MOTION PLANNING FOR MOBILE ROBOT (TOY CAR MODEL) USING RRT AND RRT* ALGORITHMS

(Prof. Reza Monferadi, Girish Kumar Ethirajan, Lavanya Suresh Kannan)

ABSTRACT:

When it comes to mobile robotics, intelligent path planning algorithms plays a significant tool. This later has a great demand in the market. This project focuses mainly on the author's toy car model which has a mobile base and a manipulator. Aiming to realise the obstacle avoidance of a mobile robot in an unstructured environment, Rapidly Exploring Randomised Tree and Rapidly Exploring Randomised Tree star are implemented in this project. The author used SOLID WORKS to build a model and also used Gazebo for simulation. The preliminary results show optimal path difference between RRT and RRT*.

Keywords: planning, RRT, RRT*, gazebo, ROS

1. INTRODUCTION:

Path planning in autonomous robotics has been an active research topic for years as of now. This process has three main steps, (i) initiating start and goal points and actions that need to be taken next, (ii) implementing the planning algorithm, (iii) markings its position. In this project we will deal with all the three iterative steps. For this project we are using author's toy car model. The model designed by the author has a 5 DOF manipulator which has 5 revolt joints that uses servo motors. Figure (1) shows the manipulator model which represents the joint movement with actuator. This robotic arm will sit in front of the base that picks and drops items from the ground in its base.

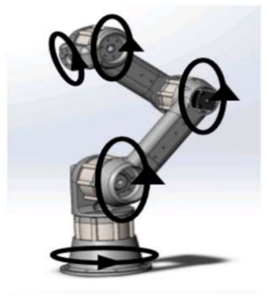

Figure 1

Since the project focuses on the mobile base part, we are ignoring the robotic arm part. The mobile base has a rack like body with four wheels which is made up of rubber tires. The wheels are driven by the effort velocity controller, Figure 2. Our ultimate aim is to implement the path planning algorithm in the working area that is filled with obstacles. One of the main issues in implementing RRT is, sometimes it does not give smooth path in a narrow passage area. For that case, we decide to implement RRT* and compare the results. These results are later simulated in gazebo, ROS.

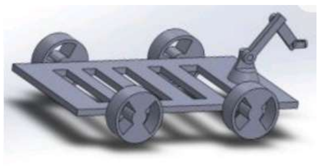

Figure 2

2. PATH PLANNING AND SIMULATION:

A. Rapidly Exploring Random Trees

RRT is one of the most popular path planning algorithms in motion planning. This algorithm is designed to efficiently search path in a non-convex high dimensional workspace. Generally, RRT explores random samples from the workspace and tries to make a feasible connection with the nearest state tree. Some drawbacks of RRT are such that it may not give optimal path all the time, it may take longer time to find the paths.

The steps involved in RRT are:

RRT pseudocode:

```
Sstart
Qgoal
iterations = n
count = 0
Graph = [ ]
Graph (vertices, edges)
while count < iterations:
  new_node = random ()
  if obstacle(new_node)==True:
        continue
  node_neighbour = neighbour (Graph
(vertices, edges), new_node)
  tree=link(new_node, node_neighbour)
  Graph.appned(tree)
  if new_node in Qgoal:
      return Graph
return Graph
```

B. Rapidly exploring Random Trees Star

RRT* is similar to that of RRT except for two steps that it performs additionally. They are,

Best parent: whenever a new node is connected to the tree, an extra step is performed in order to find the best parent. To do this a vicinity is chosen around the new node and later check the explored node. Based on the least cost-to-come, the parent node is chosen.

Rewire: once the new nodes are added, the nodes in the vicinity are then rewired based on cost-to-come These two steps performed by RRT* makes it more optimal compared to RRT.

RRT* pseudocode:

```
Goal (Vertices, Edges)
for i in range (0, n):
   new_node = Random ()
   if obstacle(new_node) == True
    nearest _node = nearest (Goal
(Vertices, Edges), new_node)
    best_parent, node_neighbour =
find_neighbours (goal (vertices,
edges), new_node, r)
     tree = chain (new_node,
best_parent)
     for a in range node_ndeighbours:
        if cost(new_node) + distance
(new_node, a) < cost(a)
             cost(a) + cost(new_node)+
       distance (new_node, a)
```

```
                parent(a) = new_node
                Goal += (new_node, a)
        Goal += tree
return Goal
```

C. RRT vs RRT*

Except for the procedure that is added to look at the path cost, every other procedure is almost similar in both RRT and RRT*. The tree grows incrementally fast in both the cases. The two procedure which was stated earlier makes RRT to be asymptotically optimal. RRT also takes less computational time and less expensive whereas RRT* take more time to find the optimal path. Though RRT* gives us an optimal solution, the rate of convergence is very slow compared to RRT.

RRT	RRT*
It is faster.	It is less fast comparatively.
The path is longer.	The path is shorter.
Does not perform an additional step to find best parent.	Perform an additional step to find parent.
Neighbour nodes are not rewired.	Rewire the neighbour nodes based on cost-to-come.

D. Implementation

The world considered here is same as considered in the previous project. The parameters considered are the following, (i) threshold distance between each node is 0.1 m, (ii) goal threshold distance is 0.1 m, (iii) effective clearance considered here is 0.4 m, and (iv) maximum number of iterations is set to 30000.

The figures 3 and 4 given below show the result obtained after implementing RRT and RRT* in a given world. As expected, RRT* took more time to complete.

The nodes exploration and the solution path planned is plotted using pygame. The solution path is shown with red marker lines in RRT and yellow marker lines in RRT*.

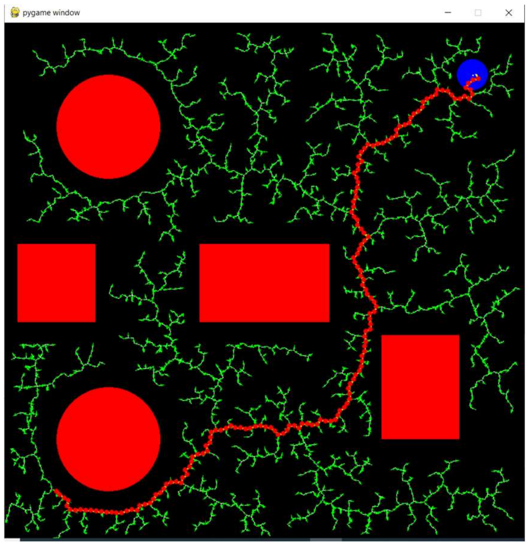

Figure 3: RRT

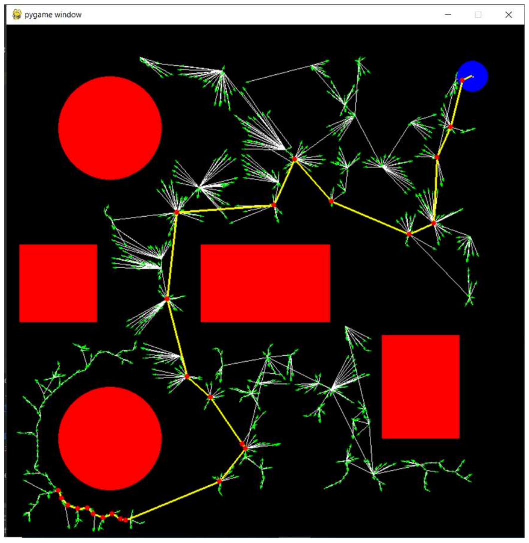

Figure 4: RRT*

3. MODEL OF TOY CAR:

Toy car given is a model which can be taken of as two parts, mobile base, and robot arm. The task was to move the mobile base. For this purpose, we had to modify the Xacro file of the given robot.

We tried out differential drive motion, by modifying the URDF file. The revolute joints where modified to have a limit of -3.14 to 3.14 and effort and velocity having values of 1 and 12 respectively. The differential drive was implemented on the front wheels and the back wheels were made passive using a differential drive plugin in Gazebo. However, it did not produce differential motion and rather moved a little and stopped.

We also tried to change the joints from revolute type to continuous. This resulted in slightly a better motion. Even so, its motion was still limited, considering we have to simulate the planned path generated using RRT and RRT* in Gazebo.

Such improper motions were observed and can be attributed to the toy car model as the author mainly designed it for its manipulator arm. We could fix the motion which will be suitable for the differential drive by tweaking the author's toy car model.

But as we wanted to use the same model without any changes, the **plugin planar move** was added, and it helped to move the robot to the corresponding twist/ velocity messages as shown in figures 5 and 6.

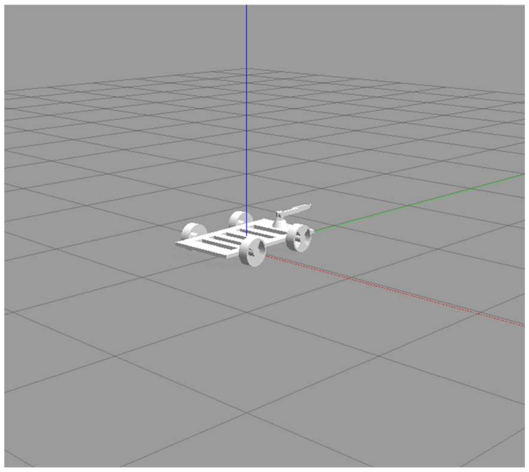

Figure 5: Toy car

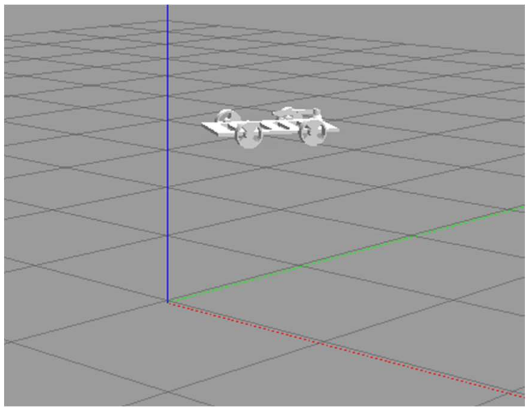

Figure 6: Toy car in motion

We also spawned the Toy Car in our world and validated its motion using teleop package in ROS. We observed that to move the toy car forward or backward, we had to publish Twist messages to the y-component of linear velocity. Normally, we would publish it to the x-component. This is due to the wheel axis setup in the toy car model.

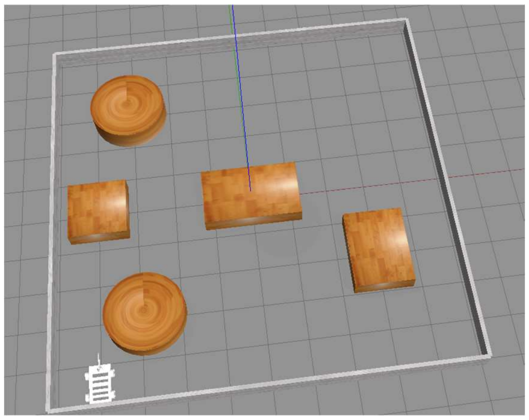

Figure 7: Toy car in Gazebo World

We wrote our custom package, 'simple navigation goals'. This package is used to move the robot to different poses in our world using an action client-server move base approach. We set the poses to reach on the parameter server inside the launch directory of this package. The poses will be read from the planned path generated using different algorithms. Later in the move_base_seq.py file (inside the scripts directory of this package), we get these parameters one at a time and move our toy car to that pose. This way of publishing onto our toy car is much more suited and reliable for our purpose and works efficiently (validated for a burger turtlebot3). However, occasionally we have to wait for a long time for the action server to start. We are yet to fix this error.

4. CONCLUSIONS:

The implementation of path planning for our toy car using different algorithms was a success. We also managed to successfully give linear and angular motions to the toy car by configuring the model to some extent and also by installing additional ROS plugins. Moreover, we also managed to write a ROS package to move a robot to different poses in the environment using action client-server move_base approach and validated it for a burger turtlebot3. However, we did not find much success in implementing our package with the toy car. Overall, it was a good learning experience.