámos várias estações que apresentam várias conexões, desempenhando um papel fundamental como pontos de transferência entre diferentes linhas, permitindo uma maior conectividade e facilidade de deslocamento para as pessoas.

Acessibilidade às principais atrações turísticas: Ao examinar a proximidade das estações em relação às principais atrações turísticas de Londres, constatámos que o Metro oferece acesso conveniente a locais como o Museu Britânico, a Torre de Londres, o Palácio de Buckingham.

Com o objetivo de visualizar e comunicar esses resultados de forma eficaz, apresentamos no relatório códigos, gráficos, mapas e outras representações visuais. O código e o mapa apresentado anteriormente fornecem-nos uma visão geral de como o algoritmo funciona e de como o podemos utilizar a nosso favor. Além disso, o Folium que criámos permite explorar a localização e as características de cada estação individualmente, facilitando a compreensão da rede em um contexto geográfico.

Essa descrição dos resultados, juntamente com as visualizações correspondentes apresentadas ao longo do relatório, permite-nos transmitir de maneira clara e objetiva as descobertas e perceções obtidas durante a análise dos dados da rede do Metro de Londres.

Conclusão:

Com a realização cuidada deste trabalho, podemos não só explorar alguns aspetos funcionais e geográficos da rede do Metro de Londres, como perceber uma possível implementação de uma estrutura de dados, denominada por grafo. Podemos entender que, de facto, a "digitalização" destas estruturas reais têm um grande impacto na sua exploração adequada e na utilidade que ferramentas desenvolvidas por cima das mesmas podem trazer. Certamente que o algoritmo Dijkstra é uma delas, podendo notificar aos passageiros, dado o sítio para onde desejam se deslocar, qual a forma mais rápida de o fazer, tendo em conta o momento temporal que se encontram.

Concluímos acima de tudo, a importância que tem a representação digital destas estruturas e o impacto que têm no quotidiano.