# python code for path planning

import rospy

import actionlib

from geometry\_msgs.msg import PoseStamped

from move\_base\_msgs.msg import MoveBaseAction, MoveBaseGoal

from nav\_msgs.msg import OccupancyGrid

from nav\_msgs.srv import GetMap

import numpy as np

from PIL import Image

from queue import PriorityQueue

import cv2

class Node:

def \_\_init\_\_(self, position, parent=None):

self.position = position

self.parent = parent

self.g = 0 # Cost from start to current node

self.h = 0 # Heuristic cost (estimated cost from current node to goal)

self.f = 0 # Total cost (g + h)

def \_\_lt\_\_(self, other):

return self.f < other.f

class PathPlanner:

def \_\_init\_\_(self):

rospy.init\_node('path\_planner')

self.move\_base\_client = actionlib.SimpleActionClient('move\_base', MoveBaseAction)

self.move\_base\_client.wait\_for\_server()

self.map\_resolution = 0.05 # Resolution of the occupancy grid map

self.map\_origin\_x = 0.0 # X-coordinate of the map origin

self.map\_origin\_y = 0.0 # Y-coordinate of the map origin

self.map = None

def get\_map(self):

rospy.wait\_for\_service('static\_map')

try:

get\_map = rospy.ServiceProxy('static\_map', GetMap)

response = get\_map()

self.map = response.map

except rospy.ServiceException as e:

rospy.logerr("Service call failed: %s", e)

def occupancy\_grid\_to\_image(self):

width = self.map.info.width

height = self.map.info.height

data = np.array(self.map.data).reshape((height, width))

img = Image.fromarray(data.astype(np.uint8), mode='L')

img = img.transpose(Image.FLIP\_TOP\_BOTTOM)

return img

def heuristic\_cost(self, position, goal):

return np.linalg.norm(np.array(position) - np.array(goal))

def get\_neighbors(self, position):

neighbors = []

rows, cols = self.map.shape

r, c = position

if r > 0 and self.map[r - 1, c] == 0: # Up

neighbors.append((r - 1, c))

if r < rows - 1 and self.map[r + 1, c] == 0: # Down

neighbors.append((r + 1, c))

if c > 0 and self.map[r, c - 1] == 0: # Left

neighbors.append((r, c - 1))

if c < cols - 1 and self.map[r, c + 1] == 0: # Right

neighbors.append((r, c + 1))

return neighbors

def reconstruct\_path(self, current\_node):

path = []

while current\_node is not None:

path.append(current\_node.position)

current\_node = current\_node.parent

return path[::-1]

def astar(self, start, goal):

rows, cols = self.map.shape

open\_list = PriorityQueue()

start\_node = Node(start)

goal\_node = Node(goal)

open\_list.put((start\_node.f, start\_node))

visited = set()

visited.add(start)

while not open\_list.empty():

current\_node = open\_list.get()[1]

current\_position = current\_node.position

if current\_position == goal:

return self.reconstruct\_path(current\_node)

neighbors = self.get\_neighbors(current\_position)

for neighbor\_position in neighbors:

if neighbor\_position not in visited:

neighbor\_node = Node(neighbor\_position, current\_node)

neighbor\_node.g = current\_node.g + 1

neighbor\_node.h = self.heuristic\_cost(neighbor\_position, goal)

neighbor\_node.f = neighbor\_node.g + neighbor\_node.h

open\_list.put((neighbor\_node.f, neighbor\_node))

visited.add(neighbor\_position)

return None

def navigate\_to\_position(self, position):

goal = MoveBaseGoal()

goal.target\_pose.header.frame\_id = 'map'

goal.target\_pose.pose.position.x = position[0]

goal.target\_pose.pose.position.y = position[1]

goal.target\_pose.pose.orientation.w = 1.0

self.move\_base\_client.send\_goal(goal)

self.move\_base\_client.wait\_for\_result()

def find\_object\_position(self):

self.get\_map()

map\_image = self.occupancy\_grid\_to\_image()

# Convert the map image to OpenCV format

map\_image\_cv = np.array(map\_image)

# Apply object detection algorithm to find the object in the map image

# Replace the code below with your specific object detection algorithm

object\_cascade = cv2.CascadeClassifier('object\_cascade.xml') # Path to object detection cascade file

gray\_image = cv2.cvtColor(map\_image\_cv, cv2.COLOR\_BGR2GRAY)

objects = object\_cascade.detectMultiScale(gray\_image, scaleFactor=1.1, minNeighbors=5)

if len(objects) > 0:

# Object detected, return the position of the first object found

x, y, w, h = objects[0]

object\_position = (x + w/2, y + h/2)

else:

# Object not found, return a default position

object\_position = (0, 0)

return object\_position

def run(self):

rospy.loginfo("Finding object position...")

object\_position = self.find\_object\_position()

rospy.loginfo("Object position: {}".format(object\_position))

rospy.loginfo("Moving to object position...")

self.navigate\_to\_position(object\_position)

rospy.loginfo("Reached object position.")

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

planner = PathPlanner()

planner.run()