

Laboratório 02 - Busca Informada (19 de março de 2020)



André Moreno¹

¹Aluno de Graduação do Instituto Tecnológico de Aeronáutica, São José dos Campos, Brasil.

E-mail: moreno6230@gmail.com

1 Busca Informada

A fim de exercitar e fixar os conhecimentos adquiridos em sala de aula sobre Busca informada, implementou-se três algoritmos descritos abaixo com o objetivo de planejar o caminho de um robô móvel.

1.1 Dijkstra

Inicialmente, obteve-se o nó referente da posição inicial e objetivo de robô utilizando o método *get_node* da classe *node_grid*, e inicializou como zero a distância total percorrida pelo robô até o momento e inicializou uma fila de prioridades.

```
pq = []

start = self.node_grid.get_node(start_position[0],
    → start_position[1])
goal = self.node_grid.get_node(goal_position[0],
    → goal_position[1])
start.f = 0

heapq.heappush(pq, (start.f, start))
```

Após, enquanto o robô não atingir o objetivo, extraía-se o nó com o custo mínimo da fila de prioridades, ou seja, o nó mais próximo do nó atual. Desta forma, marcava-o como visitado e, utilizando o método *get_position* da classe *node*, extraía-se as coordenadas do nó. Caso o nó de custo mínimo fosse o objetivo a ser alcançado pelo robô, construía-se a trajetória a ser percorrida pelo robô utilizando o método *construct_path* da classe *PathPlanner*. Por fim, o algoritmo retorna a trajetória e o custo do trajeto. Caso contrário, a distância percorrida até o *successor_node* era atualizada e atribuída-se *node* como pai de *successor_node*.

```
while pq:
    f, node = heapq.heappop(pq)
```

```
node.closed = True
node_i, node_j = node.get_position()

if node == goal:
    path_dijkstra = self.construct_path(goal)
    cost_dijkstra = goal.f
    self.node_grid.reset()
    return path_dijkstra, cost_dijkstra

for successor in
    → self.node_grid.get_successors(node_i,
    → node_j):
    successor_node =
        → self.node_grid.get_node(successor[0],
        → successor[1])

    if not successor_node.closed:
        if successor_node.f > node.f +
            → self.cost_map.get_edge_cost(
            → node.get_position(),
            → successor_node.get_position()):
            successor_node.f = node.f +
                → self.cost_map.get_edge_cost(
                → node.get_position(),
                → successor_node.get_position())

        successor_node.parent = node
        heapq.heappush(pq, (successor_node.f,
            → successor_node))
```

2 Greedy Best-First

A implementação deste método foi similar ao algoritmo Dijkstra, diferenciando-se que o próximo nó a ser extraído da fila de prioridades leva em consideração apenas a distância euclidiana do nó atual e o objetivo. A distância do nó ao objetivo é calculada utilizando o método *distance_to*. Observa-se a seguir o algoritmo implementado.

```
pq = []

start = self.node_grid.get_node(start_position[0],
    → start_position[1])
goal = self.node_grid.get_node(goal_position[0],
    → goal_position[1])

start.g = start.distance_to(goal_position[0],
    → goal_position[1])
start.f = 0
heapq.heappush(pq, (start.g, start))

while pq:
    g, node = heapq.heappop(pq)
    node.closed = True
    node_i, node_j = node.get_position()

    for successor in
        → self.node_grid.get_successors(node_i,
        → node_j):
```



```

successor_node =
    ↳ self.node_grid.get_node(successor[0],
    ↳ successor[1])

if not successor_node.closed:
    successor_node.closed = True
    successor_node.parent = node
    successor_node.f = node.f +
    ↳ self.cost_map.get_edge_cost(
    ↳ node.get_position(),
    ↳ successor_node.get_position())

if successor_node == goal:
    path_greedy = self.construct_path(goal)
    cost_greedy = goal.f
    self.node_grid.reset()
    return path_greedy, cost_greedy

successor_node.g =
    ↳ successor_node.distance_to(
    ↳ goal_position[0], goal_position[1])
heapq.heappush(pq, (successor_node.g,
    ↳ successor_node))

```

3 A*

A implementação deste método foi similar ao anterior, diferenciando-se que o critério de escolha do próximo nó a ser extraído da fila de prioridades será o nó o qual apresentar o menor valor para soma da distância real percorrida até o presente nó e a distância euclidiana do presente nó até o objetivo.

```

pq = []
start = self.node_grid.get_node(start_position[0],
    ↳ start_position[1])
goal = self.node_grid.get_node(goal_position[0],
    ↳ goal_position[1])

start.g = 0
start.f = start.distance_to(goal_position[0],
    ↳ goal_position[1])
heapq.heappush(pq, (start.f, start))

while pq:
    f, node = heapq.heappop(pq)
    node.closed = True
    node_i, node_j = node.get_position()

    if node == goal:
        path_a_star = self.construct_path(goal)
        cost_a_star = goal.f
        self.node_grid.reset()
        return path_a_star, cost_a_star

    for successor in
        ↳ self.node_grid.get_successors(node_i,
        ↳ node_j):

        successor_node =
            ↳ self.node_grid.get_node(successor[0],
            ↳ successor[1])

        if successor_node.f > node.g +
            ↳ self.cost_map.get_edge_cost(
            ↳ node.get_position(),
            ↳ successor_node.get_position()) +
            ↳ successor_node.distance_to(
            ↳ goal_position[0], goal_position[1]):

            successor_node.g = node.g +
                ↳ self.cost_map.get_edge_cost(
                ↳ node.get_position(),
                ↳ successor_node.get_position())

            successor_node.f = successor_node.g +
                ↳ successor_node.distance_to(
                ↳ goal_position[0], goal_position[1])

        successor_node.parent = node
        heapq.heappush(pq, (successor_node.f,
            ↳ successor_node))

```

4 Análise de Resultados

Utilizando *Monte Carlo*, calculou-se tempo médio e o custo do caminho médio gasto pelos três algoritmos implementados para 100 pontos de partida e chegada distintos. Observa-se na Tab. 1 os resultados obtidos.

5 Imagens do funcionamento

Nas Fig. 1, 2, 3, 4, 5 e 6 pode-se comparar o caminho planejado por cada algoritmo implementado para os seis primeiros casos comparados na seção anterior.

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	0.54107	0.29824	79.82919	38.57096
Greedy Search	0.02215	0.00300	103.34198	59.40972
A*	0.17403	0.16950	79.82919	38.57096

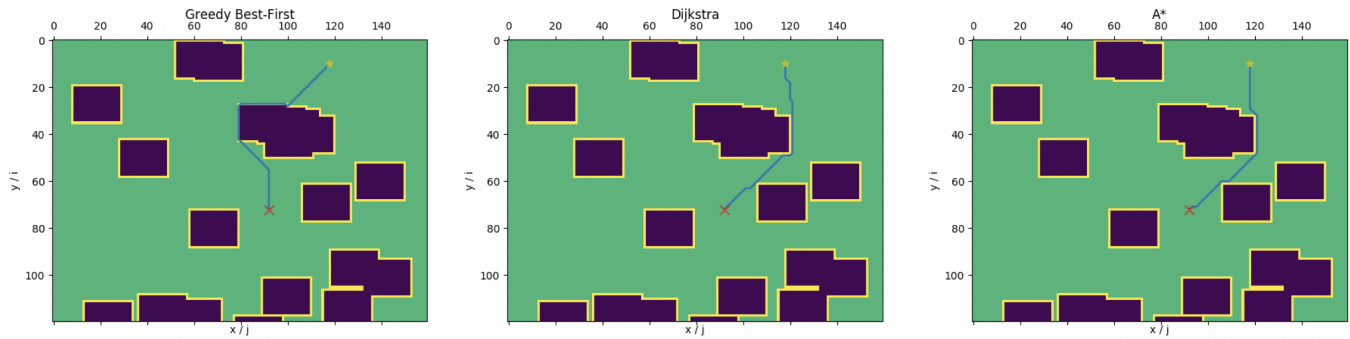


Figura 1: Comparação entre os algoritmos de planejamento de caminho - percurso 1

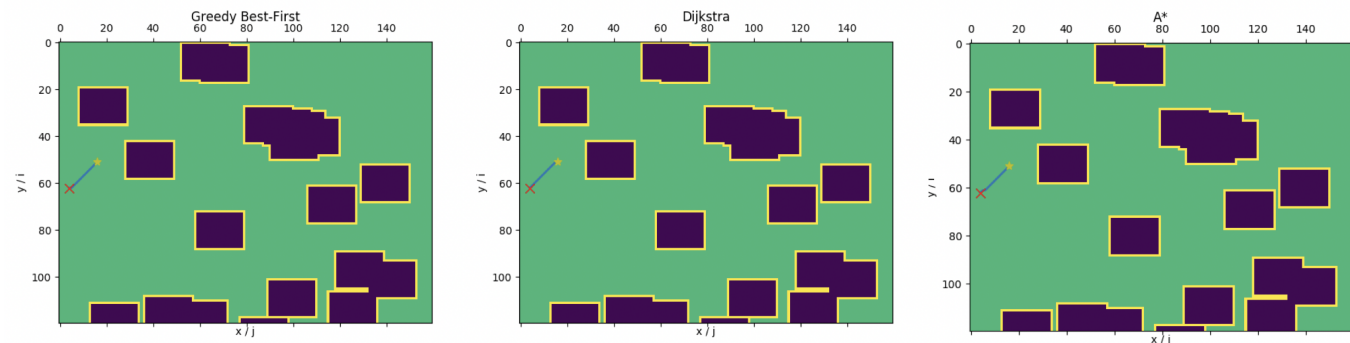


Figura 2: Comparação entre os algoritmos de planejamento de caminho - percurso 2

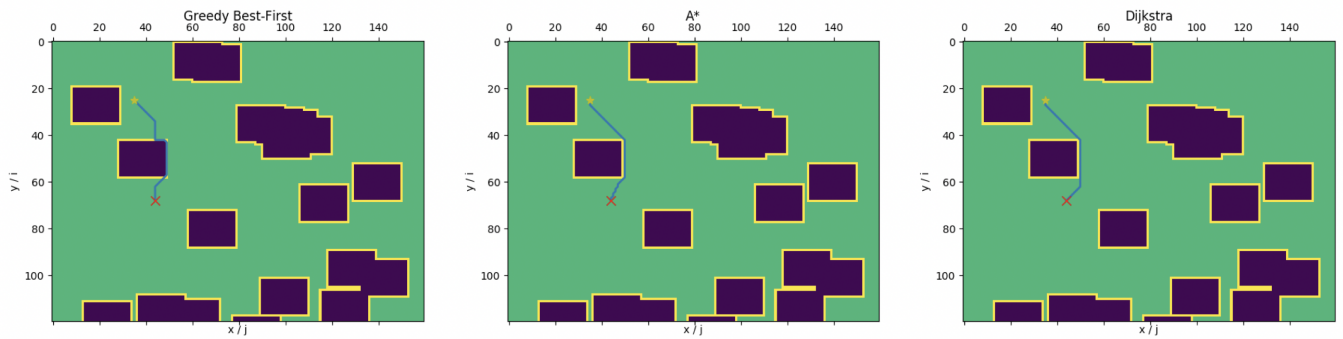


Figura 3: Comparação entre os algoritmos de planejamento de caminho - percurso 3

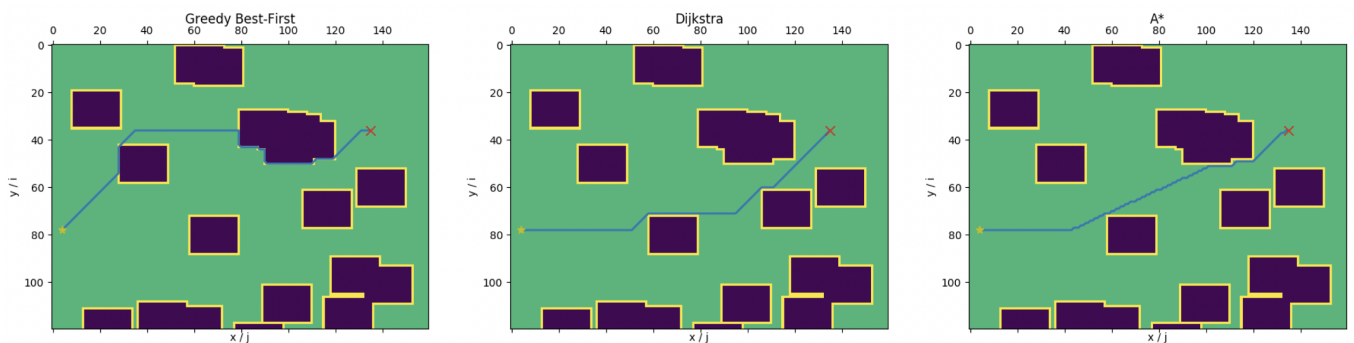


Figura 4: Comparação entre os algoritmos de planejamento de caminho - percurso 4

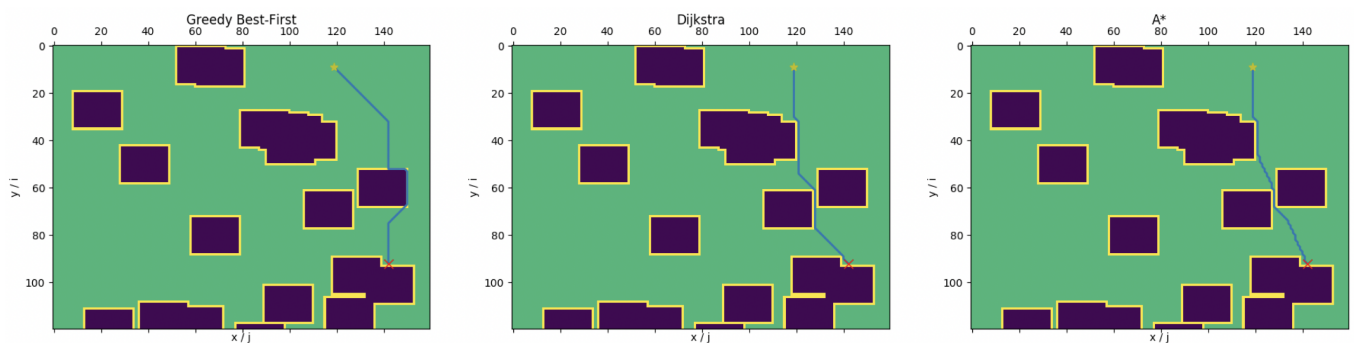


Figura 5: Comparação entre os algoritmos de planejamento de caminho - percurso 5

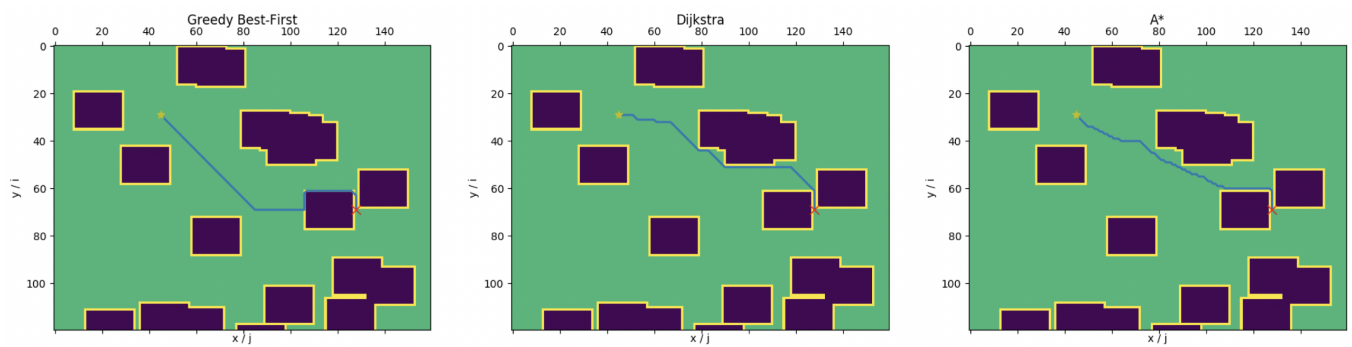


Figura 6: Comparação entre os algoritmos de planejamento de caminho - percurso 6