Instituto Tecnológico de Aeronáutica – ITA
Inteligência Artificial para Robótica Móvel – CT-213
Professor: Marcos Ricardo Omena de Albuquerque Maximo
Aluno: Carlos Matheus Barros da Silva

Lab 02:

O Lab foi desenvolvido com sucesso. Foram implementados os três métodos de busca, e funcionaram de acordo com o esperado. Foi feita uma breve explicação na parte relativa a cada um deles, bem como o trecho de código e duas imagens. Os resultados gerais de tempo computacional e custo do caminho está apresentado na Tabela 1.

Tabela 1:

Algoritmo	Tempo computacional (segundos)		Custo do caminho	
	Média	Desvio padrão	Média	Desvio Padrão
Dijkstra	0.2906153726577759	0.1661143032490671	79.84576582513603	38.57375391124076
Greedy Search	0.011470966339111328	0.0026041937694367	103.34198082325912	59.40972195676692
A*	0.09073643207550049	0.0907858256727782	79.8291972826411	38.57096237576532

Dijkstra – Implementado utilizando o custo dos nós como sendo o custo do nó inicial mais o custo para ir àquele nó, o que é calculado pelo método fornecido *get_edge_cost()*, de maneira que a condição inicial é de que o nó inicial tem custo zero.
A implementação do algoritmo pode ser vista no Código 1. Dois casos testes então apresentados nas Figuras 5 e Figura 6.

Compute time: mean: 0.2906153726577759, std: 0.1661143032490671
Cost: mean: 79.84576582513603, std: 38.57375391124076

```
def dijkstra(self, start_position, goal_position):
    found = False
    pq = []
    node = start_position
    cost = self.node_grid.get_node(node[0], node[1]).f = 0
    heapq.heappush(pq, (cost, node))

    while len(pq) != 0 and not found:

        (cost, node) = heapq.heappop(pq)
        node_node = self.node_grid.get_node(node[0], node[1])
        node_node.closed = True

        for successor in self.node_grid.get_successors(node[0], node[1]):
            successor_cost = self.node_grid.get_node(successor[0], successor[1]).f
            calculated_cost = cost + self.cost_map.get_edge_cost(node, successor)
            if successor_cost > calculated_cost:
                successor_node = self.node_grid.get_node(successor[0], successor[1])
                successor_cost = successor_node.f = calculated_cost
                heapq.heappush(pq, (successor_cost, successor))
                successor_node.parent = node_node
            if successor == goal_position:
                found = True
                break

    path = PathPlanner.construct_path(self.node_grid.get_node(goal_position[0],
```

```
goal_position[1]))
    cost = self.node_grid.get_node(goal_position[0], goal_position[1]).f
    self.node_grid.reset()

    return path, cost
```

Código 1: Método Dijkstra

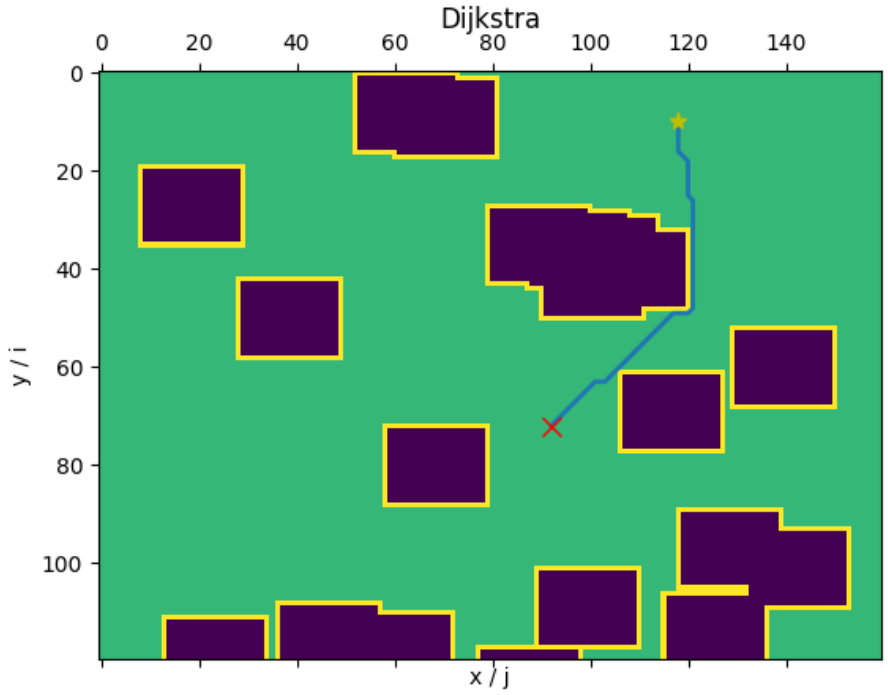

Figura 1: Algoritmo Dijkstra, caso teste dijkstra_0

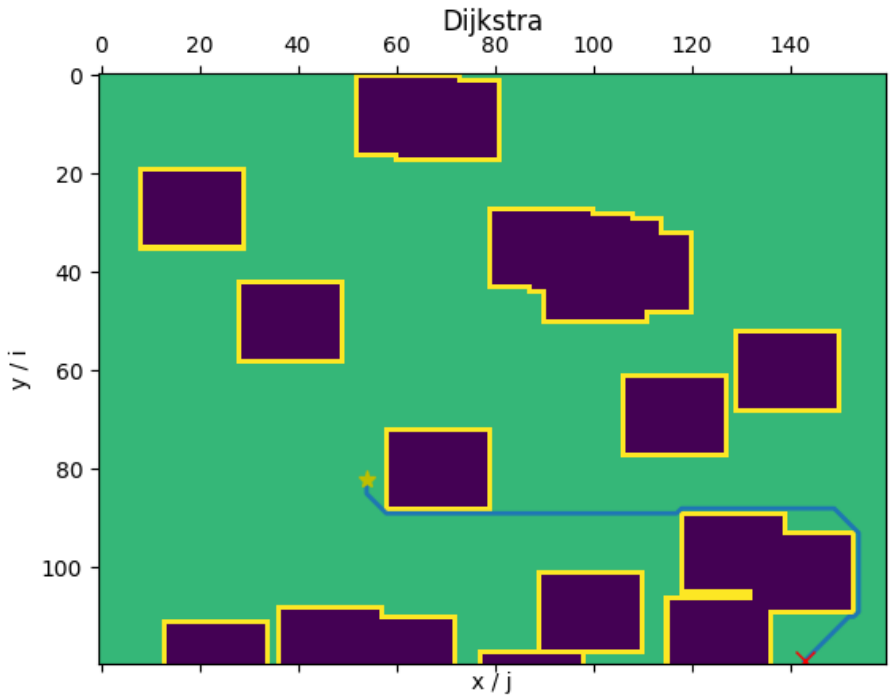

Figura 2: Algoritmo Dijkstra, caso teste dijkstra_55

Greedy – Implementado utilizando o custo dos nós como sendo o valor da distância Euclidiana do nó até o destino final, isto é, dado um nó com coordenadas (x, y) e um ponto que representa o destino final com coordenadas (x_f, y_f), o valor do custo desse nó será c dado pela equação:

$$c = \sqrt{(x - x_f)^2 + (y - y_f)^2}$$

Cada nó inserido na fila de prioridade era marcado como *closed* imediatamente, uma vez que o custo dele não seria diferente se ele fosse descoberto a partir de outro nó, já que seu custo depende apenas de sua distância ao destino.
A implementação do algoritmo pode ser vista no Código 2. O código da função da heurística de custo pode ser visto no Código 3. Dois casos testes então apresentados nas Figuras 5 e Figura 6.

Compute time: mean: 0.011470966339111328, std: 0.0026041937694367
Cost: mean: 103.34198082325912, std: 59.40972195676692

```
def greedy(self, start_position, goal_position):
    pq = []
    node = start_position
    cost = PathPlanner.heuristic(node, goal_position)
    self.node_grid.get_node(node[0], node[1]).f = 0
    heapq.heappush(pq, (cost, node))

    while len(pq) != 0:
```

```
        (cost, node) = heapq.heappop(pq)
        node_node = self.node_grid.get_node(node[0], node[1])

        node_node.closed = True

        if node == goal_position:
            break

        for successor in self.node_grid.get_successors(node[0], node[1]):
            successor_node = self.node_grid.get_node(successor[0], successor[1])
            if not successor_node.closed:
                successor_node.f = node_node.f + self.cost_map.get_edge_cost(node,
successor)
                successor_node.parent = node_node
                successor_cost = PathPlanner.heuristic(successor, goal_position)
                heapq.heappush(pq, (successor_cost, successor))
                successor_node.closed = True

    path = PathPlanner.construct_path(self.node_grid.get_node(goal_position[0],
goal_position[1]))
    cost = self.node_grid.get_node(goal_position[0], goal_position[1]).f
    self.node_grid.reset()
    return path, cost
```

Código 2: Método greedy

```
@staticmethod
def heuristic(node, goal_position):
    return ((node[0] - goal_position[0])**2 + (node[1] - goal_position[1])**2)**(1/2)
```

Código 3: Método heuristic

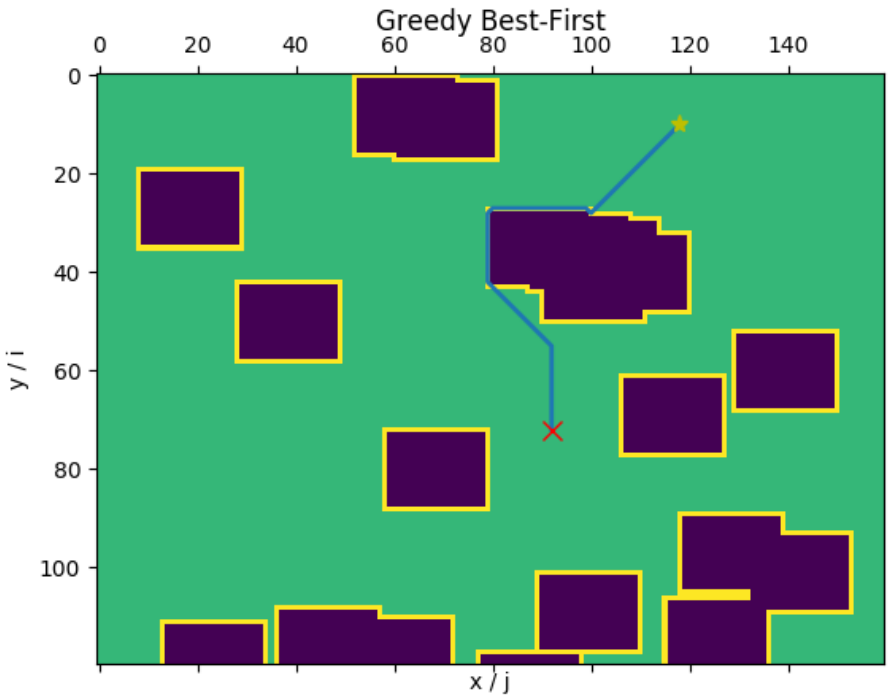

Figura 3: Algoritmo Greedy, caso teste greedy_0

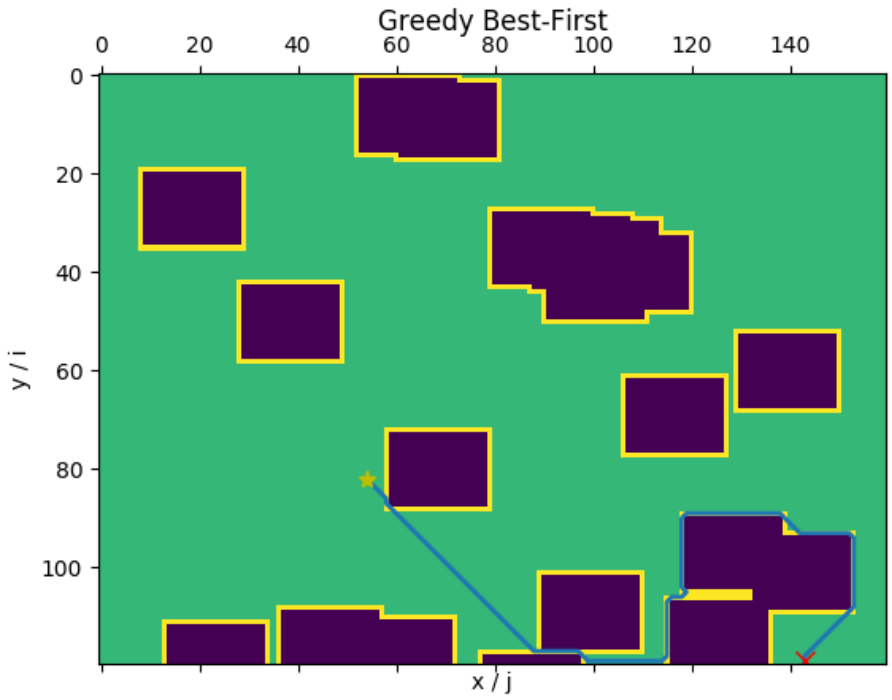

Figura 4: Algoritmo Greedy, caso teste greedy_55

---

*A** – Implementado utilizando o custo dos nós como sendo o valor que eles teriam no caso do método Dijkstra mais o valor da distância Euclidiana do nó até o destino final, isto é, dado um nó com coordenadas (x, y) e um ponto que representa o destino final com coordenadas (x_f, y_f) e custo daquele nó em Dijkstra c_d, o valor do custo desse nó será c dado pela equação:

$$c = c_d + \sqrt{(x - x_f)^2 + (y - y_f)^2}$$

A implementação do algoritmo pode ser vista no Código 4. O código da função da heurística de custo pode ser visto no Código 5. Dois casos testes então apresentados nas Figuras 5 e Figura 6.

Compute time: mean: 0.09073643207550049, std: 0.0907858256727782
Cost: mean: 79.8291972826411, std: 38.57096237576532

```
def a_star(self, start_position, goal_position):
    pq = []
    node = start_position
    node_node = self.node_grid.get_node(node[0], node[1])

    node_node.g = 0
    node_node.f = PathPlanner.heuristic(node, goal_position)

    heapq.heappush(pq, (node_node.f, node))
```

```
    while len(pq) != 0:
        (cost, node) = heapq.heappop(pq)
        node_node = self.node_grid.get_node(node[0], node[1])
        node_node.closed = True

        if node == goal_position:
            break

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

            g_calculated_cost = node_node.g + self.cost_map.get_edge_cost(node,
successor)
            f_calculated_cost = g_calculated_cost + PathPlanner.heuristic(successor,
goal_position)

            if successor_node.f > f_calculated_cost:
                if not successor_node.closed:
                    successor_node.g = g_calculated_cost
                    successor_node.f = f_calculated_cost
                    successor_node.parent = node_node
                    heapq.heappush(pq, (successor_node.f, successor))

    path = PathPlanner.construct_path(self.node_grid.get_node(goal_position[0],
goal_position[1]))
    cost = self.node_grid.get_node(goal_position[0], goal_position[1]).g

    self.node_grid.reset()
    return path, cost
```

Código 4: Método greedy

```
@staticmethod
def heuristic(node, goal_position):
    return ((node[0] - goal_position[0])**2 + (node[1] - goal_position[1])**2)**(1/2)
```

Código 5: Método heuristic

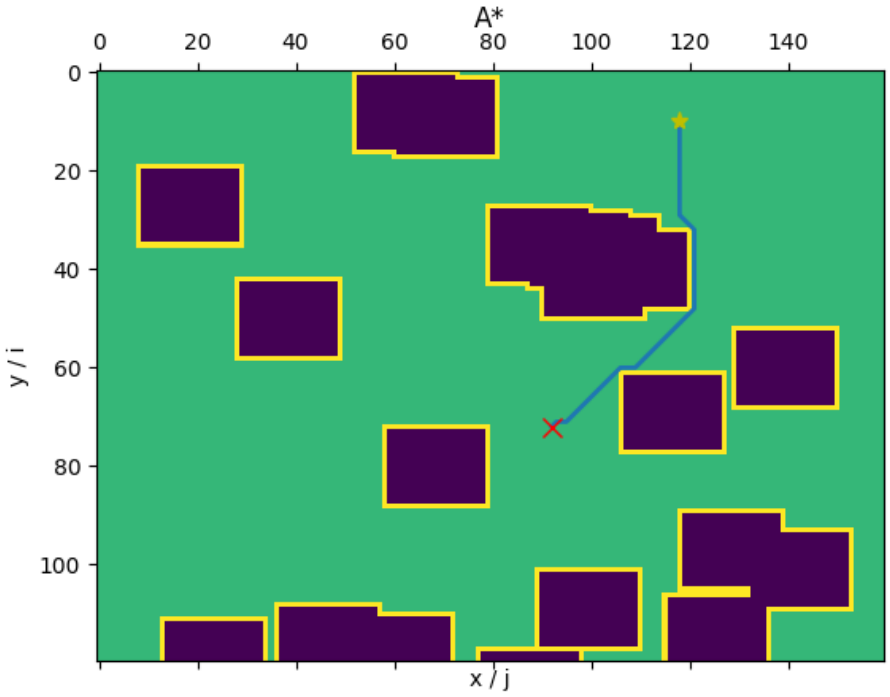

Figura 5: Algoritmo A, caso teste a_star_0*

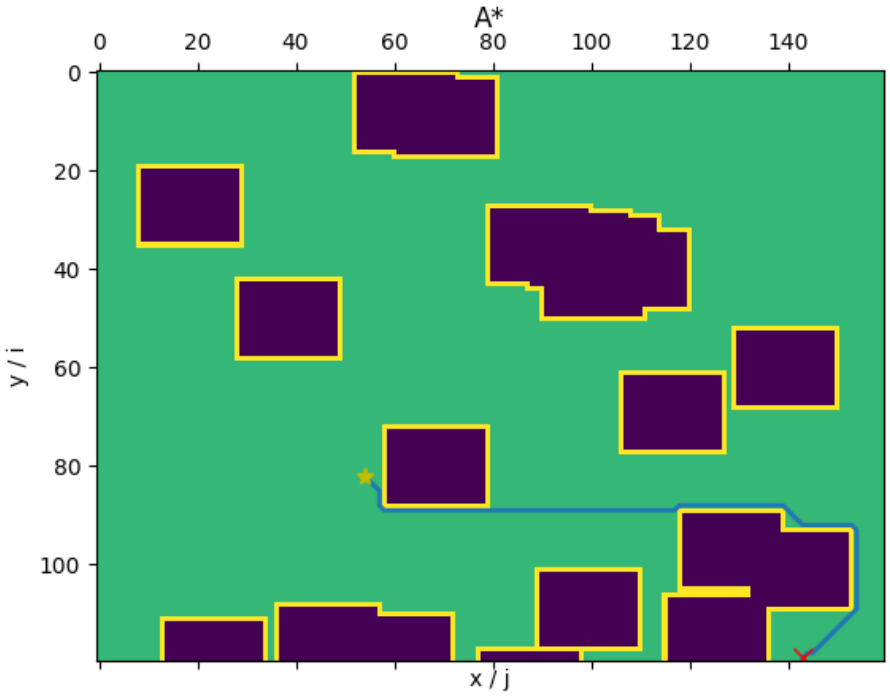

Figura 6: Algoritmo A, caso teste a_star_55*