Instituto Tecnológico de Aeronáutica - ITA
Inteligência Artificial para Robótica Móvel - CT213
Aluno: Matheus Felipe Ramos Borges

Relatório do Laboratório 2 - Busca Informada

1 Breve Explicação em Alto Nível da Implementação

O código tem como objetivo determinar o caminho mínimo entre dois pontos em um mapa. Para isso, discretiza o mapa em um grid 8-conectado e o trata como um grafo. Dessa forma, encontrar o percurso da origem até o destino torna-se um problema de busca em grafos. Foram implementados três algoritmos para determinar o caminho. Algoritmo de Dijkstra, que embora exija maior poder computacional, garante sempre a obtenção do caminho mínimo. Algoritmo Greedy Search (Busca Gulosa), que utiliza uma heurística para definir o caminho, embora não garanta a solução ótima, encontra um trajeto de forma extremamente rápida. Algoritmo A*, que também é baseado em heurística, consegue determinar o caminho de menor custo, sendo um pouco mais lento que a busca gulosa, mas mais eficiente que Dijkstra na maioria dos casos.

1.1 Algoritmo Dijkstra

A ideia do algoritmo de Dijkstra é realizar uma busca em largura no grafo enquanto calcula os custos mínimos de cada nó, até alcançar o objetivo. Esse algoritmo garante a obtenção do caminho de menor custo; no entanto, exige um alto poder computacional, pois percorre todos os nós. A baixo, está representado o pseudo-código do algoritmo de Dijkstra.

```
def dijkstra(start):
# Inicializa o custo de todos os nós como infinito
pq = PriorityQueue()
start.cost = 0
pq.insert_or_update(start.cost, start)

while not pq.empty():
    node = pq.extract_min()

    for successor in node.successors():
        if successor.cost > node.cost + cost(node, successor):
            successor.cost = node.cost + cost(node, successor)
            successor.parent = node
            pq.insert_or_update(successor.cost, successor)
```

O algoritmo de Dijkstra começa com a inicialização, onde todos os nós recebem um custo infinito, exceto o nó inicial, que é definido com custo zero. Em seguida, o nó inicial é inserido em uma fila de prioridade. Durante a exploração dos nós, enquanto houver elementos na fila, o nó com

o menor custo é retirado. Para cada vizinho desse nó, calcula-se um possível caminho mais curto passando por ele. Se esse novo custo for menor que o anteriormente registrado para o sucessor, o valor é atualizado, e o nó é reinserido na fila. O algoritmo segue para a finalização, continuando o processo até que todos os nós tenham sido processados ou até que se atinja o objetivo específico, caso o objetivo seja encontrar o caminho até apenas um nó. No lab, utilizava-se uma Min-heap como fila de prioridade, desse modo a remoção do nó de menor custo era otimizada.

1.2 Algoritmo *Greedy Search*

A ideia do algoritmo Greedy Search (busca gulosa) é utilizar uma heurística para determinar quais nós serão visitados. Nesse caso, não se considera o custo real para alcançar o objetivo, mas sim o valor da heurística. Assim, não percorremos todos os nós, mas selecionamos apenas aqueles que mais se aproximam do objetivo. A heurística utilizada no laboratório, e a mais comum em problemas de otimização de percurso em grids 8-conectados, é a distância euclidiana.

$$f(n) = h(n) = \sqrt{(\text{node.x} - \text{goal.x}) + (\text{node.y} - \text{goal.y})} \tag{1}$$

Nesse caso, o caminho encontrado normalmente não é o de custo mínimo, mas é obtido de forma muito rápida, pois sempre buscamos nós que se aproximam do objetivo. Abaixo está o pseudocódigo do algoritmo da busca gulosa.

```
def greedy_search(start, goal):
    # Inicializa node.g e node.f para infinito em todos os nós
    pq = PriorityQueue()
    start.g = 0
    start.f = h(start, goal)
    pq.insert_or_update(start.f, start)

    while not pq.empty():
        node = pq.extract_min()
        if node.content == goal:
            return node

        for successor in node.successors():
            if successor.g > node.g + cost(node, successor):
                successor.g = node.g + cost(node, successor)
                successor.f = h(successor, goal)
                successor.parent = node
                pq.insert_or_update(successor.f, successor)
```

O algoritmo da Busca Gulosa inicia atribuindo a todos os nós um valor infinito para g (custo do caminho percorrido) e f (apenas a função heurística nesse caso). O nó inicial recebe g = 0 e tem seu f calculado com base na heurística $h(n)$, que estima a distância até o objetivo. Esse nó é então inserido em uma fila de prioridade ordenada pelo valor de f. Durante a exploração, enquanto houver nós na fila, o nó com o menor valor de f é removido. Se esse nó for o objetivo,

o algoritmo retorna o caminho encontrado; caso contrário, gera seus sucessores e avalia cada um deles. Para cada sucessor, calcula-se um novo valor de f usando apenas a heurística $h(n)$, ignorando o custo acumulado g. Se esse novo valor for menor que o previamente atribuído ao sucessor, ele é atualizado, recebe o nó atual como pai e é reinserido na fila de prioridade. O processo continua até que a fila fique vazia ou o objetivo seja alcançado. Se a fila esvaziar antes de encontrar o objetivo, significa que não há um caminho viável até ele.

1.3 Algoritmo A*

A ideia do algoritmo A* é utilizar uma heurística para determinar quais nós serão visitados. Nesse caso, iremos considerar tanto a heurística do problema quanto o custo real do nó.

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

Assim, não percorremos todos os nós, mas selecionamos apenas aqueles que mais se aproximam do objetivo e visamos reduzir o custo. Nesse caso, o critério de comparação é a função f, que leva em conta tanto o custo até o nó, quanto a heurística. A heurística utilizada no laboratório, e a mais comum em problemas de otimização de percurso em grids 8-conectados, é a distância euclidiana.

$$h(n) = \sqrt{(\text{node.x} - \text{goal.x}) + (\text{node.y} - \text{goal.y})} \tag{3}$$

Então, conseguimos encontrar nesse caso o caminho de custo mínimo de forma bem mais eficiente que o algoritmo de Dijkstra, pois não passamos por nós que se afastam do objetivo.

```
def a_star(start, goal):
    # Inicializa node.g e node.f para infinito em todos os nós
    pq = PriorityQueue()
    start.g = 0
    start.f = h(start, goal)
    pq.insert_or_update(start.f, start)

    while not pq.empty():
        node = pq.extract_min()
        if node.content == goal:
            return node

        for successor in node.successors():
            if successor.g > node.g + cost(node, successor):
                successor.g = node.g + cost(node, successor)
                successor.f = successor.g + h(successor, goal)
                successor.parent = node
                pq.insert_or_update(successor.f, successor)
```

O algoritmo A* inicia atribuindo a todos os nós um valor infinito para g (custo do caminho percorrido) e f. O nó inicial recebe g = 0 e tem seu f calculado com base na heurística e no custo

até o nó atual $g(n) + h(n)$, que estima o custo total até o objetivo. Esse nó é então inserido em uma fila de prioridade ordenada pelo valor de f. Durante a exploração, enquanto houver nós na fila, o nó com o menor valor de f é removido. Se esse nó for o objetivo, o algoritmo retorna o caminho encontrado; caso contrário, gera seus sucessores e avalia cada um deles. Para cada sucessor, calcula-se um novo valor de f usando $g(n) + h(n)$. Se esse novo valor for menor que o previamente atribuído ao sucessor, ele é atualizado, recebe o nó atual como pai e é reinserido na fila de prioridade. O processo continua até que a fila fique vazia ou o objetivo seja alcançado. Se a fila esvaziar antes de encontrar o objetivo, significa que não há um caminho viável até ele.

2 Figuras Comprovando Funcionamento do Código

2.1 Algoritmo Dijkstra

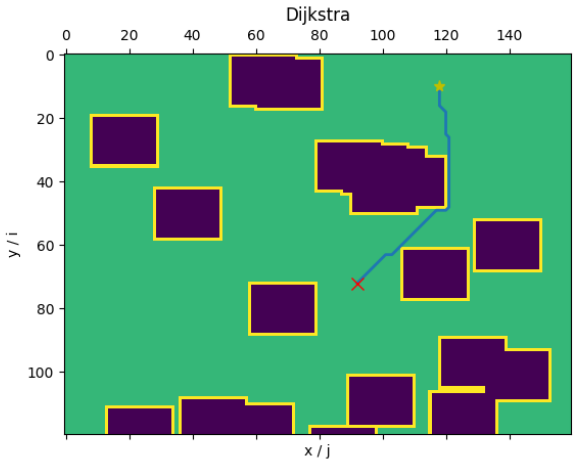

Figura 1: caminho encontrado pelo algoritmo de Dijkstra.

2.2 Algoritmo *Greedy Search*

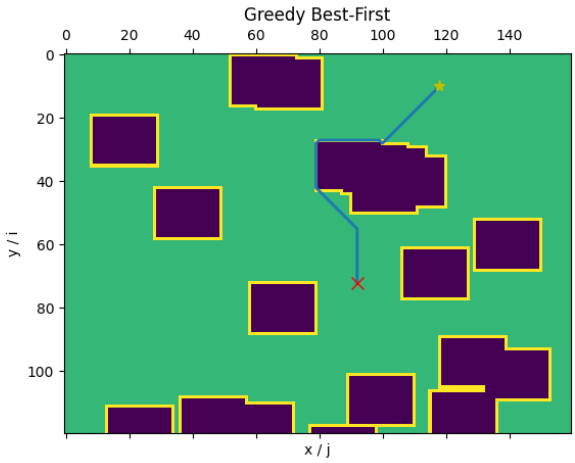

Figura 2: caminho encontrado pelo algoritmo de *Greedy Search*.

2.3 Algoritmo A*

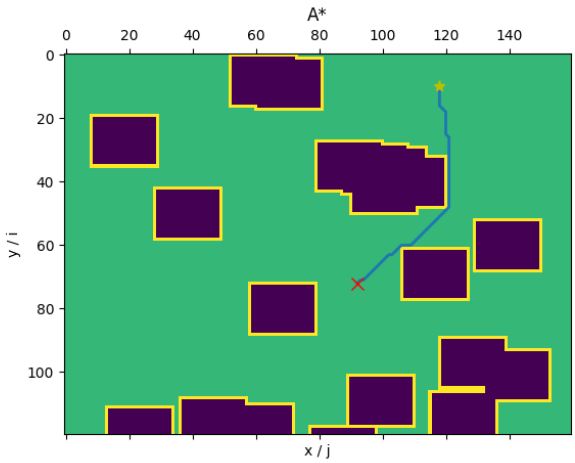

Figura 3: caminho encontrado pelo algoritmo de A*.

3 Comparação entre os Algoritmos

Tabela 1 com a comparação do tempo computacional, em segundos, e do custo do caminho entre os algoritmos usando um Monte Carlo com 100 iterações.

Tabela 1: tabela de comparação entre os algoritmos de planejamento de caminho.

Algoritmo	**Tempo computacional (s)**		**Custo do caminho**	
	Média	**Desvio padrão**	**Média**	**Desvio padrão**
Dijkstra	0.057	0.032	79.8	38.6
Greedy Search	0.003	0.001	103.1	58.8
A*	0.013	0.011	79.8	38.6