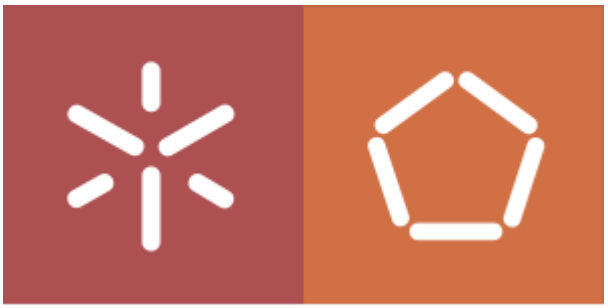


Universidade do Minho
Escola de Engenharia

Licenciatura em Engenharia Informática

Inteligência Artificial

Grupo 36

a84675, António Luís Braga Mendes
a100535, Gonçalo Nuno da Silva Loureiro
a100604, Afonso Oliveira e Silva
a100760, Diogo Filipe Paiva Martins

Resolução de problemas - Algoritmos de procura

Janeiro 2024

Resumo

Este relatório visa descrever o processo de realização do trabalho prático no contexto da disciplina de Inteligência Artificial ao longo do primeiro semestre, do terceiro ano da Licenciatura em Engenharia Informática na Universidade do Minho. O objetivo do trabalho prático foi estimular e sensibilizar os alunos na formulação de soluções para problemas de busca utilizando algoritmos de procura, centrando-se em cenários de entregas conduzidas por múltiplos estafetas, com a Health Planet como referência. O documento descreve de forma sucinta e concisa as abordagens adotadas pelo grupo para a sua resolução, bem como a fundamentação por trás das decisões realizadas ao longo da resolução.

Índice

1. Introdução

Compreender de forma precisa a natureza do problema é crucial para desenvolver uma solução eficaz. Neste contexto, parece que a Health Planet enfrenta um desafio logístico complexo que envolve a otimização das rotas de entrega, tendo em conta uma série de variáveis críticas:

- Existência de vários estafetas e de diferentes meios de transporte.
- As entregas possuem pesos e são atribuídas a estafetas específicos.
- Cada estafeta está vinculado a uma cidade ou freguesia, com uma avaliação associada.
- Os clientes podem especificar o tempo máximo que estão dispostos a esperar pela entrega.
- A empresa busca minimizar as suas emissões de CO2.
- Cada veículo possui uma velocidade média, bem como limites de peso e penalidades de velocidade dependendo do volume transportado.

Com base nestas considerações, torna-se evidente que o objetivo é encontrar métodos para calcular os trajetos mais eficientes para cada estafeta, assegurando a entrega oportuna e minimizando o tempo de transporte. Ao abordar estas variáveis, a Health Planet pode desenvolver uma solução ótima, capaz de enfrentar os desafios associados à entrega dos produtos de maneira eficiente, pontual e sustentável.

2. Formulação do problema

Como maneira de compreender melhor o problema em questão e de modo a encontrar uma solução para o mesmo mais facilmente, é importante definir bem os seguintes componentes: Representação de Estados; Estado Inicial; Estado Objetivo; Operadores; Custo da Solução.

2.1 Representação de Estado

Tipos de problema como este são problemas de estado único, ou seja, há apenas um conjunto de decisões que otimiza a função de objetivo, maximizando ou minimizando algum critério. Neste caso, o estafeta tem acesso ao ambiente antes da entrega começar e também sabe onde começará este processo, num posto da "Health Planet". Claro está que existem certas situações não esperadas e/ou não controladas quer pelo estafeta quer pela instituição que obriguem ao cálculo de uma nova solução, como um acidente por exemplo que levasse a estrada a ser cortada, tendo o estafeta que tomar outro trajeto (solução), estando assim perante um problema de contingência.

2.2 Representação do Estado Inicial

No problema em questão o estado inicial está representado quando encontramos o nosso ator (estafeta) num dado nodo, isto é, num posto onde este levanta uma ou mais encomendas para realizar a entrega. Como referido acima, podem existir situações de exceção e, nesses casos, podemos ter um novo estado inicial caso seja preciso encontrar um novo trajeto.

2.3 Representação do Estado Objetivo

O estado objetivo é sempre simples e nunca muda independentemente das situações excecionais. No contexto de entrega de encomendas, o estado objetivo será sempre o sítio em que o estafeta conclui o processo, ou seja, um nodo que representa o número da porta da entrega, não interessando o trajeto que percorre.

2.4 Operadores

O operador crucial nesta situação é a transição de um nodo para outro. Cada ação envolve um custo composto por três elementos: distância, tempo e emissões. A distância refere-se à separação física entre dois pontos, acompanhada por duas taxas que, com base nessa distância, determinam penalidades temporais (indicadores do trânsito) e emissões (indicadores do esforço do motor). Todos esses aspectos são importantes para o cálculo do custo da solução final.

2.5 Custo da Solução

A avaliação da eficácia e do impacto ambiental das estratégias de entregas no âmbito da Health Planet destaca a importância da determinação do custo da solução. A estratégia empregada concentra-se em calcular o custo relacionado à pegada ecológica, à distância percorrida e ao tempo das entregas. No contexto da nossa distribuidora, os elementos mais cruciais são o tempo e a pegada. Em relação ao tempo, é essencial assegurar que todas as entregas são realizadas dentro do prazo estipulado. No que diz respeito à pegada, o objetivo é minimizá-la ao máximo.

2.6 Abordagem alternativa

Primeiramente, gostaríamos de salientar que há diversas maneiras de abordar este problema. Uma alternativa considerada foi representar cada nodo como uma rua, tendo os nodos sucessores como as ruas acessíveis. No entanto, essa abordagem foi descartada posteriormente, uma vez que enfrentamos dificuldades em distinguir os números das portas nas diferentes ruas.

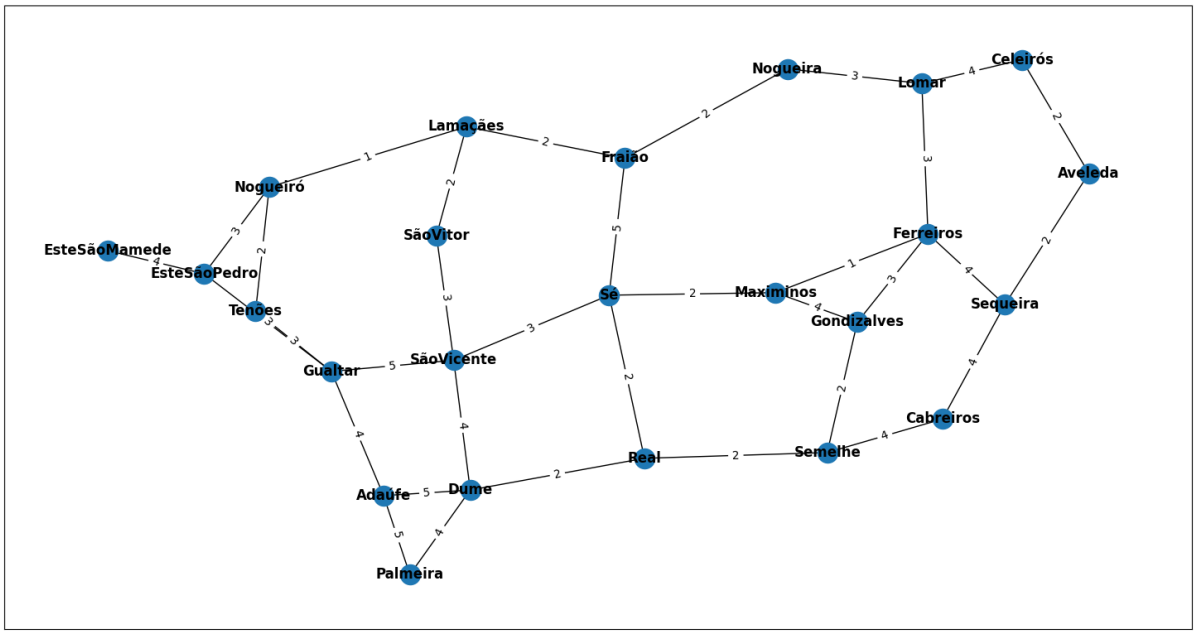
3. Circuitos

Para a idealização dos circuitos eficientes de entrega a percorrer por parte dos estafetas da Health Planet tivemos de recorrer a vários algoritmos de procura tanto informada como não informada. Abaixo estão pontos importantes a considerar para gerar as rotas que cobrem uma determinada área:

- Otimização da entrega - tentar otimizar as rotas de entrega para minimizar o consumo de energia, tempo de viagem e as emissões de carbono;
- Meios de transporte sustentáveis - considerar os diferente níveis de consumo de energia associados a cada escolha da opção de transporte (bicicleta/mota/carro);
- Restrição de prazo de entrega - ter em consideração o prazo de entrega estabelecidos pelos clientes, evitando assim atrasos que resultem em penalizações e más avaliações;
- Satisfação do cliente - otimizar as rotas com base nas classificações dos clientes, tentando maximizar a satisfação do cliente, melhorando assim o ranking de entregas do estafeta
- Características da encomenda - ser possível adaptar as rotas com base nas características da encomenda como peso e volume, para garantir uma entrega eficiente;
- Custo do serviço de entrega - incluir o custo do serviço de entrega ao gerar as rotas, tendo em conta fatores como o prazo de entrega e o meio de transporte utilizado.

Ter em atenção estes fatores é crucial no desenvolvimento de algoritmos que cumprem as necessidades logísticas da Health Planet. Promover práticas sustentáveis e eficientes na entrega de encomendas, trabalhando na implementação de algoritmos de procura que cumprem os objetivos específicos de gerar circuitos de entrega eficientes para a Health Planet é um dos objetivos fundamentais deste projeto.

4. Representação em forma de grafo



5. Implementação das estratégias de procura

Para a resolução deste trabalho usamos 7 estratégias de procura onde 4 são de procura não informada (cega) e as outras 3 de procura informada.

5.1 Procura não informada

5.1.1 DFS

Com esta estratégia expandimos sempre um dos nodos mais profundos da árvore. Este algoritmo ocupa muito pouco espaço em memória mas não pode ser utilizado em árvores de profundidade infinita, não sendo um algoritmo completo. Devolve quase sempre a primeira solução que encontra.

```
    def procura_DFS(self, start, end, path=[], visited=set()):
        path.append(start)
        visited.add(start)

        if start == end:
            custoT = self.calcula_custo(path)
            return (path, custoT)
        for (adjacente, peso) in self.m_graph[start]:
            if adjacente not in visited:
                resultado = self.procura_DFS(adjacente, end, path, visited)
                if resultado is not None:
                    return resultado
        path.pop()
        return None
```

5.1.2 BFS

Neste caso, expandimos os nodos de menor profundidade, ou seja, executamos uma procura primeiro em largura. Ocupa muito espaço na memória e demora bastante tempo para ser executado. É um algoritmo bastante completo se o fator de ramificação for finito e é ótimo se o custo de cada passo for 1, o que não acontece neste trabalho.

```
def procura_BFS(self, start, end):
    # definir nodos visitados, para evitar ciclos
    visited = set()
    fila = Queue()
    custo = 0
    # adicionar o nodo inicial à fila e aos visitados
    fila.put(start)
    visited.add(start)
    # garantir que o start node não tem pais
    parent = dict()
    parent[start] = None

    path_found = False
    while not fila.empty() and path_found == False:
        node_atual = fila.get()
        if node_atual == end:
            path_found = True
        else:
            for (adjacente, peso) in self.m_graph[node_atual]:
                if adjacente not in visited:
                    fila.put(adjacente)
                    parent[adjacente] = node_atual
                    visited.add(adjacente)

    # construir o caminho
    path = []
    if path_found:
        path.append(end)
        while parent[end] is not None:
            path.append(parent[end])
            end = parent[end]
        path.reverse()
        # função que calcula custa caminho
        custo = self.calcula_custo(path)
    return (path, custo)
```

5.1.3 Custo Uniforme

O objetivo desta estratégia consiste em guardar o custo total do caminho do estado inicial para esse nodo, existindo uma lista prioritária de estados não expandidos ordenada pelo custo do caminho. Em todos os nodos N, g(N) é o custo conhecido de ir da raiz até ao nodo N. É uma estratégia ótima e completa se o custo da deslocação for maior que alguma constante positiva. É equivalente ao algoritmo de Dijkstra.

```
def procura_custoUniforme(self, start, end):
    priority_queue = PriorityQueue()
    priority_queue.put((0, start))
    visited = set()
    cost_so_far = {start: (0, None)}

    while not priority_queue.empty():
        current_cost, current_node = priority_queue.get()
        if current_node == end:
            path = [current_node]

            while current_node != start:
                if current_node not in cost_so_far:
                    print('Node not found.')
                    return None
                aux = True
                while aux:
                    current_node = cost_so_far[current_node][1]
                    if current_node in path:
                        del cost_so_far[current_node]
                    else:
                        aux = False
                path.append(current_node)
            path.reverse()

            return path, current_cost
        visited.add(current_node)

        for neighbor, weight in self.getNeighbours(current_node):
            new_cost = current_cost + weight
            if neighbor not in visited:
                cost_so_far[neighbor] = (new_cost, current_node)
                priority_queue.put((new_cost, neighbor))
                visited.add(neighbor)

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

5.1.4 DFS iterativo

Se não soubermos o valor máximo limite, ficaremos restritos a uma estratégia de busca em profundidade primeiro e enfrentaremos o desafio de possíveis caminhos infinitos. A solução consiste na modificação do princípio da busca limitada, variando o limite entre zero e infinito. Primeiro verificar a raiz e desenvolver um DFS procurando um caminho com limite de comprimento 1. Se não houver um caminho com limite de comprimento 1, desenvolver um DFS procurando um caminho com limite de comprimento 2. Repetir este processo até encontrar uma solução. É uma estratégia completa mas não é ótima para o tipo de problema que temos em mão pois o custo não é sempre 1.

```
def procura_iterativeDFS(self, start, end, max_depth=100):
    for depth_limit in range(1, max_depth + 1):
        result = self.DFS_recursive(start, end, depth_limit, set())
        if result is not None:
            path, cost = result
            return path, cost

    print(f'Path does not exist within the depth limit of {max_depth}.')
    return None
```

5.2 Procura informada

5.2.1 Greedy

A estratégia neste tipo de algoritmo é expandir o nodo que parece estar mais perto do nodo objetivo com base numa determinada heurística. Este algoritmo não é completo pois pode entrar em ciclos e nem sempre encontra a solução ótima. Ocupa um grande espaço pois guarda todos os nós em memória.

```
def procura_Greedy(self, start, end):
    open_list = set([start])  # lista de nodos visitados, mas com vizinhos que ainda não foram visitados
    closed_list = set([])  # lista de nodos visitados
    parents = {}  # dicionário que mantem o antecessor de um nodo
    parents[start] = start
    while len(open_list) > 0:
        n = None
        # encontra nodo com a menor heuristica
        for v in open_list:
            if n is None or self.m_h[v] < self.m_h[n]:
                n = v
        if n is None:
            print('Path does not exist!')
            return None
        if n == end:
            recons_path = []
            while parents[n] != n:
                recons_path.append(n)
                n = parents[n]
            recons_path.append(start)
            recons_path.reverse()
            return (recons_path, self.calcula_custo(recons_path))
        for (m, weight) in self.getNeighbours(n):
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
        open_list.remove(n)
        closed_list.add(n)
    print('Path does not exist!')
    return None
```

5.2.2 A*

Este algoritmo combina a procura gulosa com a uniforme, as duas já citadas anteriormente, minimizando a soma do caminho já efetuado com o mínimo previsto que falta percorrer até à solução. Para chegar ao custo estimado da solução menos dispendiosa que passa pelo nodo n, usa a função f(n) = g(n) + h(n) onde g(n) é o custo total, até aquele determinado instante, para chegar ao estado n (custo do percurso) e h(n) o custo estimado para chegar ao nodo objetivo, baseado numa determinada heurística. Evita expandir caminhos que são muito dispendiosos. É um algoritmo ótimo e completo, mas ocupa muito espaço na memória.

```
def procura_aStar(self, start, end):
    open_list = {start}
    closed_list = set([])
    g = {}
    g[start] = 0
    parents = {}
    parents[start] = start
    while len(open_list) > 0:
        n = None
        for v in open_list:
            if n is None or g[v] + self.getH(v) < g[n] + self.getH(n):
                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()
            return (reconst_path, self.calcula_custo(reconst_path))
        for (m, weight) in self.getNeighbours(n):
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n

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

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

5.2.3 A* Iterativo

Como referenciado antes, o algoritmo A* utiliza uma função heurística para avaliar os nodos a serem expandidos, combinando o custo do caminho conhecido com uma estimativa do custo restante até o destino. Isso ajuda a priorizar os nós que parecem mais promissores.

O algoritmo A* iterativo é uma implementação que realiza uma procura em profundidade iterativa até atingir uma profundidade máxima especificada (max_depth). Ele chama a função "aStar_recursive" com diferentes limites de profundidade, aumentando gradualmente de 1 até max_depth. O objetivo é procurar o caminho de menor custo dentro desses limites de profundidade.

```
def procura_iterativeAStar(self, start, end, max_depth=100):
    for depth_limit in range(1, max_depth + 1):
        result = self.aStar_recursive(start, start, end, depth_limit)
        if result is not None:
            path, cost = result
            return path, cost
    print(f'Path does not exist within the depth limit of {max_depth}.')
    return None
```

6. Comparação dos resultados obtidos das diferentes estratégias com caminho mais curto e custo da solução

Numa encomenda onde o estafeta se encontra em Lamações e se quer dirigir a Palmeira, registam- se as seguintes travessias:

```
Nodo inicial -> Lamações
Nodo final -> Palmeira
1-DFS
2-BFS
3-Greedy
4-A*
5-Custo Uniforme
6-Iterative DFS
7-Iterative A*
0-Sair
Introduza a sua opção -> 1
['Lamações', 'Fraião', 'Sé', 'Real', 'Dume', 'SãoVicente', 'Gualtar', 'Adaúfe', 'Palmeira'] 29
Introduza a sua opção -> 2
['Lamações', 'SãoVitor', 'SãoVicente', 'Dume', 'Palmeira'] 13
Introduza a sua opção -> 3
['Lamações', 'Nogueiró', 'EsteSãoPedro', 'Gualtar', 'Adaúfe', 'Palmeira'] 16
Introduza a sua opção -> 4
['Lamações', 'SãoVitor', 'SãoVicente', 'Dume', 'Palmeira'] 13
Introduza a sua opção -> 5
['Lamações', 'SãoVitor', 'SãoVicente', 'Dume', 'Palmeira'] 13
Introduza a sua opção -> 6
['Lamações', 'Fraião', 'Sé', 'Real', 'Dume', 'Palmeira'] 15
Introduza a sua opção -> 7
['Lamações', 'SãoVitor', 'SãoVicente', 'Dume', 'Palmeira'] 13
```

Dado estas travessias percebemos que o algoritmos com maior custo é o DFS, e neste caso específico o BFS, custo uniforme, DFS e A* iterativos chegaram à mesma solução com custos iguais.

Numa encomenda onde o estafeta se encontra em Este São Mamede e se quer dirigir a Cabreiros, registam- se as seguintes travessias:

```
Introduza a sua opção -> 3
Nodo inicial -> EsteSãoMamede
Nodo final -> Cabreiros
1-DFS
2-BFS
3-Greedy
4-A*
5-Custo Uniforme
6-Iterative DFS
7-Iterative A*
0-Sair
Introduza a sua opção -> 1
['EsteSãoMamede', 'Nogueiró', 'EsteSãoPedro', 'Gualtar', 'SãoVicente', 'Sé', 'Maximinos', 'Gondizalves', 'Semelhe', 'Cabreiros'] 32
Introduza a sua opção -> 2
['EsteSãoMamede', 'Nogueiró', 'Lamações', 'Fraião', 'Sé', 'Real', 'Semelhe', 'Cabreiros'] 22
Introduza a sua opção -> 3
['EsteSãoMamede', 'EsteSãoPedro', 'Gualtar', 'SãoVicente', 'Sé', 'Real', 'Semelhe', 'Cabreiros'] 23
Introduza a sua opção -> 4
['EsteSãoMamede', 'Nogueiró', 'Lamações', 'Fraião', 'Sé', 'Real', 'Semelhe', 'Cabreiros'] 22
Introduza a sua opção -> 5
['EsteSãoMamede', 'Nogueiró', 'Lamações', 'Fraião', 'Sé', 'Real', 'Semelhe', 'Cabreiros'] 22
Introduza a sua opção -> 6
['EsteSãoMamede', 'Nogueiró', 'EsteSãoPedro', 'Gualtar', 'SãoVicente', 'Sé', 'Real', 'Semelhe', 'Cabreiros'] 28
Introduza a sua opção -> 7
['EsteSãoMamede', 'Nogueiró', 'Lamações', 'Fraião', 'Sé', 'Real', 'Semelhe', 'Cabreiros'] 22
```

Através da análise dos resultados obtidos neste estudo, podemos concluir que os algoritmos de Busca em Largura (BFS), Custo Uniforme e A* destacaram-se como as melhores opções para a resolução do problema em questão. Cada um desses algoritmos demonstrou eficiência em diferentes aspectos, o que ressalta a importância de escolher a abordagem adequada de acordo com as características específicas do cenário de aplicação.

7. Conclusão

Em síntese, este trabalho explorou e comparou os algoritmos de pesquisa para a resolução de problemas no ambiente especificado que era a instituição “Health Planet”. Os resultados obtidos indicam que cada algoritmo apresenta vantagens distintas, sendo que a escolha entre eles deve ser cuidadosamente ponderada de acordo com as características do problema em questão, nomeadamente o menor custo.

O algoritmo de Procura em Largura mostrou-se eficiente na exploração sistemática do espaço de estados, priorizando soluções próximas à origem. Por outro lado, o Custo Uniforme destacou-se ao considerar criteriosamente os custos associados às transições entre estados, resultando em soluções com menor custo total.

No entanto, o algoritmo A* sobressaiu-se ao incorporar uma heurística que direciona a busca para o objetivo desejado de maneira eficiente. Essa capacidade de considerar informações heurísticas possibilitou a descoberta de soluções mais rapidamente, destacando-se em situações em que a estimativa do custo até o objetivo é valiosa.

Dessa forma, a escolha entre BFS, Custo Uniforme e A* dependerá das características específicas de cada problema. Enquanto o BFS e o Custo Uniforme podem ser mais adequados para problemas simples e com custos uniformes, o A* destaca-se em situações mais complexas, nas quais a heurística é valiosa para otimizar a eficiência da busca.

Portanto, este estudo oferece uma visão abrangente das vantagens e limitações de cada algoritmo, fornecendo uma base sólida para a seleção informada da abordagem mais adequada de acordo com as demandas específicas de cada aplicação.

Concluindo, aprendemos bastante na realização deste trabalho, especialmente sobre otimização de rotas e dos custos associados às mesmas.