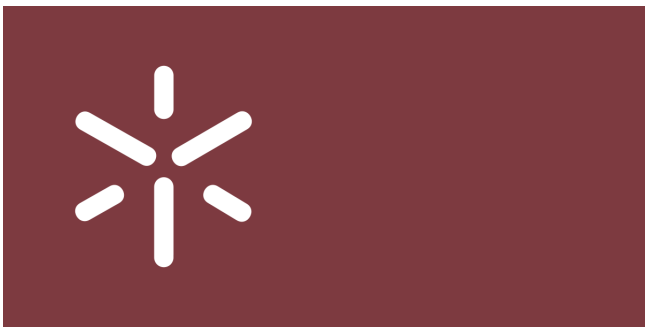

Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Projeto da U.C. Inteligência Artificial

Ano letivo 2023/2024

Realizado por:

A91672 - Luís Ferreira = 0
A93258 - Bernardo Lima = 0
A100543 - João Pastore = 0
A100554 - David Teixeira = 0

Conteúdo

1 Introdução e Descrição do problema **1**

2 Formulação do Problema **1**
2.1 Representação do Estado Inicial 1
2.2 Estado/Teste Objetivo 1
2.3 Operadores 1
2.4 Custo da Solução 1

3 Tarefas Realizadas e Decisões do Grupo **2**
3.1 Gerar Circuitos de Entrega 2
3.2 Representação dos Pontos de Entrega em Grafo 2
3.3 Estratégias de Procura Utilizadas 3
3.3.1 Procura Informada 3
3.3.2 Procura não Informada 4
3.4 Comparação de Resultados dos Algoritmos de Procura 4
3.5 Justificação das Heurísticas nos Algoritmos de Procura Informada 5
3.6 Desenvolvimento do Trabalho 6
3.6.1 Classe *main* 6
3.6.2 Classe *menu_functions* 7
3.6.3 Classe *order_functions* 8
3.6.4 Classe *CourierFunctions* 9
3.6.5 Classe *Order* 9
3.6.6 Classe *Courier* 9
3.6.7 Classe *Test* 11
3.6.8 Classes *Node*, *Graph* e *GuimaraesStreetGraphGenerator* 12

4 Sumário e Discussão dos Resultados obtidos **12**
4.1 Considerações Especiais e Limitações 13

5 Conclusão **13**

1 Introdução e Descrição do problema

Este trabalho prático aborda o desenvolvimento de algoritmos de procura para otimizar a distribuição de encomendas da empresa *Health Planet*, visando a sustentabilidade. Cada estafeta tem entregas em ruas associadas a freguesias, com clientes que definem prazos de entrega e atribuem *rankings*. As encomendas têm peso, volume e devem ser transportadas através de um meio de transporte sustentável. Restrições incluem capacidade e velocidade para bicicletas, motos e carros. O objetivo é minimizar custos, cumprir prazos e promover práticas sustentáveis.

Iniciamos o processo focando-nos na procura por soluções eficientes para a distribuição de encomendas num mapa fictício de Guimarães, utilizando diversos algoritmos de procura num grafo representativo. O objetivo primordial é minimizar o custo, considerando a distância como parâmetro. Em seguida, começamos por organizar todas as encomendas que foram registadas para facilitar a atribuição das mesmas aos estafetas no início de cada dia. Depois, priorizamos a escolha de estafetas cujos veículos possuam menores emissões de carbono, sempre que seja algo viável para a realização das entregas. Somente em casos impraticáveis, direcionamos as encomendas para veículos com maior impacto ambiental. Estas medidas visam otimizar a eficiência logística, minimizar a pegada de carbono e promover práticas sustentáveis na distribuição de encomendas.

2 Formulação do Problema

2.1 Representação do Estado Inicial

- Grafo com vários nodos diferentes, cada um associado a diferentes tipos de custos, refletindo a topologia fictícia de Guimarães.
- Ausência de encomendas a serem entregues.
- Ausência de clientes atendidos.
- Ausência de veículos designados para entregas.

2.2 Estado/Teste Objetivo

- Ordens entregues nos remetentes constatados na sua descrição;
- Ordens entregues no tempo indicado pelo cliente (ou mesmo antes);
- Lista com encomendas entregues;
- Veículos existentes para entregas;

2.3 Operadores

Os operadores consistem nas ações realizadas para transitar entre estados no problema de distribuição de encomendas em Guimarães. Exemplificando, podem incluir atribuição de encomendas a veículos, atualização do estado do grafo para refletir a entrega bem-sucedida e ajuste dos horários de entrega conforme necessário.

2.4 Custo da Solução

O custo da solução é calculado com base na distância percorrida. A minimização desse custo é o objetivo principal, visando eficiência logística, cumprimento de prazos e práticas sustentáveis.

3 Tarefas Realizadas e Decisões do Grupo

3.1 Gerar Circuitos de Entrega

Nesta etapa, abordaremos a criação dos circuitos de entrega para o problema em questão. O código *Python* abaixo utiliza uma representação fictícia de um grafo de ruas de Guimarães e configura conexões entre diferentes localidades, simulando a estrutura da cidade. São utilizadas, também, as classes *Node* e *Grafo*, disponibilizadas nas aulas TP.

```
from graph.Graph import Grafo

class GuimaraesStreetGraphGenerator:
    def generate_graph(self):
        guimaraes_graph = Grafo()

        # Edges within Freguesia 1
        guimaraes_graph.add_edge("Alameda Dom Afonso Henriques", "Avenida
                                                Conde Margaride", 2)
        guimaraes_graph.add_edge("Alameda Dom Afonso Henriques", "Rua de
                                                Santo Antonio", 3)
        # ...

        # Edges within Freguesia 2
        guimaraes_graph.add_edge("Largo da Mumadona", "Rua de Santa Maria",
                                                3)
        # ...

        # Edges within Freguesia 3
        guimaraes_graph.add_edge("Rua da Penha", "Avenida Alberto Sampaio",
                                                4)
        # ...

        # Edges within Freguesia 4
        guimaraes_graph.add_edge("Avenida D X", "Rua da Abadia", 6)
        # ...

        # Edges within Freguesia 5
        guimaraes_graph.add_edge("Largo da Feira", "Rua de Fernao Mendes
                                                Pinto", 3)
        # ...

        # Connections between freguesias
        guimaraes_graph.add_edge("Largo do Trovador", "Rua da Penha", 2)
        # ...

        return guimaraes_graph
```

3.2 Representação dos Pontos de Entrega em Grafo

Para representar (visualmente) todos os pontos de entrega sob a forma de um grafo, utilizamos o método **desenha()**.

```
    if __name__ == "__main__":
    generator = GuimaraesStreetGraphGenerator()
    guimaraes_graph = generator.generate_graph()
    print("Guimaraes Street Graph:")
    print(guimaraes_graph)
    guimaraes_graph.desenha()
```

Eis uma possível representação do grafo gerado:

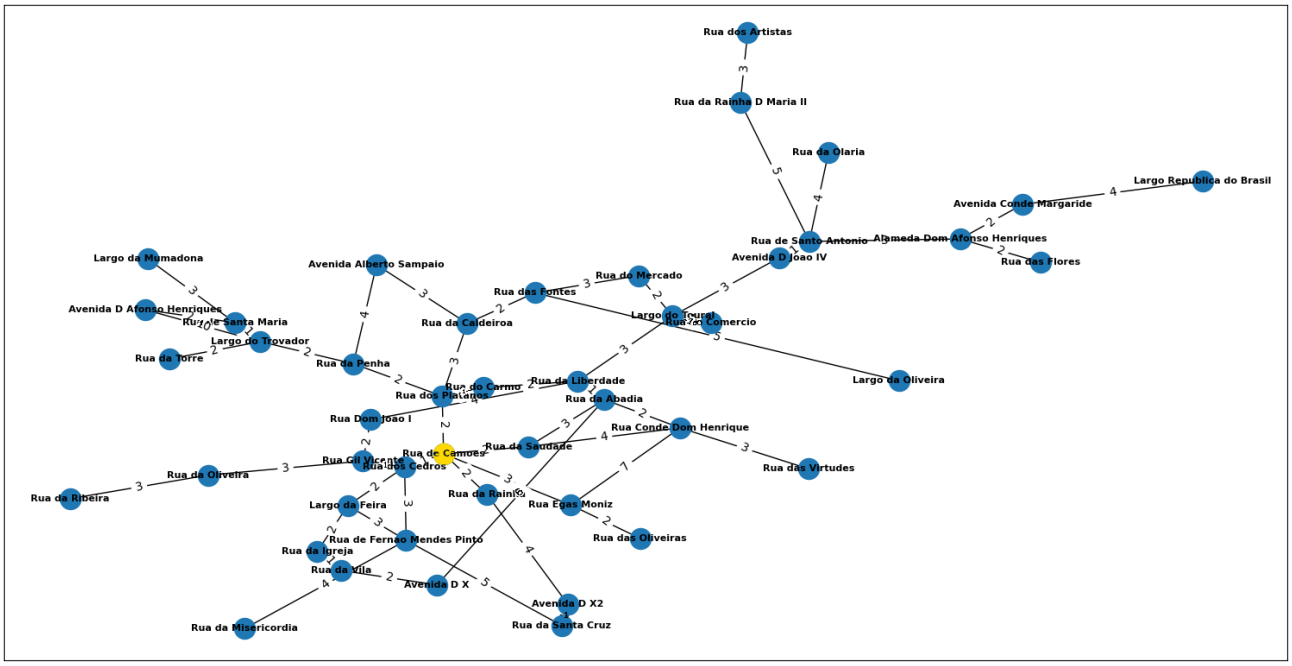

Figura 1: Representação dos Pontos de Entrega em Grafo

3.3 Estratégias de Procura Utilizadas

Para determinar o caminho mais favorável, são utilizados diferentes tipos de algoritmos de procura. De todos os disponíveis, o escolhido para a entrega é aquele que retorna o menor custo e o que consome menos. Utilizaram-se diferentes tipos de algoritmos existentes de pesquisa informada e não informada, que serão devidamente abordados nos tópicos que se seguem.

3.3.1 Procura Informada

A*: O algoritmo A* combina os benefícios da procura em largura (BFS) com uma heurística, para encontrar uma solução de uma forma eficiente. Ele utiliza uma função de custo total que corresponde à soma do custo do caminho percorrido até o estado atual, com uma estimativa (heurística) do custo do estado atual até a meta. A* prioriza estados com custos totais mais baixos.

A função utilizada por este algoritmo de procura pode ser definida como:

$$f(n) = g(n) + h(n) \tag{1}$$

onde $g(n)$ é a soma do custo do caminho percorrido até o estado atual e $h(n)$ é o valor da heurística do nó em análise.

Greedy: O algoritmo de procura gulosa seleciona sempre o nó com base (apenas) na heurística, sem considerar o custo total do caminho.

A função utilizada por este algoritmo de procura pode ser definida como:

$$f(n) = h(n) \tag{2}$$

onde $h(n)$ é o valor da heurística do nó em análise.

3.3.2 Procura não Informada

BFS (Procura em Largura): A estratégia de procura em Largura consiste em explorar sistematicamente todos os sucessores do estado atual antes de se aprofundar. A sua característica notável é a completude, assegurando que, se houver uma solução, ela será encontrada e, além disso, será a mais curta possível.

DFS (Procura em Profundidade): Ao contrário da BFS, a procura em Profundidade mergulha o mais fundo possível num caminho antes de retroceder. É uma abordagem mais ousada, explorando as profundezas do espaço de procura.

Uniform Cost Search (Procura de Custo Uniforme): A UCS expande o nó com o menor custo acumulado até o momento, sendo uma versão ponderada da BFS. Essa abordagem garante eficiência ao encontrar a solução mais curta, levando em consideração os custos associados a cada passo. Para além do já referido, também implementamos a heurística neste algoritmo para que em casos de empate fosse escolhido o nó com a menor heurística.

Iterative Deepening Search: O IDS é uma adaptação da DFS que incorpora o melhor dos dois mundos. Realizando procuras em profundidade com limites crescentes, começa com um limite pequeno e aumenta gradualmente. Essa estratégia combina a eficácia da DFS com a garantia de encontrar a solução mais rasa primeiro.

3.4 Comparação de Resultados dos Algoritmos de Procura

A análise comparativa dos resultados dos algoritmos de procura é essencial para entender o desempenho e a eficácia de cada estratégia num contexto específico. O código abaixo apresenta uma função que utiliza todos os algortimos de procura discutidos anteriormente, de modo a que seja possível encontrar caminhos num grafo e, seguidamente, comparar os custos. Além de ser apresentado o caminho mais curto e o respetivo custo da solução, também é apresentado o caminho percorrido ao longo da execução em cada um dos algoritmos desenvolvidos.

```
def compare_search_algorithm_results(guimaraes_graph, starting_node,
                                     finishing_node):
print("\n===== Compare Search Algorithm Results =====")
try:
    path_a_star, cost_a_star = guimaraes_graph.procura_aStar(
                                        starting_node, finishing_node)
    print("\nA* Search Result:")
    print("Path:", path_a_star)
    print("Cost:", cost_a_star)
except:
    print("\nA* Search Result:")
    print("\nNo path found.")

try:
    path_greedy, cost_greedy = guimaraes_graph.greedy(starting_node,
                                        finishing_node)
    print("\nGreedy Search Result:")
    print("Path:", path_greedy)
    print("Cost:", cost_greedy)
except:
    print("\nGreedy Search Result:")
    print("\nNo path found.")

try:
    path_bfs, cost_bfs = guimaraes_graph.procura_BFS(starting_node,
                                        finishing_node)
    print("\nBFS Search Result:")
    print("Path:", path_bfs)
```

```
        print("Cost:", cost_bfs)
    except:
        print("\nBFS Search Result:")
        print("\nNo path found.")

    try:
        path_dfs, cost_dfs = guimaraes_graph.procura_DFS(starting_node,
                                                         finishing_node)
        print("\nDFS Search Result:")
        print("Path:", path_dfs)
        print("Cost:", cost_dfs)
    except:
        print("\nDFS Search Result:")
        print("\nNo path found.")

    try:
        path_uc, cost_uc = guimaraes_graph.procura_uniform_cost(
                                                         starting_node, finishing_node)
        print("\nUniform cost Search Result:")
        print("Path:", path_uc)
        print("Cost:", cost_uc)
    except:
        print("\nUniform cost Search Result:")
        print("\nNo path found.")

    try:
        path_id, cost_id = guimaraes_graph.procura_IDDFS(starting_node,
                                                         finishing_node)
        print("\nIterative Deepening Search Result:")
        print("Path:", path_id)
        print("Cost:", cost_id)
    except:
        print("\nIterative Deepening Search Result:")
        print("\nNo path found.")
```

3.5 Justificação das Heurísticas nos Algoritmos de Procura Informada

As heurísticas desempenham um papel crucial em algoritmos de procura informada, fornecendo estimativas inteligentes para orientar a procura em direção a soluções mais promissoras. No contexto dos algoritmos como o A* e *Greedy Search*, as heurísticas são utilizadas para avaliar o custo ou potencial de cada nó em direção ao objetivo. No nosso caso, a heurística utilizada nos algoritmos é baseada na estimativa do custo do nó até ao objetivo. A função `calculate_turns_heuristic` implementa uma heurística simples, contando o número de vizinhos em comum entre o nó atual e o nó objetivo.

```
def calculate_turns_heuristic(self, current_node, goal_node):
    current_node_neighbors = set([neighbor for neighbor, _ in self.m_graph[
                                                    current_node]])
    goal_node_neighbors = set([neighbor for neighbor, _ in self.m_graph[
                                                    goal_node]])

    turns = len(current_node_neighbors.intersection(goal_node_neighbors))
    return turns
```

No A*, a heurística é combinada com o custo real do caminho percorrido até o momento para avaliar os nós. No Greedy, apenas a heurística é considerada para a escolha do próximo

nó. Em casos de empate no custo do Uniform Cost Search, a heurística é utilizada como critério de desempate.

A escolha da heurística depende da natureza específica do problema. No nosso contexto, a heurística de contagem de vizinhos em comum pareceu ser a mais apropriada, dado que não são associadas coordenadas aos nossos nós, não sendo possível o calculo de heurísticas como a da distância euclidiana.

3.6 Desenvolvimento do Trabalho

Nesta secção, abordaremos as classes relacionadas ao nosso projeto.

3.6.1 Classe *main*

A classe *main* é responsável pela coordenação geral do programa. Ela gere a inicialização e execução de outras classes e módulos.

O código Python fornecido é um sistema de gestão de entregas que utiliza um grafo de ruas em Guimarães. Vamos analisar as principais partes:

1. **Declarações de Importação:**

```
from datetime import datetime, timedelta
from menu_functions import *
from graph.GuimaraesStreetGraphGenerator import
                                        GuimaraesStreetGraphGenerator
from Test import Tests
```

   Essas declarações importam módulos e classes necessárias para manipulação de datas, funções de menu, gerador de grafos e testes.

2. **Função Principal `main()`:**

```
def main():
    # ...
```

   A função principal inicializa o gerador de grafos, cria listas vazias para estafetas e encomendas, gera o grafo de ruas de Guimarães e entra num *loop* para exibir um menu para o utilizador.

3. **Opções do Menu:**

```
choice = get_user_choice()

    if choice == 0:
        print("Exiting the program. Goodbye!")
        break
    elif choice == 1:
        courier = add_courier()
        if courier is not None:
            couriers.append(courier)
            print(f"{courier.name} added as a courier with {courier.
                                                    transport}.")
# ...
```

   As opções do menu incluem *features* como :

   (a) Sair do Programa

   (b) Adicionar Estafeta

(c) Adicionar Encomenda

(d) Mostrar Encomendas pendentes

(e) Mostrar Encomendas enviadas

(f) Mostrar Estafetas

(g) Avançar um dia

(h) Menu de *developer*, menu que inclui features como:

- Representar postos de entrega sob a forma de um grafo
- Utilizar algoritmos de procura
- Comparar resultados de algoritmos de procura
- Importar estafetas para fins de teste
- Importar encomendas para fins de teste
- Remover arestas do grafo gerado (para simular estradas cortadas)

3.6.2 Classe *menu_functions*

Esta classe simplesmente desenvolve as funções referidas acima. Engloba toda a lógica necessária para o correto funcionamento da classe *main*.

1. `main_menu()`
   - Imprime o menu principal para o Sistema de Entrega *Health Planet*.
2. `dev_menu()`
   - Imprime o menu de desenvolvimento para opções avançadas.
3. `get_node_final(guimaraes_graph)`
   - Recebe o *input* do utilizador para a rua de entrega e garante que seja um nó válido no grafo.
4. `get_nodes()`
   - Recebe o *input* do utilizador para os nós de início e término.
5. `get_user_choice()`
   - Recebe a escolha do utilizador para as opções do menu e trata de *inputs* inválidos.
6. `search_menu(guimaraes_graph)`
   - Exibe os algoritmos de procura disponíveis e solicita o *input* do utilizador para realizar uma procura.
7. `compare_search_algorithm_results(guimaraes_graph, starting_node, finishing_node)`
   - Compara e imprime os resultados de diferentes algoritmos de procura para um par de nós dados.
8. `add_courier()`
   - Recebe o *input* do utilizador para adicionar um novo estafeta com nome e método de transporte.

9. `add_order(guimaraes_graph)`
    - Recebe o *input* do utilizador para registar um novo pedido com detalhes do cliente e tempo de processamento.
10. `display_pending_orders(orders)`
    - Imprime todos os pedidos registados com status *Waiting*.
11. `display_sent_orders(orders)`
    - Imprime todos os pedidos registados com status *Delivered*.
12. `display_couriers(couriers)`
    - Imprime todos os estafetas registados, classificados por *rating*.
13. `choose_best_algorithm(graph, starting_node, finishing_node, order_client_name)`
    - Compara os resultados de algoritmos de procura e retorna o melhor caminho e custo para um pedido.
14. `process_orders(orders, couriers, guimaraes_graph)`
    - Processa pedidos pendentes, atribui-os a estafetas adequados e atualiza informações de encomendas e estafetas.
15. `calculate_rating_based_on_percentage_difference(estimated_time, demanded_time)`
    - Calcula uma avaliação com base na diferença percentual entre os tempos de entrega estimado e exigido.
16. `verify_removed_edges(graph, removed_edges, current_date)`
    - Verifica se alguma aresta removida (simulando bloqueios de estrada) expirou e adiciona-a de volta ao grafo.

3.6.3 Classe *order_functions*

Esta classe lida com operações relacionadas a pedidos. Engloba métodos para adicionar novos pedidos e exibir detalhes dos pedidos registados.

1. `add_order()`
    - Solicita informações ao utilizador para criar uma nova instância da classe `Order` e retorna-a.
2. `display_orders(orders)`
    - Imprime os detalhes de todas as encomendas registadas.

3.6.4 Classe *CourierFunctions*

Esta classe lida com operações relacionadas a estafetas. Engloba métodos para adicionar novos estafetas e exibir detalhes dos estafetas registados.

1. `add_courier()`
    - Solicita informações ao utilizador para criar uma nova instância da classe `Courier` e retorna-a.
2. `display_couriers(couriers)`
    - Imprime os detalhes de todas os estafetas registados.

3.6.5 Classe *Order*

Esta classe representa uma encomenda no sistema de entregas. Possui atributos como nome do cliente, peso, volume, tempo de processamento, nó inicial, último nó, caminho, custo e *status*.

```
class Order:
    def __init__(self, client_name, weight, volume, processing_time,
                                                starting_node, last_node):
        self.client_name = client_name
        self.weight = weight
        self.volume = volume
        self.processing_time = processing_time
        self.starting_node = starting_node
        self.last_node = last_node
        self.path = []
        self.cost = 0
        self.status = "Waiting"

    # Calcula o preco do pedido com base no peso, volume e tempo de
                                                processamento
    def calculate_price(self):
        return self.weight * 0.3 + self.volume * 0.1 + self.processing_time
                                                    * 0.2
```

A classe *Order* possui um construtor *__init__* que inicializa os atributos do pedido. Além disso, possui um método *calculate_price* que calcula o preço do pedido com base no peso, volume e tempo de processamento.

3.6.6 Classe *Courier*

Esta classe representa um estafeta no sistema de entregas. Possui atributos como nome, meio de transporte, peso, pedidos entregues, lista de avaliações, avaliação e métodos para adicionar entregas, obter entregas, verificar peso, calcular velocidade média, calcular capacidade máxima, calcular tempo máximo, verificar tempo, obter tempo, combinar entregas, verificar interseção de caminhos e calcular avaliação.

```
class Courier:
    def __init__(self, name, transport, weight=0, orders=None):
        self.name = name
        self.transport = transport
        self.weight = weight
        if orders is not None:
            self._deliveries.extend(orders)
        self.ratinglist = []
        self.rating = 0
```

```
    # ...

    def calculateMaxTime(self, path_cost, order_weight):
        total_weight = self.weight + order_weight
        if self.transport == "Bicycle":
            return path_cost / (self.average_speed() - (0.6 * total_weight))
                                                    * 60
        elif self.transport == "Moto":
            return path_cost / (self.average_speed() - (0.5 * total_weight))
                                                    * 60
        elif self.transport == "Car":
            return path_cost / (self.average_speed() - (0.1 * total_weight))
                                                    * 60

    # ...

    def calculate_rating(self, user_rating):
        self.ratinglist.append(user_rating)
        average_rating = sum(self.ratinglist) / len(self.ratinglist)
        self.rating = round(average_rating)

    def calculate_price(self, order_price):
        if self.transport == "Bicycle":
            return order_price + 1
        elif self.transport == "Moto":
            return order_price + 2
        elif self.transport == "Car":
            return order_price + 5
        else:
            return 0
```

1. `__init__()`
    - Inicializa os atributos de um estafeta.
2. `add_delivery(self, order)`
    - Adiciona uma entrega à lista de entregas de um estafeta.
3. `get_deliveries(self)`
    - Retorna a lista de entregas do estafeta.
4. `verifyWeight(self, order_weight)`
    - Verifica se o peso da entrega é válido para o meio de transporte do estafeta.
5. `average_speed(self)`
    - Retorna a velocidade média do estafeta com base no meio de transporte.
6. `max_capacity(self)`
    - Retorna a capacidade máxima de carga do estafeta com base no meio de transporte.
7. `calculateMaxTime(self, path_cost, order_weight)`
    - Calcula o tempo máximo estimado para a entrega com base no custo do caminho e no peso da entrega.

8. `verifyTime(self, path_cost, order_processing_time, order_weight)`
    - Verifica se o tempo de processamento da entrega é válido com base no tempo máximo estimado.
9. `getTime(self, path_cost, order_weight)`
    - Retorna o tempo máximo estimado para a entrega com base no custo do caminho e no peso da entrega.
10. `can_combine_delivery(self, new_order)`
    - Verifica se uma nova entrega pode ser combinada com as entregas existentes do estafeta.
11. `path_intersects(self, order1, order2)`
    - Verifica se os caminhos de duas entregas se intersectam.
12. `calculate_rating(self, user_rating)`
    - Calcula a classificação média do estafeta com base nas classificações dos utilizadores.
13. `calculate_price(self, order_price)`
    - Calcula o preço da entrega com base no meio de transporte.

3.6.7 Classe *Test*

Esta classe chama dois métodos estáticos, *import_couriers* e *import_orders*, que são responsáveis pelo povoamento de listas de objetos do tipo *Courier* e *Order*, respectivamente. Os objetos *Courier* representam estafetas e possuem atributos como nome e tipo de veículo. Por outro lado, os objetos *Order* representam pedidos e têm atributos como código, peso, prioridade, tempo estimado de entrega e endereços de origem e destino. Os métodos preenchem as listas de *couriers* e pedidos com dados fictícios, para serem utilizados em testes ou simulações.

```
    import random
from Courier import Courier
from Order import Order

class Tests:
    def import_couriers(couriers):
        couriers.extend([
            Courier("Joao", "Bicycle"),
            Courier("Pedro", "Moto"),
            # ...
        ])

    def import_orders(orders):
        orders.extend([
            Order("Order01", 1, 10, random.randint(1, 60), "Rua de Camoes",
                                                    "Rua da Saudade"),
            Order("Order02", 90, 20, random.randint(1, 120), "Rua de Camoes"
                                                    , "Avenida Conde Margaride
                                                    "),
            Order("Order03", 40, 30, random.randint(1, 120), "Rua de Camoes"
                                                    , "Rua da Abadia"),
            # ...
        ])
```

1. `import_couriers(couriers)`

    - Este método utiliza a função *extend* para adicionar uma lista de objetos *Courier* à lista fornecida como argumento. Cada objeto *Courier* é inicializado com um nome e um meio de transporte (*Bicycle*, *Moto* ou *Car*).

2. `import_orders(orders)`

    - Semelhante ao método anterior, este método utiliza a função *extend* para adicionar uma lista de objetos *Order* à lista fornecida como argumento. Cada objeto *Order* é inicializado com um identificador, peso, valor, tempo estimado de entrega, e endereços de origem e destino.

3.6.8 Classes *Node*, *Graph* e *GuimaraesStreetGraphGenerator*

As classes *Node* e *Graph* foram previamente desenvolvidas nas aulas TP, e portanto não serão discutidas.

GuimaraesStreetGraphGenerator já foi devidamente abordada na secção **Gerar Circuitos de Entrega**.

4 Sumário e Discussão dos Resultados obtidos

Com o objetivo de testar o projeto, implementamos uma sétima opção no menu inicial, o ***Dev Menu***. Este é responsável por:

- **Representar o grafo**
- **Testar** cada **algoritmo** de forma **independente**
- **Comparar resultados entre algoritmos** num determinado trajeto
- **Importar** *couriers* e *orders* de **controlo** pré-definidos
- **Simular bloqueios** em ruas

Para todos os testes utilizados, partimos do pressuposto que todos os *couriers* têm como **ponto de partida** a **Rua de Camoes**. Para além disso, todos os cenários foram testados com a mesma unidade de controlo, obtida através da importação disponível no ***Dev Menu***, tendo esta possíveis resultados diferentes, devido à aleatoridade do tempo de processamento de cada *order* incorporada na funcionalidade referida.

Cenários Possíveis

- **Sem *Roadblocks***: Neste caso, os testes foram realizados através de diversas unidades de teste, tendo a grande maioria dos resultados obtidos sido satisfatória, salvo algumas exceções, que iremos referir no *capítulo 4.1.*
- **Com *Roadblocks***: Neste segundo caso, foram realizados testes através das mesmas unidades de teste do caso anterior, tendo sido possível a deteção de alguns erros no código, como por exemplo, a falta de verificação de encomendas impossíveis, estando estas implementadas como considerações especiais no capítulo seguinte.

4.1 Considerações Especiais e Limitações

Condições Gerais

- **Entrega impossível devido à falta de tempo**: Caso uma *order* tenha um tempo de processamento **inferior** ao possível, esta é considerada **cancelada**. Este cancelamento poderá resultar de uma combinação do peso da mesma em conjunto com o tipo de veículo que os *couriers* disponíveis possuirem (nos casos em que não são suficientemente rápidos), ou pelo caso do tempo disponível ser simplesmente insuficiente, dada a distância entre o nó inicial e o nó objetivo, para qualquer veículo disponível.
- **Falha de entrega devido à falta de *couriers***: Não havendo *couriers* **suficientes** para todas as entregas, estas serão descartadas com base na **ordem de chegada** das mesmas.

Para o caso de existirem *Roadblocks* existe, ainda, uma consideração especial, no caso de **não existirem caminhos possíveis** devido aos mesmos, tendo assim a *order* de ser **cancelada**.

5 Conclusão

Ao longo dos testes realizados neste trabalho, tornou-se evidente que a solução otimizada para as entregas tem como base, na sua grande maioria, o algoritmo A*. Embora exista a possibilidade de outros algoritmos alcançarem uma solução igualmente eficaz (como IDS ou DFS), o A* consistentemente se destacou. Não apenas percorreu o menor número de nós, como também apresentou o menor custo. Isso permite concluir que, entre todos os algoritmos analisados neste trabalho, o A* é, de facto, o mais eficiente.