ECE276C Assignment: RRT Planning in Cluttered Environments

Goal

Your task is to implement a Rapidly-Exploring Random Tree (RRT) motion planner for a 3-link robot arm. The planner should find a collision-free path from a start configuration to a goal configuration while avoiding obstacles in the environment. The robot will then follow this path in simulation.

1 Problem Breakdown

Follow the steps below to complete the assignment.

1.1 Part 1: Edge Collision Checking

Complete the `check_edge_collision` function to ensure that no collisions occur between two joint positions. Note that the function `check_node_collision(robot_id, object_ids, joint_position)` is already written for you, which checks if a single robot configuration (node) is in collision.

Function Implementation:

```
"""
Checks for collision between two joint positions of a robot in PyBullet.
Args:
    robot_id (int): The ID of the robot in PyBullet.
    object_ids (list): List of IDs of the objects in PyBullet.
    joint_position_start (list): List of joint positions to start from.
    joint_position_end (list): List of joint positions to get to.
    discretization_step (float): maximum interpolation distance before a new
        collision check is performed.
Returns:
    bool: True if a collision is detected, False otherwise.
"""
```

1.2 Part 2: RRT Class

Implement the following methods in the `RRT` class to perform motion planning using the RRT algorithm.

1.2.1 `step(from_node, to_joint_angles)`

Compute a new node along the direction from `from_node` to `to_joint_angles`, limiting the distance to `self.step_size`.

Implementation:

```
def step(self, from_node, to_joint_angles):
    """Step from "from_node" to "to_joint_angles", that should
     (a) return the to_joint_angles if it is within the self.step_size or
     (b) only step so far as self.step_size, returning the new node within that
         distance"""
```

1.2.2 `get_nearest_node(random_point)`

Find the node in the tree closest to the `random_point`.

Implementation:

```
def get_nearest_node(self, random_point):
    """Find the nearest node in the tree to a given point."""
```

1.2.3 `plan()`

Implement the RRT algorithm to explore the space and find a path to the goal. The algorithm should randomly sample points in the configuration space with a 10% probability that it samples the goal point to generate an epsilon-greedy heuristic (and converge faster). The maximum step size is 0.5.

Implementation:

```
def plan(self):
    """Run the RRT algorithm to find a path of dimension Nx3. Limit the search to
       only max_iter iterations."""
```

Part 3: Run the RRT Planner

Use the `RRT` class to plan a path for all `goal_positions`. Create an animation of the output, and record the average and standard deviation of the time taken to solve the plans each 10 times.

Part 4: Optimal Planning

Implement the RRT* rewiring step such that the paths are more "optimal", in that end-effector Cartesian trajectory paths were shorter. Create a `RRT.plan2()` function to accomplish this so that you do not change the original `RRT.plan()` function. You may wish to add a cost variable to the Node class which is acceptable.

Generate an animation of the output.

Expected Deliverables

1. Completed code base
2. Animation of output