Path planning using RRT

Author:

- Khalid OUBLAL - khalid.oublal@polytechnique.edu (mailto:khalid.oublal@polytechnique.edu)
- Github: https://github.com/oublalkhalid/Path-planning-using-RRT (https://github.com/oublalkhalid/Path-planning-using-RRT)

There are two levels of planning, local planning and global planning. We have already studied global planning in previews works, which stores environmental information in a map and uses this map to find a feasible path. But it is not suitable in unknown environments. In this present work, we investigate local path planning, which only takes into account the instantaneous environmental information of the robot, which helps us to reduce the computation time.

Path planning can consist of four parts: obstacle avoidance, reactive planning, stochastic path search and exploration. The Vector Field Histogram, dynamic window and potential flows can be used to avoid obstacles. In the stochastic path search, the RRT algorithm and its variants are used. And the exploration of unknown environment can be done by the $RTT^\star$ algorithm \cite{karaman2011sampling}. In the rest of this practical work, we will illustrate the different steps explored on the Rapidly Exploring Random Tree RRT and one of its variants $RRT^\star$ algorithm.

Entrée [ ]:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Feb 15 19:38:27 2022
RRT_2D
@author: huiming zhou & David Filliat
Modified by Khalid Oublal
"""

# console 7/A
import os
import sys
import math
import numpy as np
import utils
import env
from utils import *
from tqdm import tqdm
import pandas as pd


# parameters
showAnimation = False
```

Entrée [ ]:

```
class Node:
    def __init__(self, n):
        self.x = n[0]
        self.y = n[1]
        self.parent = None

class Rrt:
    def __init__(self, environment, s_start, s_goal, step_len, goal_sample_rate, ite
        self.s_start = Node(s_start)
        self.s_goal = Node(s_goal)
        self.step_len = step_len
        self.goal_sample_rate = goal_sample_rate
        self.iter_max = iter_max
        self.vertex = [self.s_start]

        self.env = environment
        #self.plotting = plotting.Plotting(self.env, s_start, s_goal)
        self.utils = utils.Utils(self.env)

        self.x_range = self.env.x_range
        self.y_range = self.env.y_range
        self.obs_circle = self.env.obs_circle
        self.obs_rectangle = self.env.obs_rectangle
        self.obs_boundary = self.env.obs_boundary

    def planning(self):

        for i in range(self.iter_max):
            node_rand = self.generate_random_node(self.goal_sample_rate)
            node_near = self.nearest_neighbor(self.vertex, node_rand)
            node_new = self.new_state(node_near, node_rand)

            if node_new and not self.utils.is_collision(node_near, node_new):
                self.vertex.append(node_new)
                dist, _ = self.get_distance_and_angle(node_new, self.s_goal)

                if dist <= self.step_len:
                    self.new_state(node_new, self.s_goal)
                    return self.extract_path(node_new), i

        return None, self.iter_max

    # def generate_random_node(self, goal_sample_rate):
    #     if np.random.random() < goal_sample_rate:
    #         return self.s_goal
    #     delta = self.utils.delta
    #     return Node((np.random.uniform(self.x_range[0] + delta, self.x_range[1] -
```

Response to Q.4

Impelmentation of generate_random_node() function

Entrée [ ]:

```
nerate_random_node(self, goal_sample_rate):
if np.random.random() < goal_sample_rate:
    return self.s_goal

delta = self.utils.delta

node = Node((np.random.uniform(self.x_range[0] + delta, self.x_range[1] - delta),
            np.random.uniform(self.y_range[0] + delta, self.y_range[1] - delta)))

if np.random.randn() < 0.6:
    while 1:
        id = np.random.randint(len(self.env.obs_rectangle))
        #[x, y, w, h] = self.env.obs_rectangle[id]
        # We can use directly self.env.obs_rectangle[id][index]
        node_list =[Node((np.random.uniform(self.env.obs_rectangle[id][0] - delta, s
                           np.random.uniform(self.env.obs_rectangle[id][1] - delta,
                     Node((np.random.uniform(self.env.obs_rectangle[id][0] + self.en
                           np.random.uniform(self.env.obs_rectangle[id][1] - delta,
                     Node((np.random.uniform(self.env.obs_rectangle[id][0] - delta,
                           np.random.uniform(self.env.obs_rectangle[id][1] + self.en
                     Node((np.random.uniform(self.env.obs_rectangle[id][0] + self.en
                           np.random.uniform(self.env.obs_rectangle[id][1] + self.en
                     ]
        node = node_list[np.random.randint(len(node_list))]
        #node = Node((np.random.uniform(self.env.obs_rectangle[id][0] - delta, self.
         #           np.random.uniform(self.env.obs_rectangle[id][1] - delta, self.e
        if self.utils.is_inside_obs(node):
            break
return node
```

Entrée [ ]:

```
    @staticmethod
    def nearest_neighbor(node_list, n):
        return node_list[int(np.argmin([math.hypot(nd.x - n.x, nd.y - n.y)
                                        for nd in node_list]))]

    def new_state(self, node_start, node_end):
        dist, theta = self.get_distance_and_angle(node_start, node_end)

        dist = min(self.step_len, dist)
        node_new = Node((node_start.x + dist * math.cos(theta),
                         node_start.y + dist * math.sin(theta)))
        node_new.parent = node_start

        return node_new

    def extract_path(self, node_end):
        path = [(self.s_goal.x, self.s_goal.y)]
        node_now = node_end

        while node_now.parent is not None:
            node_now = node_now.parent
            path.append((node_now.x, node_now.y))

        return path

    @staticmethod
    def get_distance_and_angle(node_start, node_end):
        dx = node_end.x - node_start.x
        dy = node_end.y - node_start.y
        return math.hypot(dx, dy), math.atan2(dy, dx)


def get_path_length(path):
    """
    Compute path length
    """
    length = 0
    for i,k in zip(path[0::], path[1::]):
        length += math.dist(i,k)
    return length


def main():
    x_start = (2, 2)  # Starting node
    x_goal = (49, 24)  # Goal node
    environment = env.Env2()

    rrt = Rrt(environment, x_start, x_goal, 2, 0.10, 1500)
    path, nb_iter = rrt.planning()


    # average_path = 0
    # average_iteration = 0
    # N = 50
    # for i in range(0,N):
    #     rrt = Rrt(environment, x_start, x_goal, 2, 0.10, 10000)
    #     path, nb_iter = rrt.planning()
    #     if path:
```

```
    #         average_path += get_path_length(path)/N
    #         average_iteration += nb_iter/N
    #     else:
    #         print("No Path Found in " + str(nb_iter) + " iterations!")
    # print('Found path in ' + str(average_iteration) + ' iterations, length : ' + s

data_dic = { "max_iter": [1500 for i in range(50)],
    "Time_computation": [],
    "path_index": [],
    "path_length": []}

def run(data_dic,rate):
    x_start=(2, 2)  # Starting node
    x_goal=(49, 24)  # Goal node
    environment = env.Env2()
    for i in tqdm(data_dic["max_iter"]):

        t_end_list=[]
        nb_iter_list=[]
        path_list=[]

        for j in range(1): # avoid stochastic character
            t0=current_milli_time()
            rrt = Rrt(environment, x_start, x_goal, 2, rate, i)
            path, nb_iter = rrt.planning() # path and iteration

            # Compute and save data
            t_end=current_milli_time()-t0 # time

            t_end_list.append(t_end)
            nb_iter_list.append(nb_iter)
            path_list.append(path)

            # if path:
            #     #print("Time computation (ms) is:", t_end)
            #     print('Found path in ' + str(nb_iter) + ' iterations, length : ' +
            #     if showAnimation:
            #         rrt.plotting.animation(rrt.vertex, path, "RRT", True)
            #         plotting.plt.show()
            # else:
            #     print("Time computation (ms) is:", t_end)
            #     print("No Path Found in " + str(nb_iter) + " iterations!")
            #     if showAnimation:
            #         rrt.plotting.animation(rrt.vertex, [], "RRT", True)
            #         plotting.plt.show()


        data_dic= add_data(data_dic,path_list,nb_iter_list,t_end_list)

    return data_dic
```

Entrée [ ]:

```
#   data_dic = { "step_len": [0.1,0.2,1,2,5,10,20,40,50,80,100],
#   "Time_computation": [],
#   "path_index": [],
#   "path_length": []}

if __name__ == '__main__':
    #main()
    for i in [0.10]:
        data_dic = { "max_iter": [1500 for i in range(50)],"Time_computation": [],
                    "path_index": [],"path_length": []}
        print("\n ETAPE ------",i,"% \n:" )
        data_dic=run(data_dic,i)
        dataFrame=pd.DataFrame.from_dict(data_dic)
        dataFrame.to_pickle("data/RRT_Env2_50_cycle_newFunction_0p1percent.pkl")
    #data=pd.read_pickle("result_rrt_Env2_100_cycle.pkl")
```