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.

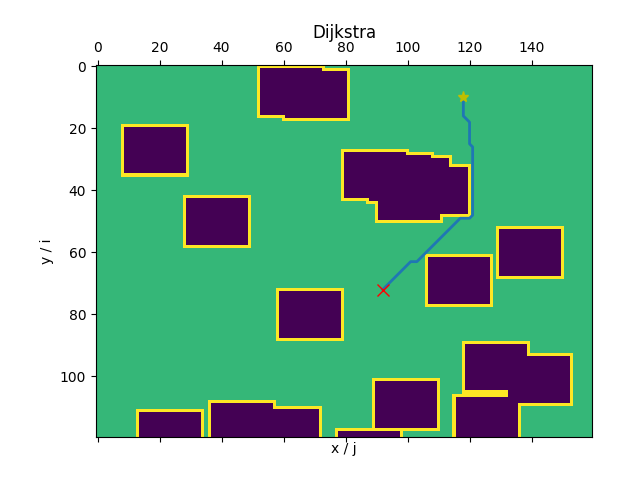
A implementação do algoritmo pode ser vista no Código 1. Dois casos testes então apresentados nas Figuras 5 e Figura 6.

Compute time: mean: 0.2906153726577759, std: 0.1661143032490671

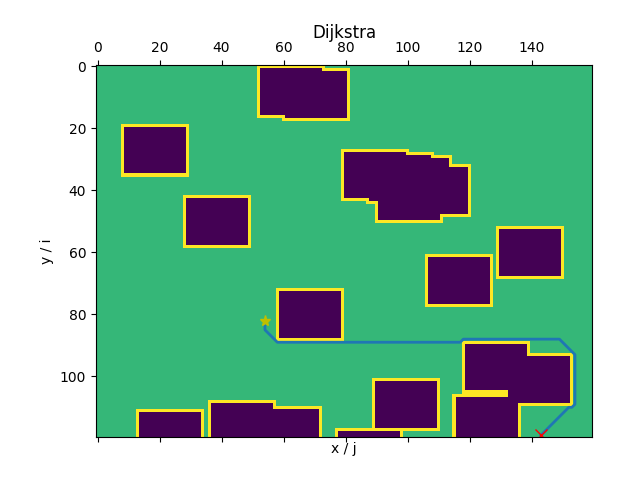
Cost: mean: 79.84576582513603, std: 38.57375391124076

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 e um ponto que representa o destino final com coordenadas , o valor do custo desse nó será dado pela equação:

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

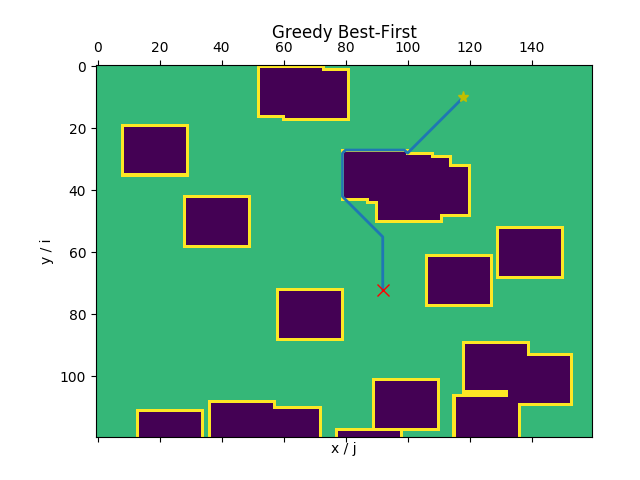
def greedy(self, start\_position, goal\_position):pq = []  
 node = start\_position  
 cost = PathPlanner.heuristic(node, goal\_position)  
 self.node\_grid.get\_node(node[0], node[1]).f = 0  
 heapq.heappush(pq, (cost, node))

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

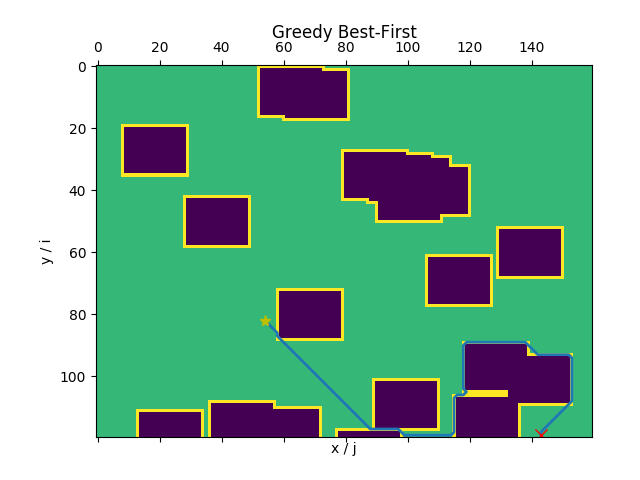
*Código 2: Método greedy*

@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 e um ponto que representa o destino final com coordenadas e custo daquele nó em Dijkstra , o valor do custo desse nó será dado pela equação:

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

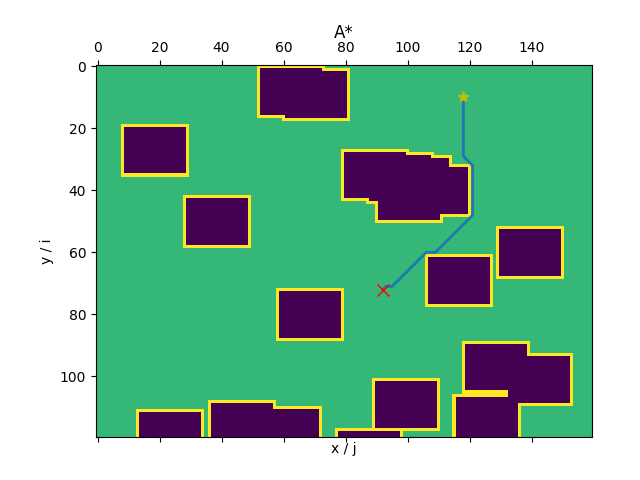
Cost: mean: 79.8291972826411, std: 38.57096237576532

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

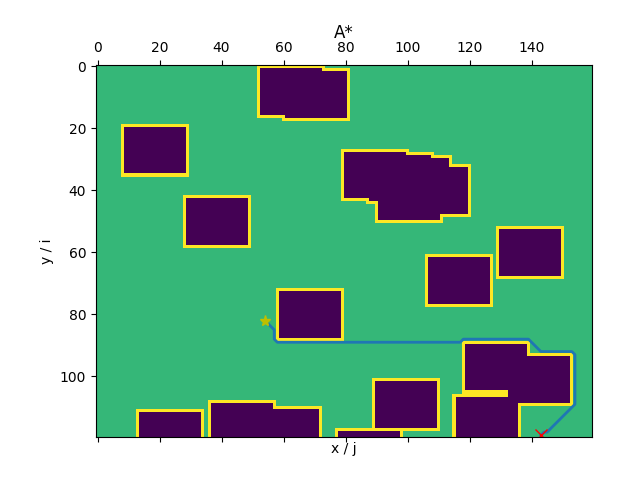
*Código 4: Método greedy*

@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*