RMP Project Report Spring 2018

Dev Takle

**Problem Statement:**

The problem was to find a collision-free path in a 2D environment for a car-like robot with front-wheel rotation from an initial state to a goal state. For this project, I implemented a Rapidly-Exploring Random tree and collision detection algorithm in Python and successfully illustrated my results using the PyGame library.

**Approach:**

This type of car-like robot is known to be *non-holonomic*, in which the control system can be expressed by –

dx = v\*cos(fi) \*cos(theta)

dy = v\*cos(fi) \*sin(theta)

dtheta = (v/L) \*sin(fi)

where theta is the orientation of the robot with respect to the x axis, fi is the steering angle of the front wheels, v is the velocity of the robot and L is the length of the robot.

I used these expressions to approximate the next possible position of the car using the Runge-Kutta Fourth Order method. To store the actual tree, I made use of a hashtable-like data structure - the Python dictionary with (child node, parent node) as the (key, value) pairs. This allowed me to backtrack in the tree to find the path once the RRT reached the goal state.

**Parameters:**

Velocity – I used a velocity of 12. The tree would cover more of the environment with a faster velocity, but accuracy will suffer.

Time step dt – I used a time step of 0.3. A greater value would result in larger steps but lower accuracy.

Iterations – The number of random points to be chosen. Since sampling is uniform, a higher number would result in a large tree. This number should increase if the initial state is far from the goal state.

Goal threshold – I used a goal threshold of 5. A higher goal threshold will result in a faster path computation, but will not be accurate in terms of the final state.

Minimum steering angle – I used an angle of -45 degrees for minimum steer.

Maximum steering angle – I used an angle of 45 degrees for maximum steer.

**Method:**

Choose initial state, final state

Add initial state to the tree

REPEAT

Pick a random state from the environment (uniform sampling)

Find the nearest neighbor of the random state in the tree

For a range of steering angles

Find the optimum next state

If this optimum state passes collision check, add it to the tree

UNTIL

Distance between final state and optimum state is less than the specified threshold

**Collision Detection Method**

Separating Axis Theorem - [Reference](https://gamedevelopment.tutsplus.com/tutorials/collision-detection-using-the-separating-axis-theorem--gamedev-169)

The theorem checks if an axis can be drawn between two simple polygons.

**Code Explanation (for important functions/classes)**

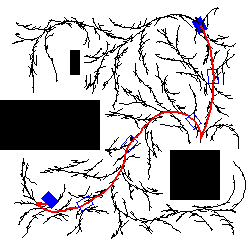
Robot class in robot.py - which is Initialized with the state, length and width of the robot. I have chosen the center of the rectangular robot to be the reference point and hence this is my state.

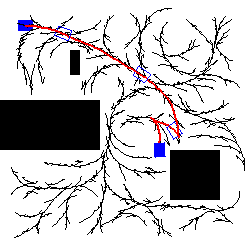
The Robot class constructs and returns a Rectangle object using the given state (center and orientation of the robot).

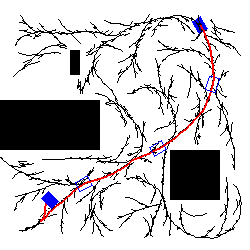
Rectangle class in robot.py – contains the code for collision detection using Separating Axis Theorem.

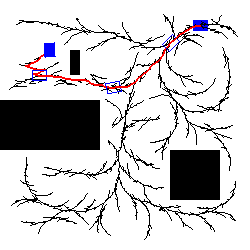
Main.py – the RRT planner and all supporting functions are implemented in this file.

Results –









**Reflection**

The planner is very successful for some combinations of final state orientation and goal state orientation while for some final state orientations, with the same iterations, it is not as successful. However, it nearly always reaches the final state given enough time. This leads me to believe that my planner would be suitable for problems like parallel parking, where orientation is critical, with some tradeoff in performance.

All the parameters listed above offer tradeoffs with regards to performance vs accuracy, and must be chosen carefully according to what the robot needs to do.

**Future work**

This approach offers O(n2)performance. The most obvious bottleneck in this approach is finding the nearest neighbor of a state in the tree. In my current implementation, I use a simple tree structure which takes O(n) time to find the nearest neighbor. A kd-tree, which uses recursive binary search, can offer O(log n) performance. I could not implement for this project, because of compiler issues regarding the recursion depth, but this could vastly improve performance, especially with when more random points are chosen.

Another part that could be improved is the distance metric. Using the Euclidean distance metric on the orientations of the robot states did not result in good performance, as the planner was not accurate enough in determining incremental states. A better distance metric with respect to orientation may be used in future work to get better accuracy and performance.

This approach may not be ideal for obstacles which are much smaller than the robot itself, and the collision detection can be extended to include line-obstacle collision detection as well.