Part01:

```
import rospy
from geometry_msgs.msg import Twist
import heapq as hq
import copy
import numpy as np
import math
from matplotlib import pyplot as plt
import cv2 as cv
from numpy import tan, deg2rad, rad2deg



SCALE_FACTOR = 2

BLUE = (255, 0, 0)
DARK_GREEN = (15, 168, 33)
GREEN = (0, 255, 0)
RED = (0, 0, 255)
YELLOW = (9, 227, 212)
BLACK = (0, 0, 0)
GRAY = (199, 198, 195)

# for coordinates
X = 0
Y = 1
THETA = 2

# map dimensions
X_MAX = 600
Y_MAX = 200
X_MAX_SCALED = X_MAX * SCALE_FACTOR
Y_MAX_SCALED = Y_MAX * SCALE_FACTOR

# used accessing node information
PARENT_COORDINATES = 4
COORDINATES = 5

# the turtlebot is 10.5 cm.
```

```
# in an unscaled simulation, 1 pixel represents a 1cm x 1cm square
# mutliply ROBOT_SIZE by SCALE_FACTOR to determine pixel representation of
robot
ROBOT_SIZE = 10
ROBOT_SIZE_SCALED = ROBOT_SIZE * SCALE_FACTOR
ROBOT_SIZE_SCALED = math.ceil(ROBOT_SIZE)


Open_List = []    # used to track all the open nodes
Closed_List = []  # {"C2G", "C2C", "TC", "node_index", "parent_coor",
"node_coor"} # used to track the closed node
Closed_Coor = set()   # closed coordinates. Used to quickly check if a
coordinate is closed
threshold_coor = set()
obstacle_points = set()  # used to quickly look up if a point is in an
obstacle
map_points = set()       # used to quickly look up if a point is in the
map

node_index = 0


"""
draw_map()
creates a (200 * SCALE_FACTOR) * (600 * SCALE_FACTOR) numpy array that
represents the world map
the map represents a 2 meter by 6 meter world
using a SCALE_FACTOR of 2, each pixel represents a 50mm x 50mm square in
the world
"""
def draw_map():
   # Background
   background_color = BLACK
   map = np.zeros((Y_MAX_SCALED, X_MAX_SCALED, 3), np.uint8)
   map[:] = background_color

   # Map boarder
   map[0:ROBOT_SIZE_SCALED                              , :
] = YELLOW    # north edge
```

```
   map[(Y_MAX_SCALED - ROBOT_SIZE_SCALED) : Y_MAX_SCALED  , :
] = YELLOW    # south edge
   map[:                                          ,
0:ROBOT_SIZE_SCALED                                ] = YELLOW    # east
edge
   map[:                                          , (X_MAX_SCALED -
ROBOT_SIZE_SCALED) : X_MAX_SCALED ] = YELLOW    # west edge

   # box 1 boundary
   pts = np.array([[150 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 0 *
SCALE_FACTOR],
                   [150 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 125 *
SCALE_FACTOR + ROBOT_SIZE_SCALED],
                   [165 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 125 *
SCALE_FACTOR + ROBOT_SIZE_SCALED],
                   [165 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 0 *
SCALE_FACTOR]],
                  np.int32)
   cv.fillPoly(map, [pts], YELLOW)

   # box 1
   pts = np.array([[150 * SCALE_FACTOR, 0 * SCALE_FACTOR],
                   [150 * SCALE_FACTOR, 125 * SCALE_FACTOR],
                   [165 * SCALE_FACTOR, 125 * SCALE_FACTOR],
                   [165 * SCALE_FACTOR, 0 * SCALE_FACTOR]],
                  np.int32)
   cv.fillPoly(map, [pts], BLUE)

   # box 2 boundary
   pts = np.array([[250 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 200 *
SCALE_FACTOR],
                   [250 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 75 *
SCALE_FACTOR - ROBOT_SIZE_SCALED],
                   [265 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 75 *
SCALE_FACTOR - ROBOT_SIZE_SCALED],
                   [265 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 200 *
SCALE_FACTOR]],
                  np.int32)
   cv.fillPoly(map, [pts], YELLOW)
```

```
    # box 2
    pts = np.array([[250 * SCALE_FACTOR, 200 * SCALE_FACTOR],
                    [250 * SCALE_FACTOR, 75 * SCALE_FACTOR],
                    [265 * SCALE_FACTOR, 75 * SCALE_FACTOR],
                    [265 * SCALE_FACTOR, 200 * SCALE_FACTOR]],
                   np.int32)
    cv.fillPoly(map, [pts], BLUE)

    # circle boundry
    cv.circle(map, (400 * SCALE_FACTOR, 90 * SCALE_FACTOR), 50 *
SCALE_FACTOR + ROBOT_SIZE_SCALED, YELLOW, -1)

    # circle
    cv.circle(map, (400 * SCALE_FACTOR, 90 * SCALE_FACTOR), 50 *
SCALE_FACTOR, BLUE, -1)

    return map


"""
get_valid_point_map()
input: np array representing the world map.  array values are the colors
on the map
output: np array of 1s and 0s representing if a point is valid to be
traveled in.
Function works by checking the color of pixel and determinining if that
color is valid or invalid
"""
def get_valid_point_map(color_map):
    valid_point_map = np.ones((Y_MAX_SCALED, X_MAX_SCALED), np.uint8)
    for x in range(0, X_MAX_SCALED):
        for y in range(0, Y_MAX_SCALED):
            pixel_color = tuple(color_map[y, x])
            if pixel_color == YELLOW or pixel_color == BLUE:
                valid_point_map[y, x] = 0
    return valid_point_map


def determine_valid_point(valid_point_map, coordinates):
    if not __point_is_inside_map(coordinates[X], coordinates[Y]):
```

```
        return False
    if valid_point_map[coordinates[Y], coordinates[X]] == 1:
        return True
    else:
        return False


def __point_is_inside_map(x, y):
    if (x > X_MAX) or (x < 0):
        return False
    elif (y > Y_MAX) or (y < 0):
        return False
    else:
        return True

def __add_point(x, y, map, color):
    map[y, x] = color
    return map


def __draw_line(p1, p2, map, color):
    pts = np.array([[p1[0], p1[1]], [p2[0], p2[1]]],
                   np.int32)
    cv.fillPoly(map, [pts], color)


def draw_node(child_coordinates, parent_coordinates, map, color):

    child_coordinates = tuple(int(SCALE_FACTOR * _) for _ in
child_coordinates)
    cv.circle(map, child_coordinates, radius=3, color=color, thickness=-1)

    if (parent_coordinates is not None):
        parent_coordinates = tuple(int(SCALE_FACTOR * _ ) for _ in
parent_coordinates)
        cv.circle(map, parent_coordinates, radius=3, color=color,
thickness=-1)
        __draw_line(child_coordinates, parent_coordinates, map, color)
```

```
#function that defines the goal position as a circle with a threshhold.
# takes in the size of the robot as the threshhold for the target
def point_in_goal(x, y):
   distance = math.sqrt((x-goal_position[0])**2 + (y-goal_position[1])**2)
   if distance <= ROBOT_SIZE_SCALED:
       return True
   else:
       return False


#function to calculate the cost to go to from the point to the goal in a
straight line
def C2G_func (n_position, g_position):
   C2G = round(((g_position[0]-n_position[0])**2 +
(g_position[1]-n_position[1])**2)**0.5, 1)
   return C2G

def explore(n,UL,UR):
   t = 0
   D=0
   Xn=n[5][X]
   Yn=n[5][Y]
   Thetan = deg2rad(n[5][THETA])

   # Xi, Yi,Thetai: Input point's coordinates
   # Xn, Yn, Thetan: End point coordintes
   while t<1:
       Thetan += (r / L) * (UR - UL) * dt * 2 * math.pi/60
       Xn += round((0.5*r * (UL + UR) * math.cos(Thetan) * dt)*2)/20
       Yn += round((0.5*r * (UL + UR) * math.sin(Thetan) * dt)*2)/20
       t = t + dt
       temp_point = (Xn, Yn)
       if temp_point in obstacle_points:
           return None

   # print("temp_point:", temp_point)
   Theta_n = round(rad2deg(Thetan), 2)
   theta_dot = (r / L) * (UR - UL) *2*math.pi/60
   x_dot = 0.5*(r/100) * (UL + UR) * math.cos(Thetan)*2*math.pi/60
   y_dot = 0.5*(r/100) * (UL + UR) * math.sin(Thetan)*2*math.pi/60
```

```
    new_position = round(Xn), round(Yn), round(Theta_n)
    D=round(D+ math.sqrt(math.pow((0.5*r * (UL + UR) * math.cos(Thetan) *
dt),2)+math.pow((0.5*r * (UL + UR) * math.sin(Thetan) * dt),2)), 2)
    new_C2C = round((n[1]+D),2)
    new_C2G = round((C2G_func(new_position, goal_position)),2)
    new_TC = round((new_C2C + new_C2G),2)

    point_vel = (float(x_dot), float(y_dot), float(theta_dot))
    new_node = (new_TC, new_C2C, new_C2G, node_index, n[5], new_position,
point_vel)

    return new_node

# main exploration function. starts by popping a node out of the open list
and checking the status of the
# popped node prior to expanding the search using the exploration
functions defined above.
# node: (C2G, C2C, TC, point_index, (x,y,theta)parent_coordinates,
(x,y,theta)coordinates)
def exploreNodes():
    global goal_found
    hq.heapify(Open_List)
    # print("OpenListLen:", len(Open_List))
    while Open_List:
        if goal_found:
            break
        popped_node = hq.heappop(Open_List)
        Closed_Coor.add((popped_node[5][0], popped_node[5][1]))

        #popped node is checked and added to the closed list as a dic
        check_popped_status(popped_node)
        popped_node_dic = {"TC": popped_node[0], "C2C": popped_node[1],
"C2G": popped_node[2], "node_index": popped_node[3], "parent_coor":
popped_node[4], "node_coor": popped_node[5], "vel": popped_node[6]}
        Closed_List.append(popped_node_dic)

        # checks if the newly created node falls within the defined map
points, outside the obstacles,
        # has not been closed already and its not within the threshhold of
other points
```

```
        # when all pass, it adds it to the open list

        #[RPM_1, RPM_1]
        new_node = explore(copy.deepcopy(popped_node), actions[0][0],
actions[0][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_1, 0]
        new_node = explore(copy.deepcopy(popped_node), actions[1][0],
actions[1][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[0, RPM_1]
        new_node = explore(copy.deepcopy(popped_node), actions[2][0],
actions[2][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_2, RPM_2]
        new_node = explore(copy.deepcopy(popped_node), actions[3][0],
actions[3][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
```

```
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_2, 0]
        new_node = explore(copy.deepcopy(popped_node), actions[4][0],
actions[4][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[0, RPM_2]
        new_node = explore(copy.deepcopy(popped_node), actions[5][0],
actions[5][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_1, RPM_2]
        new_node = explore(copy.deepcopy(popped_node), actions[6][0],
actions[6][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_2, RPM_1]
```

```
        new_node = explore(copy.deepcopy(popped_node), actions[7][0],
actions[7][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        # print("OpenList:", Open_List)
        return Open_List, Closed_Coor, Closed_List

#threshhold function that checks if the newly created point falls within
the already explored points,
# angle must be the same. point can have multiple different angle
def threshhold(nx, ny):
    if (nx, ny) in threshold_coor:
        return True
    else:
        threshold_coor.add((nx+0.5, ny))
        threshold_coor.add((nx+0.5, ny+0.5))
        threshold_coor.add((nx, ny+0.5))
        threshold_coor.add((nx-0.5, ny+0.5))
        threshold_coor.add((nx-0.5, ny))
        threshold_coor.add((nx-0.5, ny-0.5))
        threshold_coor.add((nx, ny-0.5))
        threshold_coor.add((nx, ny+0.5))
        return threshold_coor, False

#check if newly explored point has been explored previously, if so compare
C2C and update if the new C2C is
# lower than the one originally stored
def checkTC (on, n):
    global node_index
    for i, nodes in enumerate(Open_List):
        if (nodes[5][0],nodes[5][1]) == (n[5][0],n[5][1]):
            if n[0] < nodes[0]:
                new_node = (n[0], n[1], n[2], nodes[1], n[4], n[5], n[6])
                Open_List[i] = new_node
```

```
                hq.heapify(Open_List)
            return Open_List
    else:
        node_index += 1
        new = (n[0], n[1], n[2], node_index, on[5], n[5], n[6])
        hq.heappush(Open_List, new)
    return Open_List

#function to check the status of the popped node, if it matches the goal
coordinates it starts the backtracking function
def check_popped_status (n):
    global goal_found
    if point_in_goal(n[5][0], n[5][1]):
        goal_node = {"TC": n[0], "C2C": n[1], "C2G": n[2], "node_index":
n[3], "parent_coor": n[4], "node_coor": n[5], "vel": n[6]}
        Closed_List.append(goal_node)
        print("goal node position:", n[5])
        print("target position:", goal_position)
        print("Goal found")
        print("destination info:", n)
        goal_found = True
        start_backtrack ()
    else:
        return(n)


#backtracking function that takes in the last closed node from the closed
list and runs a loop using
#  the node coordinates and the parent coordinate to trace back the steps
leading to the start position
def start_backtrack ():
    print("Backtracking...")
    path_nodes = []
    path_coor = []
    path_vel = []
    current_node = Closed_List[-1]
    path_nodes.append(current_node)
    path_coor.append((current_node['node_coor'][X],
current_node['node_coor'][Y], current_node['node_coor'][THETA]))
```

```
    path_vel.append((current_node['vel'][X], current_node['vel'][Y],
current_node['vel'][THETA]))
    print("First node used:", current_node)

# (TC, C2C, C2G, point_index, (x,y,theta)parent_coordinates,
(x,y,theta)coordinates, vel(x,y,theta))
    while current_node["parent_coor"] is not None:
        search_value = current_node["parent_coor"]
        for node in Closed_List:
            if node["node_coor"] == search_value:
                # If a matching value is found, assign the entire
dictionary as the new current_node
                current_node = node
                break
        path_nodes.append(current_node)
        path_coor.append((current_node['node_coor'][X],
current_node['node_coor'][Y], current_node['node_coor'][THETA]))
        path_vel.append((current_node['vel'][X], current_node['vel'][Y],
current_node['vel'][THETA]))

    path_nodes.reverse()
    path_coor.reverse()
    path_vel.reverse()
    # move(path_vel)
    run_visualization(path_coor)



def move(path_vel):

    vel_msg = Twist()
    if ROS == True:
        rospy.init_node('turtlebot3_control')
        vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=10)
        print('starting movement!')

    rate = rospy.Rate(.7825)

    counter = 0
    for i in path_vel:
```

```
        vel_msg.linear.x = math.sqrt(i[0]**2 + i[1]**2)
        vel_msg.angular.z = (-1*i[2])

        print("[",counter,"]")
        counter+=1
        print(vel_msg)
        print()
        if ROS == True:
            vel_pub.publish(vel_msg)
            rate.sleep()

    vel_msg.linear.x = 0
    vel_msg.angular.z = 0
    if ROS == True:
        vel_pub.publish(vel_msg)

# runs visulization of path plan using OpenCV.
# path shown is the solution path, with the connected nodes from the open
list
def run_visualization(path_coordinates):
    for node in path_coordinates:
        if not point_in_goal(node[X], node[Y]):
            draw_node((node[X], node[Y]), None, color_map, GREEN)
            for action in actions:
                plot_curve(node, action[0], action[1], color_map, GRAY)
            cv.imshow('A* Algorithm', color_map)
            cv.waitKey(200)
        else:
            #color goal node red
            node = path_coordinates[-1]
            draw_node((node[X], node[Y]), None, color_map, RED)

    cv.imshow('A* Algorithm', color_map)
    cv.waitKey(0)
    cv.destroyAllWindows()
    return


def plot_curve(node, UL,UR, map, color):
    t = 0
```

```
    D=0
    Xn=node[X]
    Yn=node[Y]
    Thetan = deg2rad(node[THETA])

    # Xi, Yi,Thetai: Input point's coordinates
    # Xs, Ys: Start point coordinates for plot function
    # Xn, Yn, Thetan: End point coordintes
    while t<1:
        Xs = Xn
        Ys = Yn
        Thetan += (r / L) * (UR - UL) * dt * 2 * math.pi/60
        Xn += round((0.5*r * (UL + UR) * math.cos(Thetan) * dt)*2)/20
        Yn += round((0.5*r * (UL + UR) * math.sin(Thetan) * dt)*2)/20
        t = t + dt
        __draw_line((Xs*SCALE_FACTOR, Ys*SCALE_FACTOR), (Xn*SCALE_FACTOR,
Yn*SCALE_FACTOR), map, color)
    Xn = Xn
    Yn = Yn


if __name__ == '__main__':

    DEBUG = False
    ROS = False

    # user input required to fill out the following
    if DEBUG == False:
        start_x_position = int(input("enter start X position(20-580): "))
        start_y_position = int(input("enter start Y position(20-180): "))
        start_theta_position = int(input("enter start theta
position(0-360): "))
        print()
        goal_x_position = int(input("enter goal X position(20-580): "))
        goal_y_position = int(input("enter goal y position(20-180): "))
        RPM_1 = 10          # (rot/s)
        RPM_2 = 20           # (rot/s)
        r = 3.3             # (cm)
        L = 12              # (cm)
        dt = 0.1
```

```
        clearance = 10
        ROBOT_SIZE_SCALED = ROBOT_SIZE + clearance * SCALE_FACTOR
    else:
        start_x_position  = 50
        start_y_position  = 100
        start_theta_position  = 0
        goal_x_position  = 530
        goal_y_position  = 180
        RPM_1 = 10          # (rot/s)
        RPM_2 = 20           # (rot/s)
        r = 3.3             # (cm)
        L = 12              # (cm)
        dt = 0.1
        # step_size = 1
        # ROBOT_SIZE = 5  # (mm) 105 is the size if the robot in mm.  Will
not work for this simulation
        clearance = 10
        ROBOT_SIZE_SCALED = ROBOT_SIZE_SCALED + clearance * SCALE_FACTOR

    # 0 = left   1 = right
    actions = [[RPM_1, RPM_1], [RPM_1, 0], [0, RPM_1], [RPM_2, RPM_2],
[RPM_2, 0], [0, RPM_2], [RPM_1, RPM_2], [RPM_2, RPM_1]]

    start_position = (start_x_position, start_y_position,
start_theta_position)
    goal_position = (goal_x_position, goal_y_position)

    # initial values for the start node
    vel1 = (0, 0, 0)
    C2G1 = C2G_func(start_position, goal_position)
    C2C1 = 0
    TC1 = C2G1 + C2C1
    # (C2G, C2C, TC, point_index, (x,y,theta)parent_coordinates,
(x,y,theta)coordinates, velocity)
    start_node = (TC1, C2C1, C2G1, node_index, None, start_position, vel1)
    hq.heappush(Open_List, start_node)
    print("initial_Open_list:", Open_List)

    print("Drawing map...")
    color_map = draw_map()
```

```
    print("Determining open points and obstacle points...")
    valid_point_map = get_valid_point_map(color_map)
    # creat 2 sets, 1 containing all the possible points within map, 1
containg all possible points within the obstacle spaces
    # this will be used later to check the created points to see if they
can be used
    x_range = np.arange(0, 600, 0.5)
    y_range = np.arange(0, 200, 0.5)
    for x in np.arange(0, 600, 0.5):
        for y in np.arange(0, 200, 0.5):
            if valid_point_map[int(y * SCALE_FACTOR), int(x *
SCALE_FACTOR)] == 1:
                map_points.add((x, y))
            else:
                obstacle_points.add((x, y))

    if (start_position[0], start_position[1]) in obstacle_points:
        print("start point selected is in obstacle space, try again")
        exit()
    if (start_position[0], start_position[1]) not in map_points:
        print("start point selected is  outside the map, try again")
        exit()
    if (goal_position[0], goal_position[1]) in obstacle_points:
        print(goal_position)
        print("goal point selected is in obstacle space, try again")
        exit()
    if (goal_position[0], goal_position[1]) not in map_points:
        print("goal point selected is  outside the map, try again")
        exit()

    goal_found = False
    print("start node:", start_node)
    print("starting exploration")
    while not goal_found:
        exploreNodes()
```

Part02:

```
#!/usr/bin/env python

# Copyright (c) 2011, Willow Garage, Inc.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are
met:
#
#     * Redistributions of source code must retain the above copyright
#       notice, this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above copyright
#       notice, this list of conditions and the following disclaimer in the
#       documentation and/or other materials provided with the
distribution.
#     * Neither the name of the Willow Garage, Inc. nor the names of its
#       contributors may be used to endorse or promote products derived
from
#        this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
# ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE
# POSSIBILITY OF SUCH DAMAGE.

import rospy
from geometry_msgs.msg import Twist
import heapq as hq
```

```
import copy
import numpy as np
import math
from matplotlib import pyplot as plt
import cv2 as cv
from numpy import tan, deg2rad, rad2deg



SCALE_FACTOR = 2

BLUE = (255, 0, 0)
DARK_GREEN = (15, 168, 33)
GREEN = (0, 255, 0)
RED = (0, 0, 255)
YELLOW = (9, 227, 212)
BLACK = (0, 0, 0)
GRAY = (199, 198, 195)

# for coordinates
X = 0
Y = 1
THETA = 2

# map dimensions
X_MAX = 600
Y_MAX = 200
X_MAX_SCALED = X_MAX * SCALE_FACTOR
Y_MAX_SCALED = Y_MAX * SCALE_FACTOR

# used accessing node information
PARENT_COORDINATES = 4
COORDINATES = 5

# the turtlebot is 10.5 cm.
# in an unscaled simulation, 1 pixel represents a 1cm x 1cm square
# mutliply ROBOT_SIZE by SCALE_FACTOR to determine pixel representation of
robot
ROBOT_SIZE = 10.5
ROBOT_SIZE_SCALED = ROBOT_SIZE * SCALE_FACTOR
```

```
ROBOT_SIZE_SCALED = math.ceil(ROBOT_SIZE)


Open_List = []    # used to track all the open nodes
Closed_List = []  # {"C2G", "C2C", "TC", "node_index", "parent_coor",
"node_coor"} # used to track the closed node
Closed_Coor = set()   # closed coordinates. Used to quickly check if a
coordinate is closed
threshold_coor = set()
obstacle_points = set()  # used to quickly look up if a point is in an
obstacle
map_points = set()       # used to quickly look up if a point is in the
map

node_index = 0


"""
draw_map()
creates a (200 * SCALE_FACTOR) * (600 * SCALE_FACTOR) numpy array that
represents the world map
the map represents a 2 meter by 6 meter world
using a SCALE_FACTOR of 2, each pixel represents a 50mm x 50mm square in
the world
"""
def draw_map():
   # Background
   background_color = BLACK
   map = np.zeros((Y_MAX_SCALED, X_MAX_SCALED, 3), np.uint8)
   map[:] = background_color

   # Map boarder
   map[0:ROBOT_SIZE_SCALED                                  , :
] = YELLOW    # north edge
   map[(Y_MAX_SCALED - ROBOT_SIZE_SCALED) : Y_MAX_SCALED   , :
] = YELLOW    # south edge
   map[:                                                    ,
0:ROBOT_SIZE_SCALED                                ] = YELLOW    # east
edge
```

```
   map[:                                                , (X_MAX_SCALED -
ROBOT_SIZE_SCALED) : X_MAX_SCALED ] = YELLOW    # west edge

   # box 1 boundary
   pts = np.array([[150 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 0 *
SCALE_FACTOR],
                   [150 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 125 *
SCALE_FACTOR + ROBOT_SIZE_SCALED],
                   [165 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 125 *
SCALE_FACTOR + ROBOT_SIZE_SCALED],
                   [165 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 0 *
SCALE_FACTOR]],
                  np.int32)
   cv.fillPoly(map, [pts], YELLOW)

   # box 1
   pts = np.array([[150 * SCALE_FACTOR, 0 * SCALE_FACTOR],
                   [150 * SCALE_FACTOR, 125 * SCALE_FACTOR],
                   [165 * SCALE_FACTOR, 125 * SCALE_FACTOR],
                   [165 * SCALE_FACTOR, 0 * SCALE_FACTOR]],
                  np.int32)
   cv.fillPoly(map, [pts], BLUE)

   # box 2 boundary
   pts = np.array([[250 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 200 *
SCALE_FACTOR],
                   [250 * SCALE_FACTOR - ROBOT_SIZE_SCALED, 75 *
SCALE_FACTOR - ROBOT_SIZE_SCALED],
                   [265 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 75 *
SCALE_FACTOR - ROBOT_SIZE_SCALED],
                   [265 * SCALE_FACTOR + ROBOT_SIZE_SCALED, 200 *
SCALE_FACTOR]],
                  np.int32)
   cv.fillPoly(map, [pts], YELLOW)

   # box 2
   pts = np.array([[250 * SCALE_FACTOR, 200 * SCALE_FACTOR],
                   [250 * SCALE_FACTOR, 75 * SCALE_FACTOR],
                   [265 * SCALE_FACTOR, 75 * SCALE_FACTOR],
                   [265 * SCALE_FACTOR, 200 * SCALE_FACTOR]],
```

```
                np.int32)
    cv.fillPoly(map, [pts], BLUE)

    # circle boundry
    cv.circle(map, (400 * SCALE_FACTOR, 90 * SCALE_FACTOR), 50 *
SCALE_FACTOR + ROBOT_SIZE_SCALED, YELLOW, -1)

    # circle
    cv.circle(map, (400 * SCALE_FACTOR, 90 * SCALE_FACTOR), 50 *
SCALE_FACTOR, BLUE, -1)

    return map


"""
get_valid_point_map()
input: np array representing the world map.  array values are the colors
on the map
output: np array of 1s and 0s representing if a point is valid to be
traveled in.
Function works by checking the color of pixel and determinining if that
color is valid or invalid
"""
def get_valid_point_map(color_map):
    valid_point_map = np.ones((Y_MAX_SCALED, X_MAX_SCALED), np.uint8)
    for x in range(0, X_MAX_SCALED):
        for y in range(0, Y_MAX_SCALED):
            pixel_color = tuple(color_map[y, x])
            if pixel_color == YELLOW or pixel_color == BLUE:
                valid_point_map[y, x] = 0
    return valid_point_map


def determine_valid_point(valid_point_map, coordinates):
    if not __point_is_inside_map(coordinates[X], coordinates[Y]):
        return False
    if valid_point_map[coordinates[Y], coordinates[X]] == 1:
        return True
    else:
        return False
```

```
def __point_is_inside_map(x, y):
   if (x > X_MAX) or (x < 0):
       return False
   elif (y > Y_MAX) or (y < 0):
       return False
   else:
       return True

def __add_point(x, y, map, color):
   map[y, x] = color
   return map


def __draw_line(p1, p2, map, color):
   pts = np.array([[p1[0], p1[1]], [p2[0], p2[1]]],
                  np.int32)
   cv.fillPoly(map, [pts], color)


def draw_node(child_coordinates, parent_coordinates, map, color):

   child_coordinates = tuple(int(SCALE_FACTOR * _) for _ in
child_coordinates)
   cv.circle(map, child_coordinates, radius=3, color=color, thickness=-1)

   if (parent_coordinates is not None):
       parent_coordinates = tuple(int(SCALE_FACTOR * _ ) for _ in
parent_coordinates)
       cv.circle(map, parent_coordinates, radius=3, color=color,
thickness=-1)
       __draw_line(child_coordinates, parent_coordinates, map, color)


#function that defines the goal position as a circle with a threshhold.
# takes in the size of the robot as the threshhold for the target
def point_in_goal(x, y):
   distance = math.sqrt((x-goal_position[0])**2 + (y-goal_position[1])**2)
   if distance <= ROBOT_SIZE_SCALED:
```

```
        return True
    else:
        return False


#function to calculate the cost to go to from the point to the goal in a
straight line
def C2G_func (n_position, g_position):
    C2G = round(((g_position[0]-n_position[0])**2 +
(g_position[1]-n_position[1])**2)**0.5, 1)
    return C2G

def explore(n,UL,UR):
    t = 0
    D=0
    Xn=n[5][X]
    Yn=n[5][Y]
    Thetan = deg2rad(n[5][THETA])

    # Xi, Yi,Thetai: Input point's coordinates
    # Xn, Yn, Thetan: End point coordintes
    while t<1:
        Thetan += (r / L) * (UR - UL) * dt * 2 * math.pi/60
        Xn += round((0.5*r * (UL + UR) * math.cos(Thetan) * dt)*2)/20
        Yn += round((0.5*r * (UL + UR) * math.sin(Thetan) * dt)*2)/20
        t = t + dt
        temp_point = (Xn, Yn)
        if temp_point in obstacle_points:
            return None

    # print("temp_point:", temp_point)
    Theta_n = round(rad2deg(Thetan), 2)
    theta_dot = (r / L) * (UR - UL) *2*math.pi/60
    x_dot = 0.5*(r/100) * (UL + UR) * math.cos(Thetan)*2*math.pi/60
    y_dot = 0.5*(r/100) * (UL + UR) * math.sin(Thetan)*2*math.pi/60
    new_position = round(Xn), round(Yn), round(Theta_n)
    D=round(D+ math.sqrt(math.pow((0.5*r * (UL + UR) * math.cos(Thetan) *
dt),2)+math.pow((0.5*r * (UL + UR) * math.sin(Thetan) * dt),2)), 2)
    new_C2C = round((n[1]+D),2)
    new_C2G = round((C2G_func(new_position, goal_position)),2)
```

```
    new_TC = round((new_C2C + new_C2G),2)

    point_vel = (float(x_dot), float(y_dot), float(theta_dot))
    new_node = (new_TC, new_C2C, new_C2G, node_index, n[5], new_position,
point_vel)

    return new_node

# main exploration function. starts by popping a node out of the open list
and checking the status of the
# popped node prior to expanding the search using the exploration
functions defined above.
# node: (C2G, C2C, TC, point_index, (x,y,theta)parent_coordinates,
(x,y,theta)coordinates)
def exploreNodes():
    global goal_found
    hq.heapify(Open_List)
    # print("OpenListLen:", len(Open_List))
    while Open_List:
        if goal_found:
            break
        popped_node = hq.heappop(Open_List)
        Closed_Coor.add((popped_node[5][0], popped_node[5][1]))

        #popped node is checked and added to the closed list as a dic
        check_popped_status(popped_node)
        popped_node_dic = {"TC": popped_node[0], "C2C": popped_node[1],
"C2G": popped_node[2], "node_index": popped_node[3], "parent_coor":
popped_node[4], "node_coor": popped_node[5], "vel": popped_node[6]}
        Closed_List.append(popped_node_dic)

        # checks if the newly created node falls within the defined map
points, outside the obstacles,
        # has not been closed already and its not within the threshhold of
other points
        # when all pass, it adds it to the open list

        #[RPM_1, RPM_1]
        new_node = explore(copy.deepcopy(popped_node), actions[0][0],
actions[0][1])
```

```
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_1, 0]
        new_node = explore(copy.deepcopy(popped_node), actions[1][0],
actions[1][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[0, RPM_1]
        new_node = explore(copy.deepcopy(popped_node), actions[2][0],
actions[2][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_2, RPM_2]
        new_node = explore(copy.deepcopy(popped_node), actions[3][0],
actions[3][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
```

```
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_2, 0]
        new_node = explore(copy.deepcopy(popped_node), actions[4][0],
actions[4][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[0, RPM_2]
        new_node = explore(copy.deepcopy(popped_node), actions[5][0],
actions[5][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_1, RPM_2]
        new_node = explore(copy.deepcopy(popped_node), actions[6][0],
actions[6][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        #[RPM_2, RPM_1]
        new_node = explore(copy.deepcopy(popped_node), actions[7][0],
actions[7][1])
        if new_node is not None:
            if ((new_node[5][0], new_node[5][1])) in map_points:
```

```
                if ((new_node[5][0], new_node[5][1])) not in
obstacle_points:
                    if ((new_node[5][0], new_node[5][1])) not in
Closed_Coor:
                        if threshhold(new_node[5][0], new_node[5][1]):
                            checkTC(copy.deepcopy(popped_node), new_node)
        # print("OpenList:", Open_List)
        return Open_List, Closed_Coor, Closed_List

#threshhold function that checks if the newly created point falls within
the already explored points,
# angle must be the same. point can have multiple different angle
def threshhold(nx, ny):
    if (nx, ny) in threshold_coor:
        return True
    else:
        threshold_coor.add((nx+0.5, ny))
        threshold_coor.add((nx+0.5, ny+0.5))
        threshold_coor.add((nx, ny+0.5))
        threshold_coor.add((nx-0.5, ny+0.5))
        threshold_coor.add((nx-0.5, ny))
        threshold_coor.add((nx-0.5, ny-0.5))
        threshold_coor.add((nx, ny-0.5))
        threshold_coor.add((nx, ny+0.5))
        return threshold_coor, False

#check if newly explored point has been explored previously, if so compare
C2C and update if the new C2C is
# lower than the one originally stored
def checkTC (on, n):
    global node_index
    for i, nodes in enumerate(Open_List):
        if (nodes[5][0],nodes[5][1]) == (n[5][0],n[5][1]):
            if n[0] < nodes[0]:
                new_node = (n[0], n[1], n[2], nodes[1], n[4], n[5], n[6])
                Open_List[i] = new_node
                hq.heapify(Open_List)
            return Open_List
    else:
        node_index += 1
```

```
        new = (n[0], n[1], n[2], node_index, on[5], n[5], n[6])
        hq.heappush(Open_List, new)
    return Open_List

#function to check the status of the popped node, if it matches the goal
coordinates it starts the backtracking function
def check_popped_status (n):
    global goal_found
    if point_in_goal(n[5][0], n[5][1]):
        goal_node = {"TC": n[0], "C2C": n[1], "C2G": n[2], "node_index":
n[3], "parent_coor": n[4], "node_coor": n[5], "vel": n[6]}
        Closed_List.append(goal_node)
        print("goal node position:", n[5])
        print("target position:", goal_position)
        print("Goal found")
        print("destination info:", n)
        goal_found = True
        start_backtrack ()
    else:
        return(n)


#backtracking function that takes in the last closed node from the closed
list and runs a loop using
#  the node coordinates and the parent coordinate to trace back the steps
leading to the start position
def start_backtrack ():
    print("Backtracking...")
    path_nodes = []
    path_coor = []
    path_vel = []
    current_node = Closed_List[-1]
    path_nodes.append(current_node)
    path_coor.append((current_node['node_coor'][X],
current_node['node_coor'][Y], current_node['node_coor'][THETA]))
    path_vel.append((current_node['vel'][X], current_node['vel'][Y],
current_node['vel'][THETA]))
    print("First node used:", current_node)
```

```
# (TC, C2C, C2G, point_index, (x,y,theta)parent_coordinates,
(x,y,theta)coordinates, vel(x,y,theta))
   while current_node["parent_coor"] is not None:
       search_value = current_node["parent_coor"]
       for node in Closed_List:
           if node["node_coor"] == search_value:
               # If a matching value is found, assign the entire
dictionary as the new current_node
               current_node = node
               break
       path_nodes.append(current_node)
       path_coor.append((current_node['node_coor'][X],
current_node['node_coor'][Y], current_node['node_coor'][THETA]))
       path_vel.append((current_node['vel'][X], current_node['vel'][Y],
current_node['vel'][THETA]))

   path_nodes.reverse()
   path_coor.reverse()
   path_vel.reverse()
   move(path_vel)
   run_visualization(path_coor)



def move(path_vel):

   vel_msg = Twist()
   if ROS == True:
       rospy.init_node('turtlebot3_control')
       vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=10)
       print('starting movement!')

   rate = rospy.Rate(.775)

   counter = 0
   for i in path_vel:
       vel_msg.linear.x = math.sqrt(i[0]**2 + i[1]**2)
       vel_msg.angular.z = (-1*i[2])

       print("[",counter,"]")
```

```
        counter+=1
        print(vel_msg)
        print()
        if ROS == True:
            vel_pub.publish(vel_msg)
            rate.sleep()

    vel_msg.linear.x = 0
    vel_msg.angular.z = 0
    if ROS == True:
        vel_pub.publish(vel_msg)

# runs visulization of path plan using OpenCV.
# path shown is the solution path, with the connected nodes from the open
list
def run_visualization(path_coordinates):
    for node in path_coordinates:
        if not point_in_goal(node[X], node[Y]):
            draw_node((node[X], node[Y]), None, color_map, GREEN)
            for action in actions:
                plot_curve(node, action[0], action[1], color_map, GRAY)
            cv.imshow('A* Algorithm', color_map)
            cv.waitKey(200)
        else:
            #color goal node red
            node = path_coordinates[-1]
            draw_node((node[X], node[Y]), None, color_map, RED)

    cv.imshow('A* Algorithm', color_map)
    cv.waitKey(0)
    cv.destroyAllWindows()
    return


def plot_curve(node, UL,UR, map, color):
    t = 0
    D=0
    Xn=node[X]
    Yn=node[Y]
    Thetan = deg2rad(node[THETA])
```

```
    # Xi, Yi,Thetai: Input point's coordinates
    # Xs, Ys: Start point coordinates for plot function
    # Xn, Yn, Thetan: End point coordintes
    while t<1:
        Xs = Xn
        Ys = Yn
        Thetan += (r / L) * (UR - UL) * dt * 2 * math.pi/60
        Xn += round((0.5*r * (UL + UR) * math.cos(Thetan) * dt)*2)/20
        Yn += round((0.5*r * (UL + UR) * math.sin(Thetan) * dt)*2)/20
        t = t + dt
        __draw_line((Xs*SCALE_FACTOR, Ys*SCALE_FACTOR), (Xn*SCALE_FACTOR,
Yn*SCALE_FACTOR), map, color)
    Xn = Xn
    Yn = Yn


if __name__ == '__main__':

    DEBUG = True
    ROS = True

    # user input required to fill out the following
    if DEBUG == False:
        start_x_position = int(input("enter start X position(0-600): "))
        start_y_position = int(input("enter start Y position(0-250): "))
        start_theta_position = int(input("enter start theta
position(0-360): "))
        print()
        goal_x_position = int(input("enter goal X position(0-600): "))
        goal_y_position = int(input("enter goal y position(0-250): "))
        print()
        RPM_1 = int(input("enter RPM for wheel rotation speed 1 (0-100):
"))
        RPM_2 = int(input("enter RPM for wheel rotation speed 2 (0-100):
"))
        print()
        step_size = int(input("enter step size (0-10): "))
        clearance = int(input("enter the clearance used for navigation (in
mm): "))
```

```
        ROBOT_SIZE_SCALED = ROBOT_SIZE + clearance * SCALE_FACTOR
    else:
        start_x_position  = 50
        start_y_position  = 100
        start_theta_position  = 0
        goal_x_position  = 530
        goal_y_position  = 180
        RPM_1 = 10          # (rot/s)
        RPM_2 = 20           # (rot/s)
        r = 3.3             # (cm)
        L = 12              # (cm)
        dt = 0.1
        # step_size = 1
        # ROBOT_SIZE = 5  # (mm) 105 is the size if the robot in mm.  Will
not work for this simulation
        clearance = 10
        ROBOT_SIZE_SCALED = ROBOT_SIZE_SCALED + clearance * SCALE_FACTOR

    # 0 = left   1 = right
    actions = [[RPM_1, RPM_1], [RPM_1, 0], [0, RPM_1], [RPM_2, RPM_2],
[RPM_2, 0], [0, RPM_2], [RPM_1, RPM_2], [RPM_2, RPM_1]]

    start_position = (start_x_position, start_y_position,
start_theta_position)
    goal_position = (goal_x_position, goal_y_position)

    # initial values for the start node
    vel1 = (0, 0, 0)
    C2G1 = C2G_func(start_position, goal_position)
    C2C1 = 0
    TC1 = C2G1 + C2C1
    # (C2G, C2C, TC, point_index, (x,y,theta)parent_coordinates,
(x,y,theta)coordinates, velocity)
    start_node = (TC1, C2C1, C2G1, node_index, None, start_position, vel1)
    hq.heappush(Open_List, start_node)
    print("initial_Open_list:", Open_List)

    print("Drawing map...")
    color_map = draw_map()
```

```
    print("Determining open points and obstacle points...")
    valid_point_map = get_valid_point_map(color_map)
    # creat 2 sets, 1 containing all the possible points within map, 1
containg all possible points within the obstacle spaces
    # this will be used later to check the created points to see if they
can be used
    x_range = np.arange(0, 600, 0.5)
    y_range = np.arange(0, 200, 0.5)
    for x in np.arange(0, 600, 0.5):
        for y in np.arange(0, 200, 0.5):
            if valid_point_map[int(y * SCALE_FACTOR), int(x *
SCALE_FACTOR)] == 1:
                map_points.add((x, y))
            else:
                obstacle_points.add((x, y))

    if (start_position[0], start_position[1]) in obstacle_points:
        print("start point selected is in obstacle space, try again")
        exit()
    if (start_position[0], start_position[1]) not in map_points:
        print("start point selected is  outside the map, try again")
        exit()
    if (goal_position[0], goal_position[1]) in obstacle_points:
        print(goal_position)
        print("goal point selected is in obstacle space, try again")
        exit()
    if (goal_position[0], goal_position[1]) not in map_points:
        print("goal point selected is  outside the map, try again")
        exit()

    goal_found = False
    print("start node:", start_node)
    print("starting exploration")
    while not goal_found:
        exploreNodes()
```