CYBER-PHYSICAL SYSTEMS AND ROBOTICS

Lab 3. Motion planning

# Preparation

1. The property decorator is used as a setter and getter are used in C++ or other programming languages. This allows for validation, debugging, and more readability in the Python code.
2. (x, y) = (−4, −4) (r,c) = (8,0)

(r,c) = (0,8) (x,y) = (4,4)

# Code

## Creating an admissible heuristic

In order to create an admissible heuristic, the only condition is to make an optimistic guess of how far our robot is from the goal. The easiest way to do this when dealing with a discrete distance, like in our case, is computing the Manhattan distance between any given point and the start. This is the number of steps from the starting point to a certain square without taking walls into account. The following code is a simple implementation of this idea.

def \_compute\_heuristic(self, goal: Tuple[float, float]) -> np.ndarray:  
 *"""Creates an admissible heuristic.  
  
 Args:  
 goal: Destination location in (x,y) coordinates.  
  
 Returns:  
 Admissible heuristic.  
  
 """* map\_rows, map\_cols = np.shape(self.\_map.grid\_map)  
  
  
 heuristic = np.ndarray(shape = (map\_rows, map\_cols), dtype = int)  
  
 goal\_rc = self.\_xy\_to\_rc(goal)  
  
 for i in range(0, map\_rows):  
 for j in range (0, map\_cols):  
 heuristic[i][j] = abs(goal\_rc[0] - i) + abs(goal\_rc[1] - j)  
  
 return heuristic

Code 1: Heuristic computation

As explained before, each grid space is assigned a value depending on its manhattan distance to the start. In Figure 1 we have the resulting heuristic which we will use for the A\* algorithm implemented in the following steps.



Figure 1: Resulting heuristic array

Note that the order is flipped when compared with the map, but it is symmetrical and is only a product of naming rows and columns in a different order. Once we have it we can start calculating the optimal path.

## Searching with A\*

In order to compute the A\* algorithm we followed the following steps:

First, we transform the input data into rc coordinates and obtain information such as the number of rows and columns. We also create empty lists for closed\_list, open\_list and an empty array for the ancestors.

heuristic = self.\_compute\_heuristic(goal)  
  
g = 0  
goal\_rc = self.\_xy\_to\_rc(goal)  
start\_rc = self.\_xy\_to\_rc(start)  
  
f = 0  
  
i = 1  
  
point = (start\_rc[0], start\_rc[1], heuristic[start\_rc[0], start\_rc[1]], g) #Current point  
  
open\_list = []  
closed\_list = []  
  
neighbours = []  
  
map\_rows, map\_cols = np.shape(self.\_map.grid\_map)  
  
ancestors = np.full(shape = (map\_rows, map\_cols), fill\_value = None)  
  
reconstruct\_path = []  
  
ancestors[start\_rc[0], start\_rc[1]] = (start\_rc[0], start\_rc[1], g)

Code 2: Variable initialization

Second, inside a loop that executes until we arrive to the goal, for each direction (top, right, left and bottom) we check if we’re going straight, backwards or turning. After that, depending on the old direction and the new direction, we add to a neighbours list the 4 possible movements with their respective weights depending on the type of movement they make.

while (point[0], point[1]) != goal\_rc:  
 d\_x = point[0] - ancestors[point[0], point[1]][0]  
 d\_y = point[1] - ancestors[point[0], point[1]][1]  
 direction = [d\_x,d\_y]  
  
 new\_direction = [-1,0] #RIGHT  
 a = direction[0] \* new\_direction[0] + direction[1] \* new\_direction[1]  
 c = np.dot(np.asanyarray(direction), np.asanyarray(new\_direction)) # -> cosine of the angle  
  
 if c == 1:  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[0])) #STRAIGHT  
 elif c == -1:  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[1])) # BACKWARDS  
 elif c == 0:  
 if d\_x == 0 and d\_y != 0:  
 #Se movía en vertical  
 if d\_y > 0:  
 #Se movía aumentando y  
 if new\_direction[0] > 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 if new\_direction[0] < 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if d\_y > 0:  
 #Se movía disminuyendo y  
 if new\_direction[0] > 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if new\_direction[0] < 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 if d\_y == 0 and d\_x != 0:  
 #Se movía en horizontal  
 if d\_x > 0:  
 #Se movía aumentando x  
 if new\_direction[1] > 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 if new\_direction[1] < 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if d\_x < 0:  
 #Se movía aumentando x  
 if new\_direction[1] > 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if new\_direction[1] < 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 else:  
 neighbours.append((point[0] - 1, point[1], 1)) #FIRST STEP

Code 3: Direction computation and neighbour appending

Then, for every neighbor we check if they are reasonable data (they have real and sensible coordinates) and if they’re on top of an obstacle. If they aren’t, we calculate the new g with the one of the current point and the g corresponding to each neighbor and if that point hasn’t been added to the ancestors matrix or if they’re already in the ancestor matrix but now a new path has been found with a lower g, we take them into the open list and add them to the ancestor list. The second case is there just in the remote case that happens.

for neighbour in neighbours:  
 try:  
 if self.\_map.grid\_map[neighbour[0], neighbour[1]] != 1 and np.sign(neighbour[0]) != -1 and np.sign(neighbour[1]) != -1 and neighbour[0] < map\_rows and neighbour [1] < map\_cols:  
 g = point[3] + neighbour[2]  
 if ancestors[neighbour[0], neighbour[1]] is None or ancestors[neighbour[0], neighbour[1]][2] > g:  
 open\_list.append((neighbour[0], neighbour[1], heuristic[neighbour[0], neighbour[1]] + g, g))  
 ancestors[neighbour[0],neighbour[1]] = (point[0], point[1], g)  
 except:  
 print("Point not in map")

Code 4: Adding neighbours to the open list

Then, we sort the open list depending on the lower f, and after that we move that point to the closed list and our new current point will be that particular point.

neighbours.clear()  
  
open\_list.sort(key = lambda r: (r[2]))  
  
point = open\_list[0]  
  
closed\_list.append(open\_list[0][:2])  
  
print("Number of steps: " + str(i))  
  
i += 1  
  
del open\_list[0]

Code 5: Adding the point to the closed list

When we have exited the loop, we call the reconstruct\_path function and reverse the result obtained from the previous call, because it’s given from goal to start instead of from start to goal.

reconstruct\_path = self.\_reconstruct\_path(start\_rc, goal\_rc, ancestors = ancestors)  
  
reconstruct\_path.reverse()  
  
return reconstruct\_path

Code 6: Calliong reconstruct\_path to obtain the path

## Returning the optimal path

Since we are using the ancestors matrix, returning the optimal path is really simple and easy. We start in the goal point and we find which point discovered the goal, and we add that point to the path. We continue this algorithm until we get to the start point. That’s why we reversed the path in the A\* function.

The path is transformed to xy coordinates because our path smoothing algorithm can work with those kind of coordinates.

def \_reconstruct\_path(self, start: Tuple[float, float], goal: Tuple[float, float], ancestors: np.ndarray) -> \  
 List[Tuple[float, float]]:  
 *"""Computes the trajectory from the start to the goal location given the ancestors of a search algorithm.  
  
 Args:  
 start: Initial location in (x, y) format.  
 goal: Goal location in (x, y) format.  
 ancestors: Matrix that contains for every cell, None or the (x, y) ancestor from which it was opened.  
  
 Returns: Path to the goal (start location first) in (x, y) format.  
  
 """* path = []  
 current = goal  
  
 path.append(self.\_rc\_to\_xy(goal))  
  
 while current != start:  
 current = (ancestors[current[0], current[1]][0], ancestors[current[0], current[1]][1])  
 path.append(self.\_rc\_to\_xy(current))  
  
 return path

Code 7: Reconstructing the optimal path