Path Planning for TurtleBot using RRT*

CPSC 8810 Motion Planning

Final Project Report

Mohammad Anas Imam Khan, Manikanda Balaji V., Siddhant Srivastava, Aakanksha Smriti

Problem Statement

The aim of our project was to write a custom global planner that employs the RRT* algorithm in Python and implement this global planner to make the TurtleBot3 Burger traverse through the generated path using RRT*.

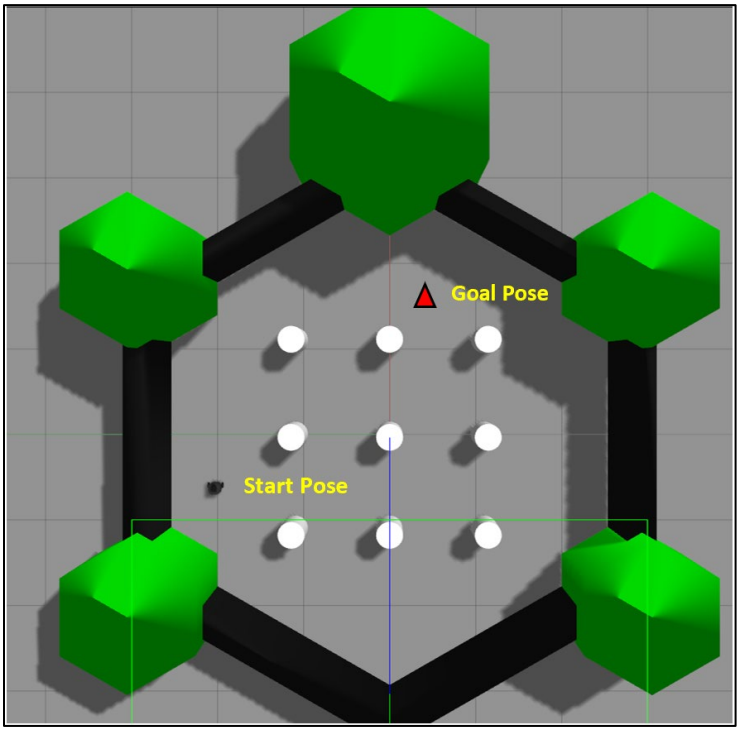

Figure 1: Gazebo world with start and goal position

Methodology

1. **Map generation:**
   An image of a known map in Gazebo was imported into MATLAB. The image was then converted from RGB to grayscale using the rgb2gray command of OpenCV. The grayscale image shown in figure 2 shows the white portions as the obstacles and the black portions as the free space.

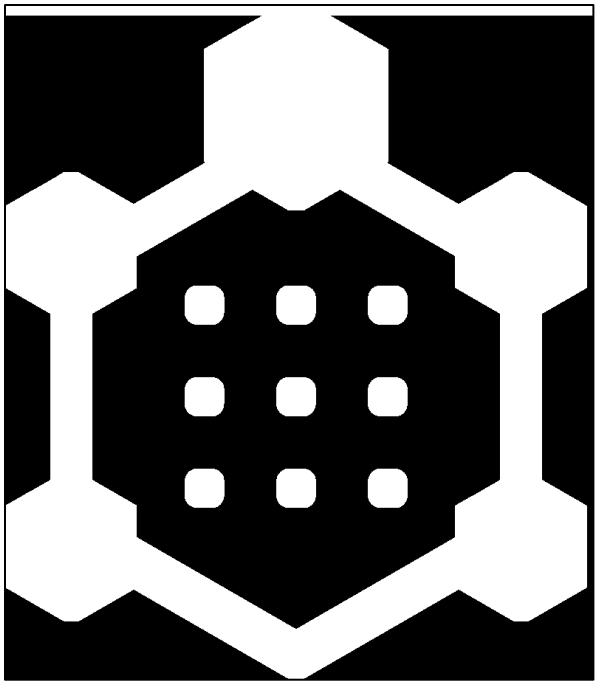

Figure 2: Grayscale inflated occupancy map

The image was then converted into binary format representing the obstacles as '1' and free space as '0'. Taking the threshold values of the obstacles, the obstacles were inflated to create a c-shape. This generated occupancy map was exported as a .csv file along with the point map.

2. **Edge collision and point collision:**

The csv occupancy map file was imported into python and then read using a custom function called *occupancy_map_generator().* The csv file was read as a NumPy array.

The function above helped in generating the function, *edge_collision_checker(x1,y1,x2,y2)* that would check for collision when two nodes are being joined. This function takes input as the x-y coordinates of the respective nodes.

The edge collision checker function utilizes the *collision_check(x,y)* to see if during interpolation points between an edge lie within the obstacle space both in the x-y coordinate frame.

3. **RRT* algorithm:**

A class named *RRTStar()* was defined that contains another class named *Node()* that helps in storing the x and y coordinates of the node as well as the cost to traverse the path and the node's parent.

The main class has four tuning variables namely:

- *expand_dis:* Used to check if the distance to traverse from one node to another is greater than the value specified in this variable. If two nodes are not connected, then the distance is larger than the value.

- *Path_resolution:* This is used to provide a physical limitation on the steps that a TurtleBot can take in the gazebo world. For the value of 60 used for this variable, the units that the TurtleBot can move 0.6 units. If the path resolution is found to be larger than the nodes are not traversed.
- *max_iter:* Used to specify the maximum number of iterations for the RRT* algorithm to compute.
- *connect_circle_dist:* Used to find nearby nodes within a radius using the formula specified in the pseudo-code below.

Psuedo-code RRT*:

a) add start node to node list
   for n iterations:
   - get random node
   - new_node = random node
   - find nearest node from node list for the random node
   - use *steer()* function

b) *steer()* function:
   - find distance between source_node and destination_node
   - new_node = source_node
   - add new_node in the path list of new node
   - check if the distance calculated is greater than expand_dis
   - If true, number of divisions = expand_dis/path_resolution
   - for range of number of divisions:
     - steer the node in x(cos theta) and y(sin theta) directions in steps of path_resolution.
   - make source node as parent of destination node
   - return the final new_node

c) check for edge collision of new_node and its parent node
   find the nearest index of new_node using *find_near_nodes()* function

d) *find_near_nodes()* function:
   - nnode = length of node list + 1
   - r = connect_circle_dist $* \sqrt{\log(nnode) / nnode}$
   - take r as minimum of expand_dis and r
   - find distance between new_node and all nodes in node list
   - take nodes that are less than r
   - return the indices of these nodes

e) *choose_parent()* function:
   - for all the near node indices:
     - apply *steer()* function for near node and new_node
     - check for edge collision between near node and new_node
       - if no collision, calculate the cost to the new node via all the near node

- choose the near node with the least cost as the parent node of new_node
- apply *steer()* function between for the above parent node and new node
- the minimum cost calculated becomes the cost of the new node

f) add the new_node in the node list

g) *rewire()* function:
- take same near node indices from *find_near_nodes()* function
- for all the near node indices:
    - apply *steer()* function for new_node and new_edge_node
    - calculate improved cost of the new_edge_node created in previous step
    - check for edge collision between new_node and new_edge_node
    - if improved cost < old cost and no collision
        - make new_node as parent of that new_edge_node
        - update the cost of new_node
    - propagate the change in cost of the new_node to its child node using *modify_child_cost()* function

h) *modify_child_cost()* function:
- run through all nodes in node list
- check if the modified cost node is parent of the node and again call *modify_child_cost()* function to propagate cost with this node to its child

The RRT* algorithm after completion gave final path and x and y coordinates of all the nodes in the path as a list. This list was then given to the global planner for ROS.

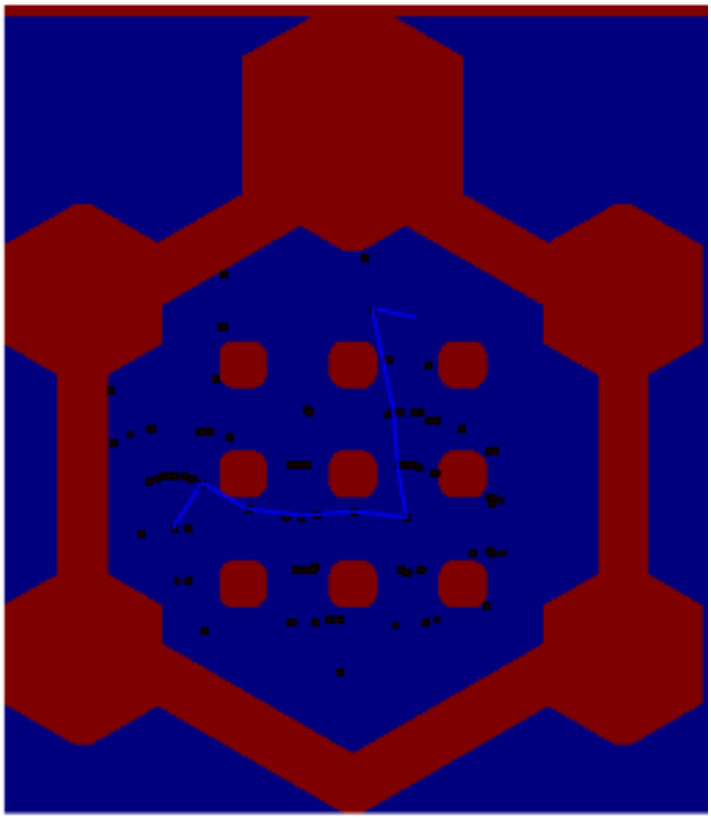

Figure 3 Generated Path (Shown in blue)

4. Global Planner for ROS

The x-y coordinates of the final path that was backtracked using the *rrtstar_path()* function were fed into Gazebo. However, before feeding the path to Gazebo, the transformation from the image co-ordinate to Gazebo co-ordinate had to be made. This transformation used x and y offsets and a scaling factor. The gazebo world's x-y coordinates were the image's y-x coordinate. The process to arrive at the y and x offset was to import the image of the map in MATLAB and then use OpenCV's function, imcircle to find coordinates of a circle and its radius at the origin of gazebo world. The co-ordinate of the center circle was the x and y offset. Radius was found to be 15 whereas the actual radius of the circle in the gazebo is 0.15 which corresponds to a scale of 100:1.

$$y_{offset} = 529, x_{offset} = 393, scale = 0.01$$

$$xgazebo = (y_{offset} - path\_y\_coordinate) * scale$$
$$ygazebo = (x_{offset} - path\_x\_coordinate) * scale$$

Figure 4 Image to Gazebo Co-ordinate Transform (Equation is written using https://www.mathcha.io/editor)

This transformation was also used to specify the starting points in the launch file for the gazebo world.

Before the x and y co-ordinates could be fed, we had to transform the TurtleBot's yaw position specified in the quaternion frame to the yaw position in the Eulerian frame. The TurtleBot publishes its position in the topic *Odom* that was subscribed. The contents of the message are shown below in Figure. The *tf* package was imported into python that would be used to transform the *x,y,z,* and *w* coordinates in the quaternion frame to roll-pitch-yaw in the Euler frame using the command *tf.transformations.euler_from_quaternion(quaternion).*

geometry_msgs/Quaternion Message

File: geometry_msgs/Quaternion.msg

Raw Message Definition

```
# This represents an orientation in free space in quaternion form.

float64 x
float64 y
float64 z
float64 w
```

Compact Message Definition

```
float64 x
float64 y
float64 z
float64 w
```

Figure 5 Quaternion Message

The path's x and y coordinates were fed individually by looping in a for a loop. The linear distance error was calculated using the Eulerian distance formula. The angular error was calculated using the formula:

$$angular\ error\ =\ \arctan((ygazebo - bot_y) - (xgazebo - bot_x)) - bot_{yaw}$$

Figure 6 Typed Using https://www.mathcha.io/editor

The linear and angular errors were fed into a proportional controller that would be used to publish the velocity to the topic *cmd_vel* to make the TurtleBot move in the Gazebo world.

Conclusion

Challenge faced: The initial objective for the project was to replace the default global planner in the RVIZ simulator with our RRT* path planner. This was difficult to achieve since it would require us to create a plugin using C++ for implementing our custom planner. Given the time constraints, this task proved to be challenging owing to the unfamiliarity with the C++ language. Hence, we adopted the approach mentioned in the Methodology section. The second objective was to perform SLAM on a map and bring up the map with the TurtleBot using the RVIZ simulator where we would send the goal pose to the move base node. This also proved challenging because it primarily requires us to create a plugin first.

References

1) https://github.com/AtsushiSakai/PythonRobotics/tree/master/PathPlanning/RRTStar

2) http://paper.ijcsns.org/07_book/201610/20161004.pdf

3) https://github.com/Mayavan/RRT-star-path-planning-with-turtlebot