



**Universidade do Minho**  
Escola de Engenharia

# **Inteligência Artificial**

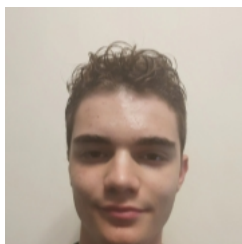
## **Trabalho prático**

Grupo 023

Alexandre de Oliveira Monsanto, a104358

Maria Inês da Rocha Pinto Gracias Fernandes, a104522

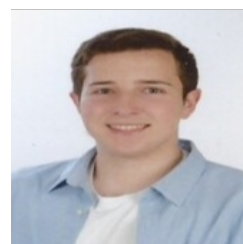
Vasco João Timóteo Gonçalves, a104527



a104358



a104522



a104527

Dezembro, 2024

## Índice

<b>Introdução .....</b>	<b>3</b>
<b>Descrição do problema .....</b>	<b>3</b>
<b>Formulação do problema .....</b>	<b>3</b>
1. Estado inicial .....	3
2. Teste objetivo .....	4
3. Operadores (ações disponíveis) .....	4
4. Custo da solução .....	5
<b>Tarefas realizadas .....</b>	<b>6</b>
• Construção do grafo (zonas de entrega e caminhos possíveis) .....	6
• Estratégias de procura (informada e não informada) .....	7
Procura não informada.....	7
Procura informada.....	8
• Implementação de algoritmos de procura .....	9
Procura não informada.....	9
Procura informada.....	10
• Testagem de algoritmos de procura .....	12
• Condições dinâmicas .....	12
<b>Resultados obtidos .....</b>	<b>14</b>

## **Introdução**

O presente relatório apresentará informações relativas ao Trabalho Prático da Unidade Curricular Inteligência Artificial, pertencente ao 3º Ano da Licenciatura em Engenharia Informática da Universidade do Minho, realizada no ano letivo 2024/2025.

O objetivo deste projeto é desenvolver algoritmos de procura que permitam otimizar a distribuição de alimentos em zonas afetadas por uma catástrofe natural. Pretende-se garantir que os recursos disponíveis sejam utilizados de forma eficiente, priorizando as áreas mais necessitadas e maximizando o número de pessoas assistidas dentro de um tempo limitado.

## **Descrição do problema**

Durante uma catástrofe natural, é prioritário fornecer suprimentos essenciais como alimentos, água e medicamento às áreas mais afetadas, que apresentam desafios específicos conforme a gravidade, densidade populacional e acessibilidade, influenciadas pela geografia e pelo clima. A distribuição eficiente dos mantimentos envolve o uso de diversos veículos (drones, helicópteros, barcos, camiões e camionetas), cada um com limitações de carga, autonomia e tempo de viagem. É crucial selecionar o transporte adequado e otimizar rotas para garantir que os suprimentos cheguem rapidamente às zonas críticas, priorizando locais com maior gravidade ou população mais vulnerável. A estratégia deve minimizar o desperdício de recursos, como alimentos perecíveis e combustível, e enfrentar obstáculos como estradas destruídas ou condições adversas. Assim, a operação maximiza sua eficiência, salvando o maior número possível de vidas.

## **Formulação do problema**

### **1. Estado inicial**

O estado inicial é caracterizado por todos os veículos (camiões, drones, helicópteros, barcos ou camionetas) estarem na sua respetiva base de origem (Porto, Coimbra, Lisboa ou Faro), devidamente atestados e prontos para agir em caso de necessidade. Quanto aos suprimentos, estão armazenados estrategicamente e podem ser alimentos, água, medicamentos básicos ou especializados e kits de primeiros socorros. Existem também postos de reabastecimento de combustível (localizados no Norte, Centro, Lisboa, Alentejo e Algarve) de forma a serem facilmente acedidos para reabastecer os veículos.

## 2. Teste objetivo

O objetivo é garantir que todos os suprimentos necessários sejam entregues às zonas afetadas, respeitando as seguintes condições:

1. Cada zona afetada recebeu uma quantidade mínima necessária de alimentos, água e medicamentos.
2. As entregas foram realizadas dentro da janela de tempo crítica de cada zona.
3. As entregas priorizaram as zonas com maior severidade e maior população impactada, conforme o sistema de prioridades.
4. Os recursos limitados foram utilizados de forma eficiente, minimizando o desperdício de alimentos perecíveis, combustível, e otimizando a capacidade de carga dos veículos.

**Critério de paragem:** o problema é considerado resolvido quando:

- Todas as zonas afetadas atingirem o nível mínimo de suprimentos exigido.
- Nenhuma janela de tempo crítica foi ultrapassada.
- O uso de recursos respeitou o princípio da eficiência, evitando desperdício desnecessário.

## 3. Operadores (ações disponíveis)

Os operadores definem as ações que podem ser realizadas em cada estado para transitar para outro estado no espaço de procura. Estas ações permitem a movimentação e alocação eficiente de recursos e veículos ao longo do grafo. No contexto do problema, os operadores disponíveis são:

- **Carregar Suprimentos:**

**Descrição:** Um veículo é carregado com suprimentos numa base.

**Restrições:**

A carga total não pode exceder a capacidade do veículo.

Os tipos e quantidades de suprimentos carregados devem atender às necessidades das zonas afetadas.

- **Mover Veículo:**

**Descrição:** Um veículo desloca-se de um nodo (posto ou zona) para outro no grafo.

**Restrições:**

A rota deve ser acessível (sem bloqueios ou incompatibilidades de terreno).

A autonomia do veículo deve ser suficiente para alcançar o destino.

O tempo de deslocamento deve ser considerado em relação à janela de tempo crítica das entregas.

- **Entregar Suprimentos:**

**Descrição:** Um veículo entrega uma parte ou a totalidade dos suprimentos transportados a uma zona afetada.

**Restrições:**

Os suprimentos entregues devem corresponder às necessidades da zona.

A entrega deve ocorrer dentro da janela de tempo crítica da zona.

- **Reabastecer Veículo:**

**Descrição:** Um veículo retorna a um posto de reabastecimento para reabastecer combustível.

**Restrições:**

Deve estar num posto de reabastecimento devidamente assinalado.

- **Priorizar Rotas Alternativas:**

**Descrição:** Replanear a rota devido a mudanças dinâmicas, como bloqueios ou alterações climáticas.

**Restrições:**

A rota alternativa deve ser acessível e otimizada em relação ao custo de deslocamento.

#### **4. Custo da solução**

O objetivo é minimizar o custo total cumprindo os objetivos definidos no teste objetivo (entregar suprimentos dentro do prazo e evitar desperdícios). No contexto do problema, o custo vai ser definido consoante o custo de cada nodo, tendo também em conta o custo do impacto dos eventos dinâmicos (que serão abordados mais adiante no relatório). Assim, o custo da solução dependerá de caso para caso.

## Tarefas realizadas

- Construção do grafo (zonas de entrega e caminhos possíveis)

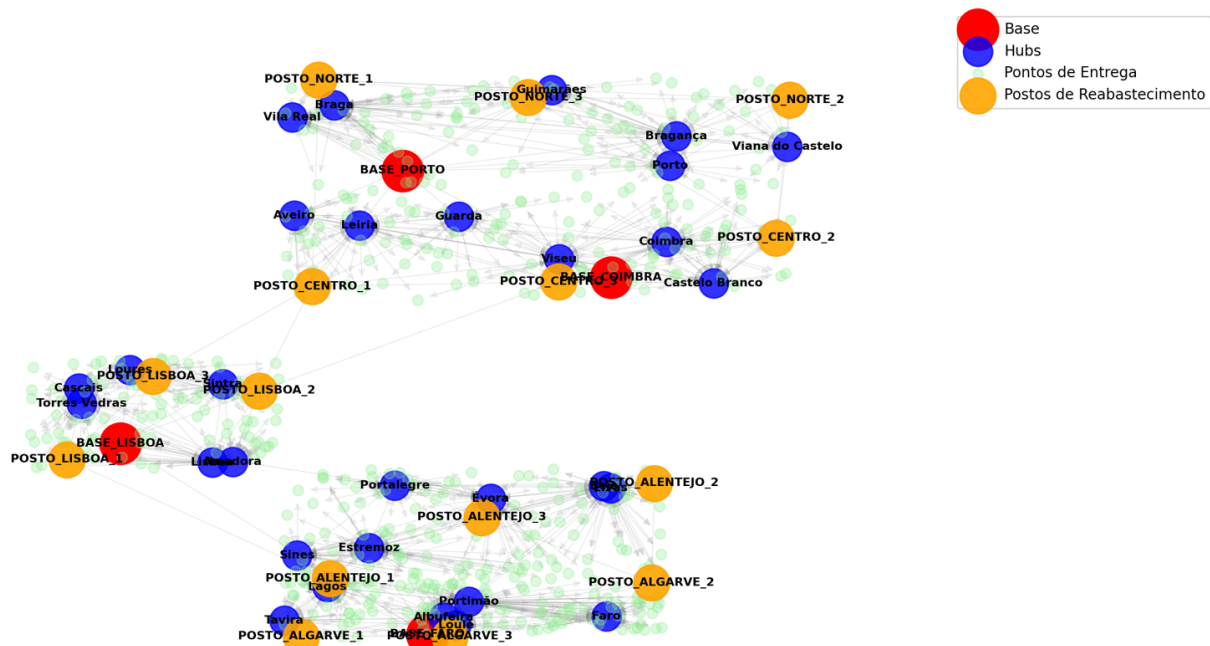


Figura 1 – Grafo que representa a rede de distribuição de suprimentos

O processo de construção do grafo começa pela definição dos diferentes tipos de nodos e arestas, que refletem os elementos principais do sistema:

**Bases (nodos vermelhos):** Representam os pontos de origem dos veículos (Porto, Coimbra, Lisboa e Faro). É a partir dessas bases que os veículos iniciam as suas operações, devidamente carregados e abastecidos.

**Hubs (nodos azuis):** Servem como pontos intermediários para facilitar a distribuição de suprimentos. Esses *hubs* estão estrategicamente posicionados para facilitar o alcance às zonas mais remotas ou isoladas.

**Pontos de Entrega (nodos verdes):** São as zonas afetadas que necessitam de suprimentos.

**Postos de Reabastecimento (nodos laranja):** Pontos onde os veículos podem reabastecer combustível ou carregar mais suprimentos, garantindo que possam continuar operacionais mesmo em condições adversas.

**Arestas:** Representam as rotas possíveis entre os diferentes nodos, com pesos associados que refletem o custo e o tempo de deslocamento. Esses pesos podem variar devido a condições climáticas ou bloqueios.

Para criar estas arestas (conexões) entre os nodos, definiram-se funções que calculam os custos e tempos necessários para viajar entre dois nodos.

```

criar_grafo.py 4 criar_grafo.py/PortugalDistributionGraph/gerar_coordenadas_posto
11 class PortugalDistributionGraph:
12     def calcula_distancia(self, coord1, coord2):
13         return hs.haversine(coord1, coord2, unit=Unit.KILOMETERS)
14
15     def calcular_custo_tempo(self, coord1, coord2):
16         dist = self.calcula_distancia(coord1, coord2)
17         custo_base = dist * 0.08
18         tempo_base = dist * 0.01
19         custo = custo_base * random.uniform(0.8, 1.2)
20         tempo = tempo_base * random.uniform(0.9, 1.3)
21         return round(custo, 2), round(tempo, 3)

```

Figura 2 - Funções que calculam os custos de distância e de tempo

```

criar_grafo.py 4 criar_grafo.py/PortugalDistributionGraph/gerar_coordenadas_posto
11 class PortugalDistributionGraph:
12     def _criar_conexoes(self):
13         nodes = list(self.grafo.nodes(data=True))
14
15         # Conectar bases com bases e postos de abastecimento mais próximas
16         bases = [n for n, d in nodes if d['tipo'] == 'base']
17
18         for base1 in bases:
19             coord_base1 = self.grafo.nodes[base1]['coordenadas']
20             # Conectar com a base mais próxima
21             bases_dist = [(b, self.calcula_distancia(coord_base1, self.grafo.nodes[b]['coordenadas']))
22                           for b in bases if b != base1]
23             bases_dist.sort(key=lambda x: x[1])
24
25             base_mais_proxima, dist_proxima = bases_dist[0]
26             custo, tempo = self.calcular_custo_tempo(coord_base1, self.grafo.nodes[base_mais_proxima]['coordenadas'])
27             self.grafo.add_edge(base1, base_mais_proxima, custo=custo, tempo=tempo)
28             self.grafo.add_edge(base_mais_proxima, base1, custo=custo, tempo=tempo)
29
30             # Conectar com o posto de abastecimento mais próximo
31             postos = [n for n, d in nodes if d['tipo'] == 'posto']
32             postos_dist = [(p, self.calcula_distancia(coord_base1, self.grafo.nodes[p]['coordenadas']))
33                            for p in postos]
34             postos_dist.sort(key=lambda x: x[1])
35
36             for posto, dist in postos_dist[:2]:
37                 custo, tempo = self.calcular_custo_tempo(coord_base1, self.grafo.nodes[posto]['coordenadas'])
38                 self.grafo.add_edge(base1, posto, custo=custo, tempo=tempo)
39                 self.grafo.add_edge(posto, base1, custo=custo, tempo=tempo)

```

Figura 3 – Exemplo de conexões entre nodos, que usam as funções da Figura 2

- Estratégias de procura (informada e não informada)

## Procura não informada

### Procura BFS (Breadth-First Search)

O algoritmo Breadth-First Search é um algoritmo de procura num grafo em largura. Esta estratégia vai começar por procurar em todos os nodos de menor profundidade, isto é, vai percorrer todos os nodos de um certo nível de profundidade e quando todos estes forem percorridos, passa a procurar no nível seguinte. Normalmente esta procura demora muito tempo e ocupa muito espaço, pelo que apenas deve ser utilizada em problemas pequenos. Contudo, apresenta sempre soluções com o menor número de nodos e, se o custo de todas as arestas for igual, será também a solução com menor custo.

No contexto deste projeto, ele pode ser utilizado para explorar caminhos iniciais sem considerar prioridades ou condições dinâmicas, como bloqueios de rotas (o que não é o ideal para o objetivo pretendido).

## Procura DFS (Depth-First Search)

O objetivo desta procura passa por expandir o nodo atual sempre num nodo mais profundo da árvore. Uma das suas vantagens é a utilização de pouca memória, o que faz com que seja um algoritmo ideal para problemas com muitas soluções. Por outro lado, este algoritmo não pode ser usado em grafos com profundidade infinita, uma vez que, neste tipo de grafos, entra, por vezes, em caminhos errados.

No contexto pretendido, a DFS não considera prioridades ou condições variáveis diretamente, sendo menos eficiente para situações que procurem soluções otimizadas ou em condições dinâmicas.

## Procura informada

### Heurística utilizada

Os algoritmos de procura informada baseiam-se numa heurística para encontrar o próximo melhor nodo, de entre os nodos disponíveis. Assim, neste caso, foi decidido pelo grupo definir apenas uma heurística baseada no custo mínimo entre os nodos.

### Procura Gulosa (*Greedy*)

O estado escolhido por esta procura é o estado com menor heurística dos estados atualmente expandidos. Este algoritmo recorre, assim, a uma *open list* (conjunto de nodos que podem vir a ser visitados) e a uma *closed list* (conjunto de nodos que já não podem ser visitados). Este algoritmo começa por inicializar a *open list* com o estado inicial, escolhendo em seguida, a cada iteração, o nodo dessa lista com menor heurística, adicionando à mesma, no final, os vizinhos não visitados. O processo é repetido até se encontrar um nodo objetivo. Embora seja rápido, não garante soluções ótimas.

Aliás, analisando o contexto do projeto, a Gulosa pode ser útil para situações onde se procura atender zonas prioritárias rapidamente, mas pode falhar em otimizar recursos ou respeitar janelas de tempo críticas.

### Procura A\*

A procura A\* opta por uma estratégia semelhante à Gulosa, no entanto armazena o custo até cada um dos estados alcançados. O método de seleção do próximo estado da *open list* é a minimização da soma do custo atingido pelo nodo com a heurística desse nodo. Este algoritmo encontra solução ótima caso a heurística seja aceitável.

Neste contexto, a A\* é ideal para incorporar fatores como a prioridade das zonas, condições variáveis (como bloqueios ou rotas inadequadas para determinados veículos) e janelas de tempo críticas. Assim, esta abordagem é possivelmente a mais adequada para resolver o problema de distribuição eficiente de suprimentos num cenário dinâmico.

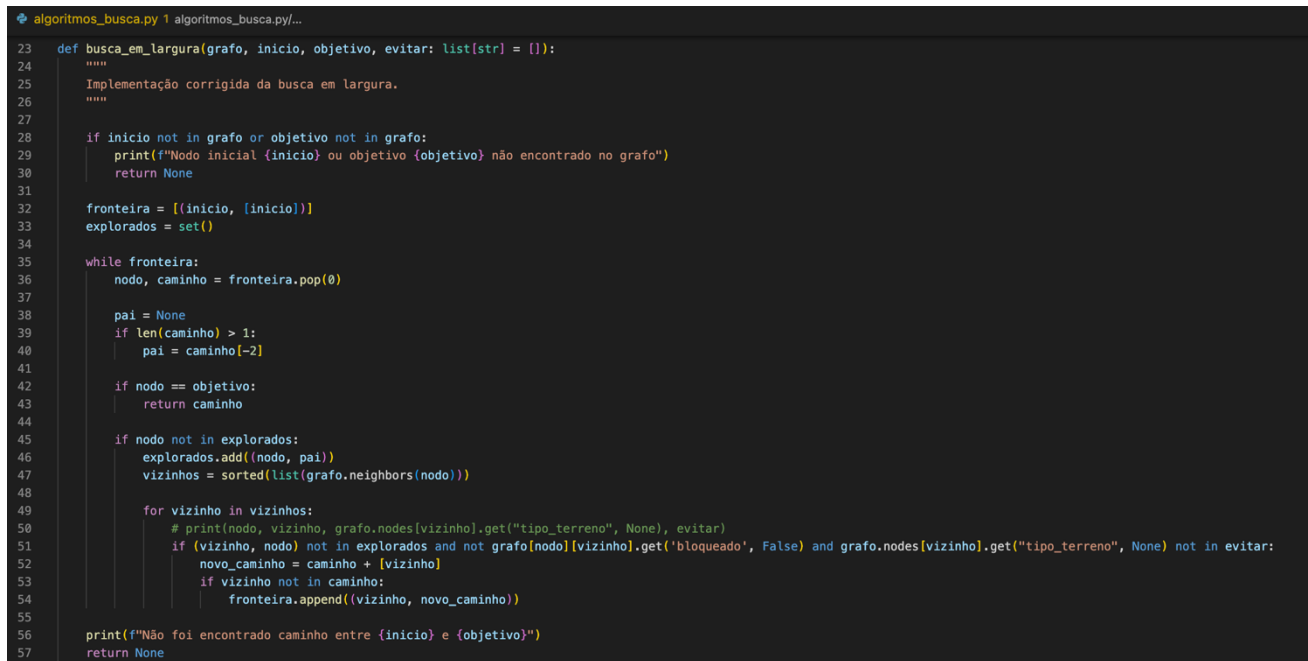


- **Implementação de algoritmos de procura**

Como explicado anteriormente, o objetivo do trabalho é otimizar a distribuição de suprimentos em zonas afetadas por uma catástrofe natural. De modo a definir qual a melhor rota para cada veículo, têm de ser implementados e testados os algoritmos de procura apresentados.

## Procura não informada

### Procura BFS (Breadth-First Search)



```
23 def busca_em_largura(grafo, inicio, objetivo, evitar: list[str] = []):
24     """
25     Implementação corrigida da busca em largura.
26     """
27
28     if inicio not in grafo or objetivo not in grafo:
29         print(f"Node inicial {inicio} ou objetivo {objetivo} não encontrado no grafo")
30         return None
31
32     fronteira = [(inicio, [inicio])]
33     explorados = set()
34
35     while fronteira:
36         nodo, caminho = fronteira.pop(0)
37
38         pai = None
39         if len(caminho) > 1:
40             pai = caminho[-2]
41
42         if nodo == objetivo:
43             return caminho
44
45         if nodo not in explorados:
46             explorados.add((nodo, pai))
47             vizinhos = sorted(list(grafo.neighbors(nodo)))
48
49             for vizinho in vizinhos:
50                 # print(nodo, vizinho, grafo.nodes[vizinho].get("tipo_terreno", None), evitar)
51                 if (vizinho, nodo) not in explorados and not grafo[nodo][vizinho].get('bloqueado', False) and grafo.nodes[vizinho].get("tipo_terreno", None) not in evitar:
52                     novo_caminho = caminho + [vizinho]
53                     if vizinho not in caminho:
54                         fronteira.append((vizinho, novo_caminho))
55
56     print(f"Não foi encontrado caminho entre {inicio} e {objetivo}")
57     return None
```

Figura 4 – Implementação do algoritmo BFS

Este algoritmo explora os nodos por nível, garantindo soluções com o menor número de arestas. Demonstrou limitações em cenários grandes e dinâmicos devido ao consumo de memória e falta de otimização de custos.

## Procura DFS (Depth-First Search)

```
algoritmos_busca.py 1 algoritmos_busca.py/...
59 def busca_em_profundidade(grafo, inicio, objetivo, evitar: list[str] = []):
60     """
61     Implementação corrigida da busca em profundidade.
62     """
63     if inicio not in grafo or objetivo not in grafo:
64         print(f"Nodo inicial {inicio} ou objetivo {objetivo} não encontrado no grafo")
65         return None
66
67     fronteira = [(inicio, [inicio])]
68     explorados = set()
69
70     while fronteira:
71         nodo, caminho = fronteira.pop()
72
73         if nodo == objetivo:
74             return caminho
75
76         if nodo not in explorados:
77             explorados.add(nodo)
78             vizinhos = sorted(list(grafo.neighbors(nodo)), reverse=True)
79
80             for vizinho in vizinhos:
81                 # print(nodo, vizinho, grafo.nodes[vizinho].get("tipo_terreno", None), evitar)
82                 if vizinho not in explorados and not grafo[nodo][vizinho].get('bloqueado', False) and grafo.nodes[vizinho].get("tipo_terreno", None) not in evitar:
83                     novo_caminho = caminho + [vizinho]
84                     if vizinho not in [n for n, _ in fronteira]:
85                         fronteira.append((vizinho, novo_caminho))
86
87     print(f"Não foi encontrado caminho entre {inicio} e {objetivo}")
88     return None
```

Figura 5 – Implementação do algoritmo DFS

Este algoritmo explorou trajetos em profundidade, mas foi menos eficiente em encontrar soluções otimizadas devido à ausência de consideração de custos ou prioridades.

## Procura informada

### Heurística utilizada

O cálculo da heurística foi feito da seguinte forma:

```
algoritmos_busca.py 1 algoritmos_busca.py/...
7 def calcular_heuristica(grafo, objetivo):
8     """
9     Calcula uma heurística baseada no custo mínimo entre os nodos.
10    """
11    heuristica = {}
12    for nodo in grafo.nodes():
13        try:
14            # Usa o algoritmo de Dijkstra para encontrar o caminho mais curto
15            caminho = nx.dijkstra_path(grafo, nodo, objetivo, weight='custo')
16            custo = sum(grafo[caminho[i]][caminho[i+1]]['custo']
17                       for i in range(len(caminho)-1))
18            heuristica[nodo] = custo
19        except (nx.NetworkXNoPath, nx.NodeNotFound):
20            heuristica[nodo] = float('inf')
21    return heuristica
```

Figura 6 - Cálculo da heurística

## Procura Gulosa (Greedy)

```
algoritmos_busca.py | algoritmos_busca.py/...
90 def busca_gulosa(grafo, inicio, objetivo, heuristica=None, evitar: list[str] = []):
91     """
92     Implementação corrigida da busca gulosa.
93     """
94     if inicio not in grafo or objetivo not in grafo:
95         print(f"Node inicial {inicio} ou objetivo {objetivo} não encontrado no grafo")
96         return None
97
98     if heuristica is None:
99         heuristica = calcular_heuristica(grafo, objetivo)
100
101     fronteira = [(heuristica[inicio], inicio, [inicio])]
102     explorados = set()
103
104     while fronteira:
105         _, nodo, caminho = sorted(fronteira, key=lambda x: x[0])[0]
106         fronteira = [(h, n, p) for h, n, p in fronteira if n != nodo]
107
108         if nodo == objetivo:
109             return caminho
110
111         if nodo not in explorados:
112             explorados.add(nodo)
113
114             for vizinho in sorted(grafo.neighbors(nodo)):
115                 # print(nodo, vizinho, grafo.nodes[vizinho].get("tipo_terreno", None), evitar)
116                 if vizinho not in explorados and not grafo[nodo][vizinho].get('bloqueado', False) and grafo.nodes[vizinho].get("tipo_terreno", None) not in evitar:
117                     if vizinho not in [n for _, n, _ in fronteira]:
118                         novo_caminho = caminho + [vizinho]
119                         fronteira.append((heuristica[vizinho], vizinho, novo_caminho))
120
121     print(f"Não foi encontrado caminho entre {inicio} e {objetivo}")
122     return None
```

Figura 7 - Implementação do algoritmo Gulosa

Este algoritmo mostrou-se útil para atender rapidamente zonas críticas, mas gerou custos operacionais mais elevados.

## Procura A\*

```
algoritmos_busca.py | algoritmos_busca.py/...
124 def busca_a_estrela(grafo, inicio, objetivo, heuristica=None, evitar: list[str] = []):
125     """
126     Implementação corrigida do A*.
127     """
128     if inicio not in grafo or objetivo not in grafo:
129         print(f"Node inicial {inicio} ou objetivo {objetivo} não encontrado no grafo")
130         return None
131
132     if heuristica is None:
133         heuristica = calcular_heuristica(grafo, objetivo)
134
135     open_list = {inicio}
136     closed_list = set()
137
138     g = {inicio: 0}
139     parents = {inicio: None}
140
141     while open_list:
142         n = min(open_list, key=lambda x: g[x] + heuristica[x])
143
144         if n == objetivo:
145             path = []
146             while n is not None:
147                 path.append(n)
148                 n = parents[n]
149             path.reverse()
150             return path
151
152         open_list.remove(n)
153         closed_list.add(n)
154
155         for vizinho in sorted(grafo.neighbors(n)):
156             if grafo[n][vizinho].get('bloqueado', False) or grafo.nodes[vizinho].get("tipo_terreno", None) in evitar:
157                 continue
158
159             tentative_g = g[n] + grafo[n][vizinho]['custo']
160
161             if vizinho in closed_list and tentative_g >= g.get(vizinho, float('inf')):
162                 continue
163
164             if vizinho not in open_list or tentative_g < g.get(vizinho, float('inf')):
165                 parents[vizinho] = n
166                 g[vizinho] = tentative_g
167                 if vizinho not in open_list:
168                     open_list.add(vizinho)
169
170     print(f"Não foi encontrado caminho entre {inicio} e {objetivo}")
171     return None
```

Figura 8 - Implementação do algoritmo A\*

Este algoritmo considera múltiplos fatores, como o custo do deslocamento, a prioridade da zona, restrições dos veículos (autonomia, compatibilidade com o terreno) e as janelas de tempo críticas.

- **Testagem de algoritmos de procura**

Após a testagem de todos os algoritmos, verifica-se que o algoritmo A\* é a melhor escolha, como tínhamos previsto.

```
=== Avaliação de Algoritmos ===  
  
A testar Busca em Largura...  
Resultados para Busca em Largura:  
- Tempo médio de execução: 0.0002 segundos  
- Tempo da rota: 1.08 minutos  
- Custo da rota: 9.50 unidades  
  
A testar Busca em Profundidade...  
Resultados para Busca em Profundidade:  
- Tempo médio de execução: 0.0002 segundos  
- Tempo da rota: 1.08 minutos  
- Custo da rota: 9.50 unidades  
  
A testar Busca Gulosa...  
Resultados para Busca Gulosa:  
- Tempo médio de execução: 0.0000 segundos  
- Tempo da rota: 1.08 minutos  
- Custo da rota: 9.50 unidades  
  
A testar A*...  
Resultados para A*:  
- Tempo médio de execução: 0.0000 segundos  
- Tempo da rota: 1.00 minutos  
- Custo da rota: 7.93 unidades  
  
Melhor algoritmo escolhido: A*  
Score final: 0.7235
```

Figura 9 - Testagem de algoritmos de procura

- **Condições dinâmicas**

No decorrer do projeto, foram implementadas condições meteorológicas adversas. O custo, o tempo e o nível do bloqueio variam consoante o impacto destas condições.

```
condicoes_meteorologicas.py 1 condicoes_meteorologicas.py/...  
6 class CondicaoMeteorologica(Enum):  
7     NORMAL = "normal"  
8     CHUVA_LEVE = "chuva_leve"  
9     CHUVA_FORTE = "chuva_forte"  
10    NEVOEIRO = "nevoeiro"  
11    TEMPESTADE = "tempestade"  
12    NEVE = "neve"  
13  
14 class GestorMeteorologico:  
15     def __init__(self, grafo: nx.DiGraph):  
16         self.grafo = grafo  
17         self.condicoes_por_regiao = {}  
18         self.multiplicadores = {  
19             CondicaoMeteorologica.NORMAL: {  
20                 'custo': 1.0,  
21                 'tempo': 1.0,  
22                 'bloqueio': 0.0  
23             },  
24             CondicaoMeteorologica.CHUVA_LEVE: {  
25                 'custo': 1.1,  
26                 'tempo': 1.1,  
27                 'bloqueio': 0.05  
28             },  
29             CondicaoMeteorologica.CHUVA_FORTE: {  
30                 'custo': 1.1,  
31                 'tempo': 1.3,  
32                 'bloqueio': 0.15  
33             },  
34             CondicaoMeteorologica.NEVOEIRO: {  
35                 'custo': 1.3,  
36                 'tempo': 1.8,  
37                 'bloqueio': 0.1  
38             },  
39             CondicaoMeteorologica.TEMPESTADE: {  
40                 'custo': 1.7,  
41                 'tempo': 2,  
42                 'bloqueio': 0.25  
43             },  
44             CondicaoMeteorologica.NEVE: {  
45                 'custo': 1.8,  
46                 'tempo': 1.8,  
47                 'bloqueio': 0.2  
48             }  
49         }  
50         self.valores_originais = {  
51             (u, v): {  
52                 'custo': data['custo'],  
53                 'tempo': data['tempo']  
54             }  
55             for u, v, data in self.grafo.edges(data=True)  
56         }  
57         self.inicializar_condicoes()
```

```
34         CondicaoMeteorologica.NEVOEIRO: {  
35             'custo': 1.3,  
36             'tempo': 1.8,  
37             'bloqueio': 0.1  
38         },  
39         CondicaoMeteorologica.TEMPESTADE: {  
40             'custo': 1.7,  
41             'tempo': 2,  
42             'bloqueio': 0.25  
43         },  
44         CondicaoMeteorologica.NEVE: {  
45             'custo': 1.8,  
46             'tempo': 1.8,  
47             'bloqueio': 0.2  
48         }  
49     }  
50     self.valores_originais = {  
51         (u, v): {  
52             'custo': data['custo'],  
53             'tempo': data['tempo']  
54         }  
55         for u, v, data in self.grafo.edges(data=True)  
56     }  
57     self.inicializar_condicoes()
```

Figuras 10 e 11 – Impacto das condições meteorológicas

Após esta implementação, foi decidido pelo grupo que se simulariam também alguns eventos imprevisíveis, de forma a refletir essa realidade num cenário de emergência. São eles a existência de obstáculos (como inundações ou deslizamentos de terra) e a existência de

eventos imprevisíveis (como falhas na comunicação ou alguma falha estrutural). Estas condições impactam, nomeadamente, os custos das rotas e a velocidade dos veículos (afetando o tempo de resposta), assim como a acessibilidade às áreas afetadas.

```
eventos_dinamicos.py 1 eventos_dinamicos.py/GestorEventos/_init_  
  
6 class TipoObstaculo(Enum):  
7     INUNDACAO = "inundacao"  
8     DESLIZAMENTO = "deslizamento"  
9     QUEDA_ARVORES = "queda_arvores"  
10    EROSAO = "erosao"  
11    DESMORONAMENTO = "desmoronamento"  
12  
13 class TipoEvento(Enum):  
14    FALHA_COMUNICACAO = "falha_comunicacao"  
15    EVACUACAO = "evacuacao"  
16    RESGATE_EM_ANDAMENTO = "resgate_em_andamento"  
17    OBRA_EMERGENCIA = "obra_emergencia"  
18    FALHA_ESTRUTURAL = "falha_estrutural"
```

Figura 12 – Criação de eventos dinâmicos

```
eventos_dinamicos.py 1 eventos_dinamicos.py/GestorEventos/_init_  
  
20 class GestorEventos:  
21     def __init__(self, grafo: nx.DiGraph):  
22  
30         self.multiplicadores_obstaculos = {  
31             TipoObstaculo.INUNDACAO: {  
32                 'custo': 1.2,  
33                 'tempo': 1.6,  
34                 'duracao': (72, 240),  
35                 'prob_remocao': 0.05  
36             },  
37             TipoObstaculo.DESLIZAMENTO: {  
38                 'custo': 1.25,  
39                 'tempo': 1.9,  
40                 'duracao': (48, 168),  
41                 'prob_remocao': 0.03  
42             },  
43             TipoObstaculo.QUEDA_ARVORES: {  
44                 'custo': 1.1,  
45                 'tempo': 1.3,  
46                 'duracao': (24, 72),  
47                 'prob_remocao': 0.2  
48             },  
49             TipoObstaculo.EROSAÇÃO: {  
50                 'custo': 1.15,  
51                 'tempo': 1.6,  
52                 'duracao': (36, 120),  
53                 'prob_remocao': 0.1  
54             },  
55             TipoObstaculo.DESMORONAMENTO: {  
56                 'custo': 1.3,  
57                 'tempo': 2,  
58                 'duracao': (72, 240),  
59                 'prob_remocao': 0.02  
60             }  
61         }  
62     }
```

Figura 13 – Impacto dos eventos dinâmicos (obstáculos)

## Resultados obtidos

Os objetivos impostos pelo enunciado passaram por formular um problema de procura (de forma a otimizar a distribuição de alimentos em zonas afetadas por uma catástrofe natural), gerar um grafo de estados, implementar algoritmos de procura informada e não informada e criar condições dinâmicas e imprevisíveis que pudessem acontecer neste cenário de emergência.

O grafo criado modelou eficientemente a realidade do problema, com nodos a representar bases, *hubs* intermediários, postos de reabastecimento e zonas afetadas, assim como as arestas representativas das conexões (com custos e tempos respetivos).

A implementação e testagem dos algoritmos, bem como a simulação de condições dinâmicas, permite-nos tirar algumas conclusões. Na procura não informada, o algoritmo BFS, embora tivesse garantido soluções válidas, revelou-se ineficiente em cenários grandes e dinâmicos. O DFS mostrou ser limitado em eficiência, frequentemente explorando caminhos irrelevantes em cenários complexos. Nos algoritmos de procura informada, a procura gulosa foi útil para priorizar zonas críticas rapidamente, mas com custos elevados devido à falta de otimização geral. Já o algoritmo A\* demonstrou ser o mais eficiente, equilibrando custo total, tempo e atendimento às prioridades das zonas afetadas, mesmo em condições adversas, como tempestades ou rotas ineficientes.

Quanto ao impacto das condições dinâmicas, como bloqueios e mudanças meteorológicas, foi simulado com realismo, permitindo testar a resiliência das soluções propostas.

Em jeito de conclusão, o trabalho demonstrou a eficácia de algoritmos de procura em resolver problemas complexos e dinâmicos de distribuição de suprimentos em cenários de emergência. A modelagem com grafos e a integração de condições variáveis tornaram o sistema adaptável e eficiente, destacando o A\* como a melhor abordagem para esse contexto.

Estes resultados reforçam a importância de considerar fatores dinâmicos, como o clima e bloqueios, e de utilizar heurísticas robustas para otimizar soluções em problemas reais. Com melhorias futuras, como a inclusão de mais tipos de veículos ou heurísticas personalizadas, o sistema pode ser ainda mais refinado, garantindo respostas rápidas e eficientes em cenários de emergência.