# University of California, Riverside

## EE/ME144/EE283A

## Foundations of Robotics

## Fall 2021

# Lab 7 Report

**Nov 17, 2021**

| Name | SID | Section | Group Number |
|---|---|---|---|
| Simraj Singh | 862155747 | Thursday | 4 |
| Shaheriar Malik | 862154387 | Thursday | 4 |

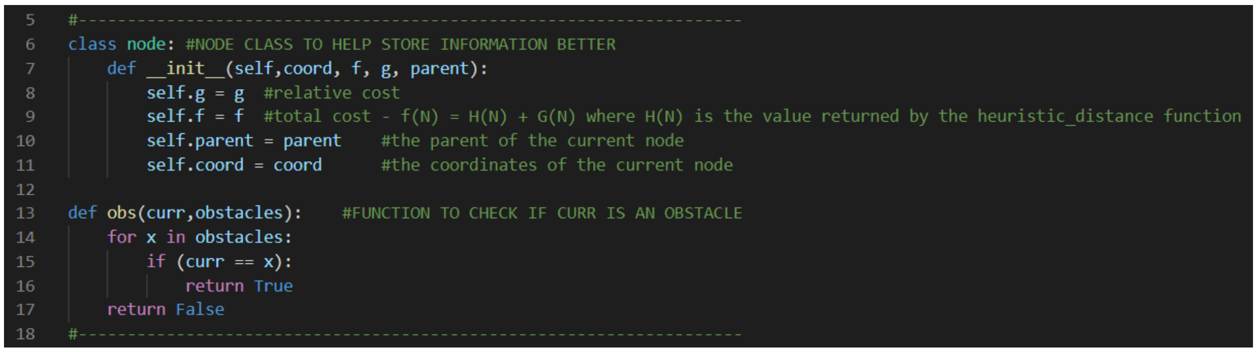
## 1. Problem Statement

In Lab 7, we were tasked with successfully implementing A* algorithm in order to find an optimal path from the start point to the end point in the grid, without colliding with obstacles. We did this by using Euclidean distance as the heuristic function.

## 2. Design Idea

The main idea behind this lab was to implement A* algorithm in python so we could search for waypoints and ultimately program the physical Turtlebot to find an optimal path in Lab 8. We started our solution with a node class in order to more easily access information on the properties of nodes (relative/total cost, parent node, and node coordinates). Continuing to the implementation of A*, we filled out the neighbors() function by defining the list of a node's 4 neighbors (up, down, left, and right). For the heuristic_distance() function, we simply return the distance between the current point and the goal point by using the distance function (Euclidean heuristic).

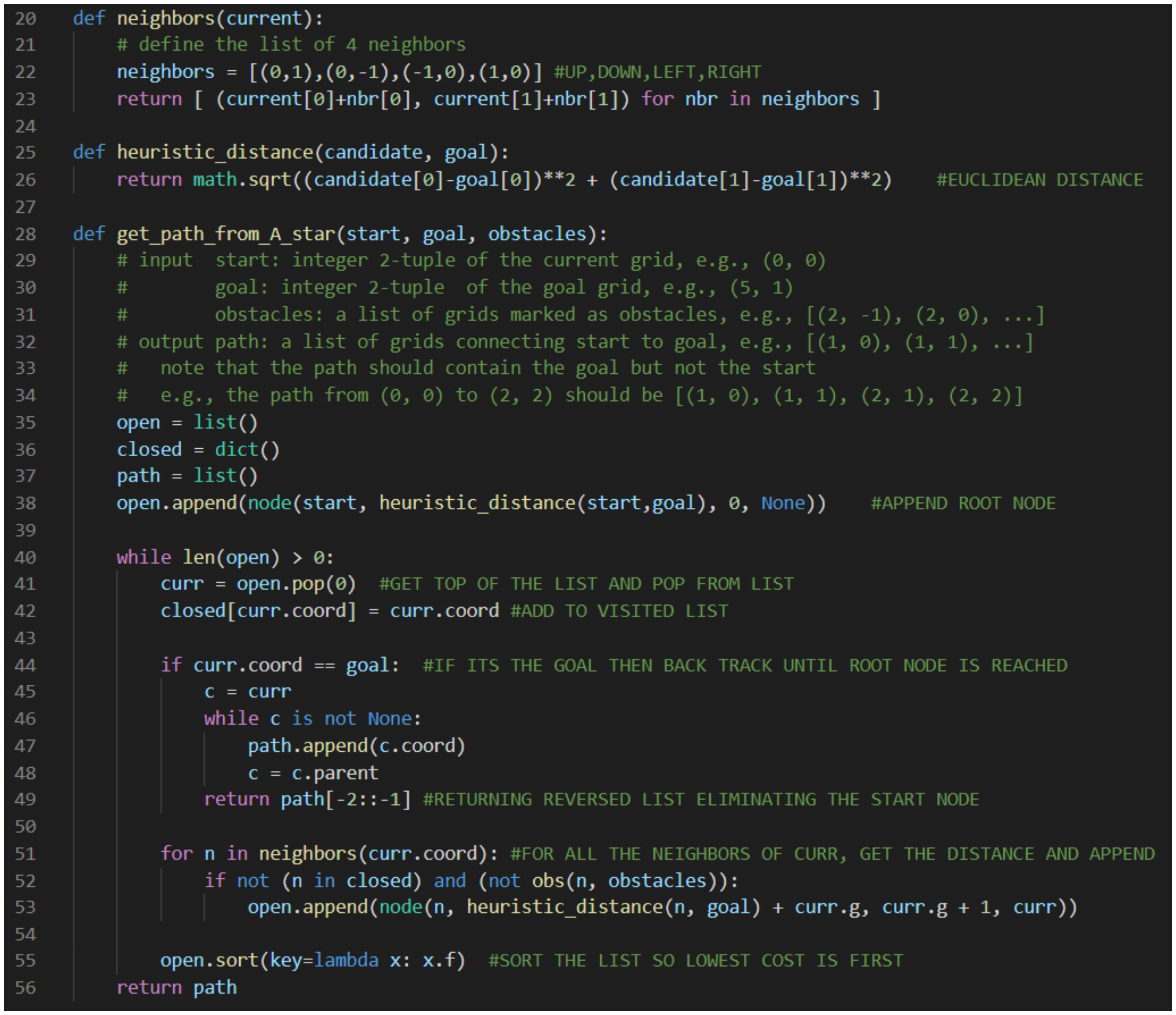
Lastly, we fill out the get_path_from_A_star() function. Starting this function, we define a dictionary for visited nodes and 2 lists for a priority queue and returning the final path. While our queue is non-empty, we get the top node of the queue and add it to our visited list. If we have reached our goal node, we backtrack until we reach the root node and return the path by excluding the start node and returning a reversed list. If it's not the goal node, we get the distance using our heuristic function and append it to our queue. We then sort the queue so we expand the lowest cost first. This logic follows from the pseudocode provided in the lab manual. See figure 7.1 and 7.2 for pictures of our code.



```python
#-------------------------------------------------------------------
class node: #NODE CLASS TO HELP STORE INFORMATION BETTER
    def __init__(self,coord, f, g, parent):
        self.g = g  #relative cost
        self.f = f  #total cost - f(N) = H(N) + G(N) where H(N) is the value returned by the heuristic_distance function
        self.parent = parent    #the parent of the current node
        self.coord = coord      #the coordinates of the current node

def obs(curr,obstacles):    #FUNCTION TO CHECK IF CURR IS AN OBSTACLE
    for x in obstacles:
        if (curr == x):
            return True
    return False
#-------------------------------------------------------------------
```

Figure 7.1: Code implemented for node class with comments



```python
def neighbors(current):
    # define the list of 4 neighbors
    neighbors = [(0,1),(0,-1),(-1,0),(1,0)] #UP,DOWN,LEFT,RIGHT
    return [ (current[0]+nbr[0], current[1]+nbr[1]) for nbr in neighbors ]

def heuristic_distance(candidate, goal):
    return math.sqrt((candidate[0]-goal[0])**2 + (candidate[1]-goal[1])**2)    #EUCLIDEAN DISTANCE

def get_path_from_A_star(start, goal, obstacles):
    # input  start: integer 2-tuple of the current grid, e.g., (0, 0)
    #        goal: integer 2-tuple  of the goal grid, e.g., (5, 1)
    #        obstacles: a list of grids marked as obstacles, e.g., [(2, -1), (2, 0), ...]
    # output path: a list of grids connecting start to goal, e.g., [(1, 0), (1, 1), ...]
    #   note that the path should contain the goal but not the start
    #   e.g., the path from (0, 0) to (2, 2) should be [(1, 0), (1, 1), (2, 1), (2, 2)]
    open = list()
    closed = dict()
    path = list()
    open.append(node(start, heuristic_distance(start,goal), 0, None))    #APPEND ROOT NODE

    while len(open) > 0:
        curr = open.pop(0)  #GET TOP OF THE LIST AND POP FROM LIST
        closed[curr.coord] = curr.coord #ADD TO VISITED LIST

        if curr.coord == goal:  #IF ITS THE GOAL THEN BACK TRACK UNTIL ROOT NODE IS REACHED
            c = curr
            while c is not None:
                path.append(c.coord)
                c = c.parent
            return path[-2::-1] #RETURNING REVERSED LIST ELIMINATING THE START NODE

        for n in neighbors(curr.coord): #FOR ALL THE NEIGHBORS OF CURR, GET THE DISTANCE AND APPEND
            if not (n in closed) and (not obs(n, obstacles)):
                open.append(node(n, heuristic_distance(n, goal) + curr.g, curr.g + 1, curr))

        open.sort(key=lambda x: x.f)  #SORT THE LIST SO LOWEST COST IS FIRST
    return path
```

Figure 7.2: Code implementation for A* search algorithm with comments

## 3. Results

Ultimately, we were able to successfully implement A* search algorithm in order to find an optimal path from the start node to the goal node while avoiding obstacles in the grid. We didn’t run into any troubles during this lab. As seen by Figures 7.1 and 7.2, we followed a slightly more CS approach to this A* implementation as opposed to the ME approach discussed in the lab. Our prior experience with implementing A* in C++ from other CS classes, namely CS170 - Intro to Artificial Intelligence, definitely proved to be useful for this lab. In Lab 8, we will be using this A* algorithm to program the physical Turtlebot to find the optimal trajectory and then traverse that path while ignoring obstacles.

## 4. Appendix

How to run the code for lab 7:

- Make a test python script with a start, goal, and a list of obstacles. These values are tuples. Then call import get_path_from_A_star and call the function and test if the path is the shortest.
- To run the A* algorithm on its own, do the following:
    - `roscd ee144f21/scripts`
    - `python motion_planning.py`