**Date:4/4/24 EXPERIMENT 11**

**Name:Rithika shree**

**21R231**

**Path Planning Algorithm using ROS**

**Aim:**

To learn the Path Planning Algorithms (A\* and Dijkstra’s algorithms).

**Software/ Package Used:**

<https://realitybytes.blog/2018/08/17/graph-based-path-planning-a/amp/>

<https://github.com/SakshayMahna/Robotics-Playground/tree/main/turtlebot3_ws>.

ROS

**Programs:**

**A\* algorithm :**

**CODE:**

#! /usr/bin/env python

import matplotlib.pyplot as plt

from copy import deepcopy

COLOR\_MAP = (0, 8)

class PathPlanner:

def \_\_init\_\_(self, grid, visual=False):

self.grid = grid

self.visual = visual

self.heuristic = None

self.goal\_node = None

def calc\_heuristic(self):

row = len(self.grid)

col = len(self.grid[0])

self.heuristic = [[0 for x in range(col)] for y in range(row)]

for i in range(row):

for j in range(col):

row\_diff = abs(i - self.goal\_node[0])

col\_diff = abs(j - self.goal\_node[1])

self.heuristic[i][j] = int(abs(row\_diff - col\_diff) + min(row\_diff, col\_diff) \* 2)

print ("Heuristic:")

for i in range(len(self.heuristic)):

print self.heuristic[i]

def a\_star(self, start\_cart, goal\_cart):

goal = [goal\_cart[1], goal\_cart[0]]

self.goal\_node = goal

init = [start\_cart[1], start\_cart[0]]

# Calculate the Heuristic for the map

self.calc\_heuristic()

print init, goal

if self.visual:

viz\_map = deepcopy(self.grid)

fig = plt.figure(figsize=(12, 12))

ax = fig.add\_subplot(111)

ax.set\_title('Occupancy Grid')

plt.xticks(visible=False)

plt.yticks(visible=False)

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=COLOR\_MAP)

ax.set\_aspect('equal')

plt.pause(2)

viz\_map[init[0]][init[1]] = 5 # Place Start Node

viz\_map[goal[0]][goal[1]] = 6

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=COLOR\_MAP)

plt.pause(2)

# Different move/search direction options:

delta = [[-1, 0], # go up

[0, -1], # go left

[1, 0], # go down

[0, 1]] # go right

delta\_name = ['^ ', '< ', 'v ', '> ']

# If you wish to use diagonals:

# delta = [[-1, 0], # go up

# [0, -1], # go left

# [1, 0], # go down

# [0, 1], # go right

# [-1, -1], # upper left

# [1, -1], # lower left

# [-1, 1], # upper right

# [1, 1]] # lower right

# delta\_name = ['^ ', '< ', 'v ', '> ', 'UL', 'LL', 'UR', 'LR']

# Heavily used from some of the A\* Examples by Sebastian Thrun:

closed = [[0 for col in range(len(self.grid[0]))] for row in range(len(self.grid))]

shortest\_path = [[' ' for \_ in range(len(self.grid[0]))] for \_ in range(len(self.grid))]

closed[init[0]][init[1]] = 1

expand = [[-1 for col in range(len(self.grid[0]))] for row in range(len(self.grid))]

delta\_tracker = [[-1 for \_ in range(len(self.grid[0]))] for \_ in range(len(self.grid))]

cost = 1

x = init[0]

y = init[1]

g = 0

f = g + self.heuristic[x][y]

open = [[f, g, x, y]]

found = False # flag that is set when search is complete

resign = False # flag set if we can't find expand

count = 0

while not found and not resign:

if len(open) == 0:

resign = True

if self.visual:

plt.text(2, 10, s="No path found...", fontsize=18, style='oblique', ha='center',

va='top')

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=COLOR\_MAP)

plt.pause(5)

return -1

else:

open.sort()

open.reverse()

next = open.pop()

x = next[2]

y = next[3]

g = next[1]

expand[x][y] = count

count += 1

if x == goal[0] and y == goal[1]:

found = True

if self.visual:

viz\_map[goal[0]][goal[1]] = 7

plt.text(2, 10, s="Goal found!", fontsize=18, style='oblique', ha='center',

va='top')

plt.imshow(viz\_map,origin='upper',interpolation='none', clim=COLOR\_MAP)

plt.pause(2)

else:

for i in range(len(delta)):

x2 = x + delta[i][0]

y2 = y + delta[i][1]

if len(self.grid) > x2 >= 0 <= y2 < len(self.grid[0]):

if closed[x2][y2] == 0 and self.grid[x2][y2] == 0:

g2 = g + cost

f = g2 + self.heuristic[x2][y2]

open.append([f, g2, x2, y2])

closed[x2][y2] = 1

delta\_tracker[x2][y2] = i

if self.visual:

viz\_map[x2][y2] = 3

plt.imshow(viz\_map,origin='upper',interpolation='none',

clim=COLOR\_MAP)

plt.pause(.5)

current\_x = goal[0]

current\_y = goal[1]

shortest\_path[current\_x][current\_y] = '\* '

full\_path = []

while current\_x != init[0] or current\_y != init[1]:

previous\_x = current\_x - delta[delta\_tracker[current\_x][current\_y]][0]

previous\_y = current\_y - delta[delta\_tracker[current\_x][current\_y]][1]

shortest\_path[previous\_x][previous\_y] =

delta\_name[delta\_tracker[current\_x][current\_y]]

full\_path.append((current\_x, current\_y))

current\_x = previous\_x

current\_y = previous\_y

full\_path.reverse()

print "Found the goal in {} iterations.".format(count)

print "full\_path: ", full\_path[:-1]

for i in range(len(shortest\_path)):

print shortest\_path[i]

if self.visual:

for node in full\_path:

viz\_map[node[0]][node[1]] = 7

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=COLOR\_MAP)

plt.pause(.5)

# Animate reaching goal:

viz\_map[goal[0]][goal[1]] = 8

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=COLOR\_MAP)

plt.pause(5)

return init, full\_path[:-1]

if \_\_name\_\_ == '\_\_main\_\_':

test\_grid = [[0, 0, 0, 0, 0, 0],

[0, 1, 1, 1, 1, 0],

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 1, 0]]

test\_start = [0, 0] # [x, y]

test\_goal = [5, 7] # [x, y]

# test\_grid = [[0, 0, 0, 0, 0, 0, 0, 0],

# [0, 0, 0, 0, 0, 0, 0, 0],

# [0, 0, 0, 0, 0, 0, 0, 0],

# [1, 1, 1, 1, 1, 1, 1, 1],

# [1, 0, 0, 1, 1, 0, 0, 1],

# [1, 0, 0, 1, 1, 0, 0, 1],

# [1, 0, 0, 1, 1, 0, 0, 1],

# [1, 0, 0, 0, 0, 0, 0, 1],

# [1, 0, 0, 0, 0, 0, 0, 1],

# [1, 0, 0, 0, 0, 0, 0, 1],

# [1, 0, 0, 0, 0, 0, 0, 1],

# [1, 0, 0, 0, 0, 0, 0, 1],

# [1, 1, 1, 1, 1, 1, 1, 1]]

# test\_start = [2, 4] # [x, y]

# test\_goal = [6, 11] # [x, y]

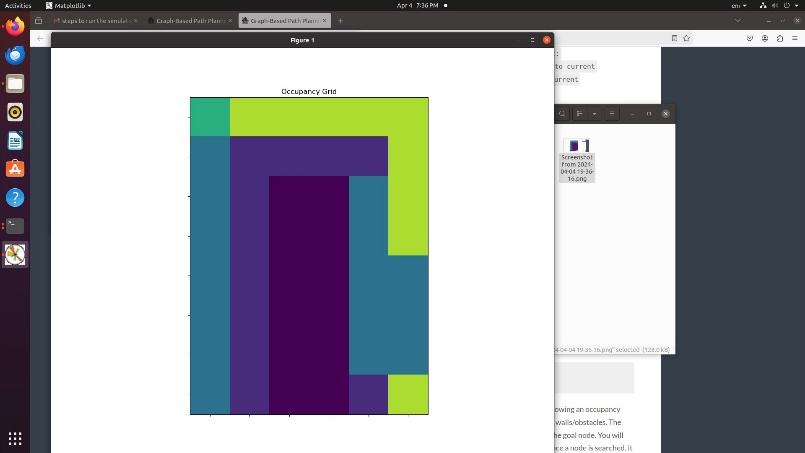
# Create an instance of the PathPlanner class:

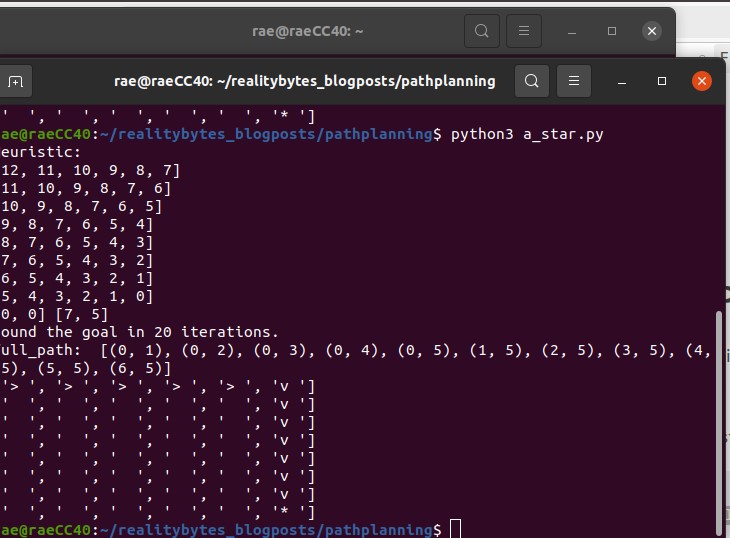
test\_planner = PathPlanner(test\_grid, True)

# Plan a path.

test\_planner.a\_star(test\_start, test\_goal)

**COMMANDS:**

* git clone https://github.com/atomoclast/realitybytes\_blogposts.git
* /realitybytes\_blogposts/pathplanning/
* cd pathplanning
* pathplanning$ chmod +x a\_star.py
* python3 a\_star.py



**Dijistra’s Algorithm**

**CODE:**

from \_\_future\_\_ import print\_function

import numpy as np

import math

import matplotlib.pyplot as plt

import pprint

def dijkstras(occupancy\_map, x\_spacing, y\_spacing, start, goal):

DEBUG = False

VISUAL = True

colormapval = (0, 8)

goal\_found = False

# Setup Map Visualizations:

if VISUAL == True:

viz\_map=occupancy\_map

fig = plt.figure(figsize=(12,12))

ax = fig.add\_subplot(111)

ax.set\_title('Occupancy Grid')

plt.xticks(visible=False)

plt.yticks(visible=False)

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=colormapval)

ax.set\_aspect('equal')

plt.pause(2)

# We will use this delta function to search surrounding nodes.

delta = [[-1, 0], # go up

[0, -1], # go left

[1, 0], # go down

[0, 1]] # go right

# Each node on the map "costs" 1 step to reach.

cost = 1

# Convert numpy array of map to list of map, makes it easier to search.

occ\_map = occupancy\_map.tolist()

if DEBUG == True:

print("occ\_map: ")

pprint.pprint(occ\_map)

# Converge start and goal positions to map indices.

x = int(math.ceil((start.item(0) / x\_spacing) - 0.5)) # startingx

y = int(math.ceil((start.item(1) / y\_spacing) - 0.5)) # startingy

goalX = int(math.ceil((goal.item(0) / x\_spacing) - 0.5))

goalY = int(math.ceil((goal.item(1) / y\_spacing) - 0.5))

print("Start Pose: ", x, y)

print("Goal Pose: ", goalX, goalY)

# Make a map to keep track of all the nodes and their cost distance values.

possible\_nodes = [[0 for row in range(len(occ\_map[0]))] for col in range(len(occ\_map))]

row = y

col = x

# Show the starting node and goal node.

# 5 looks similar to S and 6 looks similar to G.

possible\_nodes[row][col] = 5

if VISUAL == True:

viz\_map[row][col] = 5

viz\_map[goalY][goalX] = 6

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=colormapval)

plt.pause(2)

if DEBUG == True:

print("Possible Nodes: ")

pprint.pprint(possible\_nodes)

g\_value = 0

frontier\_nodes = [(g\_value, col, row)] # dist, x, y

searched\_nodes = []

parent\_node = {} # Dictionary that Maps {child node : parent node}

loopcount = 0

while len(frontier\_nodes) != 0:

if DEBUG == True:

"\n>>>>>>>>>>>>LOOP COUNT: ", loopcount, "\n"

frontier\_nodes.sort(reverse=True) #sort from shortest distance to farthest

current\_node = frontier\_nodes.pop()

if DEBUG == True:

print("current\_node: ", current\_node)

print("frontier nodes: ", searched\_nodes)

if current\_node[1] == goalX and current\_node[2] == goalY:

print("Goal found!")

goal\_found = True

if VISUAL == True:

plt.text(2, 10, s="Goal found!", fontsize=18, style='oblique', ha='center', va='top')

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=colormapval)

plt.pause(2)

break

g\_value, col, row = current\_node

# Check surrounding neighbors.

for i in delta:

possible\_expansion\_x = col + i[0]

possible\_expansion\_y = row + i[1]

valid\_expansion = 0 <= possible\_expansion\_y < len(occupancy\_map[0]) and 0 <=

possible\_expansion\_x < len(occ\_map)

if DEBUG == True:

print("Current expansion Node: ", possible\_expansion\_x, possible\_expansion\_y)

if valid\_expansion:

try:

unsearched\_node =

possible\_nodes[possible\_expansion\_y][possible\_expansion\_x] == 0

open\_node = occ\_map[possible\_expansion\_y][possible\_expansion\_x] == 0

if DEBUG == True:

print("Check Open or Wall: ",

occ\_map[possible\_expansion\_y][possible\_expansion\_x])

except:

unsearched\_node = False

open\_node = False

if unsearched\_node and open\_node:

possible\_nodes[possible\_expansion\_y][possible\_expansion\_x] = 3

possible\_node = (g\_value + cost, possible\_expansion\_x, possible\_expansion\_y)

frontier\_nodes.append(possible\_node)

if DEBUG == True:

print("frontier\_nodes:", frontier\_nodes)

if VISUAL == True:

viz\_map[possible\_expansion\_y][possible\_expansion\_x] = 3

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=colormapval)

plt.pause(.5)

# This now builds parent/child relationship

parent\_node[possible\_node] = current\_node

if DEBUG == True:

print("Parent Node: \n", parent\_node)

print("While Possible Nodes: ")

pprint.pprint(possible\_nodes)

loopcount = loopcount+1

if goal\_found == True:

print("Generating path...")

route = []

child\_node = current\_node

while child\_node in parent\_node:

route.append(parent\_node[child\_node])

child\_node = parent\_node[child\_node]

route.sort()

# route back to metric units:

if DEBUG == True:

print("Route: ", route)

if VISUAL == True:

for i in range(0, len(route)):

viz\_map[route[i][2]][route[i][1]] = 7

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=colormapval)

plt.pause(.5)

viz\_map[goalY][goalX] = 7

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=colormapval)

plt.pause(5)

path = []

position = [start.item(0), start.item(1)] # Starting point passed in by function

path.append(position) # Add it to the list for the path

for i in range(0, len(route)):

position = [round((route[i][1]+0.5)\*x\_spacing, 3), round((route[i][2]+0.5)\*y\_spacing,

3)]

path.append(position)

position = [goal.item(0), goal.item(1)]

path.append(position)

print("Path: ")

pprint.pprint(path)

# Convert to numpy array and return.

path = np.array(path)

return path

else:

if VISUAL == True:

plt.text(2, 10, s="No path found...", fontsize=18, style='oblique', ha='center', va='top')

plt.imshow(viz\_map, origin='upper', interpolation='none', clim=colormapval)

plt.pause(5)

return False

def test():

test\_map1 = np.array([

[1, 1, 1, 1, 1, 1, 1, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 1, 1, 1, 1, 1, 1, 1]])

x\_spacing1 = 0.13

y\_spacing1 = 0.2

start1 = np.array([[0.3], [0.3], [0]])

goal1 = np.array([[0.6], [1], [0]])

path1 = dijkstras(test\_map1,x\_spacing1,y\_spacing1,start1,goal1)

true\_path1 = np.array([

[0.3, 0.3], [0.325, 0.3], [0.325, 0.5],[0.325, 0.7], [0.325, 0.9], [0.325, 1.1],[0.455, 1.1],

[0.585, 1.1], [0.6, 1.0] ])

if np.array\_equal(path1,true\_path1):

print("Path 1 passes")

test\_map2 = np.array([ [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [1,

1, 1, 1, 1, 1, 1, 1], [1, 0, 0, 1, 1, 0, 0, 1], [1, 0, 0, 1, 1, 0, 0, 1], [1, 0, 0, 1, 1, 0, 0, 1], [1, 0, 0,

0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1]])

start2 = np.array([[0.5], [1.0], [1.5707963267948966]])

goal2 = np.array([[1.1], [0.9], [-1.5707963267948966]])

x\_spacing2 = 0.2

y\_spacing2 = 0.2

path2 = dijkstras(test\_map2,x\_spacing2,y\_spacing2,start2,goal2)

true\_path2 = np.array([[ 0.5, 1.0], # [2, 5]

[ 0.5, 1.1], # [2, 5]

[ 0.5, 1.3], # [2, 6]

[ 0.5, 1.5], # [2, 7]

[ 0.7, 1.5], # [3, 7]

[ 0.9, 1.5], # [4, 7]

[ 1.1, 1.5], # [5, 7]

[ 1.1, 1.3], # [5, 6]

[ 1.1, 1.1], # [5, 5]

[ 1.1, 0.9] # [5, 4]

])

if np.array\_equal(path2,true\_path2):

print("Path 2 passes")

test\_map3 = np.array([

[1, 1, 1, 1, 1, 1, 1, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 1, 1, 1, 1, 1, 1, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 0, 0, 0, 0, 0, 0, 1],

[1, 1, 1, 1, 1, 1, 1, 1]])

x\_spacing3 = 0.13

y\_spacing3 = 0.2

start3 = np.array([[0.3], [0.3], [0]])

goal3 = np.array([[0.6], [1], [0]])

path3 = dijkstras(test\_map3, x\_spacing3, y\_spacing3, start3,goal3)

if path3 == False:

print("Path 3 passes")

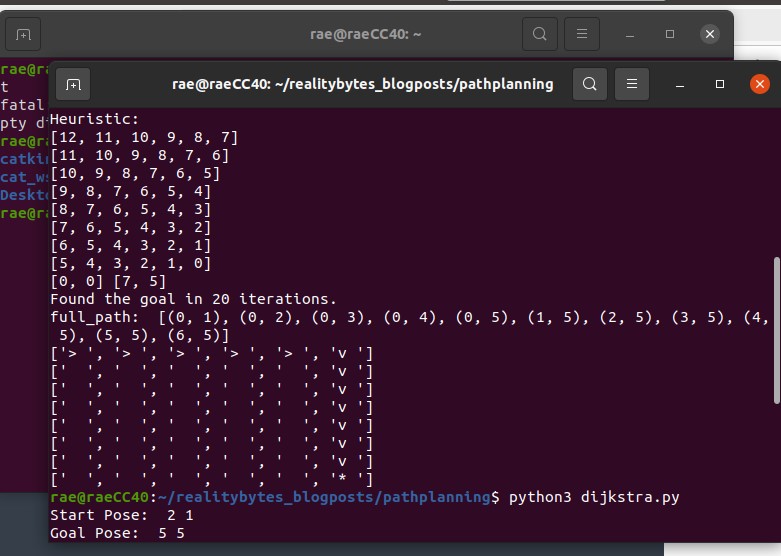
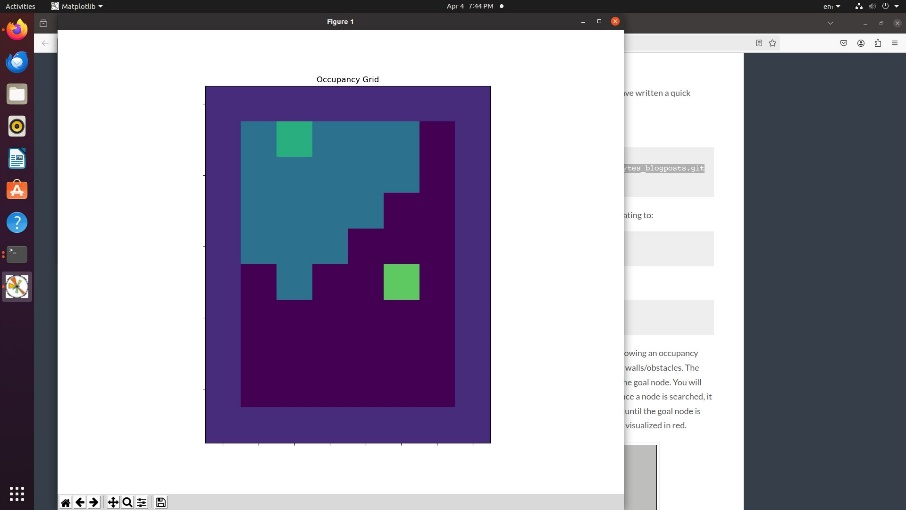
def main():

test()

if \_\_name\_\_ == '\_\_main\_\_':

main()

**COMMANDS:**

* git clone https://github.com/atomoclast/realitybytes\_blogposts.git
* /realitybytes\_blogposts/pathplanning/
* cd pathplanning
* pathplanning$ chmod +x dijkstra.py
* python3 dijkstra.py

**a)PATH PLANNING**

#!/usr/bin/env python

import rospy

from pp\_msgs.srv import PathPlanningPlugin, PathPlanningPluginResponse

from geometry\_msgs.msg import Twist

from gridviz import GridViz

from algorithms.dijkstra import dijkstra

from algorithms.astar import astar

from algorithms.greedy import greedy

from algorithms.q\_learning import q\_learning

from algorithms.lpastar import lpastar

previous\_plan\_variables = None

def make\_plan(req):

Callback function used by the service server to process

requests from clients. It returns a msg of type PathPlanningPluginResponse

global previous\_plan\_variables

# costmap as 1-D array representation

costmap = req.costmap\_ros

# number of columns in the occupancy grid

width = req.width

# number of rows in the occupancy grid

height = req.height

start\_index = req.start

goal\_index = req.goal

# side of each grid map square in meters

resolution = 0.05

# origin of grid map

origin = [-10, -10, 0]

viz = GridViz(costmap, resolution, origin, start\_index, goal\_index, width)

# time statistics

start\_time = rospy.Time.now()

# calculate the shortes path

path, previous\_plan\_variables = lpastar(start\_index, goal\_index, width, height, costmap, resolution, origin, viz, previous\_plan\_variables)

if not path:

rospy.logwarn("No path returned by the path algorithm")

path = []

else:

execution\_time = rospy.Time.now() - start\_time

print("\n")

rospy.loginfo('++++++++ Path Planning execution metrics ++++++++')

rospy.loginfo('Total execution time: %s seconds', str(execution\_time.to\_sec()))

rospy.loginfo('++++++++++++++++++++++++++++++++++++++++++++')

print("\n")

rospy.loginfo('Path sent to navigation stack')

resp = PathPlanningPluginResponse()

resp.plan = path

return resp

def clean\_shutdown():

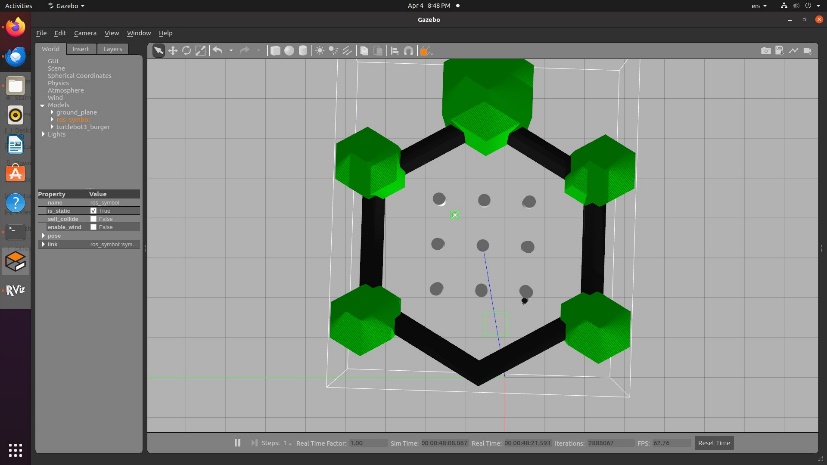
cmd\_vel.publish(Twist())

rospy.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('path\_planning\_service\_server', log\_level=rospy.INFO, anonymous=False)

make\_plan\_service = rospy.Service("/move\_base/SrvClientPlugin/make\_plan", PathPlanningPlugin, make\_plan)



**b)A\* ALGORITHM**

#! /usr/bin/env python3

import rospy

from math import sqrt

from algorithms.neighbors import find\_neighbors

def euclidean\_distance(index, goal\_index, width):

""" Heuristic Function for A Star algorithm"""

index\_x = index % width

index\_y = int(index / width)

goal\_x = goal\_index % width

goal\_y = int(goal\_index / width)

distance = (index\_x - goal\_x) \*\* 2 + (index\_y - goal\_y) \*\* 2

return sqrt(distance)

def astar(start\_index, goal\_index, width, height, costmap, resolution, origin, grid\_viz, previous\_plan\_variables):

'''

Performs A Star shortest path algorithm search on a costmap with a given start and goal node

'''

# create an open\_list

open\_list = []

# set to hold already processed nodes

closed\_list = set()

# dict for mapping children to parent

parents = dict()

# dict for mapping g costs (travel costs) to nodes

g\_costs = dict()

# dict for mapping f costs (heuristic + travel) to nodes

f\_costs = dict()

# set the start's node g\_cost and f\_cost

g\_costs[start\_index] = 0

f\_costs[start\_index] = 0

# add start node to open list

start\_cost = 0 + euclidean\_distance(start\_index, goal\_index, width)

open\_list.append([start\_index, start\_cost])

shortest\_path = []

path\_found = False

rospy.loginfo('A Star: Done with initialization')

# Main loop, executes as long as there are still nodes inside open\_list

while open\_list:

# sort open\_list according to the lowest 'g\_cost' value (second element of each sublist)

open\_list.sort(key = lambda x: x[1])

# extract the first element (the one with the lowest 'g\_cost' value)

current\_node = open\_list.pop(0)[0]

# Close current\_node to prevent from visting it again

closed\_list.add(current\_node)

# Optional: visualize closed nodes

grid\_viz.set\_color(current\_node,"pale yellow")

# If current\_node is the goal, exit the main loop

if current\_node == goal\_index:

path\_found = True

break

# Get neighbors of current\_node

neighbors = find\_neighbors(current\_node, width, height, costmap, resolution)

# Loop neighbors

for neighbor\_index, step\_cost in neighbors:

# Check if the neighbor has already been visited

if neighbor\_index in closed\_list:

continue

# calculate g\_cost of neighbour considering it is reached through current\_node

g\_cost = g\_costs[current\_node] + step\_cost

h\_cost = euclidean\_distance(neighbor\_index, goal\_index, width)

f\_cost = g\_cost + h\_cost

# Check if the neighbor is in open\_list

in\_open\_list = False

for idx, element in enumerate(open\_list):

if element[0] == neighbor\_index:

in\_open\_list = True

break

# CASE 1: neighbor already in open\_list

if in\_open\_list:

if f\_cost < f\_costs[neighbor\_index]:

# Update the node's g\_cost and f\_cost

g\_costs[neighbor\_index] = g\_cost

f\_costs[neighbor\_index] = f\_cost

parents[neighbor\_index] = current\_node

# Update the node's g\_cost inside open\_list

open\_list[idx] = [neighbor\_index, f\_cost]

# CASE 2: neighbor not in open\_list

else:

# Set the node's g\_cost and f\_cost

g\_costs[neighbor\_index] = g\_cost

f\_costs[neighbor\_index] = f\_cost

parents[neighbor\_index] = current\_node

# Add neighbor to open\_list

open\_list.append([neighbor\_index, f\_cost])

# Optional: visualize frontier

grid\_viz.set\_color(neighbor\_index,'orange')

rospy.loginfo('AStar: Done traversing nodes in open\_list')

if not path\_found:

rospy.logwarn('AStar: No path found!')

return shortest\_path

# Reconstruct path by working backwards from target

if path\_found:

node = goal\_index

shortest\_path.append(goal\_index)

while node != start\_index:

shortest\_path.append(node)

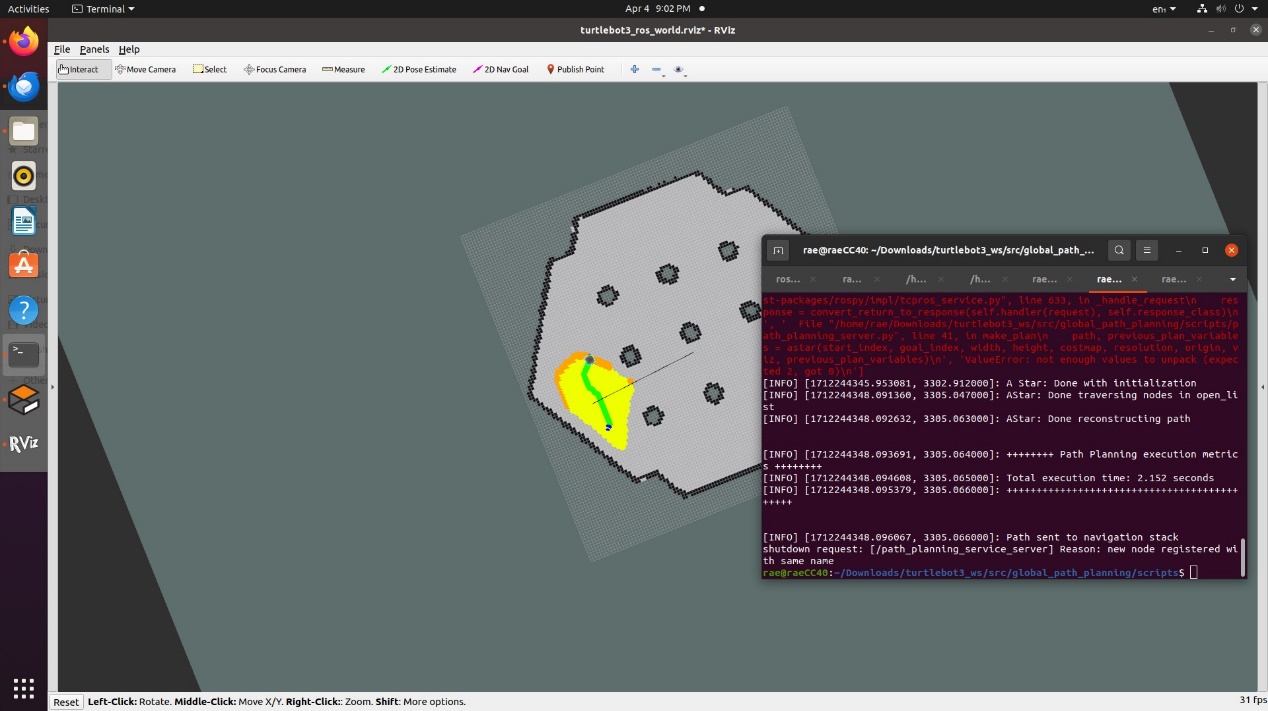
# get next node

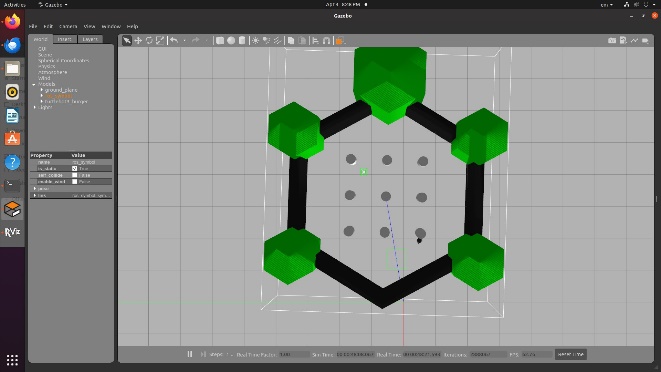
node = parents[node]

# reverse list

shortest\_path = shortest\_path[::-1]

rospy.loginfo('AStar: Done reconstructing path')





**c)D STAR**

#! /usr/bin/env python3

"""

Dijkstra's algorithm path planning exercise solution

Author: Roberto Zegers R.

Copyright: Copyright (c) 2020, Roberto Zegers R.

License: BSD-3-Clause

Date: Nov 30, 2020

Usage: roslaunch unit2\_pp unit2\_solution.launch

"""

import rospy

**from algorithms.neighbors import find\_neighbors**

**def dijkstra(start\_index, goal\_index, width, height, costmap, resolution, origin, grid\_viz, previous\_plan\_variables):**

'''

**Performs Dijkstra's shortes path algorithm search on a costmap with a given start and goal node**

'''

# create an open\_list

open\_list = []

# set to hold already processed nodes

closed\_list = set()

**# dict for mapping children to parent**

parents = dict()

# dict for mapping g costs (travel costs) to nodes

g\_costs = dict()

# set the start's node g\_cost

g\_costs[start\_index] = 0

# add start node to open list

open\_list.append([start\_index, 0])

shortest\_path = []

path\_found = False

rospy.loginfo('Dijkstra: Done with initialization')

# Main loop, executes as long as there are still nodes inside open\_list

while open\_list:

# sort open\_list according to the lowest 'g\_cost' value (second element of each sublist)

open\_list.sort(key = lambda x: x[1])

# extract the first element (the one with the lowest 'g\_cost' value)

current\_node = open\_list.pop(0)[0]

# Close current\_node to prevent from visting it again

**closed\_list.add(current\_node)**

# Optional: visualize closed nodes

grid\_viz.set\_color(current\_node,"pale yellow")

# If current\_node is the goal, exit the main loop

if current\_node == goal\_index:

path\_found = True

break

**# Get neighbors of current\_node**

neighbors = find\_neighbors(current\_node, width, height, costmap, resolution)

# Loop neighbors

for neighbor\_index, step\_cost in neighbors:

# Check if the neighbor has already been visited

**if neighbor\_index in closed\_list:**

continue

# calculate g\_cost of neighbour considering it is reached through current\_node

g\_cost = g\_costs[current\_node] + step\_cost

# Check if the neighbor is in open\_list

in\_open\_list = False

for idx, element in enumerate(open\_list):

if element[0] == neighbor\_index:

in\_open\_list = True

break

# CASE 1: neighbor already in open\_list

if in\_open\_list:

if g\_cost < g\_costs[neighbor\_index]:

# Update the node's g\_cost inside g\_costs

g\_costs[neighbor\_index] = g\_cost

parents[neighbor\_index] = current\_node

# Update the node's g\_cost inside open\_list

open\_list[idx] = [neighbor\_index, g\_cost]

# CASE 2: neighbor not in open\_list

else:

# Set the node's g\_cost inside g\_costs

g\_costs[neighbor\_index] = g\_cost

parents[neighbor\_index] = current\_node

# Add neighbor to open\_list

open\_list.append([neighbor\_index, g\_cost])

# Optional: visualize frontier

grid\_viz.set\_color(neighbor\_index,'orange')

rospy.loginfo('Dijkstra: Done traversing nodes in open\_list')

if not path\_found:

rospy.logwarn('Dijkstra: No path found!')

return shortest\_path

# Reconstruct path by working backwards from target

if path\_found:

node = goal\_index

shortest\_path.append(goal\_index)

while node != start\_index:

shortest\_path.append(node)

# get next node

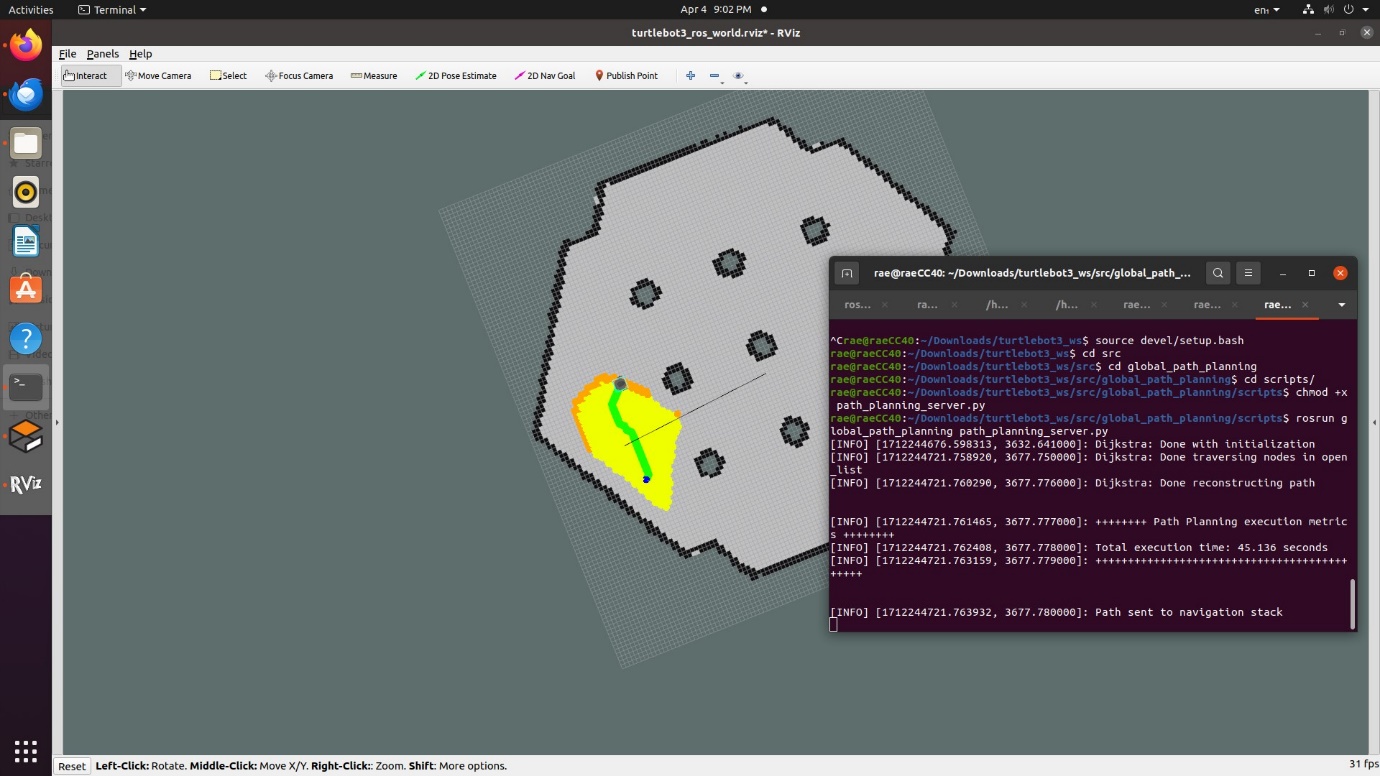
node = parents[node]

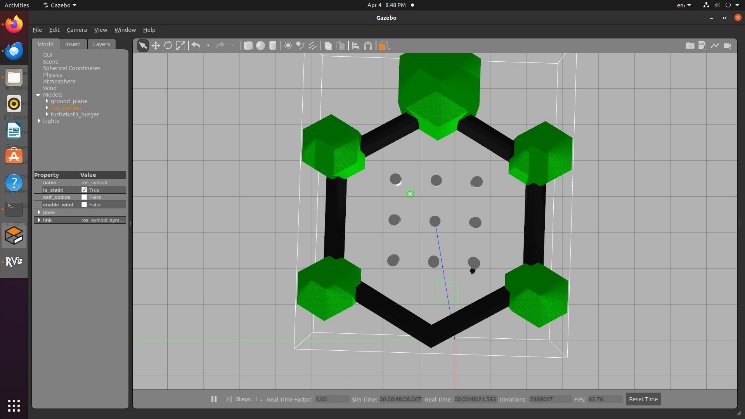
# reverse list

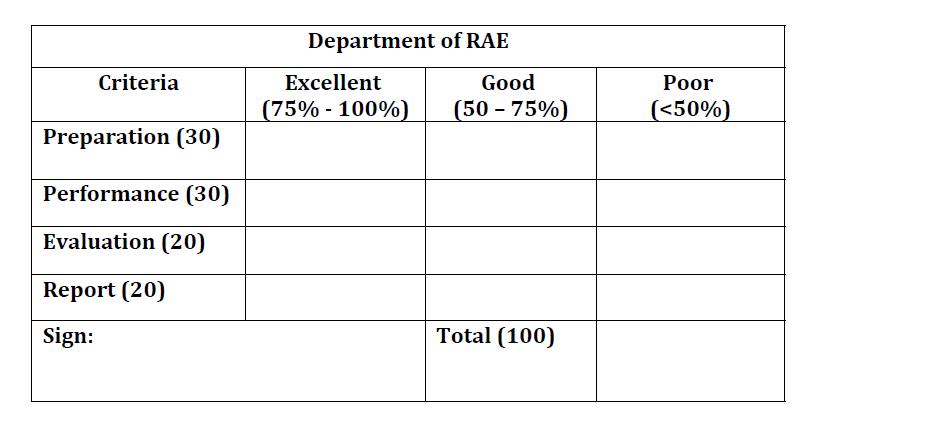
shortest\_path = shortest\_path[::-1]

rospy.loginfo('Dijkstra: Done reconstructing path')

return shortest\_path, None







**Result:**

The Path Planning algorithms were learnt using ROS.