

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Inteligência Artificial

Trabalho prático

Dezembro 2024

Desenvolvido por:

Rodrigo Ferreira - a104531
Rodrigo Fernandes - a104175
Diogo Esteves - a104004

1 Avaliação do Grupo

Rodrigo Miguel Granja Ferreira (a104531) = 0

Diogo José Fernandes Esteves (a104004) = 0

Rodrigo Oliveira Fernandes (a104175) = 0

Índice

1 Avaliação do Grupo

2 Introdução

3 Descrição do problema

3.1	map.csv
3.2	vehicles.csv
3.3	medicine.csv
3.4	food.csv
3.5	equipment.csv

4 Formulação do Problema

4.1	Representação do Estado
4.2	Estado Inicial
4.3	Teste Objetivo
4.4	Ações
4.5	Custo da Solução

5 Tarefas e Decisões

5.1	Tarefas Principais
5.2	Decisões

6 Implementação

6.1	Pesquisas não informadas
6.1.1	DFS
6.1.2	BFS
6.1.3	Custo Uniforme
6.1.4	Iterative Deepening Search (IDS) e Depth-Limited Search (DLS)
6.2	Pesquisas informadas
6.2.1	Greedy
6.2.2	A*
6.2.3	SMA*

7 Interface

8 Discussão dos Resultados

9 Conclusão

2 Introdução

O projeto aborda o desenvolvimento de uma estratégia para a distribuição eficiente de suprimentos essenciais em áreas afetadas por catástrofes naturais. A tarefa é caracterizada por desafios logísticos complexos, como a variação na gravidade das zonas, a partir das necessidades, a população e a acessibilidade geográfica, agravados por condições meteorológicas adversas e dinâmicas. Para enfrentar essas dificuldades, é necessário criar um modelo que priorize zonas críticas, levando em conta critérios como gravidade da situação e número de pessoas afetadas, ao mesmo tempo em que otimiza o uso de recursos, quer estes sejam os suprimentos em si, quer sejam os recursos dos meios de transporte.

O sistema proposto deverá incorporar a alocação e o planejamento de rotas para diferentes tipos de veículos de transporte (drones, helicópteros, barcos, caminhões, entre outros), cada um com capacidades específicas, como carga máxima, autonomia e tempos de viagem. A estratégia deve minimizar o desperdício de recursos, como alimentos perecíveis e combustível, e lidar com restrições operacionais, incluindo limitações de acesso devido a estradas bloqueadas ou destruídas.

Além disso, o modelo precisa integrar um sistema de prioridades, assegurando que as regiões mais necessitadas sejam atendidas com rapidez, e deve ser capaz de adaptar-se às mudanças constantes nas condições ambientais. A solução ideal combina elementos de otimização de rotas, alocação eficiente de recursos e tomada de decisão baseada em dados dinâmicos, visando maximizar a cobertura das zonas afetadas, reduzir desperdícios e garantir o maior impacto humanitário possível. O resultado esperado é um sistema que contribua significativamente para operações de socorro eficazes em cenários de desastres naturais, onde o tempo e a eficiência são fatores críticos para salvar vidas.

3 Descrição do problema

O ambiente com que lidamos é um ambiente não determinístico e parcialmente acessível, definindo assim um problema de contingência. A evolução do ambiente não pode ser determinada de forma única a partir das ações do agente sobre o mesmo. Por exemplo, as condições de acesso a uma determinada área podem variar consoante as condições da estrada e da meteorologia, podendo surgir bloqueios inesperados em determinadas rotas, que impedem o agente (veículo) de proceder por certas rotas. Assim, as ações do agente não afetam diretamente o estado do ambiente e o mesmo não tem acesso a todas as informações acerca do ambiente, devido às condições dinâmicas do mesmo ao longo do tempo, tornando-o assim não determinístico e parcialmente acessível.

O trabalho aborda a otimização da distribuição desses suprimentos. Utilizamos algoritmos de procura (tanto informada quanto não informada) para maximizar a eficiência da operação.

Neste contexto, modelámos as áreas de entrega e as rotas disponíveis como um grafo, onde cada nó representa uma zona específica e as arestas indicam os caminhos possíveis. A solução considera limitações como capacidade de transporte e rotas bloqueadas.

O trabalho foi inspirado no mapa mundo, utilizando coordenadas geográficas reais para tornar o problema mais próximo de situações reais.

Para garantir a modularidade e a reutilização do código, foram criadas classes específicas para representar os diferentes componentes do problema, como *graph*, *node*, *area*, *supply*, *vehicle*, *travel* e *algs_handler*.

Para suportar a modelação e a resolução do problema, foram utilizados ainda ficheiros CSV para armazenar os dados necessários à execução do sistema. Abaixo estão descritos os ficheiros utilizados, seus formatos e exemplos de dados:

3.1 map.csv

Este ficheiro contém informações geográficas e demográficas de diversos países, incluindo suas coordenadas (latitude e longitude), países adjacentes, população, região e um índice de acessibilidade. O objetivo é fornecer uma visão geral sobre a localização e as relações geopolíticas entre os países listados, bem como dados sobre sua população e acessibilidade.

Nome do campo	Descrição	Exemplo
País	Nome do país	Brasil
Latitude	Latitude geográfica	-14.2350
Longitude	Longitude geográfica	-51.9253
Região	Região geopolítica	América do Sul
População	População estimada	211049527
Países Adjacentes	Lista de países vizinhos	Argentina, Paraguai, Bolívia
Acessibilidade	Índice de acessibilidade	0.85

Table 1: map.csv

3.2 vehicles.csv

Este ficheiro contém informações sobre diferentes meios de transporte, incluindo sua capacidade máxima, autonomia, consumo médio, tempo de viagem e o tipo de transporte (aéreo, aquático ou terrestre). O objetivo é fornecer uma visão geral das características de cada veículo, como sua eficiência e o tipo de ambiente em que pode operar.

Nome do campo	Descrição	Exemplo
Nome	Nome do veículo	Camião
MaxCapacity	Capacidade máxima do veículo	20000
Autonomy	Autonomia do veículo	1200
AverageConsumption	Consumo médio de combustível	0.35
TravelTime	Tempo de viagem estimado	3
Type	Tipo de transporte	Aéreo

Table 2: vehicles.csv

3.3 medicine.csv

Este ficheiro contém informações sobre diversos medicamentos, incluindo o peso da embalagem e a vida útil (*Shelf Life*) de cada produto. O objetivo é fornecer detalhes sobre as características de cada medicamento, como sua durabilidade e o peso da embalagem.

Nome do campo	Descrição	Exemplo
Medicamento	Nome do medicamento	Paracetamol
PesoPack	Peso da embalagem do medicamento (em kg)	10
ShelfLife	Vida útil do medicamento (em minutos)	15001

Table 3: medicine.csv

3.4 food.csv

Este ficheiro contém informações sobre alimentos, incluindo o peso da embalagem e a vida útil de cada produto. O campo ‘ShelfLife’ indica o número de dias até a expiração do produto, sendo que o valor ”inf” representa produtos com validade indefinida ou sem data de validade conhecida.

Nome do campo	Descrição	Exemplo
Comida	Nome do alimento	Arroz
PesoPack	Peso da embalagem do alimento (em kg)	20
ShelfLife	Vida útil do alimento (em minutos) ou ”inf” se indefinido	50987

Table 4: food.csv

3.5 equipment.csv

Este ficheiro contém informações sobre diversos equipamentos, incluindo o peso da embalagem de cada item. O objetivo é fornecer detalhes sobre os equipamentos, como o peso das embalagens, para facilitar o armazenamento e o transporte.

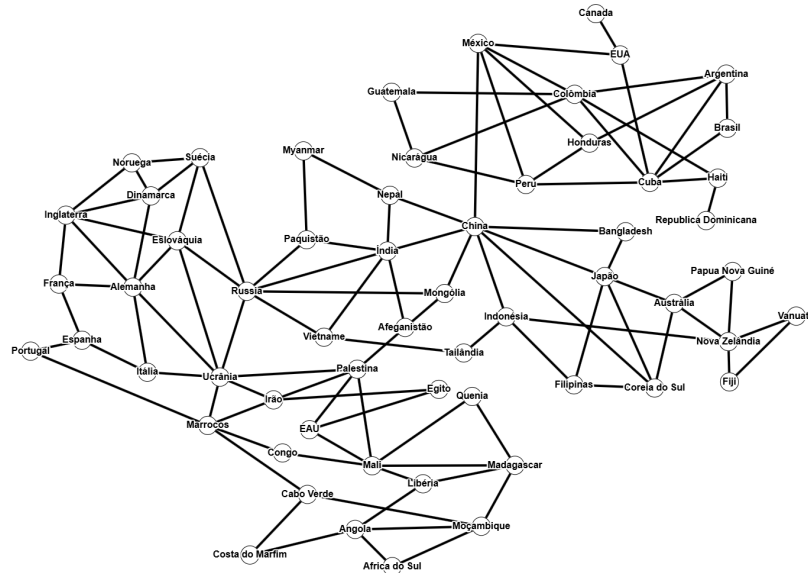
Nome do campo	Descrição	Exemplo
Equipamento	Nome do equipamento	Lanterna
PesoPack	Peso da embalagem do equipamento (em kg)	10

Table 5: equipment.csv

4 Formulação do Problema

4.1 Representação do Estado

A representação do ambiente e do estado atual de cada agente pode ser feita através da representação do ambiente em forma de grafo. Assim, definindo as áreas do nosso ambiente como sendo as 20 localizações a nível global com a maior taxa de catástrofes naturais, obtemos o seguinte grafo:



4.2 Estado Inicial

Para formulação do problema de forma coerente, é necessário definir um estado inicial do ambiente e dos agentes, para que tenhamos um ponto de partida. Assim, este estado inicial deverá ter o seguinte contexto:

- Existe um conjunto de zonas afetadas por uma catástrofe, cada uma com a sua gravidade, população, necessidades e condições de acesso.
- Existem veículos disponíveis com características específicas (capacidade de carga, autonomia de combustível, velocidade em condições ideais) localizados no ponto de partida.
- As condições geográficas e meteorológicas iniciais de cada zona são conhecidas.

4.3 Teste Objetivo

Todos os suprimentos essenciais são entregues a tempo nas zonas de maior necessidade, considerando as restrições de janela de tempo e capacidade operacional dos veículos.

4.4 Ações

No que toca à interação com o ambiente, os agentes realizam ações, tais como:

- Escolha da rota: definição da sequência de zonas a serem visitadas pelo veículo escolhido, considerando acessibilidade e prioridades.
- Carregamento de suprimentos: alocação de recursos ao veículo, respeitando a sua capacidade de carga.
- Realizar entrega: concluir a entrega em uma zona, atualizando a quantidade de recursos disponíveis e o tempo restante.
- Reabastecimento: Enviar o veículo para reabastecer combustível ou carregar mais suprimentos, se necessário.
- Alterar rota: Alterar rota devido a condições meteorológicas e bloqueios.

4.5 Custo da Solução

Para que possamos escolher com critério qual as melhores rotas a seguir, é necessário definir um custo associado a cada rota, tendo em conta os critérios que afetam diretamente a rota de um veículo. Assim, podemos definir sendo a distância total do percurso feito pelo agente.

5 Tarefas e Decisões

Nesta seção, detalhamos as principais tarefas que o sistema deve executar para alcançar os objetivos estabelecidos.

5.1 Tarefas Principais

- **Alocação de Recursos:** Distribuição eficiente dos suprimentos essenciais entre as zonas afetadas.
- **Planeamento de Rotas:** Definição das rotas mais eficientes para os veículos, considerando restrições geográficas e meteorológicas.
- **Monitorização de Condições:** Atualização contínua das condições geográficas e meteorológicas para adaptar as operações em tempo real.
- **Gestão de Veículos:** Controlo da disponibilidade, capacidade dos veículos utilizados nas operações.

5.2 Decisões

- **Seleção dos Veículos:** Escolher os veículos mais adequado com base na capacidade de carga, autonomia e condições atuais.
- **Escolha da Rota:** Determinar a sequência de zonas a serem visitadas, priorizando áreas com maior necessidade e acessibilidade.
- **Supply:** Decidir a quantidade de suprimentos a serem alocados a cada veículo, respeitando suas capacidades.
- **Realização de Entregas:** Decidir quando e onde realizar as entregas, atualizando o *stock* de recursos e o tempo disponível.
- **Reabastecimento e Manutenção:** Decidir quando enviar veículos para reabastecimento de combustível ou manutenção, garantindo a continuidade das operações.
- **Adaptação de Rotas:** Alterar rotas em resposta a mudanças nas condições ambientais ou bloqueios inesperados.

6 Implementação

Apresentámos as várias estratégias de procura implementadas no nosso trabalho, sendo que posteriormente apresentaremos resultados de cada uma face a um mesmo input no nosso programa, como termo de comparação.

6.1 Pesquisas não informadas

6.1.1 DFS

A pesquisa em profundidade (DFS) é um algoritmo de busca utilizado para explorar grafos, percorrendo-os o mais fundo possível ao longo de um único caminho antes de retroceder. A partir de um nó inicial, o algoritmo visita recursivamente todos os vizinhos de cada nó antes de explorar os vizinhos desses, avançando até atingir o nó de destino ou esgotar as possibilidades de caminho. No contexto do código apresentado, a DFS é utilizada para encontrar o melhor caminho entre dois pontos, levando em consideração as necessidades de carga e consumo de combustível do veículo, além do tempo de viagem e da compatibilidade entre os nós no grafo.

```
def procura_DFS(self, start, end, path, visited, vehicle: Vehicle):
    path.append(start)
    visited.add(start)
    minimum = 0
    distance = 0
    needs = self.get_node_by_name(end).getNeeds()

    for need in needs:
        minimum += need.getSupplyWeightLoad()

    if start == end:
        custoT = self.calculate_cost(path)
        area = self.get_node_by_name(end)
        refuel = area.getRefuel()
        vehicle.updateVehicle(distance, needs, True, refuel)
        self.finish_travel(end)
        return path, custoT

    for adjacente, custo in self.graph[start]:
        if adjacente not in visited and self.compatibleConection(start, adjacente, vehicle) and
custo != -1:
            distance = self.calculate_distance(
                self.get_node_by_name(start).getLatitude(),
                self.get_node_by_name(start).getLongitude(),
                self.get_node_by_name(adjacente).getLatitude(),
                self.get_node_by_name(adjacente).getLongitude()
            )
            time = vehicle.calculateTravelTime(distance)

            refuel = self.get_node_by_name(adjacente).getRefuel()
            self.update_grafo(time)
            if vehicle.updateVehicle(distance, needs, False, refuel) is False:
                break

            area = self.get_node_by_name(adjacente).getArea()
            resultado = self.procura_DFS(adjacente, end, path, visited, vehicle)
            if resultado[1] is not float('inf'):
                return resultado

    path.pop()
    return [], float('inf')
```

Figure 1: DFS

6.1.2 BFS

A pesquisa em largura (BFS) é um algoritmo de busca utilizado para explorar grafos, visitando os nós de forma iterativa, nível por nível. Começa no nó inicial, visita todos os seus vizinhos, depois passa para os vizinhos dos vizinhos e assim por diante, até encontrar o nó de destino ou explorar todos os caminhos possíveis. Ao contrário da DFS, que vai o mais fundo possível em um caminho antes de retroceder, o BFS explora primeiro os caminhos mais curtos antes de avançar para os mais longos. No contexto do código apresentado, o BFS é utilizado para encontrar o caminho mais eficiente entre dois pontos, levando em consideração a compatibilidade entre os nós, as necessidades de carga e o consumo de combustível do veículo, além do tempo de viagem necessário. A utilização de uma fila para fazer a gestão dos nós a serem visitados permite que a busca seja feita de maneira ordenada, garantindo a exploração de caminhos mais curtos primeiro.

```

def procura_BFS(self, start, end, vehicle):
    visited = set()
    fila = Queue()

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

    parent = dict()
    parent[start] = None

    minimum = 0
    distance = 0
    needs = self.get_node_by_name(end).getNeeds()

    for need in needs:
        minimum += need.getSupplyWeightLoad()

    path_found = False
    while not fila.empty() and path_found == False:
        nodo_atual = fila.get()
        if nodo_atual == end:
            path_found = True
            area = self.get_node_by_name(end)
            refuel = area.getRefuel()
            vehicle.updateVehicle(distance, needs, True, refuel)
            self.finish_travel(end)
        else:
            for (adjacente, peso) in self.graph[nodo_atual]:
                if adjacente not in visited and self.compatibleConnection(nodo_atual, adjacente,
vehicle) and peso != -1:
                    fila.put(adjacente)
                    parent[adjacente] = nodo_atual
                    visited.add(adjacente)
                    distance = self.calculate_distance(
                        self.get_node_by_name(start).getLatitude(),
                        self.get_node_by_name(adjacente).getLatitude(),
                        self.get_node_by_name(adjacente).getLongitude()
                    )
                    time = vehicle.calculateTravelTime(distance)
                    refuel = self.get_node_by_name(adjacente).getRefuel()
                    self.update_grafo(time)
                    if vehicle.updateVehicle(distance, needs, False, refuel) is False:
                        break
                    self.update_grafo(time)

    path = []
    custoT = float('inf')
    if path_found:
        path.append(end)
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse()
        custoT = self.calculate_cost(path)
    return (path, custoT)

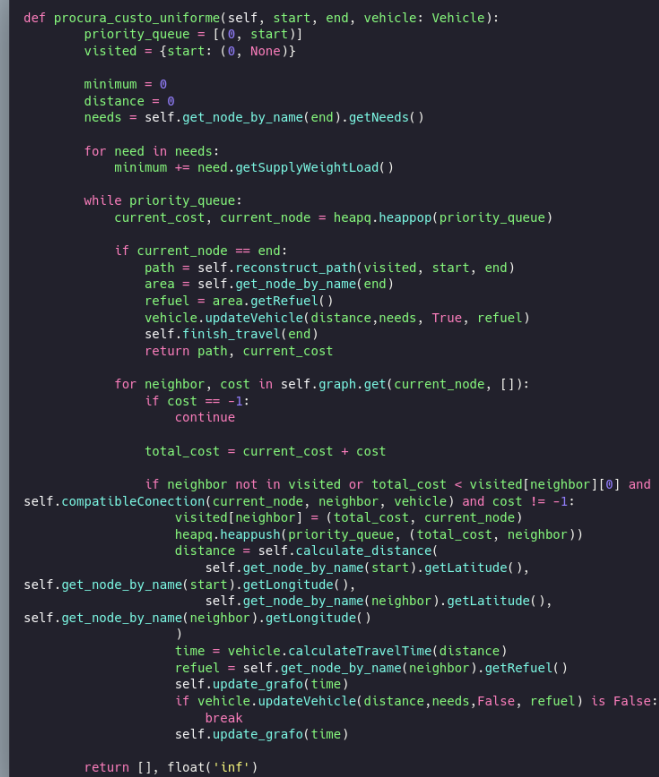
```

Figure 2: BFS

6.1.3 Custo Uniforme

A busca de custo uniforme é um algoritmo de busca que visa encontrar o caminho de menor custo entre dois pontos em um grafo. Diferente de algoritmos como o DFS e BFS, que se concentram na exploração do grafo de maneira profunda ou ampla, respectivamente, a busca de custo uniforme prioriza os caminhos de menor custo, explorando nós de acordo com o custo acumulado até o momento. O algoritmo utiliza uma fila de prioridade, onde os nós com menor custo total

são explorados primeiro, garantindo que o caminho encontrado seja o de custo mínimo. No código apresentado, a busca de custo uniforme é utilizada para encontrar o melhor caminho entre dois pontos, considerando as necessidades de carga, o consumo de combustível e o tempo de viagem do veículo. Além disso, é verificada a compatibilidade entre os nós e o veículo antes de explorar o próximo nó. A busca termina assim que o nó de destino é encontrado, retornando o caminho e o custo acumulado.



```
def procura_custo_uniforme(self, start, end, vehicle: Vehicle):
    priority_queue = [(0, start)]
    visited = {start: (0, None)}

    minimum = 0
    distance = 0
    needs = self.get_node_by_name(end).getNeeds()

    for need in needs:
        minimum += need.getSupplyWeightLoad()

    while priority_queue:
        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node == end:
            path = self.reconstruct_path(visited, start, end)
            area = self.get_node_by_name(end)
            refuel = area.getRefuel()
            vehicle.updateVehicle(distance, needs, True, refuel)
            self.finish_travel(end)
            return path, current_cost

        for neighbor, cost in self.graph.get(current_node, []):
            if cost == -1:
                continue

            total_cost = current_cost + cost

            if neighbor not in visited or total_cost < visited[neighbor][0] and
self.compatibleConnection(current_node, neighbor, vehicle) and cost != -1:
                visited[neighbor] = (total_cost, current_node)
                heapq.heappush(priority_queue, (total_cost, neighbor))
                distance = self.calculate_distance(
                    self.get_node_by_name(start).getLatitude(),
self.get_node_by_name(start).getLongitude(),
                    self.get_node_by_name(neighbor).getLatitude(),
self.get_node_by_name(neighbor).getLongitude()
                )
                time = vehicle.calculateTravelTime(distance)
                refuel = self.get_node_by_name(neighbor).getRefuel()
                self.update_grafo(time)
                if vehicle.updateVehicle(distance, needs, False, refuel) is False:
                    break
                self.update_grafo(time)

    return [], float('inf')
```

Figure 3: Custo Uniforme

6.1.4 Iterative Deepening Search (IDS) e Depth-Limited Search (DLS)

O algoritmo Depth-Limited Search (DLS) é uma técnica de busca que explora o grafo até uma profundidade máxima definida. Ele é especialmente útil para evitar loops em grafos cíclicos, limitando a profundidade de exploração. No código apresentado, o DLS percorre os vizinhos de um nó, verificando as restrições

de compatibilidade entre os nós, consumo de combustível e necessidades de reabastecimento. Caso o nó de destino seja alcançado, o custo e o caminho são retornados, atualizando também o estado do veículo.

O Iterative Deepening Search (IDS) combina a profundidade limitada do DLS com uma busca em profundidade iterativa. Ele incrementa gradualmente o limite de profundidade até encontrar uma solução ou atingir o limite máximo especificado. Este método permite uma busca completa e com menor consumo de memória do que o algoritmo A*, embora seja menos eficiente em termos de tempo.

No código, o IDS chama repetidamente o DLS com limites de profundidade crescentes. Caso uma solução seja encontrada em um dos níveis, o caminho e o custo são retornados.

```
def depth_limited_search(self, start, goal, depth_limit, vehicle: Vehicle):
    def dls_recursive(current, goal, depth, visited, path, vehicle):
        if depth < 0:
            return [], float('inf')

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

        # Check if goal is reached
        if current == goal:
            cost = self.calculate_cost(path)
            area = self.get_node_by_name(goal)
            refuel = area.getRefuel()

            # Get the distance for vehicle update
            if len(path) > 1:
                distance = self.calculate_distance(
                    self.get_node_by_name(path[-2]).getLatitude(),
                    self.get_node_by_name(path[-2]).getLongitude(),
                    self.get_node_by_name(current).getLatitude(),
                    self.get_node_by_name(current).getLongitude()
                )
            else:
                distance = 0

            needs = self.get_node_by_name(goal).getNeeds()
            vehicle.updateVehicle(distance, needs, True, refuel)
            self.finish_travel(goal)
            return path, cost

        # Explore neighbors within depth limit
        for adjacente, custo in self.graph[current]:
            if (adjacente not in visited and
                self.compatibleConnection(current, adjacente, vehicle) and
                custo != -1):

                # Calculate distance and update vehicle state
                distance = self.calculate_distance(
                    self.get_node_by_name(current).getLatitude(),
                    self.get_node_by_name(current).getLongitude(),
                    self.get_node_by_name(adjacente).getLatitude(),
                    self.get_node_by_name(adjacente).getLongitude()
                )

                time = vehicle.calculateTravelTime(distance)
                refuel = self.get_node_by_name(adjacente).getRefuel()
                self.update_grafo(time)

                needs = self.get_node_by_name(goal).getNeeds()
                if vehicle.updateVehicle(distance, needs, False, refuel) is False:
                    continue

                result_path, result_cost = dls_recursive(adjacente, goal, depth - 1,
                    visited.copy(), path.copy(), vehicle)
                if result_cost != float('inf'):
                    return result_path, result_cost

        return [], float('inf')

    # Initialize empty path and visited set
    initial_path = []
    initial_visited = set()

    return dls_recursive(start, goal, depth_limit, initial_visited, initial_path, vehicle)

def IDS(self, start, goal, max_depth, vehicle: Vehicle):
    for depth in range(max_depth):
        path, cost = self.depth_limited_search(start, goal, depth, vehicle)
        if cost != float('inf'):
            return path, cost

    return [], float('inf')
```

Figure 4: Algoritmo Iterative Deepening Search (IDS) e Depth-Limited Search (DLS)

6.2 Pesquisas informadas

6.2.1 Greedy

O algoritmo *Greedy* (ou guloso) é uma abordagem de procura que toma decisões baseadas na escolha local ótima em cada etapa, com a esperança de que isso levará a uma solução global ótima. No contexto de grafos, o algoritmo seleciona o próximo nó a ser explorado com base em uma heurística, que é uma estimativa do custo restante até o destino. O algoritmo *Greedy* não garante a obtenção da solução ótima, mas pode ser eficiente em termos de tempo, especialmente em grandes grafos.

No código apresentado, o algoritmo *Greedy* é utilizado para encontrar um caminho entre dois pontos, priorizando os nós que possuem a menor heurística (uma medida de quão "próximo" o nó está do destino). O algoritmo considera as necessidades de carga e o consumo de combustível, além de verificar a compatibilidade entre os nós e o veículo antes de avançar para o próximo nó. O caminho é reconstruído ao final, e o veículo é atualizado com a distância percorrida e o reabastecimento necessário. Se o destino não for alcançado, o algoritmo retorna que o caminho não existe.


```

def greedy(self, start, end, vehicle: Vehicle):
    open_list = set([start])
    closed_list = set([])

    parents = {}
    parents[start] = start

    minimum = 0
    distance = 0
    needs = self.get_node_by_name(end).getNeeds()

    for need in needs:
        minimum += need.getSupplyWeightLoad()

    while len(open_list) > 0:
        n = None

        for v in open_list:
            node = self.get_node_by_name(v)
            if n is None or node.getHeuristic() < node.getHeuristic():
                n = v

        if n is 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()
            area = self.get_node_by_name(end)
            refuel = area.getRefuel()
            vehicle.updateVehicle(distance, needs, True, refuel)
            self.finish_travel(end)
            return (reconst_path, self.calculate_cost(reconst_path))

        for (m, weight) in self.getNeighbours(n):
            if m not in open_list and m not in closed_list and self.compatibleConnection(n, m,
vehicle) and weight != -1:
                open_list.add(m)
                parents[m] = n
                distance = self.calculate_distance(
                    self.get_node_by_name(start).getLatitude(),
                    self.get_node_by_name(m).getLatitude(), self.get_node_by_name(m).getLongitude()
                )
                time = vehicle.calculateTravelTime(distance)
                refuel = self.get_node_by_name(m).getRefuel()
                if vehicle.updateVehicle(distance, needs, False, refuel) is False:
                    break
                self.update_grafo(time)

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

    return [], float('inf')

```

Figure 5: Algoritmo Greedy

6.2.2 A*

O algoritmo A* é uma técnica de busca heurística que combina os benefícios da busca em largura com uma heurística para orientar a busca de maneira mais eficiente. Ele utiliza uma função de avaliação $f(n)=g(n)+h(n)$, onde $g(n)$ é o custo do caminho até o nó N e $h(n)$ é uma estimativa do custo restante até o destino. O algoritmo A* é otimizado para encontrar o caminho mais curto entre dois pontos, considerando tanto o custo acumulado quanto a heurística de aproximação.

No código apresentado, o algoritmo A* é utilizado para buscar o melhor caminho entre um nó de partida e um nó de destino, considerando as necessidades de carga, o consumo de combustível e a compatibilidade entre os nós. O algoritmo mantém duas listas: uma lista aberta, contendo os nós a serem explo-

rados, e uma lista fechada, contendo os nós já explorados. Em cada iteração, o algoritmo escolhe o nó com a menor função de avaliação $f(n)$, avançando para ele e atualizando os valores de custo e heurística. O caminho é reconstruído ao final, e o veículo é atualizado com a distância percorrida e o reabastecimento necessário. Se o destino não for alcançado, o algoritmo retorna que o caminho não existe.

```
def procura_astar(self, start, end, vehicle: Vehicle):
    open_list = [start]
    closed_list = set()

    g = {start: 0}
    parents = {start: None}

    distance = 0
    needs = self.get_node_by_name(end).getNeeds()
    h_start = self.get_node_by_name(start).getHeuristic()

    while open_list:
        n = min(open_list, key=lambda node: g[node] + self.get_node_by_name(node).getHeuristic())

        if n == end:
            path = []
            while n:
                path.append(n)
                n = parents[n]
            path.reverse()

            refuel = self.get_node_by_name(end).getRefuel()
            vehicle.updateVehicle(distance, needs, True, refuel)
            self.findPathTravelled()
            return path, self.calculate_cost(path)

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

        for n, weight in self.getNeighbours(n):
            if not self.compatibleConnection(n, n, vehicle) or weight == -1:
                continue

            g_new = g[n] + weight

            if n in closed_list and g_new >= g.get(n, float('inf')):
                continue

            if n not in open_list or g_new < g.get(n, float('inf')):
                g[n] = g_new
                parents[n] = n

                distance = self.calculate_distance(
                    self.get_node_by_name(start).getLongitude(),
                    self.get_node_by_name(n).getLatitude(), self.get_node_by_name(n).getLongitude()
                )
                time = vehicle.calculateTravelTime(distance)
                refuel = self.get_node_by_name(n).getRefuel()
                if not vehicle.updateVehicle(distance, needs, False, refuel):
                    continue

                self.update_grafo_time()
                open_list.add(n)

    return [], float('inf')
```

Figure 6: Algoritmo A*

6.2.3 SMA*

O algoritmo Simplified Memory-Bounded A* é uma versão adaptada do A* para cenários com restrições de memória. Ele combina a busca heurística eficiente do A* com a capacidade de limitar a quantidade de nós mantidos em memória, garantindo que o consumo de recursos permaneça sob controle, mesmo em grafos densos.

O algoritmo utiliza a função de avaliação $f(n) = g(n) + h(n)$, onde:

- $g(n)$: custo do caminho percorrido até o nó n .
- $h(n)$: estimativa heurística do custo restante até o destino.

No código apresentado, o algoritmo busca o melhor caminho entre um nó inicial e um nó final, considerando restrições como o consumo de combustível,

as necessidades de carga, e a compatibilidade entre os nós. A principal característica desta abordagem é a introdução de um limite de memória, onde nós com maiores valores de $f(n)$ são removidos quando a capacidade é excedida.

O processo ocorre em cinco etapas principais:

1. Inicializa as listas de nós a explorar (aberta) e já explorados (fechada).
2. Seleciona o nó com menor $f(n)$ para expandir.
3. Remove nós com maiores $f(n)$ se o limite de memória for atingido.
4. Atualiza os custos e a heurística para os vizinhos do nó atual.
5. Reconstrói o caminho e atualiza o veículo caso o destino seja alcançado.

Se nenhum caminho for encontrado, o algoritmo retorna que não há solução.

7 Interface

Vamos apresentar como está organizada a interface utilizada durante os testes, ou seja, as escolhas que o utilizador terá para executar as diversas funcionalidades do programa. Nesta estão compreendidas todas as opções necessárias para o utilizador poder executar qualquer função do nosso programa.

```
===== MENU INICIAL =====  
1. Visualizar o grafo  
2. Escolher o tipo de algoritmo  
0. Sair  
=====  
Escolha uma opção: █
```

Figure 7: Menu inicial

Na figura 6 conseguimos observar o menu inicial, com opções de visualizar o grafo e ainda a opção de escolher o tipo de algoritmo de procura. Seguidamente apresenta-se um exemplo, visto que o grafo muda sempre que executámos o programa.

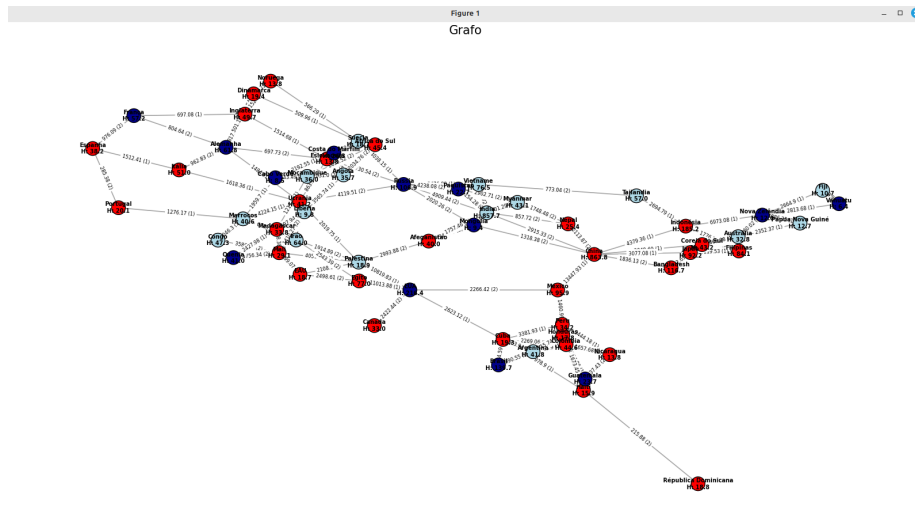


Figure 8: Visualização do grafo

Na Figura 7 visualizamos um mapa onde as áreas a vermelho indicam zonas afetadas com necessidade urgente de suprimentos, enquanto as áreas em tons de azul representam zonas não afetadas. Nestas últimas, distinguimos dois tipos: as de azul escuro, onde é possível realizar reabastecimento de combustível (*refuel*), e as de azul claro, que representam zonas em estado normal. Adicionalmente, é possível observar a heurística. O mapa exibe em cada ligação entre áreas dois valores numéricos essenciais: o custo do trajeto e o tipo de conexão:

- **Tipo 0:** Trajeto acessível por meios aéreos ou terrestres
- **Tipo 1:** Trajeto acessível por meios aéreos ou aquáticos
- **Tipo 2:** Trajeto acessível por todos os meios de transporte



Figure 9: Conexão compatível (tipos)

8 Discussão dos Resultados

Realizámos vários testes de forma a mostrar o comportamento do nosso algoritmo principal utilizando diferentes estratégias de procura face a um mesmo cenário, determinando assim quais as mais eficazes em obter a solução ótima, bem como o tempo e memória que cada uma demora a executar. De seguida, apresentamos os vários resultados ao executar o algoritmo que determina o melhor percurso.

```
Escolha o algoritmo: 1
Executando a Busca em Profundidade...
-----Nepal-----

===== Relatório de Execução =====

Tempo total de execução: 0.2027 segundos
Uso de memória atual: 0.0132 MB
Pico de memória: 0.0150 MB

Resumo por veículo:

Camião:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Barco:
- Destinos alcançados: 2
- Distância total percorrida: 45844.57 km
- Caminho completo: Nepal -> China -> China -> India -> India -> Nepal -> China -> México -> Honduras -> Peru -> Nicarágua -> Guatemala -> Colômbia -> Argentina -> Brasil -> Cuba -> EUA

Drone:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Carro:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Moto:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:
```

Figure 10: Algoritmo DFS

```
Executando Busca em Largura (BFS)...
-----Nepal-----

===== Relatório de Execução =====

Tempo total de execução: 3.0311 segundos
Uso de memória atual: 0.0079 MB
Pico de memória: 0.0143 MB

Resumo por veículo:

Camião:
- Destinos alcançados: 2
- Distância total percorrida: 5020.18 km
- Caminho completo: Nepal -> China -> China -> India

Barco:
- Destinos alcançados: 27
- Distância total percorrida: 454199.66 km
- Caminho completo: Nepal -> China -> México -> EUA -> EUA -> México -> China -> Bangladesh -> Bangladesh -> Japão -> Japão -> China -> India -> Nepal -> Myanmar -> Paquistão -> Rússia -> Vietname -> Tailândia -> Indonésia -> Filipinas -> Filipinas -> Indonésia -> China -> India -> Afeganistão -> Polónia -> Mali -> Egipto -> Egipto -> EUA -> México -> China -> India -> Vietname -> Vietname -> Rússia -> Ucrânia -> Irão -> Irão -> Marrocos -> Portugal -> Espanha -> França -> Inglaterra -> Eslováquia -> Rússia -> Vietname -> Tailândia -> Tailândia -> Vietname -> Rússia -> Ucrânia -> Itália -> Itália -> Ucrânia -> Marrocos -> Cabo Verde -> Moçambique -> África do Sul -> África do Sul -> Angola -> Libéria -> Mali -> Palestina -> Afeganistão -> India -> Myanmar -> Myanmar -> Nepal -> China -> México -> Colômbia -> Colômbia -> México -> China -> India -> Paquistão -> Rússia -> Ucrânia -> Ucrânia -> Marrocos -> Marrocos -> Cabo Verde -> Costa do Marfim -> Costa do Marfim -> Cabo Verde -> Marrocos -> Ucrânia -> Rússia -> Paquistão -> Paquistão -> Rússia -> Ucrânia -> Marrocos -> Portugal -> Portugal -> Marrocos -> Irão -> Palestina -> EUA -> México -> Colômbia -> Mali -> Mali -> República Dominicana -> República Dominicana -> Mali -> Colômbia -> México -> Honduras -> Honduras -> México -> China -> India -> Paquistão -> Rússia -> Eslováquia -> Eslováquia -> Rússia -> Suécia -> Noruega -> Noruega -> Inglaterra -> Alemanha -> Ucrânia -> Irão -> Palestina -> Palestina -> EUA -> México -> China -> Indonésia -> Indonésia -> Nova Zelândia -> Nova Zelândia -> Indonésia -> China -> India -> Afeganistão -> Botsuana -> Botsuana -> India -> China -> Indonésia -> Nova Zelândia -> Fiji -> Fiji -> Vanuatu

Drone:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Carro:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:
```

Figure 11: Algoritmo BFS

```

Executando Busca de Custo Uniforme...
-----Nepal-----

===== Relatório de Execução =====

Tempo total de execução: 3.111 segundos
Tamanho da memória atual: 8.0035 MB
Pico de memória: 8.0078 MB

Resumo por veículo:

Camêlo:
  Destinos alcançados: 3
  Distância total percorrida: 45844.57 km
  Caminho completo: Nepal -> China -> China -> Índia -> Índia -> Nepal -> China -> México -> Honduras -> Peru -> Nicarágua -> Guatemala -> Colômbia -> Argentina -> Brasil -> Cuba -> EUA

Gorila:
  Destinos alcançados: 11
  Distância total percorrida: 146222.50 km
  Caminho completo: Nepal -> China -> Bangladesh -> Bangladesh -> Japão -> Japão -> Filipinas -> Filipinas -> Coreia do Sul -> China -> Índia -> Afeganistão -> Palestina -> Iraque -> Egito -> Iraque -> Afeganistão -> Índia -> Vietnã -> Vietnã -> Índia -> Afeganistão -> Palestina -> Iraque -> Iraque -> Alemanha -> Alemanha -> Suécia -> Rússia -> Vietnã -> Tailândia -> Tailândia -> Vietnã -> Rússia -> Ucrânia -> Índia -> Itália -> Espanha -> Portugal -> Marrocos -> Cabo Verde -> Costa do Marfim -> Argélia -> África do Sul -> Argélia -> Líbia -> Mali -> Palestina -> Ucrânia -> Rússia -> Índia -> Myanmar -> Paquistão -> Paquistão -> Rússia -> Mongólia -> Afeganistão -> Palestina -> EUA -> Cuba -> Colômbia

Gorona:
  Destinos alcançados: 15
  Distância total percorrida: 137788.39 km
  Caminho completo: Nepal -> Índia -> Afeganistão -> Palestina -> Ucrânia -> Ucrânia -> Marrocos -> Marrocos -> Cabo Verde -> Costa do Marfim -> Costa do Marfim -> Marrocos -> Ucrânia -> Rússia -> Paquistão -> Paquistão -> Índia -> Afeganistão -> Palestina -> Ucrânia -> Itália -> Espanha -> Portugal -> Portugal -> Espanha -> Itália -> Espanha -> Itália -> Cabo Verde -> Mali -> Mali -> República Dominicana -> República Dominicana -> Mali -> Ce -> Itália -> Espanha -> Portugal -> México -> China -> Mongólia -> El Salvador -> Suécia -> Noruega -> Dinamarca -> Alemanha -> Ucrânia -> Iraque -> Palestina -> Palestina -> Afeganistão -> Índia -> Vietnã -> Tailândia -> Indonésia -> Nova Zelândia -> Nova Zelândia -> Indonésia -> China -> Mongólia -> Mongólia -> Índia -> Vietnã -> Tailândia -> Indonésia -> Nova Zelândia -> Fiji -> Fiji -> Vanuatu

Carro:
  Destinos alcançados: 6
  Distância total percorrida: 8.00 km
  Caminho completo:

Moto:
  Destinos alcançados: 0
  Distância total percorrida: 0.00 km
  Caminho completo:

```

Figure 12: Algoritmo Custo Uniforme

```

Buscando Balsa em Aprorfundamento Iterativo...
    -> Nepal -> ...

===== Relatório de Execução =====

Tempo Total de execução: 20.428 segundos
Uso de memória atual: 8.603 MB
Pico de memória: 0.0613 MB

Resumo por veículo:

Carro:
  Destinos alcançados: 0
  Distância total percorrida: 0.00 km
  Caminho completo:

Barco:
  Destinos alcançados: 6
  Distância total percorrida: 4051.31 km
  Caminho completo: Nepal -> China -> China -> Índia -> Índia -> China -> México -> EUA -> EUA -> México -> China -> Bangladesh -> Bangladesh -> Japão -> Japão -> Filipinas

Prone:
  Destinos alcançados: 29
  Distância total percorrida: 316611.75 km
  Caminho completo: Nepal -> Índia -> Arábia Saudita -> Palestina -> Iraque -> Egito -> Egito -> Iraque -> Ucrânia -> Rússia -> Vietnã -> Vietnã -> Rússia -> Ucrânia -> Iraque -> Iraque -> Ucrânia -> Rússia -> Vietnã -> Tailândia -> Tailândia -> Ucrânia -> Itália -> Itália -> Ucrânia -> Marrocos -> Cabo Verde -> Marrocos -> África do Sul -> África do Sul -> Angola -> Libéria -> Mali -> Palestina -> Arábia Saudita -> Índia -> Níger -> Níger -> Nepal -> China -> México -> Colômbia -> Colômbia -> México -> China -> Rússia -> Ucrânia -> Rússia -> Marrocos -> Marrocos -> Cabo Verde -> Costa do Marfim -> Costa do Marfim -> Cabo Verde -> Marrocos -> Ucrânia -> Rússia -> Paquistão -> Marrocos -> Portugal -> Portugal -> Marrocos -> Iraque -> Palestina -> EUA -> Cuba -> Haiti -> República Dominicana -> República Dominicana -> Haiti -> Colômbia -> México -> Honduras -> Honduras -> México -> China -> Rússia -> Eslovênia -> Eslovênia -> Suécia -> Noruega -> Noruega -> Inglaterra -> Alemanha -> Ucrânia -> Iraque -> Palestina -> Gênes -> México -> China -> Indonésia -> Nova Zelândia -> Nova Zelândia -> Indonésia -> China -> Mongólia -> Mongólia -> Índia -> China -> Indonésia -> Nova Zelândia -> Fiji -> Fiji -> Vanuatu

Carrão:
  Destinos alcançados: 0
  Distância total percorrida: 0.00 km
  Caminho completo:

Moto:
  Destinos alcançados: 0
  Distância total percorrida: 0.00 km
  Caminho completo:

```

Figure 13: Algoritmo IDS

```

Desculpa, Greeny...
----: Nepal ----
===== Relatório de Execução =====

Tempo total de execução: 1.0336 segundos
Uso de memória atual: 0.0027 MB
Pico de memória: 0.0097 MB

Resumo por veículo:

Camião:
Destinos alcançados: 2
Distância total percorrida: 5029.19 km
Caminho completo: Nepal -> China -> China -> Índia

Barco:
Destinos alcançados: 27
Distância total percorrida: 466056.36 km
Caminho completo: China -> Índia -> Myanmar -> Paquistão -> Rússia -> Ucrânia -> Iraque -> Palestina -> EUA -> EUA -> México -> China -> Bangladesh -> Bangladesh -> Japão -> Japão -> Filipinas -> Filipinas -> Coreia do Sul -> Coreia do Sul -> Mongólia -> Afeganistão -> Palestina -> Iraque -> Egito -> Egito -> Rússia -> México -> China -> Indonésia -> Tailândia -> Vietnã -> Vietnã -> Rússia -> Ucrânia -> Iraque -> Iraque -> Ucrânia -> Alemanha -> Itália -> Espanha -> França -> Inglaterra -> Dinamarca -> Suécia -> Rússia -> Vietnã -> Tailândia -> Tailândia -> Vietnã -> Rússia -> Ucrânia -> Itália -> Itália -> Espanha -> Portugal -> Marrocos -> Cabo Verde -> Moçambique -> África do Sul -> África do Sul -> Países Baixos -> Marrocos -> Ucrânia -> Paquistão -> Rússia -> Paquistão -> Rússia -> México -> Colômbia -> México -> China -> Índia -> Afeganistão -> Palestina -> Ucrânia -> Rússia -> Marrocos -> Marrocos -> Espanha -> Alemanha -> Itália -> Espanha -> França -> Inglaterra -> Noruega -> Suécia -> Rússia -> Mongólia -> Afeganistão -> Palestina -> Mali -> Líbéria -> Angola -> Costa da Marinha -> Cabo Verde -> Marrocos -> Ucrânia -> Rússia -> Paquistão -> Paquistão -> Myanmar -> Nepal -> China -> Índia -> Vietnã -> Rússia -> Ucrânia -> Itália -> Espanha -> Portugal -> Espanha -> Itália -> Ucrânia -> Iraque -> Palestina -> EUA -> Cuba -> Haiti -> Haiti -> República Dominicana -> República Dominicana -> Haiti -> Cuba -> Peru -> Honduras -> Honduras -> México -> China -> Índia -> Paquistão -> Rússia -> Eslováquia -> Eslováquia -> Índia -> Índia -> França -> Espanha -> Itália -> Ucrânia -> Iraque -> Palestina -> Ucrânia -> Rússia -> Vietnã -> Tailândia -> Indonésia -> Nova Zelândia -> Nova Zelândia -> Austrália -> Coreia do Sul -> China -> Mongólia -> Mongólia -> Índia -> Vietnã -> Tailândia -> Indonésia -> Nova Zelândia -> Fiji -> Fiji -> Vanuatu

Drone:
Destinos alcançados: 0
Distância total percorrida: 0.00 km
Caminho completo:

Carro:
Destinos alcançados: 0
Distância total percorrida: 0.00 km
Caminho completo:

Navio:
Destinos alcançados: 0
Distância total percorrida: 0.00 km
Caminho completo:

```

Figure 14: Algoritmo Greedy

```

Executando A*...
-----Nepal-----

===== Relatório de Execução =====

Tempo total de execução: 1.8127 segundos
Uso de memória atual: 0.0023 MB
Pico de memória: 0.0115 MB

Resumo por veículo:

Camião:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Barco:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Drone:
- Destinos alcançados: 29
- Distância total percorrida: 332069.23 km
- Caminho completo: Nepal -> China -> China -> Índia -> Índia -> Afeganistão -> Palestina -> EUA -> EUA -> México -> China -> Bangladesh -> Bangladesh -> Japão -> Japão -> Filipinas -> Filipinas -> Coreia do Sul -> China -> Índia -> Afeganistão -> Palestina -> Iraque -> Iraque -> Afeganistão -> Índia -> Vietnã -> Tailândia -> Tailândia -> Vietnã -> Rússia -> Ucrânia -> Itália -> Itália -> Espanha -> Portugal -> Marrocos -> Cabo Verde -> Costa do Marfim -> África do Sul -> Angola -> Libéria -> Mali -> Palestina -> Afeganistão -> Índia -> Myanmar -> Myanmar -> Nepal -> China -> México -> Colômbia -> Colômbia -> México -> China -> Mongólia -> Rússia -> Ucrânia -> Marrocos -> Marrocos -> Cabo Verde -> Costa do Marfim -> Costa do Marfim -> Cabo Verde -> Marrocos -> Ucrânia -> Rússia -> Paquistão -> Paquistão -> Índia -> Afeganistão -> Palestina -> Ucrânia -> Itália -> Espanha -> Portugal -> Portugal -> Espanha -> Itália -> Ucrânia -> Iraque -> Palestina -> EUA -> Cabo -> Mali -> Mali -> República Dominicana -> República Dominicana -> Mali -> Colômbia -> México -> Honduras -> Honduras -> México -> China -> Mongólia -> Rússia -> Eslováquia -> Eslováquia -> Suécia -> Noruega -> Dinamarca -> Alemanha -> Ucrânia -> Iraque -> Palestina -> Palestina -> Afeganistão -> Índia -> Vietnã -> Tailândia -> Indonésia -> Nova Zelândia -> Nova Zelândia -> Indonésia -> China -> Mongólia -> Mongólia -> Índia -> Vietnã -> Tailândia -> Indonésia -> Nova Zelândia -> Fiji -> Fiji -> Vanuatu

Carro:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Moto:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

```

Figure 15: Algoritmo A*

```

Executando SMA*...
-----Nepal-----

===== Relatório de Execução =====

Tempo total de execução: 0.5556 segundos
Uso de memória atual: 0.0009 MB
Pico de memória: 0.0043 MB

Resumo por veículo:

Camião:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Barco:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Drone:
- Destinos alcançados: 2
- Distância total percorrida: 5029.19 km
- Caminho completo: Nepal -> China -> China -> Índia

Carro:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

Moto:
- Destinos alcançados: 0
- Distância total percorrida: 0.00 km
- Caminho completo:

```

Figure 16: Algoritmo SMA*

A partir dos resultados apresentados e da comparação dos valores obtidos, podemos concluir que houve diferenças significativas nos desempenhos dos diversos algoritmos de procura testados. Apesar de alguns algoritmos retornarem soluções diferentes, os algoritmos A*, DFS, BFS, Greedy, Custo Uniforme, IDS e SMA* foram capazes de encontrar uma solução para este cenário. É relevante observar que, embora esses algoritmos tenham retornado soluções, a eficiência em termos de tempo e memória variou significativamente entre eles.

Por exemplo, o DFS e o BFS exploraram muitos nós antes de chegar à solução, mostrando-se menos eficientes em termos de tempo e consumo de

memória. O IDS apresentou uma abordagem mais sistemática que combina a profundidade limitada com a busca em profundidade, tornando-se mais eficiente do que o DFS puro em alguns casos. Por outro lado, o A*, ao usar heurísticas adequadas, encontrou consistentemente a solução ótima com menor esforço computacional, destacando-se como a abordagem mais eficiente neste cenário. O Greedy, embora rápido, não garante sempre a solução ótima, uma vez que pode ficar preso em soluções locais devido à natureza da sua heurística.

Ainda assim, é importante ressaltar que a eficiência e a precisão de cada algoritmo podem variar dependendo da configuração do grafo, do ponto de partida, do destino e da heurística utilizada. Quando o objetivo for garantir a solução ótima de maneira eficiente, o uso de algoritmos como o A* é recomendado, desde que a heurística seja admissível e consistente.

Para ilustrar de forma prática, analisamos a aplicação do algoritmo A* em um cenário específico. Nele, determinou-se a rota ideal para realizar um conjunto de entregas, distribuídas entre diferentes veículos (por exemplo, caminhão, barco e drone). Observou-se que o A* conseguiu otimizar o número de viagens e minimizar custos associados, como distância total percorrida e tempo de execução. Comparativamente, outros algoritmos, como o DFS, o IDS e o Greedy, tiveram desempenho inferior, tanto em qualidade da solução quanto em tempo de processamento.

Portanto, recomendamos o uso do A* como referência para avaliar os resultados de outros algoritmos, especialmente em cenários onde a solução ótima é crítica.

9 Conclusão

Dada a conclusão deste trabalho realizado no âmbito da disciplina de Inteligência Artificial, acreditamos ter compreendido e implementado com sucesso todos os algoritmos de procura mencionados, desde os princípios que guiam o seu funcionamento até à sua implementação prática no contexto do problema em análise.

Em suma, consideramos ter cumprido todos os objetivos e metas estabelecidos no trabalho prático. O objetivo principal era desenvolver um programa capaz de determinar os caminhos mais eficientes para diferentes cenários. Com os algoritmos implementados, como o A*, SMA*, Greedy, DFS, BFS, Custo Uniforme e IDS, fomos capazes de encontrar soluções otimizadas, destacando a eficácia de algoritmos heurísticos como o A* na obtenção do valor ótimo.

Acreditamos que este trabalho demonstra a importância dos algoritmos de procura na resolução de problemas complexos e oferece uma base sólida para a aplicação prática destes métodos em cenários do mundo real.