University of California, Riverside

EE 106

PROGRAMMING PRACTICAL ROBOTS

Spring 2024

Lab 6 Report

05/25, 2024

Name	SID	Section	Group Number
Thong Thach	862224662	XX	XX

1. Problem Statement

The objective of this lab is to perform the motion planning for a robot based on a grid map with certain cells designated as obstacles. The task is to use the A* algorithm to find an optimal path from the start cell to the goal cell without colliding with obstacles. You can use Manhattan distance or Euclidean distance as the heuristic function. Other heuristic functions are also acceptable. Diagonal movement is not allowed, meaning the path should be based on a 4-connected graph.

2. Design Idea

The main goal of this lab is to create the open list for all the possible nodes that are connected to the start node

Once all the nodes are planned, the next step is to calculate the cost to take from each node to the other node

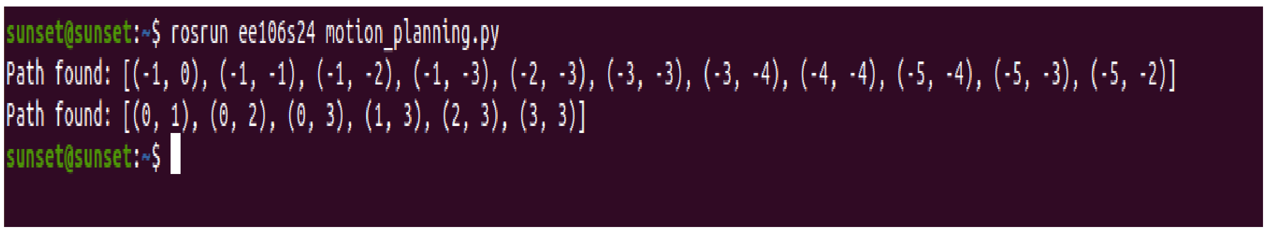
If one of the node is the obstacle or the obstacle is inside the closed_list, then it will be skipped

If the current node has a better path then it will add to the set or update to the cost of the node and add it to the parent node. If the node is found the shortest path, it will update the current node to that parent cost-to-go amount

It will keep repeating again, and again until it reaches the goal

As the motion planning starts to mapping out all the possible nodes and paths, until it reaches the goal. Motion planning will calculate all the cost-to-go to each nodes and starts to eliminate all the longest cost to go from one node to other node, only leaves with small cost. After that, it will update to the current node. More explanation will be based on the "How to run code section".

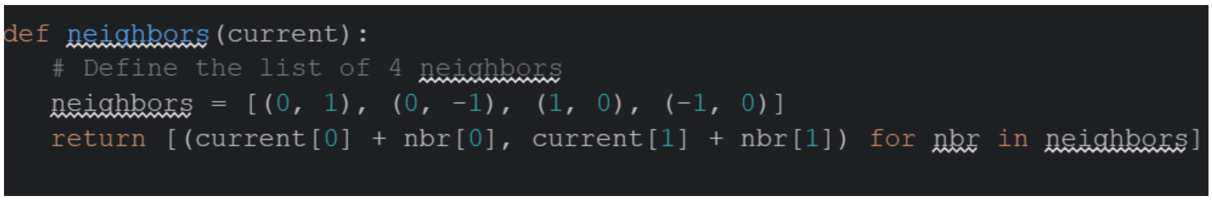
3. Results



```
sunset@sunset:~$ rosrun ee106s24 motion_planning.py
Path found: [(-1, 0), (-1, -1), (-1, -2), (-1, -3), (-2, -3), (-3, -3), (-3, -4), (-4, -4), (-5, -4), (-5, -3), (-5, -2)]
Path found: [(0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (3, 3)]
sunset@sunset:~$
```

4. Appendix (optional)

1) How to run the code



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

This is the coordinate for the robot to move around:

Each square is 1 for the cost to go to each node

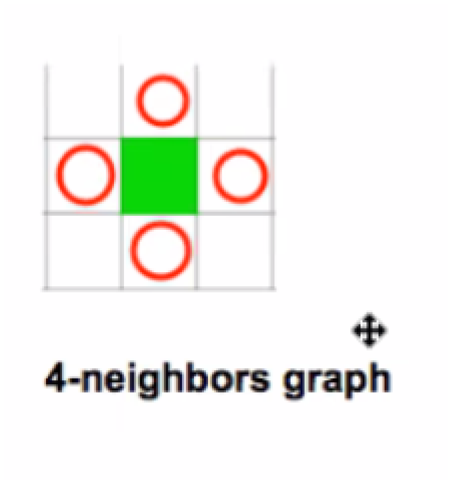
Up: (0,1)

Down(0,-1)

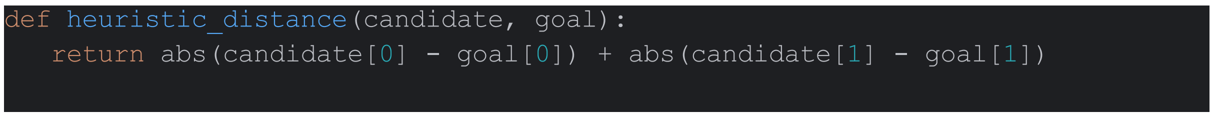
Right(1,0)

Left(-1,0)

Using 4-connected graph



For the pathfinding algorithm:



```python
def heuristic_distance(candidate, goal):
    return abs(candidate[0] - goal[0]) + abs(candidate[1] - goal[1])
```

Based on the code above, it basically finds the new cost by adding the past cost and the heuristic distance. The robot will move in grid representation, an occupancy grid maps in 2D space.

For the get_path_from_A_star

Setting up the arrays for the coordinate of the robot to move from start to the goal

```
def get_path_from_A_star(start, goal, obstacles):
   open_list = [(0, start)]
   past_cost = {}
   past_cost[start] = 0
   parent = {}
   parent[start] = None
   closed_list = set()
```

Open_list array will initialized with the cost of 0 with the starting point

The past_cost records the cost for each nodes. For the first node, the past node will be at 0.

The parent_node[] will construct the path when the goal is done

The closed_list will check if the node in the array is repeated again. If the node is repeated, the closed_list will stop the list of the array.

```
 while open_list:
       current = open_list.pop(0)[1]
       closed_list.add(current)

       if current == goal:
           path = []
           while current is not None:
               path.append(current)
               current = parent[current]

           return path[::-1][1:]

       for candidate in neighbors(current):
           if candidate in obstacles or candidate in closed_list:
               continue

           tentative_past_cost = past_cost[current] + 1  # Cost of moving to
the neighbor is always 1

           if candidate not in past_cost or tentative_past_cost <
past_cost[candidate]:
               past_cost[candidate] = tentative_past_cost
               parent[candidate] = current
               new_cost = tentative_past_cost + heuristic_distance(candidate,
goal)
```

```
               heappush(open_list, (new_cost, candidate))

   return None  # No path found
```

First, the robot will look for the first node.

After that, the closed_list will will add that current node in order to check if the node whether it is repeated again or not.

If the current node is a goal, then it will add that node into the path array. That's the whole section of this node will do as the pseudo code shows below:

```
 if current == goal:
           path = []
           while current is not None:
               path.append(current)
               current = parent[current]
```

Next, the robot will loop through all the current node

If one of the node is the obstacle or the node is moving into the closed_list, then it will be skipped

```
 for candidate in neighbors(current):
           if candidate in obstacles or candidate in closed_list:
               continue
```

Next, the robot will calculate the cost for moving each node. The cost for moving each will be 1 as shown below

```
tentative_past_cost = past_cost[current] + 1  # Cost of moving to the
neighbor is always 1
```

If the current node is not in the past_cost, is not checked yet, or the current node has a better path, `tentative_past_cost < past_cost[candidate]`, then it will add to the set or update to the cost of the node and add it to the parent node as shown below:

```
  if candidate not in past_cost or tentative_past_cost <
past_cost[candidate]:
               past_cost[candidate] = tentative_past_cost
               parent[candidate] = current
               new_cost = tentative_past_cost + heuristic_distance(candidate,
goal)
               heappush(open_list, (new_cost, candidate))
```

For the last one, set up the start coordinate, the goal coordinate, and the obstacle coordinate will tell the robot to start motion planning.

Call the function **`get_path_from_A_star()`** so the robot start to do motion planning , moving from start to goal and avoid the obstacle by planning out its path.

```
if __name__ == '__main__':
   start = (0, 0)
   goal = (-5, -2)
   obstacles = [(-2, 1), (-2, 0), (-2, -1), (-2, -2), (-4, -2), (-4, -3)]
   path = get_path_from_A_star(start, goal, obstacles)
   if path:
       print("Path found:", path)
   else:
       print("No path found")

   # For the test case
   start = (0, 0)
   goal = (3, 3)
```

2) References

The external source is only: [Welcome to UCR EE106 — UCR EE106 Spring 2024 0.1 documentation (ucr-ee106.readthedocs.io)] and the help from TA

3) Scripts

Copy and paste the scripts you have talked about in the above sections, such that the reader can locate the code snippets in the script file. This is just for the completeness of the report.

```
#!/usr/bin/env python3

from heapq import heappop, heappush

def neighbors(current):
   # Define the list of 4 neighbors
   neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
   return [(current[0] + nbr[0], current[1] + nbr[1]) for nbr in
neighbors]

def heuristic_distance(candidate, goal):
   return abs(candidate[0] - goal[0]) + abs(candidate[1] - goal[1])

def get_path_from_A_star(start, goal, obstacles):
   open_list = [(0, start)]
   past_cost = {}
   past_cost[start] = 0
   parent = {}
   parent[start] = None
   closed_list = set()

   while open_list:
       current = open_list.pop(0)[1]
       closed_list.add(current)

       if current == goal:
           path = []
           while current is not None:
               path.append(current)
               current = parent[current]

           path.reverse()
           return path[1:]


       for candidate in neighbors(current):
           if candidate in obstacles or candidate in closed_list:
               print("Skip this iteration")
               continue

           tentative_past_cost = past_cost[current] + 1  # Cost of
moving to the neighbor is always 1

           if candidate not in past_cost or tentative_past_cost <
past_cost[candidate]:
               past_cost[candidate] = tentative_past_cost
               parent[candidate] = current
               new_cost = tentative_past_cost +
heuristic_distance(candidate, goal)
               heappush(open_list, (new_cost, candidate))

   return None  # No path found

if __name__ == '__main__':
   start = (0, 0)
   goal = (-5, -2)
   obstacles = [(-2, 1), (-2, 0), (-2, -1), (-2, -2), (-4, -2), (-4,
-3)]
   path = get_path_from_A_star(start, goal, obstacles)
```

```
if path:
    print("Path found:", path)
else:
    print("No path found")

# For the test case
start = (0, 0)
goal = (3, 3)
obstacles = []
path = get_path_from_A_star(start, goal, obstacles)
if path:
    print("Path found:", path)
else:
    print("No path found")
```