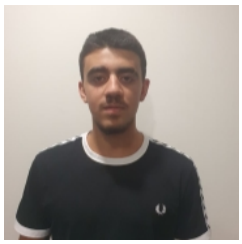
Inteligência Artificial

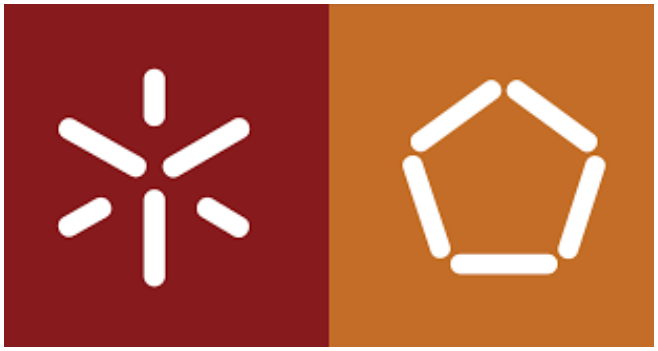
Trabalho Prático

LEI - 2023/2024

João Barroso	João Silva	Luís Vilas
A95195	91671	A91697



Grupo 35



Universidade do Minho

Avaliação por Pares

Número	Nome	Avaliação
A95195	João Barroso	0
A91671	João Silva	0
A91697	Luís Vilas	0

Resumo

Este relatório foi elaborado no contexto da elaboração do trabalho prático associado à disciplina de **Inteligência Artificial**, pertencente ao terceiro ano do curso de Licenciatura em Engenharia Informática. A seguir, serão apresentadas de forma sucinta as etapas relativas ao processo de tomada de decisões e à formulação dos objetivos delineados pelos docentes. Incluem-se, entre esses objetivos, o desenvolvimento de sensibilidade para a resolução de problemas mediante sua formulação, bem como a conceção e implementação de algoritmos de procura.

Índice

Lista de Figuras

Lista de Tabelas

1 Introdução

No âmbito da unidade curricular de **Inteligência Artificial** foi proposto a realização dum trabalho prático em grupo que consiste no desenvolvimento de vários algoritmos de procura tendo em conta um tópico relevante para a sociedade atual, a sustentabilidade. Para tal foi desenvolvida uma aplicação sobre o nome de ***Health Planet***.

Neste sentido a empresa tem como objetivo utilizar o meio de de entrega de encomendas mais sustentável/ecológico para o planeta e, para tal, tem a seu dispor um número determinado de trabalhadores (estafetas) e veículos. Nesse contexto, existem os seguintes meios de transporte, com diferentes níveis de consumos de energia e cargas máximas:

- **Bicicletas** - transporte máximo de 5kg e velocidade média 10km/h, decréscimo de 0.6km/h por cada Kg transportado
- **Motos** - transporte máximo de 20kg e velocidade média de 35km/h, decréscimo de 0.5km/h por cada Kg transportado
- **Carros** - transporte máximo de 100kg e velocidade média de 50km/h, decréscimo de 0.1km/h por cada Kg transportado

Cada estafeta tem a si designado um conjunto de entregas e efetuar por determinadas ruas duma cidade assim como um rank global que é afetado pelo seu desempenho na entrega das encomendas assim como a satisfação dos clientes.

Cada cliente poderá indicar um prazo para a entrega da sua encomenda, sendo este limite, naturalmente, controlado para que não sejam feitas exigências impossíveis de cumprir. Além disto também compete ao cliente, como mencionado em cima, a avaliação do serviço prestado por cada estafeta, sendo este avaliado numa escala de 0 a 5 estrelas.

As encomendas devem ter a informação necessária como os produtos a ser transportados, que estafeta irá realizar o pedido o local de entrega etc. O preço da entrega deverá ser influenciado pelos custos de transporte, dependendo do tipo de veículo utilizado, assim como os prazos indicados pelo cliente.

Depois duma análise cautelosa da futura estrutura do projeto foi decidido, pelo grupo, dividir a resolução do em duas *"fases"*.

- **1º Fase** - Geração inicial do grafo, o desenvolvimento dos vários algoritmos e comparação dos seus desempenhos
- **2º Fase** - Formulação da aplicação ***Health Planet*** permitindo a clientes escolher o local da entrega, assim como os produtos e prazos a entregar.

2 Formulação do Problema

2.1 Representação do Estado

- O estado atual do mapa em grafo (estradas cortadas ou com trânsito)
- O estado das encomendas e entregas (entregues ou não)
- Cada estado reflete uma disposição específica dos recursos disponíveis

2.2 Estado Inicial

- Nenhuma rua se apresenta com obstruções (trânsito alto ou estradas cortadas)
- Nenhum estafeta nem veículo saiu do armazém
- Nenhuma encomenda efetuada pelos clientes

2.3 Estado Objetivo

- Todas as encomendas foram entregues aos destinos correspondentes utilizando o meio e o caminho de transporte mais económico

2.4 Operadores

- Movimentação dos veículos pelas ruas
- Inicio da procura do melhor caminho
- A avaliação feita pelo cliente (é o último passo para finalizar uma entrega)

2.5 Custo da Solução

- Valor dependente de parâmetros como o peso dos produtos da entrega, a distância percorrida e o índice de poluição de cada veículo

3 Decisões Primordiais da Resolução

No inicio do desenvolvimento deste trabalho o grupo entendeu que a implementação inicial do mapa em que seriam feitas as travessias era da maior importância para um bom desenrolar do processo de implementação das *features* do projeto, uma vez que o código fornecido das aulas práticas já cobria muitos dos requisitos necessários da aplicação.

Sendo assim, inicialmente, foi preparada um simples *parser* de um ficheiro de texto que descrevia um conjunto de ruas manualmente intercaladas para formarem uma região fictícia.

```
"Rua de Barros" [(Rua da Universidade, 150)(Rua do Semaforo, 100)(Rua Raul, 120)] 200
"Rua da Universidade" [(Rua de Barros, 200)(Rua Raul, 120)] 150
"Rua do Semaforo" [(Rua de Barros, 200)(Rua Raul, 120)] 100
"Rua Raul" [(Rua de Barros, 200)(Rua da Universidade, 150)(Rua do Semaforo, 100)] 120
"Rua Central" [(Rua Larga, 160)(Avenida Principal, 180)(Rua Raul, 120)] 200
"Avenida Principal" [(Rua Central, 200)(Rua Larga, 160)(Avenida Larga, 200)(Rua Estreita, 70)(Avenida Secundaria, 90)(Rua dos Arcos, 70)] 180
"Rua Larga" [(Rua Central, 200)(Avenida Principal, 180)(Avenida Larga, 200)] 160
"Avenida Larga" [(Avenida Principal, 180)(Rua Larga, 160)(Rua Estreita, 70)] 200
"Rua Estreita" [(Avenida Principal, 180)(Avenida Larga, 200)(Avenida Secundaria, 90)] 70
"Avenida Secundaria" [(Avenida Principal, 180)(Rua Estreita, 70)(Rua da Praca, 60)(Avenida da Colina, 80)] 90
"Rua dos Arcos" [(Avenida Principal, 180)(Rua do Largo, 90)] 70
"Rua do Largo" [(Rua dos Arcos, 70)(Rua da Praca, 60)] 90
"Rua da Praca" [(Avenida Secundaria, 90)(Rua do Largo, 90)(Avenida da Colina, 80)(Rua das Flores, 60)(Rua das Oliveiras, 80)] 60
"Avenida da Colina" [(Avenida Secundaria, 90)(Rua da Praca, 60)(Rua da Fonte, 80)(Rua do Bosque, 180)] 80
"Rua das Flores" [(Rua da Praca, 60)(Rua da Fonte, 80)] 60
"Rua da Fonte" [(Avenida da Colina, 80)(Rua das Flores, 60)(Rua das Oliveiras, 80)(Rua do Bosque, 180)] 80
"Rua das Oliveiras" [(Rua da Praca, 60)(Rua da Fonte, 80)(Rua dos Jardins, 80)] 80
"Rua do Bosque" [(Avenida da Colina, 80)(Rua da Fonte, 80)(Avenida do Parque, 40)(Rua dos Jardins, 80)(Rua das Pedras, 40)(Rua do Mirante, 40)] 180
"Avenida do Parque" [(Rua do Bosque, 180)(Rua dos Jardins, 80)(Rua das Pedras, 40)] 40
"Rua dos Jardins" [(Rua das Oliveiras, 80)(Rua do Bosque, 180)(Avenida do Parque, 40)(Rua do Monte, 50)] 80
"Rua das Pedras" [(Rua do Bosque, 180)(Avenida do Parque, 40)(Rua do Riacho, 30)(Avenida do Vale, 40)] 40
"Rua do Riacho" [(Rua das Pedras, 40)(Avenida do Vale, 40)(Avenida da Serra, 50)(Rua do Penhasco, 30)] 30
"Avenida do Vale" [(Rua das Pedras, 40)(Rua do Riacho, 30)(Rua do Monte, 50)] 40
```

Figura 1. Excerto do mapa original

Como é percetível na figura, a estrutura do ficheiro já é alusiva a uma representação num dicionário de *python*. O último valor é alusivo ao tamanho da rua que será a *key* do dicionário.

$$Nome\ Rua : [(Nome\ Rua\ Adjacente, Tamanho\ Rua\ Adjacente)] \tag{1}$$

Naturalmente, esta solução não é uma solução realista, dinâmica nem aceitável para o desenvolvimento duma aplicação de grande escala, uma vez que os mapas não são realistas e são fixos à luz do que estiver escrito nos ficheiros de texto.

Após alguma deliberação e pesquisa foi encontrada uma solução que apelava à ambição que o grupo tinha de implementar uma resolução do projeto exemplar. Esta solução ganhou forma a partir da utilização da biblioteca ***OSMnx*** [1] que é um pacote Python para descarregar, modelar, analisar e visualizar redes de ruas e outros recursos geoespacial do OpenStreetMap criada por Geoff Boeing, um professor na Universidade do Sul da Califórnia. O *nx* referesse ao facto de ser construida em cima da biblioteca ***NetworkX*** [2], que por sua vez é uma biblioteca útil para estudar a estrutura e dinâmica de redes/grafos complexas.

A utilização deste pacote permite-nos a expandir o universo de procura para um nível mundial, tornando a aplicação usável em, possivelmente, todas as circunstancias possíveis. Isto tudo é exequível com apenas linha de código.

```python
def run(location):
    # Download the street network
    G =ox.graph_from_place(location, network_type="drive")
```

O método *graph_from_place* recebe um parâmetro que é o tipo de rede, que pode ser uma das seguintes opções:

- **drive** - obtém as vias públicas onde seja possível conduzir (à exceção de estradas de serviço)
- **drive_service** - obtém as vias públicas onde seja possível conduzir incluindo estradas de serviço
- **walk** - obtém todas as estradas e caminhos que os peões podem usar (este tipo ignora a existência de sentido único
- **bike** - obtém todas as estradas e caminhos que podem ser utilizados por ciclistas
- **all** - obtém todas as estradas e caminhos do *Open Street Maps*

O grafo que é devolvido pela biblioteca **OSMnx** é um *MultiDiGraph* (multigrafo) com peso, isto é, um grafo direcionado, em que podem haver várias *edges* entre os mesmos *nodes*. Neste grafo, os *edges* representam estradas ou segmentos que conectam nodos, a direção das *edges* indicam o sentido de trânsito e estas também contêm informação sobre o seu comprimento, tipo de via, número de secção, entre outros.

Esta biblioteca é extremamente rica em conteúdo que poderia ser explorado numa enorme panóplia de projetos, mas para o que nos diz respeito apenas iremos estudar alguns aspetos limitados que iremos falar mais à frente.

4 Implementação do grafo

4.1 Location.py

Foi decidido pelo grupo que deveria ser feita a filtragem de alguma informação fornecida pela biblioteca **OSMnx** que não parece ser relevante para o nosso problema

```
for node, data in G_filtered.nodes(data=True):
    data.pop('name', None)
    data.pop('length', None)
    data.pop('highway', None)
    data.pop('ref', None)

for u, v, k, data in G_filtered.edges(keys=True, data=True):
    if data.get('junction') !='roundabout' and data.get('name') is None:
        data.pop('osmid', None)
        data.pop('reversed', None)
        data.pop('geometry', None)
        data.pop('lanes', None)
    else:
        data.pop('highway', None)
        data.pop('osmid', None)
        data.pop('reversed', None)
        data.pop('geometry', None)
        data.pop('lanes', None)
```

Aqui foi removida informação relativa aos *nodes*, nomeadamente do nome, tamanho, se é autoestrada ou não e o código de referência. Nas *edges* foi removida a informação relativamente ao seu identificador do *OSM*, se a direção é revertida ou não, forma geométrica e número de vias de trânsito no mesmo sentido, respetivamente caso seja uma junção.

Também houve necessidade de editar o nome de algumas ruas uma vez que uma rua poderia aparecer várias vezes numa possível lista de *edges*, isto porque, por exemplo, uma avenida pode ter várias interseções ao longo do seu percurso, neste grafo é representado uma *edge* diferente sempre que uma rua é "interrompida"por uma interseção.

```
name_counter ={}
for u, v, k, data in G_filtered.edges(keys=True, data=True):
    name =data.get('name')
    if name is not None:
        name_key =str(name)
        if name_key in name_counter:
            name_counter[name_key] +=1
            data['name'] =f"{name} ({name_counter[name_key]})"
        else:
            name_counter[name_key] =1
```

A solução que o grupo formulou foi inserir um indicador no nome de cada *edge* que refletia a ordem da secção da rua.

Para pôr fim a esta parte da formulação do grafo falta-nos mencionar a criação das estruturas que iram guardar os *nodes*, *edges* e o conjunto destes que iram ser usados pela classe `Graph` que irá ser explorada posterioemente.

```
neighb =create_neighborhood_dict(G_filtered)
edgesList =create_edges_list(G_filtered)
nodeList =create_nodes_list(G_filtered)
```

As três estruturas de dados que iram fazer parte da definição do Grafo são:

- **Dicionário *graph*** - Um dicionário que represente cada nodo e os seus adjacentes
- **Lista *edges*** - Uma lista de todos os *edges* entre *nodes*
- **Lista *nodes*** - Uma lista de todos os *nodes*

```
def create_neighborhood_dict(graph):
    neighborhood_dict ={}

    for u, v, k, data in graph.edges(keys=True, data=True):

        if u not in neighborhood_dict.keys():
            neighborhood_dict[u] =[(v, data['length'], k)]
        elif (v, data['length'], k) not in neighborhood_dict[u]:
            neighborhood_dict[u].append((v, data['length'], k))

    return neighborhood_dict
```

Aqui geramos um dicionário em que as *keys* são o identificador de um *node* e o *value* é uma lista que contém a informação relevante relativa aos nodos adjacentes do *node* representado na *key*.

```
def create_nodes_list(graph):
    list =[]

    for node, data in graph.nodes(data=True):
        id =node
        x =data["x"]
        y =data["y"]
        pos =Positions(x, y)
        street_count =data["street_count"]

        nodo =Node(id, pos, street_count)
        if nodo not in list:
            list.append(nodo)

    return list
```

Nesta função produzimos uma simples lista de objetos *nodes*, que incluí as coordenadas de latitudinais e longitudinais e a quantidade de ruas que nele incide.

```
def create_edges_list(graph):
    edges =[]

    for u, v, k, data in graph.edges(keys=True, data=True):
        name =data.get('name', "")
        origem =u
        destino =v

        oneway =data.get('oneway', False)
        highway =data.get('highway', [])
        rotunda =data.get('junction', [])
        ref =data.get('ref', False)
        ponte =data.get('bridge', False)
        tunnel ='tunnel' in data
        access =data.get('access', [])
        vel =data.get('maxspeed', [])
        length =data.get('length', 0)

        if isinstance(highway, str): highway =[highway]
        if isinstance(vel, str): vel =[vel]

        rua =randomizacao_de_cortadas_transito(name, origem, destino, oneway, highway,
                                               ↪ rotunda, ponte, tunnel, access, vel,
                                               ↪  length, ref)
        edges.append(rua)

    return edges
```

Para o caso das *edges* foi necessária alguma atenção uma vez que existem vários tipos de ruas, quer sejam vias de apenas um sentido, autoestradas, pontes, túneis e afins. Também é nesta fase que é implementada um dos aspetos dinâmicos do projeto na forma de passar certas ruas a trânsito excessivo ou até mesmo cortadas.

```
def randomizacao_de_cortadas_transito(name, origem, destino, oneway, highway, rotunda,
                                      ↪ ponte, tunnel, accesso, vel, length, ref):
    random1 =random.randint(0,9)
    random2 =random.randint(0,9)

    if (random1 ==1):
        rua =Ruas(name, origem, destino, oneway, highway, rotunda, ponte, tunnel, accesso,
                                               ↪ vel, length, ref, True,
                False)
    elif (random2 ==1):
        rua =Ruas(name, origem, destino, oneway, highway, rotunda, ponte, tunnel, accesso,
                                               ↪ vel, length, ref, False,
                True)
    else:
        rua =Ruas(name, origem, destino, oneway, highway, rotunda, ponte, tunnel, accesso,
                                               ↪ vel, length, ref)
    return rua
```

A probabilidade de cada uma das situações é de 10% e não pode haver ruas com trânsito e cortadas ao mesmo tempo.

4.2 Graph.py

A classe **Grafo** tem parecenças com o código fornecido pelos docentes, porém, como já é percetível, tem algumas diferenças que foram apontadas em cima.

```
def __init__(self, nodes=[], graph={}, edges=[]):
    self.m_nodes =nodes
    self.m_graph =graph
    self.m_edges =edges
    self.m_h ={}
```

Também foram elaborados novos métodos para suportar a manipulação e utilização destas estruturas de dados.

```
    def get_node_by_id(self, id :int) ->Node:
        """
        Get the node that has the identifier id
        :param id: The id of the node
        :return: Returns a Node object or None if there is no node with that id
        """
        for node in self.m_nodes:
            if(node.m_id ==id):
                return node
        return None
```

```
    def get_edge_by_nodes(self, origem :Node, destino :Node) ->Ruas | None:
        """
        Get the edge that is formed by the 2 input nodes
        :param origem: The node of origin
        :param destino: the node of destinations
        :return: The edge object that is formed by the 2 input nodes or None if there is not
                                                    ↪ such edge
        """
        for edge in self.m_edges:
            if edge.getOrigem() ==origem.getId() and edge.getDestino() ==destino.getId():
                return edge
        return None
```

5 Fase 1

O grupo sentiu a necessidade de dividir este projeto em duas fase sendo que esta primeira fase irá se focar na construção de uma boa base que possa suportar o desenvolvimento elaborado da aplicação *Health Planet*.

Nesta primeira fase é feita, essencialmente, a análise do desempenho dos algoritmos no contexto de procura que foi formulado. No final da fase serão tiradas as respetivas conclusões sobre os algoritmos a ser usados na **Fase 2**.

5.1 Estruturas de apoio

Para facilitar a gestão da grande quantidade de *nodes* e *edges* e as suas respetivas informações retornadas pelo pacote *OSMnx* foi necessária a criação de algumas classes e a alteração de outras.

```
class Node:
   def __init__(self, id=0, posicao=Positions(), street_count=0):
      self.m_id =id
      self.pos =posicao
      self.street_count =street_count
   ...
```

A classe `Node` representa as interseções entre as ruas, elas possuem um identificador providenciado pelo pacote *OSMnx*, uma instância da classe `Positions` que represeta as posições latitudinais e longitudinais do respetivo *node* e também um contador que quantas ruas se intersetam no nodo.

```
class Positions:
   def __init__(self, x=-1, y=-1):
      self.x =x
      self.y =y
```

```
class Ruas:
   def __init__(self, name="", origem=0, destino=0, oneway=False, highway="", rotunda=
                                                    ↪ False, ponte=False, tunnel=False, vel=[
                                                    ↪ ], len=0, ref=False, cortada=False,
                                                    ↪ transito=False):
      self.nome =name
      self.nodo_origem =origem
      self.nodo_destino =destino
      self.oneway =oneway
      self.highway =highway
      self.rotunda =rotunda
      self.ponte =ponte
      self.tunnel =tunnel
      self.vel_max =vel
      self.length =len
      self.cortada =cortada
      self.transito =transito
```

A representação das ruas em si é feita pela classe `Ruas` alusiva às *edges* que ligam os *nodes* no nosso multigrafo.

Atributo	Tipo	Descrição
nome	string	Nome da rua
nodo_origem	int	Identificador do *node* de origem
nodo_destino	int	Identificador do *node* de destino
oneway	bool	Se a rua é de sentido único ou não
highway	string	Tipo de via
rotunda	bool	Se é uma rotunda ou não
ponte	bool	Se é uma ponte ou não
tunnel	tunnler	Se é um tunel ou não
vel_max	float	Velocidade máxima numa determinada rua
length	float	Tamanho da rua
cortada	bool	Se a estrada está cortada ou não
transito	bool	Se a estrada está com trânsito ou não

Tabela 1. Atributos dos *edges*

5.2 Implementação e Análise dos Algoritmos de Procura Não Informada

Agora passamos a discutir e descrever a implementação dos algoritmos escolhidos pelo grupo para o nosso caso de estudo.

5.2.1 DFS - Depth First Search

```
def procura_DFS(self, start :Node, end :Node, path=[], visited=set(), lista_transito=[]
                                                ↪ , n_nos_explorados=0) ->tuple[list,
                                                ↪ float, int] | None:

    path.append(start)
    visited.add(start)

    if start ==end:
        custoT =self.calcula_custo(path, lista_transito)
        return (path, custoT, n_nos_explorados)

    if start.getId() in self.m_graph.keys():
        for(adjacente, peso, k) in self.m_graph[start.getId()]:
            nodo =self.get_node_by_id(adjacente)
            if nodo not in visited and not self.get_edge_by_nodes(start, nodo).isCortada()
                                                                ↪ :
                if self.get_edge_by_nodes(start, nodo).isTransito():
                    lista_transito.append(self.get_edge_by_nodes(start, nodo))
                resultado =self.procura_DFS(nodo, end, path, visited, lista_transito,
                                                            ↪ n_nos_explorados +1)
                if resultado is not None:
                    return resultado
    path.pop()

    return None
```

A busca em profundidade é uma abordagem sistemática de exploração de grafos que prioriza a exploração tão longe quanto possível antes de retroceder. Como o nosso grafo pode ter extensões relativamente grandes este

algoritmos pode e vai ter um desempenho baixo, ou seja, é lento a encontrar a solução, mas estes valores vão ser discutidos mais à frente.

A função começa inicializando uma lista chamada *path* para armazenar o caminho atual sendo explorado. O nó de início é adicionado a essa lista e marcado como visitado. A função, então, verifica se o nó de início é igual ao nó de destino. Se for, o caminho encontrado é retornado, junto com o custo total do caminho e o número total de nós explorados. Caso o nó de início não seja igual ao nó de destino, a função prossegue para explorar os nós adjacentes ao nó de início. Para cada nó adjacente, a função verifica se o nó já foi visitado e se a rua correspondente não está cortada. Se essas condições forem atendidas, a função é chamada recursivamente para explorar o nó adjacente.

Durante essa exploração, a função considera ruas com trânsito, adicionando-as a uma lista se estiverem presentes no caminho. Esse acompanhamento de arestas de trânsito é utilizado para alterar o custo da passagem caso o caminho passe numa rua com trânsito.

Se a chamada recursiva encontrar um caminho, o resultado é imediatamente retornado, interrompendo a exploração de outros ramos. Caso contrário, o nó atual é desmarcado como visitado, removido do caminho e a função retorna `None`.

Esse processo de exploração em profundidade continua até que todos os caminhos possíveis tenham sido considerados ou um caminho válido seja encontrado. A função retorna o caminho, o custo total e o número total de nós explorados, ou `None` se nenhum caminho for encontrado. Essa implementação reflete a natureza recursiva e sistemática do algoritmo de busca em profundidade.

5.2.2 BFS - Breadth First Search

```
def procura_BFS(self, start :Node, end :Node) ->tuple[list[Node | None], float, int]:
    lista_transito=[]
    n_nos_explorados=0
    visited =set()
    fila =Queue()

    fila.put(start)
    visited.add(start)

    parent =dict()
    parent[start] =None

    path_found =False
    while not fila.empty() and not path_found:
        nodo_atual =fila.get()
        if nodo_atual ==end:
            path_found =True

        elif nodo_atual.getId() in self.m_graph.keys():
            for (adjacente, peso, k) in self.m_graph[nodo_atual.getId()]:
                nodo =self.get_node_by_id(adjacente)
                if nodo not in visited and not self.get_edge_by_nodes(nodo_atual, nodo).
                                                                ↪ isCortada():
                    if self.get_edge_by_nodes(nodo_atual, nodo).isTransito():
                        lista_transito.append(self.get_edge_by_nodes(nodo_atual, nodo))
                    fila.put(nodo)
                    parent[nodo] =nodo_atual
                    visited.add(nodo)
                    n_nos_explorados +=1

    path =[]
    custo =0.0
    if path_found:
        path.append(end)
        while parent[end] is not None:
            path.append(parent[end])
            end =parent[end]
        path.reverse()
        custo =self.calcula_custo(path, lista_transito)
    return (path, custo, n_nos_explorados)
```

A busca em largura é um algoritmo que se baseia na exploração gradual dos nós vizinhos antes de avançar para os nós mais distantes. A implementação desta função segue a lógica clássica do BFS.

A função inicia marcando o nó de início como visitado e colocando-o em uma fila para processamento. Em seguida, um *loop* é iniciado para explorar os nós em largura. Durante o processo, a função explora os nós adjacentes ao nó atual, verificando se eles já foram visitados e se a rua correspondente não está cortada. Caso o nó adjacente atenda a essas condições, ele é adicionado à fila para exploração futura.

O algoritmo continua esse processo até que a fila esteja vazia ou que um caminho válido seja encontrado até o nó de destino. A função mantém um registo dos pais de cada nó visitado, permitindo a reconstrução do

caminho posteriormente assim como o registo de uma lista de ruas com trânsito que pode ser usada para calcular o custo correto de passar por ruas com grande afluência de trânsito.

O custo total do caminho é calculado considerando os custos associados ás arestas de trânsito. Por fim, a função retorna um tuplo contendo o caminho reconstruido, o custo total e o número total de nós explorados.

Esta implementação da busca em largura é eficiente e organizada, oferecendo uma abordagem sistemática para encontrar caminhos em um grafo, levando em consideração fatores como arestas de trânsito e custos associados.

5.2.3 Procura Bidirecional

```
def procura_bidirecional(self, start :Node, end :Node) ->tuple[list, float, int]:
    lista_transito=[]
    path_found =False
    n_nos_explorados=0

    forward_queue =Queue()
    forward_visited =set()
    forward_parent =dict()

    backward_queue =Queue()
    backward_visited =set()
    backward_parent =dict()

    forward_queue.put(start)
    backward_queue.put(end)

    forward_visited.add(start)
    backward_visited.add(end)

    forward_parent[start] =None
    backward_parent[end] =None

    meeting_point =None

    while (not forward_queue.empty() and not backward_queue.empty()) and not path_found:
        current_forward =forward_queue.get()
        current_backward =backward_queue.get()
        intersection =forward_visited.intersection(backward_visited)

        if intersection:
            path_found =True
            meeting_point =intersection.pop()

        if current_forward.getId() in self.m_graph.keys():
            for (neighbor_fwd, cost_fwd, k) in self.m_graph[current_forward.getId()]:
                node_fwd =self.get_node_by_id(neighbor_fwd)
                if node_fwd not in forward_visited and not self.get_edge_by_nodes(
                                                        ↪ current_forward, node_fwd).
                                                        ↪ isCortada():
                    edge_fwd =self.get_edge_by_nodes(current_forward, node_fwd)
                    if edge_fwd.isTransito():
                        lista_transito.append(edge_fwd)
                    forward_queue.put(node_fwd)
```

```
                forward_parent[node_fwd] =current_forward
                forward_visited.add(node_fwd)
                n_nos_explorados +=1

        if current_backward.getId() in self.m_graph.keys():
            for neighbor_bwd in self.get_predecessors(current_backward):
                node_bwd =self.get_node_by_id(neighbor_bwd)
                if node_bwd not in backward_visited and self.get_edge_by_nodes(node_bwd,
                                                        ↪ current_backward) is not
                                                        ↪ None and not self.
                                                        ↪ get_edge_by_nodes(node_bwd,
                                                        ↪  current_backward).
                                                        ↪ isCortada():
                    edge_bwd =self.get_edge_by_nodes(node_bwd, current_backward)
                    if edge_bwd.isTransito():
                        lista_transito.append(edge_bwd)
                    backward_queue.put(node_bwd)
                    backward_parent[node_bwd] =current_backward
                    backward_visited.add(node_bwd)
                    n_nos_explorados +=1

    path =self.reconstruct_path_bidirectional(path_found, meeting_point, forward_parent,
                                              ↪ backward_parent)
    costT =self.calcula_custo(path, lista_transito)
    return (path, costT, n_nos_explorados)
```

A busca bidirecional é uma abordagem que simultaneamente explora a partir do nó inicial e do nó final, encontrando um ponto de encontro no meio. A função segue esta lógica adaptada ao contexto do grafo e das restrições do problema.

A função inicia duas filas, uma para a procura no sentido direto (do nó de início para o ponto de encontro) e outra para a procura no sentido inverso (do nó de destino para o ponto de encontro). Além disso, mantém conjuntos de nós visitados (`forward_visited` e `backward_visited`) e dicionários de pais (`forward_parent` e `backward_parent`) para reconstrução do caminho.

O algoritmo continua a execução enquanto ambas as filas não estão vazias e um caminho válido ainda não foi encontrado. Durante cada iteração, são explorados os nós adjacentes aos atuais nos dois sentidos. Se há uma interseção nos conjuntos de nós visitados, um ponto de encontro foi encontrado, indicando que a busca bidirecional foi bem-sucedida. A função regista o ponto de encontro e ativa a *flag* `path_found`

Durante a exploração, a função verifica se os nós adjacentes não foram visitados e se as arestas correspondentes não estão cortadas. Os nós adjacentes válidos são adicionados ás filas para exploração futura, seus pais são registados, e os conjuntos de visitados são atualizados.

Após encontrar o ponto de encontro, a função reconstrói o caminho a partir dos pais nos dois sentidos, utilizando os dicionários `forward_parent` e `backward_parent`.

Finalmente, a função calcula o custo total do caminho considerando as arestas de trânsito encontradas durante a busca e retorna um tuplo contendo o caminho reconstruído, o custo total e o número total de nós explorados.

```
def get_predecessors(self, node :Node) ->list:
    """
    Method to get the predecessors of a certin node, it is needed to make the backwards
                                                    ↪ exploration in the bidirectional
                                                    ↪ search
    :param node: The current node that is going to be searched for adjacent ocurrences
                                                    ↪ in the graph
    :raturn: A list of the predecessor's node's names
    """
    predecessors =[]
    for (nodo, adj) in self.m_graph.items():
        for (neighbor, cost, k) in adj:
            if neighbor ==node.getId():
                predecessors.append(nodo)
    return predecessors
```

Para ser possível a exploração no sentido contrário (do fim para o inicio) é necessário, também, fazer uma exploração dos vizinhos ao contrário, ou seja, explorar os nodos cujo um dos adjacentes seja o nodo atual. Para este efeito foi feito o método `get_predecessors`

```
def reconstruct_path_bidirectional(self, path_found :bool, meet :Node, forward_dic :
                                                ↪ dict, backward_dic :dict) ->list[Node]
                                                ↪ | list:
    start_path=[]
    end_path=[]
    final_path =[]
    current =meet
    if path_found:
        start_path.append(current)
        while forward_dic[current] is not None:
            start_path.append(forward_dic[current])
            current =forward_dic[current]
        start_path.reverse()
        current =meet
        while backward_dic[current] is not None:
            end_path.append(backward_dic[current])
            current =backward_dic[current]

        final_path =start_path +end_path
    return final_path
```

Da mesma maneira, também é necessário fazer uma reconstrução do caminho obtido de maneira especial, uma vez que as ordens dos nodos dentro dos dicionários está trocada.

5.2.4 Custo Uniforme

```
def procura_custo_uniforme(self, start :Node, end :Node) ->tuple[list[Node | None],
                                                    ↪ float, int]:
    n_nos_explorados=0
    lista_transito=[]
    priority_queue =[(0, start)]
    visited =set()
    parents =dict()
    parents[start] =None
    path_found =False

    while priority_queue and not path_found:
        current_prio, current_node =heapq.heappop(priority_queue)

        if current_node ==end:
            path_found =True

        if current_node in visited:
            continue

        visited.add(current_node)

        if current_node.getId() in self.m_graph.keys():
            for (adj, cost, k) in self.m_graph[current_node.getId()]:
                prox_nodo =self.get_node_by_id(adj)
                if prox_nodo not in visited and not self.get_edge_by_nodes(current_node,
                                                                ↪ prox_nodo).isCortada():
                    edge =self.get_edge_by_nodes(current_node, prox_nodo)
                    if edge.isTransito():
                        lista_transito.append(edge)
                    parents[prox_nodo] =current_node
                    heapq.heappush(priority_queue, (current_prio+cost, prox_nodo))
                    n_nos_explorados +=1

    path=[]
    custoT =0.0
    if path_found:
        path.append(end)
        while parents[end] is not None:
            path.append(parents[end])
            end =parents[end]
        path.reverse()
        custoT =self.calcula_custo(path, lista_transito)
    return (path, custoT, n_nos_explorados)
```

A busca de custo uniforme é um algoritmo que prioriza a exploração dos nós com o menor custo acumulado até o momento. A implementação utiliza uma fila de prioridade (min-heap) para manter o controle dos movimentos com custos mais baixos.

A função inicia uma fila de prioridade com um tuplo contendo o custo acumulado até o nó atual e o próprio nó de início. Um conjunto de nós visitados (*visited*) é mantido, juntamente com um dicionário de pais (*parents*) para reconstrução do caminho. O algoritmo continua a execução enquanto a fila de prioridade não está vazia e um caminho válido ainda não foi encontrado. Durante cada iteração, o nó com o menor custo acumulado é

retirado da fila de prioridade. Se o nó atual é o nó de destino, a busca foi bem-sucedida, e a função ativa a *flag* `path_found`.

Durante a exploração, a função verifica se os nós adjacentes não foram visitados e se as arestas correspondentes não estão cortadas. Os nós adjacentes válidos são adicionados à fila de prioridade, seus pais são registados, e o conjunto de visitados é atualizado. Após encontrar o nó de destino, a função reconstrói o caminho a partir dos pais, utilizando o dicionário `parents`.

Finalmente, a função calcula o custo total do caminho considerando as arestas de trânsito encontradas durante a busca e retorna uma tuplo contendo o caminho reconstruído, o custo total e o número total de nós explorados.

5.2.5 Procura Iterativa - Aprofundamento Progressivo

```
def procura_iterativa(self, start :Node, end :Node) ->tuple[list, float, int]:
    for limit in range(1, sys.maxsize):
        result =self.procura_iterativa_ciclo(start, end, limit, [], set(), [], 0)
        if result is not None:
            resultado, lista, n_nos =result
            custoT =self.calcula_custo(resultado, lista)
            return (resultado, custoT, n_nos)


def procura_iterativa_ciclo(self, current :Node, end :Node, depth_limit :int, path :
                                                    ↪ list, visited :set, lista_transito :
                                                    ↪ list, n_nos_explorados :int) ->tuple[
                                                    ↪ list, list, int] | None:
    if current ==end:
        path.append(current)
        visited.add(current)
        return (path, lista_transito, n_nos_explorados)

    if depth_limit ==0:
        return None

    path.append(current)
    visited.add(current)

    if current.getId() in self.m_graph.keys():
        for (adjacente, peso, k) in self.m_graph[current.getId()]:
            nodo =self.get_node_by_id(adjacente)
            if nodo not in visited and not self.get_edge_by_nodes(current, nodo).isCortada
                                                                ↪ ():
                edge =self.get_edge_by_nodes(current, nodo)
                if edge.isTransito():
                    lista_transito.append(edge)
                resultado =self.procura_iterativa_ciclo(nodo, end, depth_limit -1, path,
                                                        ↪ visited, lista_transito,
                                                        ↪ n_nos_explorados+1)
                if resultado is not None:
                    return resultado

    path.pop()

    return None
```

A busca iterativa é uma estratégia que realiza buscas em profundidade com limites crescentes. A função `procura_iterativa` inicia um *loop* que aumenta progressivamente o limite de profundidade até que um caminho seja encontrado.

Dentro do *loop*, a função chama a função `procura_iterativa_ciclo`, que é basicamente uma cópia do algoritmo de busca em profundidade (**DFS**) com algumas verificações adicionais. Esta função recebe um nó atual, o nó de destino, o limite de profundidade atual, o caminho atual, um conjunto de nós visitados, uma lista de arestas de trânsito e o número total de nós explorados até o momento. Se o nó atual é o nó de destino, a função retorna o caminho encontrado, a lista de arestas de trânsito e o número total de nós explorados. Se o limite de profundidade atingiu zero, a função retorna *None*, indicando que o limite de profundidade foi alcançado sem encontrar um caminho.

Durante a exploração, a função verifica se o nó atual não foi visitado e se a aresta correspondente não está cortada. Os nós válidos são adicionados ao caminho, à lista de visitados e à lista de arestas de trânsito. Em seguida, a função faz uma chamada recursiva com o nó adjacente. Após encontrar um caminho, a função `procura_iterativa` calcula o custo total do caminho considerando as arestas de trânsito encontradas durante a busca e retorna um tuplo contendo o caminho reconstruído, o custo total e o número total de nós explorados.

5.3 Implementação e Análise dos Algoritmos de Procura Informada

Neste secção vamos falar do processo de decisão por de trás da definição da nossa heurística e também da implementação dos algoritmos de procura informada.

5.3.1 Heurísticas

Inicialmente, foi ponderado pelo grupo usar uma heurística que tivesse em conta apenas a distância direta (distância de *manhattan*), porém, esta solução apresenta os seus problemas, como por exemplo, duas interseções podem estar muito próximas uma da outra mas o caminho real que se tem de fazer com um veículo pode obrigar a dar uma volta muito grande, havendo assim uma discrepância muito grande entre o que vemos de "cima"e a realidade.

Por isso, no final, o grupo decidiu utilizar um método mais minucioso que permitiria refletir a realidade. Foi usado um método parecido a de uma procura em largura.

```
def calcula_heuristica_global(self, destino :Node) ->None:
    dicBike =self.heurisitcas_by_vehicle(destino, 10)
    dicMoto =self.heurisitcas_by_vehicle(destino, 35)
    dicCar =self.heurisitcas_by_vehicle(destino, 50)

    for nodo, heuristica in dicBike.items():
        self.m_h[nodo] =(dicBike[nodo], dicMoto[nodo], dicCar[nodo])

def heurisitcas_by_vehicle(self, destino :Node, vel :int):
    dic ={}
    n1 =destino
    heuristica =0
    aux =destino.getId()
    dic[aux] =heuristica

    queue =[aux]

    while queue:
        current =queue.pop(0)
        if current in self.m_graph.keys():
            for (adj, custo, _) in self.m_graph[current]:
                if adj not in dic.keys():
                    n2 =adj
                    dic[adj] =dic[current] +self.calculate_time(custo, vel)
                    queue.append(adj)
        else:
            dic[current] =float('inf')

    return dic

def calculate_time(self, length_in_meters, speed_kmh):
    speed_ms =speed_kmh *1000 /3600
    time_seconds =length_in_meters /speed_ms

    return round(time_seconds, 2)
```

Começando no nodo destino, exploramos os adjacentes e adicionamos ao dicionário de heurísticas os nodos como chave e como valor o tempo que demoraria de ir do destino para esse nodo. Sendo assim, é necessário de fazer um dicionário para cada tipo de veículo já que cada um tem uma velocidade média diferente.

5.3.2 Greedy

```
def greedy(self, start, end):
    open_list ={start}
    closed_list =set([])
    transito_list=[]
    n_nos_explorados=0
    parents ={}
    parents[start] =start
    while len(open_list) >0:
        n =None
        for v in open_list:
            if n ==None or self.m_h[v.getId()] <self.m_h[n.getId()]:
                n =v

        if n ==None:
            print('Path does not exist!')
            return None

        if n ==end:
            reconst_path =[]

            while parents[n] !=n:
                reconst_path.append(n)
                n =parents[n]

            reconst_path.append(start)
            reconst_path.reverse()
            return (reconst_path, self.calcula_custo(reconst_path, transito_list),
                                                    ↪ n_nos_explorados)

        if self.getH(n.getId()) !=float('inf'):
            for (adj, weight) in self.getNeighbours(n.getId()):
                m =self.get_node_by_id(adj)
                if m not in open_list and m not in closed_list:
                    edge =self.get_edge_by_nodes(n,m)
                    if edge.isTransito():
                        transito_list.append(edge)
                    open_list.add(m)
                    parents[m] =n
                    n_nos_explorados +=1

            open_list.remove(n)
            closed_list.add(n)

    print('Path does not exist!')
    return None
```

A função `greedy` implementa o algoritmo *Greedy*, uma procura informada que seleciona o nó com a menor heurística em cada passo, sem considerar o custo acumulado até o momento.

A função inicia uma lista `open_list` contendo o nó de início e um conjunto `closed_list` vazio. Além disso, mantém um conjunto de arestas de trânsito (`transito_list`), um dicionário `parents` que regista os pais de cada nó e uma variável `n_nos_explorados` que conta o número total de nós explorados.

O *loop* principal continua enquanto a `open_list` não está vazia. A cada iteração, o nó com a menor heurística é selecionado da `open_list` para ser explorado. Se o nó atual é o nó de destino, o algoritmo reconstrói o caminho a partir dos pais e retorna um tuplo contendo o caminho reconstruído, o custo total e o número total de nós explorados. Para cada vizinho do nó atual, o algoritmo verifica se o vizinho não está na `open_list` nem na `closed_list`. Se não estiver, o vizinho é adicionado à `open_list`, e seus pais e arestas de trânsito são atualizados.

Se o vizinho já está na `open_list` ou na `closed_list`, o algoritmo verifica se é mais eficiente alcançar o vizinho através do nó atual. Se sim, os pais e as arestas de trânsito são atualizados. Após explorar todos os vizinhos do nó atual, este é removido da `open_list` e adicionado à `closed_list`. O algoritmo continua até que a `open_list` esteja vazia ou o nó de destino seja encontrado.

5.3.3 A estrela

```
def procura_aStar(self, start, end):
    startId =start.getId()
    endId =end.getId()
    open_list ={startId}
    closed_list =set([])
    transito_list=[]
    n_nos_explorados=0
    g ={}
    g[startId] =0

    parents ={}
    parents[startId] =startId
    n =None
    while len(open_list) >0:
        calc_heurist ={}
        flag =0
        for v in open_list:
            if n ==None:
                n =v
            else:
                flag =1
                calc_heurist[v] =g[v] +self.getH(v)
        if flag ==1:
            min_estima =self.calcula_est(calc_heurist)
            n =min_estima
        if n ==None:
            print('Path does not exist!')
            return None

        if n ==endId:
            reconst_path =[]

            while parents[n] !=n:
```

```
            reconst_path.append(self.get_node_by_id(n))
            n =parents[n]

        reconst_path.append(start)

        reconst_path.reverse()

        return (reconst_path, self.calcula_custo(reconst_path, transito_list),
                                                    ↪ n_nos_explorados)

    if self.getH(n) !=float('inf'):
        for (m, weight) in self.getNeighbours(n):

            if m not in open_list and m not in closed_list:
                node =self.get_node_by_id(n)
                adj =self.get_node_by_id(m)
                if self.get_edge_by_nodes(node, adj).isTransito():
                    transito_list.append(self.get_edge_by_nodes(node, adj))
                open_list.add(m)
                parents[m] =n
                g[m] =g[n] +weight
                n_nos_explorados +=1

            else:
                if g[m] >g[n] +weight:
                    g[m] =g[n] +weight
                    parents[m] =n

                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None
```

A função `procura_aStar` implementa o algoritmo A*, que é uma busca informada que utiliza uma combinação de custo atual (g) e uma heurística (h) para decidir a ordem de exploração dos nós.

A função inicia uma lista `open_list` contendo o nó de início e um conjunto *closed_list* vazio. Além disso, mantém um conjunto de arestas de trânsito (`transito_list`), um dicionário g que armazena os custos acumulados até cada nó, um dicionário `parents` que regista os pais de cada nó e uma variável que conta o número total de nós explorados.

O *loop* principal continua enquanto a `open_list` não está vazia. A cada iteração, o nó com o menor valor da função de avaliação f() (soma do custo atual g() e da heurística h()) é retirado da `open_list` para ser explorado. Se o nó atual é o nó de destino, o algoritmo reconstrói o caminho a partir dos pais e retorna um tuplo contendo o caminho reconstruído, o custo total e o número total de nós explorados.

Para cada vizinho do nó atual, o algoritmo verifica se este não está na `open_list` nem na `closed_list`. Se não estiver, o vizinho é adicionado à `open_list`, e seus pais, custos acumulados e arestas de trânsito são

atualizados. Se o vizinho já está na `open_list` ou na `closed_list`, o algoritmo verifica se é mais eficiente alcançar o vizinho através do nó atual. Se sim, os pais, custos acumulados e arestas de trânsito são atualizados.

Após explorar todos os vizinhos do nó atual, este é removido da `open_list` e adicionado à `closed_list`. O algoritmo continua até que a `open_list` esteja vazia ou o nó de destino seja encontrado.

5.4 Desempenho dos Algoritmos e Conclusões

A variável `n_nos_explorados` devolvida por cada um dos algoritmos é usada, em conjunto com outras estatísticas, para medir o desempenho de cada um destes. Os parâmetros escolhidos para este medição são:

- **Tempo de Execução**
- **Número de Chamadas de Funções Feitas**
- **Tamanho do Caminho Encontrado**
- **Custo da Solução**
- **Número de Nós Explorados**

Para tal foi usada a biblioteca **cProfile** para realizar uma medida do tempo de execução e de chamadas de funções feitas. Também foi necessário a utilização da biblioteca **pstats** para extrair a informação necessária dos *profilers*.

Para exemplificar, usaremos um exemplo de procura entre a **Rua de Souselas** e **Rua Bernardo Sequeira** da cidade de Braga, Portugal.

Algoritmo	Tempo	Nº de Chamadas	Tamanho Path	Custo Solução	Número de Nós Explorados
`DFS`	65.073	309430722	335	26723.731	334
`BFS`	9.675	47454359	34	3451.331	1510
`Bidirecional`	6.876	28954277	34	3451.331	699
`Custo Uniforme`	5.041	25196416	39	3054.917	983
`Procura Iterativa`	125.106	567491797	55	6487.604	54
`Greedy`	1.343	5593677	55	4769.021	153
`A*`	3.844	16276443	38	2886.855	408

Tabela 2. Desempenho dos algoritmos

Com estes resultados podemos concluir que o algoritmo mais rápido é o `Greedy` mas o que consegue a melhor solução é o `A*`. Sendo assim, na Fase 2, iremos apenas utilizar o `A*` uma vez que é do interesse da empresa **Health Planet** conseguir o caminho menos poluente, ou com menor custo.

6 Fase 2

A segunda fase reflete o desenvolvimento da aplicação **Health Planet**, a implementação do sistema de encomendas e entregas, Permitindo a um cliente criar e consultar encomendas de vários produtos e fazer avaliação da entrega. É possível simular a entrega das encomendas pedidas sendo possível forçar comportamentos dinâmico por parte dos estafetas.

6.1 Estruturas de apoio

```
class Client:
   current_id =0
   def __init__(self, name="", password="", history=[]):
      Client.current_id +=1
      self.id =Client.current_id
      self.name =name
      self.order_history =history
      self.password =password
```

O cliente é representado pelo seu respetivo identificador, nome, palavra-passe e histórico de entregas.

```
class Delivery:
   current_id =0
   def __init__(self, produtos=[], worker=worker(), price=0.0, vehicle=Vehicle(), rating=0
                                                    ↪ .0):
      Delivery.current_id +=1
      self.id =Delivery.current_id
      self.list_products =produtos
      self.worker =worker
      self.price =price
      self.vehicle =vehicle
      self.rating =rating
```

As entregas são definidas por uma lista de produtos, um estafeta, um preço, um veículo e uma avaliação feita pelo cleinte na finalização da mesma.

```
class Order:
   current_id =0
   def __init__(self, weight=0.0, localizacao="", goods=[], origem="", destino="", client=
                                                    ↪ Client()):
      Order.current_id +=1
      self.id =Order.current_id
      self.weight =weight
      self.localizacao =localizacao
      self.goods =goods
      self.start_street =origem
      self.delivery_street =destino
      self.delivery_time =time
      self.client =client
      self.entregue =False
```

A encomenda efetuada pelo cliente é definida pelo seu peso total, a cidade em que vai ser entregue, a lista de produtos que vão ser encomendados, a origem e o destino, o próprio cliente e se foram entregues ou não.

```
class Produto:
   def __init__(self, nome="", peso=0.0, preco=0.0):
      self.name =nome
      self.peso =peso
      self.preco =preco
      self.entregue =False
```

Os produtos em si apenas contêm a informação relativa ao nome, peso, preço e se ja foram entregues ou não

```
class Vehicle():
   current_id=0
   def __init__(self):
      Vehicle.current_id +=1
      self.id =Vehicle.current_id

class Bike(Vehicle):
   def __init__(self):
      Vehicle.__init__(self)
      self.max_weight =5
      self.velocity =10
      self.polution =0

class Motorcycle(Vehicle):
   def __init__(self):
      Vehicle.__init__(self)
      self.max_weight =20
      self.velocity =35
      self.polution =0.13

class Car(Vehicle):
   def __init__(self):
      Vehicle.__init__(self)
      self.max_weight =100
      self.velocity =50
      self.polution =0.37
```

Os veículos estão definidos como sub-classes, mas todos têm os mesmos atributos, um peso máximo, uma velocidade média e um nível de poluição.

```
class worker:
   current_id =0
   def __init__(self, name=""):
      worker.current_id +=1
      self.id =worker.current_id
      self.name =name
      self.ocupados =False
      self.average_rank =0.0
      self.num_deliveries =0
```

Para a representação do estafeta foi feita a classe *worker* que contém o identificador deste, o seu nome, se está neste momento ocupado a fazer alguma entrega, a média das avaliações recebidas e o numero de entregas feitas.

```
class HealthPlanet:
   def __init__(self, users=dict(), encomendas=dict(), workers=[], entrega=dict(), grafos=
                                                    ↪ dict(), estafetas_por_localizacao=dict
                                                    ↪ ()):
      self.users =users
      self.encomendas =encomendas
      self.workers =workers
      self.entrega =entrega
      self.grafos ={}
      self.rating_needed =[]
      self.estafetas_por_localizacao =estafetas_por_localizacao
```

Finalmente, é necessária a definição da classe que irá tratar a lógica de negócio. A classe *Health Planet* guarda em si a informação toda relevante para o funcionamento do sistema. Guarda os clientes, as encomendas, os estafetas, as entregas, os grafos de cada cidade, as entregas que necessitam de ser avaliadas e também os estafetas de cada cidade.

6.2 Interface

Primeiro vamos discutir a elaboração do interface do utilizador, este foi feito no terminar e possuí algumas funcionalidades com valor de mencionar.

O processo de pedir ao cliente destino da encomenda, sendo que a origem é sempre o armazém da empresa **Health Planet**, é o seguinte:

- O cliente indica uma rua
- O cliente deve indicar outra rua que faça interseção com a primeira de maneira a que este interseção seja o mais próxima do destino pertendido

```
def seleciona_origem_destino(graph):
      testD1, testD2 =str(), str()

   print("[SYS] Os pontos de origem e destino sao nodos que sao representados pela
                                                    ↪ intersecao de duas ou mais ruas")
   print("[SYS] O processo de selecao do destino e o seguinte: ")
   print("[SYS] -> Indicar uma rua")
   print("[SYS] -> Indicar outra rua que faca intersecao com a original")

   while isinstance(testD1, str):
      ruaDestino1 =input("[SYS] Indique a rua de destino principal: ")
      testD1 =graph.get_edge_by_name(ruaDestino1)
      print(testD1)


   while isinstance(testD2, str):
      ruaDestino2 =input("[SYS] Indique a rua que faca interceceo com a original de
                                                    ↪ destino: ")
```

```
        testD2 =graph.get_edge_by_name(ruaDestino2)
        print(testD2)

    sleep(2)
    intersectionDetino =graph.get_intersection_node(testD1, testD2)
    os.system('cls')
    start =graph.get_node_by_id(intersectionOrigem)
    end =graph.get_node_by_id(intersectionDetino)
    return start, end
```

Note que esta função está definida de maneira diferente na fase 1, uma vez que nessa já é pertinente escolher a origem.

Esta função faz uso dos método `get_edge_by_name` que realiza uma funcionalidade interessante.

```
    def get_edge_by_name(self, street :str) ->Ruas | str:
        exact =self.get_edge_by_name_exact(street)
        if len(exact) >0:
            return exact

        suggestion =self.get_edge_by_name_suggestion(street)
        return f"Rua nao encontrada. \n Quereria dizer: {', '.join(suggestion)}"
```

```
    def get_edge_by_name_suggestion(self, street :str, threshold=80) ->list[str]:
        list_of_names=[]
        for edge in self.m_edges:
            if isinstance(edge.getName(), list):
                for names in edge.getName():
                    list_of_names.append(names)
            else: list_of_names.append(edge.getName())

        suggestions =process.extract(street, list_of_names, limit=10)
        filtered_suggestions =[suggestion[0] for suggestion in suggestions if suggestion[1]
                                                    ↪ >=threshold]

        return filtered_suggestions
```

É utilizada a biblioteca **fuzzywuzzy** que utiliza um algoritmo baseado na distância de *Levenshtein* para saber os nomes de ruas que são mais parecidos com o que o utilizador escreveu. Isto executado apenas se o utilizador não escrever o nome correto duma rua.

O resto to interface é relativamente básico, uma sequência de impressões no terminal num formato pré-definido onde se pode consultar e manipular a informação do sistema.

6.3 Lógica de Negócio

Na classe principal do sistema (`Health Plannet` temos vários métodos para a manipulação das estruturas de dados. Estas são imperativas para permitir à introdução de nova informação originada pelo cliente.

```
def adicionar_cliente(self, username :str, password :str):
    if username not in self.users:
        novo_cliente =Client(username, password)
        self.users[username] =novo_cliente
        print(f"\n{username} adicionado com sucesso!")
    else:
        print("\n Erro: Este nome de Cliente ja esta em uso.")
```

```
def adicionar_encomenda(self, client :Client,localizacao :str, weight :float, goods :
                                                    ↪ list, origem :str, destino :str, time :
                                                    ↪ str) ->None:
    lista_partida=[]
    if weight >100:
        soma=0
        lista_prod=[]
        for pod in goods:
            if soma+pod.getPeso() >=100:
                lista_partida.append(lista_prod)
                lista_prod=[pod]
                soma =pod.getPeso()
            else:
                lista_prod.append(pod)
                soma +=pod.getPeso()
        if lista_prod:
            lista_partida.append(lista_prod)
    else:
        new_order =Order(weight,localizacao, goods, origem, destino, time, client)
        #if new_order not in self.encomendas.values():
        self.encomendas[new_order.getId()] =new_order
        print(f"\nEncomenda {new_order.getId()} adicionada com sucesso!")
        # else:
        #    print("\nErro ao adicionar a encomenda")

    if len(lista_partida) >0:
        for enc in lista_partida:
            new_order =Order(weight, enc, origem, destino, time, client)
            if new_order not in self.encomendas.values():
                self.encomendas[new_order.getId()] =new_order
                print(f"\nEncomenda {new_order.getId()} adicionada com sucesso!")
            else:
                print("\nErro ao adicionar a encomenda")
```

```
def adicionar_workers(self, nome :str) ->None:
    if nome not in self.workers:
        new_worker =worker(nome)
        self.workers.append(new_worker)
        print(f"\nEstafeta {new_worker.getId()} adicionado com sucesso!")
    else:
        print("\nJa existe um worker com esse nome!")
```

```
def adicionar_entrega(self, produtos :list[Produto], vehicle :Vehicles) ->None:
    estafetas =self.get_estafetas_livres()
    if len(estafetas)>0:
        preco =self.calcula_preco(produtos)
        new_delivery =Delivery(produtos, estafetas[0], preco)
        print()
        if new_delivery not in self.entrega.values():
            self.entrega[new_delivery.getId()] =new_delivery
            print(f"\nEntrega {new_delivery.getId()} adicionada com sucesso!")
        else:
            print("\nErro ao adicionar a entrega")
    else:
        print("[SYS] Nao exitem estafetas disponiveis!")
```

```
def adicionar_grafo(self,nome :str,grafo :Grafo,grafob :Grafo) ->None:
    if nome not in self.grafos:
        self.grafos[nome] =(grafo,grafob)
        print(f"\nGrafo de {nome} adicionado com sucesso!")
    else:
        print("\nJa existe um grafo dessa cidade!")
```

Além destes os outros métodos são úteis para tirar a informação possível mostrar ao utilizador.

6.4 Algortimo de Distribuição de Entregas e Simulador

Por último, construimos o nosso algoritmo de decisão de Entregas. Restringimos o maximo de Entregas a serem geridas de uma so vez a 5 entregas por localização e aplicamos um algoritmo que, apesar de não abringir todos os casos possiveis, ja abrange uma grande parte deles como por exemplo verificar se não se poupa em termos de poluição se um estafeta quando sai do armazem levar produtos para mais que uma entrega, evitando, assim, ter de percorrer mais caminho e por sua vez ter um maior consumo a todos os niveis. Para desenvolver este algortimo guardamos os trajetos que cada entregador terá de fazer assim que o sistema decida processar as entregas pendentes (Que acaba por ser a simulação de um dia de trabalho para a nossa empresa de entregas). Consequentemente, desenvolvemos uma simulação para representar os nossos entregadores a Realizar as suas Entregas. Para isso utilizámos a livraria *tkinter*, pois as suas funcionalidades gráficas são muito extensas e poderosas. Começamos então por criar uma nova classe, a classe *VehicleSimulation*, que recebe como argumentos uma lista de Ruas e o veículo com o qual o Entregador vai realizar as Encomendas.

```
class VehicleSimulation:
def __init__(self, root, nodos, caminhos, vehicle_image_path):
    self.root =root
```

```
        self.canvas =tk.Canvas(root, width=4000, height=200)
        self.canvas.pack(side=tk.TOP, fill=tk.BOTH, expand=True)

        self.nodos =nodos
        self.caminhos =caminhos

        # Draw nodes and streets
        self.draw_nodes_and_streets()

        # Car initial position
        vehicle_size =10
        self.vehicle_image =tk.PhotoImage(file=vehicle_image_path).subsample(int(100/
                                                    ↪ vehicle_size), int(100/vehicle_size)
                                                    ↪ )
        self.car =self.canvas.create_image(0, 0, anchor=tk.CENTER, image=self.vehicle_image)

        self.current_path =self.create_path()
        self.current_path_iter =iter(self.current_path)

        # Add scrollbar
        self.scrollbar =ttk.Scrollbar(root, orient=tk.HORIZONTAL, command=self.canvas.xview)
        self.scrollbar.pack(side=tk.BOTTOM, fill=tk.X)
        self.canvas.configure(xscrollcommand=self.scrollbar.set)

        self.root.after(1000, self.move_veicule) # Start moving after 1 second
```

Esta classe implementa métodos a nível gráfico que permitem visualizarmos os nossos veículos a moverem-se, simulando o Entregador a fazer uma Entrega. Gostaríamos de sublinhar o método *animate smoothly* que faz com que o veículo que está na simulação, percorra a mesma de forma uniforme e controlada, otimizando a visualização da simulação e, concomitantemente, permitindo que seja possível realizar ações de forma a conseguirmos controlar alterações no mapa como estradas cortadas.

```
def animate_smoothly(self, x_start, y_start, x_end, y_end, step):
if step <=1:
    x_car =x_start +(x_end -x_start) *step
    y_car =y_start +(y_end -y_start) *step
    self.canvas.coords(self.car, x_car, y_car)

    self.root.after(10, self.animate_smoothly, x_start, y_start, x_end, y_end, step +0.
                                                ↪ 005)
else:
    try:
        next_edge =next(self.current_path_iter)
        self.move_vehicle(next_edge)
    except StopIteration:
        print("Simulation completed!")
```

Para fazermos essas alterações criamos uma *Thread* que tem como *target* a função *thread input nodo cortado*, que tem como função cortar nodos por onde o Entregador iria passar e por isso correr novamente o algoritmo e determinar a nova melhor solução de atingir o destino. Os nodos são cortados pelo sistema e o Entregador assim

que passa por eles apercebe-se que a próxima rua está cortada e comunica à Central pedindo caminho alternativo.

```
def thread_input_nodo_cortado(self,localizacao):
  while True:
     try:
        grafo =int(input("[SYS] Indique qual dos grafos pretende cortar: 1:
                                                    ↪ GrafoBicicleta, 2: GrafoMota, 3:
                                                    ↪ GrafoCarro, 4: Exit Selecione uma
                                                    ↪  opcao: "))
        if grafo==4:
           print(f"[SYS] Processo de cortar ruas terminado para a {localizacao}")
           break
        grafoAtual, grafoAtualb =self.grafos(localizacao)
        if grafo==2 or grafo ==3:
           seleciona_rua_cortar(grafoAtual)
        else:
           seleciona_rua_cortar(grafoAtualb)
     except ValueError:
        print("Entrada invalida. Por favor, digite um numero inteiro.")
```

Para fazermos a Simulção usamos 3 imagens *.png* para representar o carro, a mota e a bicicleta:

Figura 2. Icon para representar o carro

Figura 3. Icon para representar a mota

Figura 4. Icon para representar a bicicleta

Deixamos também aqui uma amostra da Simulação de um Entregador que fazia a Entrega de carro entre dois nodos do nosso mapa da cidade de Braga:

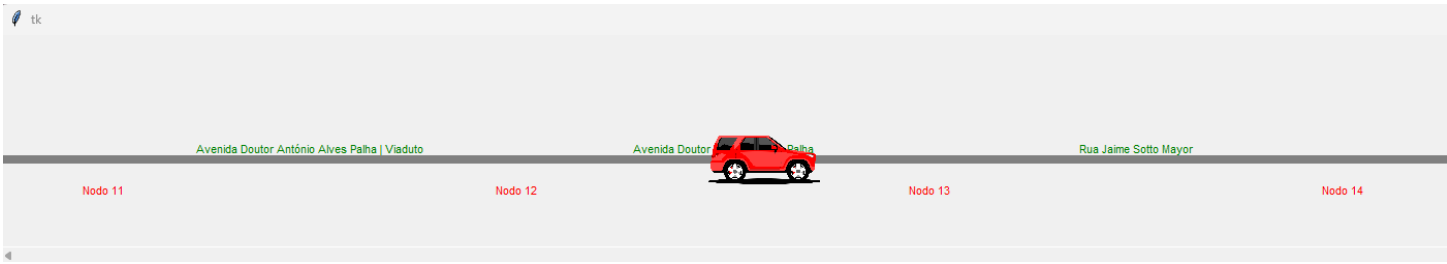


Figura 5. Simulação Carro

7 Anotações e Limitações

Este projeto apresentou-se ser um grande desafio para o grupo, uma vez que a cada sessão de trabalho as ideias iam crescendo e amontando-se, isto é mais evidente na segunda fase do trabalho. Nesta o grupo tinha várias propostas que podiam ter sido implementadas mas não viram a luz do dia devido à escassez do tempo. Entre estas está, por exemplo, uma implementação distribuída, assim como aperfeiçoar da parte da aplicação em si.

Outra parte que se apresentou como um grande obstáculo foi a implementação da simulação das entregas na aplicação, podemos dizer com vontade que este projeto poderia ser subdividido em 3 outros. Uma implementação de algoritmos de procura, uma aplicação orientada a objetos com a capacidade de atender a uma necessidade distribuída e também um simulador de todo o trabalho anterior.

8 Conclusões e Resultados Obtidos

O desenvolvimento deste projeto no âmbito da disciplina de Inteligência Artificial ofereceu uma perspetiva abrangente sobre a implementação e análise de algoritmos de busca. A exploração prática de diferentes abordagens, incluindo *DFS*, *BFS*, *Busca Bidirecional*, *Custo Uniforme*, *A** e *Greedy*, revelou nuances significativas em relação à complexidade computacional e eficiência.

A adaptação desses algoritmos a um contexto específico, no caso, a modelagem de um sistema de entrega sustentável, enfatizou a importância de considerar as características do domínio em questão. A escolha e implementação de heurísticas apropriadas, especialmente para algoritmos informados como *A** e *Greedy*, desempenharam um papel crucial na eficácia dessas abordagens.

A implementação prática proporcionou uma compreensão mais profunda dos conceitos teóricos. Desafios reais, como a representação do grafo, manipulação de dados geoespaciais e considerações ambientais, acrescentaram complexidade e relevância ao projeto. A exploração de ferramentas, como **OSMnx**, demonstrou a importância de aproveitar recursos existentes para otimizar o desenvolvimento e análise de sistemas complexos. A combinação de dados geográficos e algoritmos de busca permitiu soluções mais robustas e contextualmente relevantes.

A avaliação de desempenho, utilizando métricas como o número de nós explorados, o custo total e o tempo de execução, forneceu *insights* valiosos sobre a eficácia prática dos algoritmos implementados. A comparação entre diferentes abordagens enriqueceu a compreensão dos pontos fortes e limitações de cada uma.

Em resumo, este projeto não apenas aprofundou o entendimento teórico dos algoritmos de busca, mas também destacou a importância de sua aplicação em cenários do mundo real. A interseção entre teoria e prática proporcionou uma experiência de aprendizado valiosa, preparando para futuras explorações no vasto campo da inteligência artificial.

9 Bibliotecas Utilizadas

- **osmnx** - Biblioteca para extrair os mapas do Open Street Maps em forma de um grafo da biblioteca Networkx
- **networkx** - Biblioteca com a definição do multigrafo
- **prettytable** - Biblioteca para representar dicionários em tabelas legíveis
- **cProfile** - Biblioteca utilizada para medir o desempenho dos algoritmos
- **pstats** - Biblioteca para extrair as estatísticas do cProfile
- **fuzzywuzzy** - Biblioteca utilizada para sugerir palavras parecidas ao utilizador
- **python-Levenshtein** - Biblioteca que permite a utilização do algoritmos Levenshtein para saber o nível de parecenças entre duas palavras
- **matplotlib** - Biblioteca para representar graficamente o grafo

Referências

1. Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, 2017.
2. Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.