


## Trabalho 2

UC: Estruturas de Dados e Algoritmos

Maio de 2023

Gonalo Giro 111515

Joo Magara 111640



```
81 return array(
82     'code' => $captcha_config['code'],
83     'image_src' => $image_src
84 );
85 }
86
87
88 if ( function_exists('hex2rgb') ) {
89     function hex2rgb($hex_str, $return_string = false, $separator = ',') {
90         $hex_str = preg_replace("/[^0-9A-Fa-f]/", '', $hex_str); // Gets a p
91         $rgb_array = array();
92         if ( strlen($hex_str) == 6 ) {
93             $color_val = hexdec($hex_str);
94             $rgb_array['r'] = 0xFF & ($color_val >> 0x10);
95             $rgb_array['g'] = 0xFF & ($color_val >> 0x08);
96             $rgb_array['b'] = 0xFF & $color_val;
97         } elseif ( strlen($hex_str) == 3 ) {
98             $rgb_array['r'] = hexdec(str_repeat(substr($hex_str, 0, 1), 2));
99             $rgb_array['g'] = hexdec(str_repeat(substr($hex_str, 1, 1), 2));
100             $rgb_array['b'] = hexdec(str_repeat(substr($hex_str, 2, 1), 2));
101         } else {
102             return false;
103         }
104         return $return_string ?
105             serialize($rgb_array) : $rgb_array;
106     }
107 }
```

# Índice

Introdução .....	3
Métodos e Resultados.....	3
Aquisição de Dados .....	3
Limpeza dos Dados.....	4
Análise dos Dados .....	5
Recolha de resultados .....	7
Descrição da análise de resultados e Visualização .....	10
Discussão e Conclusões .....	13
Referências .....	13

## Introdução

O objetivo deste trabalho é representar, visualizar e analisar informação sobre a rede do metro de Londres utilizando um grafo. Este trabalho utiliza os conjuntos de dados (data sets) fornecidos em Stations.csv e Connections.csv. A implementação do grafo que representa a rede do metro de Londres em Python e possui uma funcionalidade de visualização do grafo utilizando bibliotecas e pacotes adequados a visualização de grafos (como por exemplo o NetworkX). Esta implementação do grafo que representa a rede de 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.

O grupo estava menos à vontade com o Networkx, para obter mais informação e aprender a utilizar as funções já definidas, vimos os tutoriais.

Quanto à parte do grafo já tinha sido abordada em aula e implementámos as classes Node, Edge e Graph o que nos ajudou depois a compreender mais rapidamente o pacote Networkx.

## Métodos e Resultados

### Aquisição de Dados

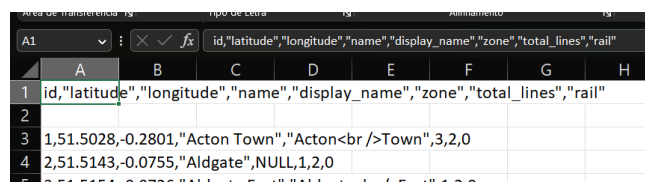
Quanto à recolha da informação dos *datasets* disponibilizados (stations.csv e connections.csv) optamos por utilizar a biblioteca csv e desenvolvemos funções para ler e armazenar os dados de cada *dataset*.

Para a leitura do *dataset* stations.csv desenvolvemos uma função que retorna a lista de linhas do csv(list\_stations).

def list\_stations:

é criada uma lista onde vai ser guardadas as linhas do *dataset*;

neste *dataset* com cabeçalho e uma linha em branco, usamos o next() para não guardar na lista essas primeiras 2 linhas.



A1	B	C	D	E	F	G	H
1	id,"latitude","longitude","name","display_name","zone","total_lines","rail"						
2							
3	1,51.5028,-0.2801,"Acton Town","Acton Town",3,2,0						
4	2,51.5143,-0.0755,"Aldgate",NULL,1,2,0						
5	3 51 5154 -0 0726 "Aldgate Fact" "Aldgatechr />Fact" 1 2 0						

Figura 1- Ficheiro "stations.csv"

```
def list_stations(self):
    stations_list = []
    with open('stations.csv') as stations1:
        stations2 = csv.reader(stations1, delimiter=',')
```

```

        next(stations2)
        next(stations2)
    for station in stations2:
        stations_list.append(station)
    return stations_list

```

Para a leitura do *dataset* connections.csv desenvolvemos uma função que retorna a lista de linhas do csv(list\_connections).

def list\_connections:

é criada uma lista onde vai ser guardadas as linhas do *dataset*;

neste *dataset* com cabeçalho, usamos o next() para não o guardar na lista.

```

def list_connections(self):
    list_connections = []
    with open("connections.csv") as connections1:
        connections2 = csv.reader(connections1, delimiter=',')
        next(connections2)
        for line in connections2:
            list_connections.append(line)
    return list_connections

```

## Limpeza dos Dados

Para a limpeza do *dataset* stations.csv, que é onde estão armazenadas as informações relativamente às estações que vão ser os nossos Nodes, desenvolvemos uma função que guarda a informação necessária da lista nos Nodes(stations).

def stations:

utilizando a função list\_stations extraímos a lista com a informação necessária;

recorrendo à biblioteca Networkx adicionamos os Nodes onde guardamos essa informação.

```

def stations(self):
    stations_list = self.list_stations()
    for station in stations_list:
        pos = float(station[1]), float(station[2])
        self.graph.add_node(station[0], pos=pos,
zone=float(station[5]), n_lines=float(station[6]), name=station[3])

```

station[0] é o id da estação, station[1] e station[2] são respetivamente latitude e longitude, station[3] é o nome da estação, station[5] é a zona a que a estação pertence, station[6] é o numero de linhas que passa na estação.

Para a limpeza do *dataset* connections.csv, que é onde estão armazenadas as informações relativamente às conexões que vão ser os nossos Edges, desenvolvemos uma função que guarda a informação necessária da lista nos Edges(connections).

def connections:

utilizando a função list\_connections extraímos a lista com a informação necessária;

recorrendo à biblioteca Networkx adicionamos os Edges onde guardamos essa informação.

```
def connections(self):
    connections = self.list_connections()
    for l in connections:
        self.graph.add_edge(l[1], l[2], distance=float(l[3]),
line=float(l[0]), tnormal=float(l[4]), t7_10=float(l[5]),
t10_16=float(l[6]))
```

l[1] é o node/estação em que começa o edge, l[2] é o node/estação em que acaba o edge, l[3] é a distancia entre o começo e o fim do edge, l[0] é a linha a que pertence o edge, l[4] é o tempo que demora a percorrer aquele edge fora das horas de peak, l[5] é o tempo que demora a percorrer aquele edge entre as 7h00 e as 10h00(AM peak), l[6] é o tempo que demora a percorrer aquele edge entre as 10h00 e as 16h00(PM peak).

## Análise dos Dados

Para devolver o número total de stations e de edges o processo é facilitado pela utilização do pacote Networkx.

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

Nesta fase, recuámos ao nosso ficheiro stations.csv e percebemos que em relação à zona a que cada estação pertence(station[5]), existem estações com valores não inteiros associados, por exemplo, uma estação pertencente à zona 1.5. Para estes casos, a interpretação que fizemos foi associar a estação a duas zonas, o que para o exemplo significaria que a estação pertencia às zonas imediatamente acima e abaixo (zonas 1 e 2).

```
84,51.4943,-0.1001,"Elephant & Castle","Elephant &<br
/>Castle",1.5,2,1
```

def n\_stations\_zones:

Através de um ciclo for verificamos, para cada node pertencente a self.graph a zona a que pertence. Se o valor da zona subtraído ao inteiro da zona for diferente de 0 então assume-se como a menor zona o inteiro da zona e a maior como esse valor mais uma unidade. Se for igual a 0 então esse é o único valor da zona de cada estação.

Depois faz-se a verificação se cada uma das zonas já está no dicionário “zone\_count” se sim apenas se adiciona 1 unidade, se não o valor associado à chave passa a 1.

```
def n_stations_zones(self):
    zone_count = {}
    for node, arrt in self.graph.nodes(data=True):
        zona = arrt['zone']
```

```

    if zona - int(zona) != 0:
        menor = int(zona)
        maior = int(zona + 0.5)
        if menor in zone_count:
            zone_count[menor] += 1
        else:
            zone_count[menor] = 1
        if maior in zone_count:
            zone_count[maior] += 1
        else:
            zone_count[maior] = 1
    else:
        if zona in zone_count:
            zone_count[zona] += 1
        else:
            zone_count[zona] = 1
total = 0
for z, n in zone_count.items():
    total += n
    print(f'Zona {z} : {n}')
return f'Total : {total}'

```

Para devolver o número de edges por linha, criámos um dicionário. Para cada edge em `self.graph`, se a linha não estiver associada à chave, o valor associado passa a ser 1, de outra forma, soma-se uma unidade apenas.

```

def n_edges_line(self):
    edges = {}
    for start, end, arrt in self.graph.edges(data=True):
        if arrt['line'] not in edges.keys():
            edges[arrt['line']] = 1
        else:
            edges[arrt['line']] += 1
    for e, n in edges.items():
        print(f'A linha {e}, tem {n} edges')

```

Para devolver o grau médio das estações, entendeu-se que seria a soma do número de linhas referente a cada estação, dividido pelo número total de estações. Como o grafo em questão é direcionado, este valor de linhas terá de ser multiplicado por dois devido aos sentidos das linhas.

```

def mean_degree(self):
    soma = 0
    total = 0
    for node, arrt in self.graph.nodes(data=True):
        soma += (arrt['n_lines'] * 2)
        total += 1
    return soma/total

```

Por fim, dado o tipo de peso, o processo para devolver o peso médio é realizado da seguinte forma:

Para os edges em `self.graph` é retirado o peso e somado a um total até serem vistos todos os edges do peso em questão e depois é dividido pelo número total de edges através da função `self._n_edges()` já apresentada anteriormente, sendo retornado o resultado da divisão.

```
def mean_weight(self, weight):
    total = 0
    n = self.n_edges()
    for start, end, arrrt in self.graph.edges(data=True):
        total += int(arrrt[weight])
    return total/n
```

## Recolha de resultados

Para a visualização do grafo usando o pacote Matplotlib.pyplot, decidimos fazer a visualização propriamente dita através de um scatter plot. A função começa definindo duas listas vazias x e y, percorre as arestas do grafo usando um loop, dentro do loop, a função obtém as coordenadas dos nodes de início e fim da aresta a partir do grafo. Onde 'pos' é uma chave que armazena as coordenadas do node no grafo. Depois, a função utiliza a função plt.plot para traçar uma linha entre essas coordenadas no gráfico. Após percorrer todas as arestas e desenhar as linhas correspondentes, a função utiliza a função plt.scatter. Finalmente, a função chama plt.show() para exibir o gráfico e retorna para encerrar a função.

```
def visualize_scatter_plot(self):
    x = []
    y = []
    for start, end, arrrt in self.graph.edges(data=True):
        coords1 = self.graph.nodes[start]['pos']
        coords2 = self.graph.nodes[end]['pos']
        plt.plot([coords1[1], coords2[1]], [coords1[0], coords2[0]],
        color='darkblue', marker='o', markersize=3.5)
    plt.scatter(x, y)
    plt.show()
    return
```

Para a visualização do grafo usando o pacote Networkx utilizamos a função draw(), tivemos de recorrer ao Matplotlib.pyplot mais especificamente à função show(). Nesta função tentámos recorrer apenas ao pacote NetworkX, desenhando o grafo, no entanto para a demonstração gráfica tivemos de aceder à biblioteca Matplotlib.

```
def visualize_nx(self):
    nx.draw(self.graph, with_labels=True)
    plt.show()
    return
```

Para a visualização do grafo usando o pacote Folium, primeiro definimos o mapa, depois com um iterador a percorrer os nodes vão sendo adicionados com a respetiva 'pos' (que é uma lista com a latitude e longitude) com o folium.Marker(), para ligar os nodes utilizamos dois iteradores nos edges que "guardavam" as posições do início e do fim do edge que são adicionados ao mapa com o folium.PolyLine(). Para visualizarmos o mapa de Londres foi necessário pesquisar as coordenadas num motor de busca.

```
def visualize_graph(self):
    mapa = folium.Map(location=[51.5074, -0.1278], zoom_start=12)

    for node, arrrt in self.graph.nodes(data=True):
```

```

        folium.Marker(arrrt['pos'], popup=arrrt['name']).add_to(mapa)

    for start, end, arrrt in self.graph.edges(data=True):
        coords1 = self.graph.nodes[start]['pos']
        coords2 = self.graph.nodes[end]['pos']
        #color = self.get_color(arrrt['line'])
        folium.PolyLine([coords1, coords2],
color='darkblue').add_to(mapa)
    mapa.save("LondonMetroNetwork.html")
    return

```

Para gerar pontos e horas aleatórias utilizamos o pacote Random e desenvolvemos as seguintes funções:

```

def gera_ponto(self):
    x = random.uniform(51.3962, 51.7090)
    y = random.uniform(-0.6110, 0.1159)
    return x, y

def gera_hora(self):
    return random.randint(0, 23)

```

Verificamos os extremos de latitude e longitude das estações e com isso definimos os limites para gerar um ponto (latitude e longitude).

Para definir a estação mais próxima do ponto gerados fizemos a função distância com o pacote Math que nos ajudou a definir a qual seria essa estação:

```

def distancia(self, ponto1, ponto2):
    return math.sqrt((float(ponto1[0])-float(ponto2[0]))**2 +
(float(ponto1[1])-float(ponto2[1]))**2)

```

```

def station_prox(self, ponto):
    min_distancia = 9999999999
    prox = None
    for start, end, arrrt in self.graph.edges(data=True):
        ponto2 = self.graph.nodes[start]['pos']
        distancia = self.distancia(ponto, ponto2)
        if distancia <= min_distancia:
            min_distancia = distancia
            prox = start
    return prox

```

Com esta função(*station\_prox(ponto)*) que dado um ponto devolve a estação que está mais próxima do mesmo, visto que nem todas as estações têm ligação a outra estação, procuramos nos edges e não nos nodes, como todas as conexões têm ida e vinda comparamos a distância apenas do 'start'.

Com todas estas funções, desenvolvemos uma que gerasse dois pontos (start e end), uma hora, e nos devolvesse as coordenadas dos pontos que gerou e o caminho mais rápido entre si de acordo com a hora do dia (dado o tipo de peso).

```

def caminho_rapido(self):
    s = self.gera_ponto()

```



```

start = self.station_prox(s)
print(f'Start: {start}')
e = self.gera_ponto()
end = self.station_prox(e)
print(f'End: {end}')
hora = self.gera_hora()
print(f'Hour: {hora}')
caminho = nx.dijkstra_path(self.graph, start, end,
weight='tnormal')
if 7 <= hora < 10:
    caminho = nx.dijkstra_path(self.graph, start, end,
weight='t7_10')
elif 10 <= hora < 16:
    caminho = nx.dijkstra_path(self.graph, start, end,
weight='t10_16')
return s, e, caminho

```

“s” e “e” são pontos aleatórios, start é a estação mais próxima do ponto s e end a estação mais próxima do ponto e, ‘hora’ é a hora gerada, com estes dados irá calcular o caminho mais rápido entre a estação ‘start’ e ‘end’. Os “if” são para ajustar o tempo que demora de uma estação para a outra alterando o “peso”, de acordo com a hora do dia.

Para mostrar o caminho mais rápido no mapa, primeiro definimos o mapa e adicionamos os dois pontos gerados (*folium.Marker()*), neste caso *s*, *e*, *caminho* = *data* é para fazer *mostrar\_caminho(caminho\_rapido)*.

```

def mostrar_caminho(self, data):
    s, e, caminho = data
    mapa = folium.Map(location=[51.5074, -0.1278], zoom_start=12)
    folium.Marker(s).add_to(mapa)
    folium.Marker(e).add_to(mapa)

    for start, end, arrt in self.graph.edges(data=True):
        coords1 = self.graph.nodes[start]['pos']
        coords2 = self.graph.nodes[end]['pos']
        folium.PolyLine([coords1, coords2],
color='darkgreen').add_to(mapa)

    for n in range(0, len(caminho)-1):
        coords1 = self.graph.nodes[caminho[n]]['pos']
        coords2 = self.graph.nodes[caminho[n+1]]['pos']
        folium.PolyLine([coords1, coords2], color='red').add_to(mapa)
    mapa.save("DijkstraNX.html")
    return

```

Depois como no *visualize\_graph* com um for a percorrer os edges utilizamos dois iteradores que “guardavam” as posições do inicio e do fim do edge que são adicionados ao mapa com o

folium.PolyLine()). E com a lista de edges que formam o caminho mais rápido da estação 'start' à estação 'end', é traçado no mapa de uma cor diferente.

## Descrição da análise de resultados e Visualização

```
>>> L.n_stations_zones()
Zona 3.0 : 70
Zona 1.0 : 64
Zona 2.0 : 96
Zona 4.0 : 44
Zona 5.0 : 29
Zona 6.0 : 20
Zona 7 : 3
Zona 10.0 : 2
Zona 9.0 : 1
Zona 8.0 : 2
Total : 331
```

Figura 2-Output "n\_stations\_zones()"

```
>>> L.n_edges_line()
A linha 10.0, tem 79 edges
A linha 4.0, tem 118 edges
A linha 3.0, tem 22 edges
A linha 6.0, tem 16 edges
A linha 9.0, tem 102 edges
A linha 1.0, tem 44 edges
A linha 7.0, tem 52 edges
A linha 8.0, tem 54 edges
A linha 2.0, tem 98 edges
A linha 12.0, tem 2 edges
A linha 11.0, tem 26 edges
A linha 5.0, tem 16 edges
```

Figura 3-Output "n\_edges\_line()"

```
>>> L.caminho_mais_curto()
Start: 118
End: 117
Hour: 13
((51.41850751360916, -0.5164370311496417), (51.43091714737023, -0.5689425384666886), [118, 117])
```

Figura 4-Output "caminho\_mais\_curto()"

```
>>> L.caminho_mais_curto()
Start: 271
End: 291
Hour: 16
((51.527763841822456, -0.4995809766417549), (51.57334081674087, -0.3545094815539319), [271, 125, 134, 220, 222, 75, 210, 291])
```

Figura 5-Output "caminho\_mais\_curto()"

```
>>> L.dijkstra()
((51.50167080806332, -0.17770609147067734), (51.59976504882969, -0.4825835339295216), [236, 146, 133, 107, 28, 11, 249, 254, 94, 115, 168, 214])
```

Figura 6-Output "dijkstra()" com coordenadas do ponto inicial, ponto final e caminho mais rápido

```
def visualize_scatter_plot()
```

Da função surge a visualização legendada com latitude e longitude com acesso à biblioteca matplotlib.pyplot num scatter plot.

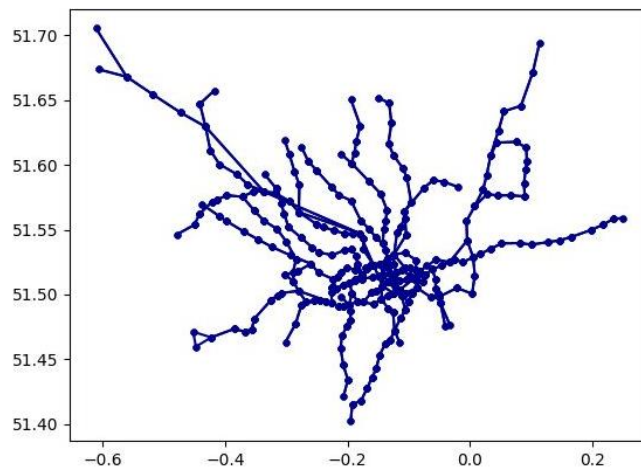


Figura 7-Output de "visualize\_scatter\_plot()"

Neste gráfico podemos visualizar utilizando apenas a função draw do NetworkX e a função show do matplotlib.

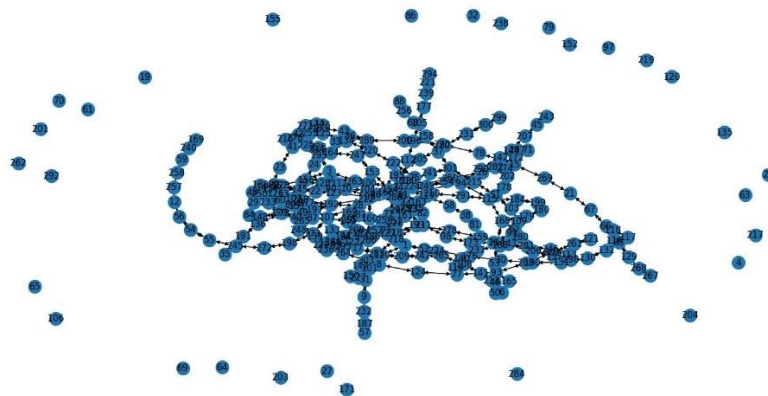


Figura 8-Output de "visualize\_nx()"

Aqui surge a visualização do mapa de Londres com recurso à biblioteca Folium, com os ícones de cada estação a azul e onde as linhas, a verde, representam cada conexão existente.

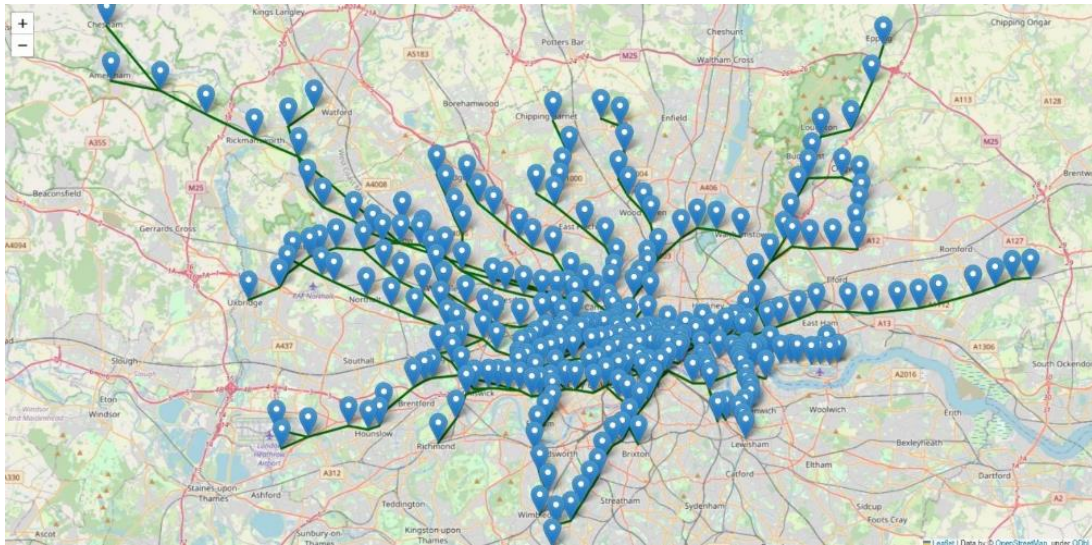


Figura 9-Output de "visualize\_graph()"

```
def mostrar_caminho()
```

Na função é representado todo o grafo, são gerados os pontos aleatórios, representados pelos ícones a azul, sendo o início e fim da linha a vermelho respetivamente a estação de partida e de chegada (as mais próximas dos pontos gerados) e o caminho ótimo, o mais rápido para a hora do dia gerada.

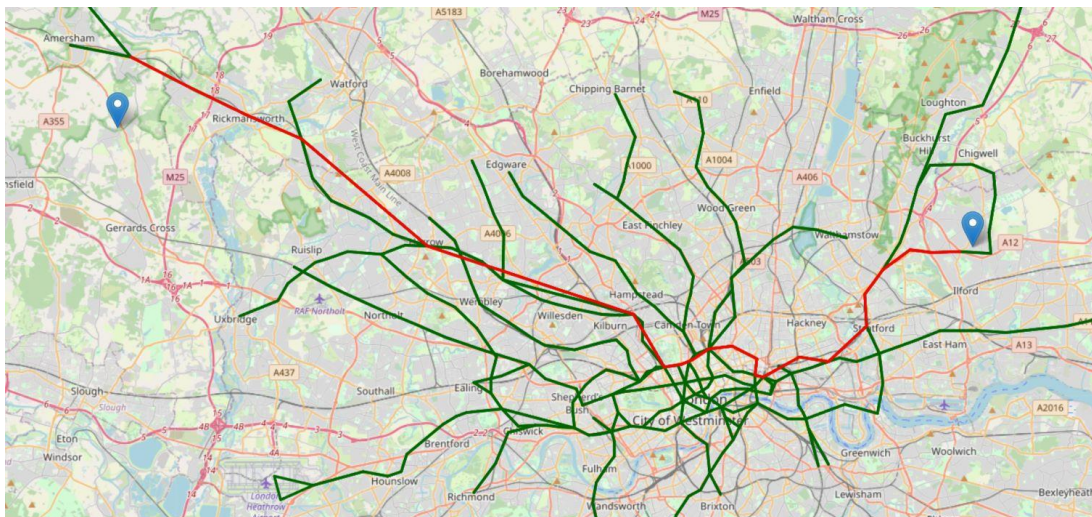


Figura 10-Output de "mostrar\_caminho()"

Realizámos um segundo teste para verificar que de facto os pontos eram gerados aleatoriamente. Comprova-se, surgem pontos com coordenadas diferentes, tal como o trajeto percorrido, visualizado a vermelho.



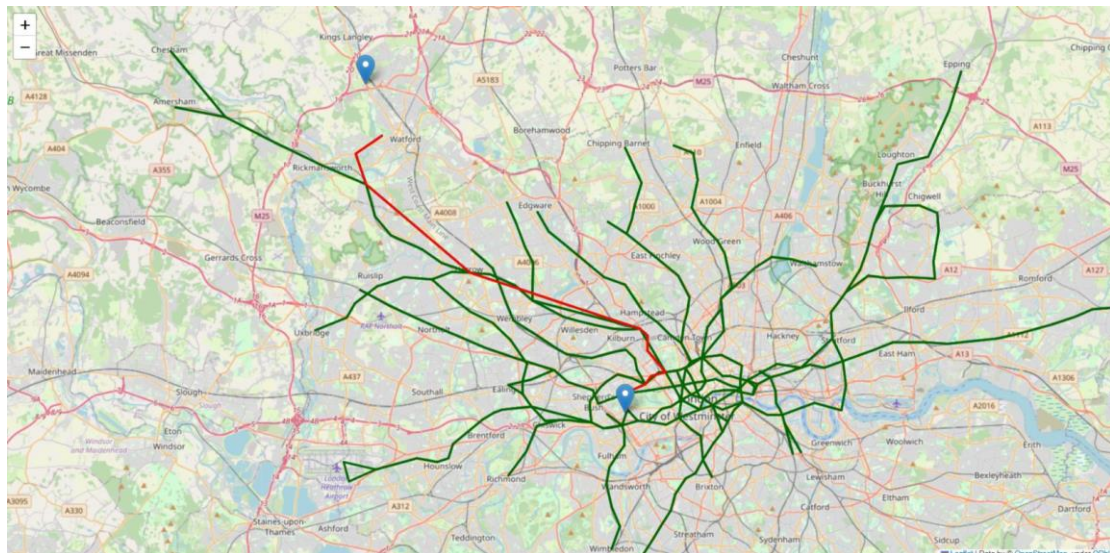


Figura 11-Output de "mostrar\_caminho()"

## Discussão e Conclusões

Para a análise de resultados e visualização verifica-se que o pacote NetworkX não consegue desenhar um volume de dados tão grande e a visualização que é bem mais limitada surge menos próxima da realidade do metro de Londres do que as restantes. Ou seja, este pacote permite uma escrita de código mais rápida e menos complexa, no entanto a visualização é que menos ajuda na leitura das conexões e estações do metro.

Para as outras duas visualizações, o scatter plot permite uma perceção melhor das coordenadas de cada estação e conexão devido à escala de existente, no entanto, a função que utiliza a biblioteca Folium permite uma visualização mais interativa, onde se destacam as estações das conexões mais facilmente e permite a localização de pontos de interesse que pode ser relevante e mais intuitivo para um mapa de transportes públicos.

## Referências

NetworkX. (2023). Tutorial. <https://networkx.org/documentation/latest/tutorial.html>