Report for the Date : 2023/11/08
Name : Kasha Singh
ITE No : ITE001456
Things I did today :

1. Writing code for optimal path of RRT* algorithm

In [1]:
```
import math
import numpy as np

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


class RrtStar:
    def __init__(self, x_start, x_goal, step_len,
                 goal_sample_rate, search_radius, iter_max):
        self.s_start = Node(x_start)
        self.s_goal = Node(x_goal)
        self.step_len = step_len
        self.goal_sample_rate = goal_sample_rate
        self.search_radius = search_radius
        self.iter_max = iter_max
        self.vertex = [self.s_start]
        self.path = []


    def planning(self):
        for k 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.is_collision(node_near, node_new):
                neighbor_index = self.find_near_neighbor(node_new)
                self.vertex.append(node_new)
```

```
                if neighbor_index:
                    self.choose_parent(node_new, neighbor_index)
                    self.rewire(node_new, neighbor_index)

        index = self.search_goal_parent()
        self.path = self.extract_path(self.vertex[index])


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

        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 choose_parent(self, node_new, neighbor_index):
        cost = [self.get_new_cost(self.vertex[i], node_new) for i in neighbor_index]

        cost_min_index = neighbor_index[int(np.argmin(cost))]
        node_new.parent = self.vertex[cost_min_index]

    def rewire(self, node_new, neighbor_index):
        for i in neighbor_index:
            node_neighbor = self.vertex[i]

            if self.cost(node_neighbor) > self.get_new_cost(node_new, node_neighbor):
                node_neighbor.parent = node_new

    def search_goal_parent(self):
        dist_list = [math.hypot(n.x - self.s_goal.x, n.y - self.s_goal.y) for n in self.vertex]
        node_index = [i for i in range(len(dist_list)) if dist_list[i] <= self.step_len]
```

```
        if len(node_index) > 0:
            cost_list = [dist_list[i] + self.cost(self.vertex[i]) for i in node_index
                         if not self.utils.is_collision(self.vertex[i], self.s_goal)]
            return node_index[int(np.argmin(cost_list))]

        return len(self.vertex) - 1

    def get_new_cost(self, node_start, node_end):
        dist, _ = self.get_distance_and_angle(node_start, node_end)

        return self.cost(node_start) + dist

    def generate_random_node(self, goal_sample_rate):
        delta = self.utils.delta

        if np.random.random() > goal_sample_rate:
            return 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)))

        return self.s_goal

    def find_near_neighbor(self, node_new):
        n = len(self.vertex) + 1
        r = min(self.search_radius * math.sqrt((math.log(n) / n)), self.step_len)

        dist_table = [math.hypot(nd.x - node_new.x, nd.y - node_new.y) for nd in self.vertex]
        dist_table_index = [ind for ind in range(len(dist_table)) if dist_table[ind] <= r and
                            not self.utils.is_collision(node_new, self.vertex[ind])]

        return dist_table_index

    def is_collision(self, start, end):
        if self.is_inside_obs(start) or self.is_inside_obs(end):
            return True
```

```
    @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]))]

    @staticmethod
    def cost(node_p):
        node = node_p
        cost = 0.0

        while node.parent:
            cost += math.hypot(node.x - node.parent.x, node.y - node.parent.y)
            node = node.parent

        return cost

    def update_cost(self, parent_node):
        OPEN = queue.QueueFIFO()
        OPEN.put(parent_node)

        while not OPEN.empty():
            node = OPEN.get()

            if len(node.child) == 0:
                continue

            for node_c in node.child:
                node_c.Cost = self.get_new_cost(node, node_c)
                OPEN.put(node_c)

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

```
        while node.parent is not None:
            path.append([node.x, node.y])
            node = node.parent
        path.append([node.x, node.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 main():
    x_start = (18, 8)  # Starting node
    x_goal = (37, 18)  # Goal node

    rrt_star = RrtStar(x_start, x_goal, 10, 0.10, 20, 10000)
    rrt_star.planning()


if __name__ == '__main__':
    main()
```

2. Learnt about navigation concepts of Nav2 Library
-> Action Server
-> Behaviour trees
-> Environmental representation

3. Learnt how to use gazebo

4. Worked on turtle sim package
-> Creating publisher and subscriber in C++ and Python
-> Creating a service client with C++ to manage the turtle
-> Implemented Custom Interfaces