Dubins Path Planning
Due date: Never

The purpose of this document and corresponding skeleton code is to guide you towards one possible way for using planning in your final project. Despite using words like 'assignment' and 'submission', you do not actually have to turn anything in for this document.

1 Path Planning

We will first purely focus on the planning aspect of this assignment. Instead of doing local control, we will do an efficient global search over the map. This should enable us to compute less greedy trajectories, and thereby avoid blindly going down a path that ends up being distasteful to our robot. Suggested skeleton code for this portion of the assignment can be found here. Each of the following sub-sections will describe the modules that need to be implemented. A high level summary of the algorithm that you will implement is as follows:

1. Set the current node to be the starting configuration
2. Compute feasible Dubins paths to the current node's neighbors
3. Among all nodes that are on the frontier (not just the current neighbors), update the current node to be the node that is most promising according to the sum of the cost-to-come and heuristic cost-to-go
4. Repeat steps 2 and 3 until the goal configuration is reached
5. Return the found path from start to goal

1.1 Dubins Motion Primitives

In order to generate motion primitives for our car, we will model our car using the Dubins car model (for which we can analytically solve the 2-point boundary value problem). The Dubins car model assumes that the car moves forward with a constant speed, and that the car has some non-zero minimum turning radius. Under these assumptions and given two configurations (x, y, θ) in SE(2) (refers to poses in 2D), it can be shown that the shortest path between the configurations consists of no more than three motion primitives:

1. Straight (S)
2. Hard right (R)
3. Hard left (L)

Furthermore, it can be shown that the optimal path can always be specified as one of the following combinations:

[LRL, RLR, LSL, LSR, RSL, RSR]

where, for example, LSL corresponds to turning hard-left for τ_1 seconds, then going straight for τ_2 seconds, and then again turning hard-left for τ_3 seconds. Fig. 1 illustrates RSL and RLR paths.

We can view each of these combinations as a function that computes the shortest path between two configurations using the corresponding strategy. Each time that we want to find a path between two neighbors in our graph, we ask each of these six functions to compute a path, and then choose the path that ends up being shortest. **In order to do this, implement the dubins_path_planning_from_origin() function inside of Dubins.py.** Note that outside modules should not call this function directly. In fact, the only function(s) called by modules outside of this file should be dubins_path_planning() and potentially path_length().

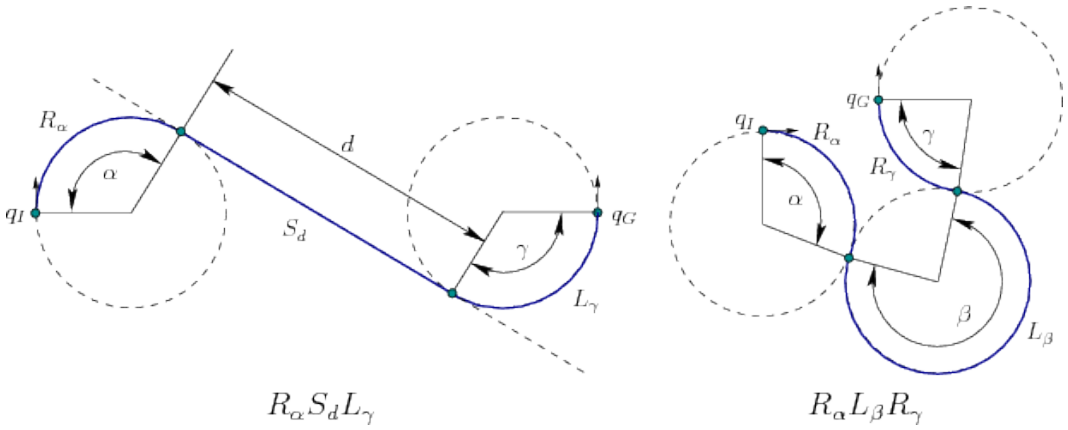

Figure 1: RSL and RLR Dubins paths. q_I and q_G, are the initial and goal configurations respectively. Image from Steven LaValle.

1.2 ObstacleManager

To ensure that the robot does not run into obstacles, such as walls, we need to use the map of the environment to check for collisions. Here, we will assume that the robot has a rectangular shape, and that if any part of our rectangular robot intersects with the wall, then it is in collision. **Implement the get_state_validity() function in ObstacleManager.py, which checks if a single configuration is in collision. Then use this function to implement get_edge_validity(), which checks if an edge between two configurations is in collision.**

1.3 Graph Generator

Instead of planning on a dense grid with a fixed discretization, we form a graph by pseudo-randomly placing points in the robot's configuration space SE(2). The specific method that we use for generating these points is based on Halton Sequences. Using Halton Sequences for generating pseudo-random values gives us a number of favorable properties in terms of the generated graph, which you can read about here. **You do not need to implement any of functions inside of GraphGenerator.py, but you will need to generate a graph (and save it into a graphml file) for Sieg Hall.** Examine the main function of GraphGenerator.py for an idea of how to do this. Note that you should try different values of the parameters (such as halton_points and disc_radius) so that you get a reasonable graph in terms of size and connected-ness.

1.4 Halton Planner and Halton Environment

We will be planning on a Halton Graph. The HaltonEnvironment module manages this graph, and provides a number of useful functions for planning in the graph. The HaltonPlanner module should use the Halton Environment in order to do A* planning over the graph. **You do not need to implement any functions inside of HaltonEnvironment.py, but you should become familiar with its functions. In HaltonPlanner.py, you will implement A* planning. In addition, you will implement a function to post-process the plan in order to try to find a shorter path.** Although A* should return the optimal path over our graph, it is possible that we could find shortcuts in the path by trying to connect nodes that were not neighbors in the original graph.

1.5 Planner Node and Planner Test

All of the code for planning is wrapped into a ROS service. As opposed to the publisher/subscriber framework that constantly sends/receives information to/from a topic, a service only operates when it receives a request from a client. In short, the server (i.e the planning node) is waiting to receive requests for a plan. When a client gives it a source and target to create a plan for, the service generates the plan, returns it, and then goes back to waiting around for more plan requests. You can read more about services and clients here. **Once you have finished implementing the above code,**

you should launch your planner service using Planner.launch. You can test that your planner is returning reasonable plans using planner_test.py, which also demonstrates how to setup a client for requesting plans from the planner service. You should also inspect srv/GetPlan.srv to understand what information is passed between the client and the server. Also note that you should be able to test all of the code up to this point without having to put it on the robot.

1.6 Submission

1. Submit visualizations of the path found for all three of the source/target pairs in planner_test.py. What is the path length of the plan found for each pair?
2. For the first of the above source/target pairs, disable post-processing of the plan. Submit the resulting visualization of the path. Are the paths any different? If so, how?

1.7 Extra Credit: Lazy A*

Implement the Lazy A* algorithm by delaying collision checks until the last possible moment. Compared to your original algorithm, does Lazy A* return paths that are typically shorter, longer or the same length? How does the runtime of Lazy A* compare to your original algorithm?

1.8 Extra Credit: Lazy SP

Implement the Lazy SP algorithm by performing search under the principle of optimism in the face of uncertainty. Compared to your original algorithm, does Lazy SP return paths that are typically shorter, longer or the same length? How does the runtime of Lazy SP compare to your original algorithm?

2 MPPI Planning

Now we will integrate our planner into MPPI. At a high-level, the algorithm should work as follows:

1. Specify a valid pose in the map (in RVIZ) for the robot to reach
2. Use a service client to get a plan from the robot's current pose to the clicked pose
3. Use the returned plan as a series of waypoints for the robot to follow using MPPI (note that you don't have to use all of the points in the plan). The robot should follow the chosen waypoints until it reaches the goal. The details of exactly how waypoints are chosen from the plan are left to you.

2.1 Submission

1. How did you choose waypoints from your plan? Why did you choose this strategy?
2. How does your robot's performance using both MPPI and planning compare to just using MPPI?
3. Describe an environment/situation in which it would be better to just do MPPI because adding planning would have a negligible effect on performance.
4. Describe an environment/situation in which combining MPPI and planning would do much better than just doing MPPI.

A big thanks to Aditya Mandalika for writing almost all of the TA's implementation of the planning code!