COS 495 – Autonomous Robot Navigation

Lab 8

Probabilistic Road Map (PRM) Motion Planning

INTRODUCTION

Given a robot's location in a known environment, a motion planning algorithm can be used to construct a collision-free trajectory that connects a start configuration to a goal configuration. Then, the robot can follow the trajectory to safely arrive at the goal location.

In this lab, you will implement a single-query Probabilistic Road Map (PRM) motion planning algorithm. At a high level, the algorithm generates a randomly expanding roadmap, consisting of nodes and edges to connect the start and goal configurations. More specifically, the algorithm roots the road map with a node at the start configuration, then randomly expands the road map by adding new edges and nodes until one of the new nodes has a collision-free edge connecting it to the goal configuration.

BACKGROUND

An example single-query PRM algorithm is:

```
1. Add start configuration c_start to R(N,E)
2. Loop
3.      Randomly Select New Node c to expand
4.      Randomly Generate new Node c' from c
5.      If edge e from c to c' is collision-free
6.           Add (c', e) to R
7.           If c' belongs to endgame region, return path
8.      Return if stopping criteria is met
```

The key steps are step 3, 4 and 7. To save time, Kindel et. al.'s grid cell based weighted sampling scheme has been coded for you to help in step 3. For step 5 and 7, a collision-checking algorithm has been coded for you.

HINT: For debugging the motion planner, set the estimated states to exactly equal the actual states in simulator mode (e.g. x_est = x). This way you won't deal with any residual localization problems.

EXPERIMENTS

Download the most recent version of the base code for lab 5. The main control loop `Robot :: RunControlLoop(CWiRobotSDK* m_MOTSDK_rob)` in `Robot.cpp` now contains the following functions:

```
// Localize
MotionPrediction(m_MOTSDK_rob);
LocalizeRealStateWithOdometry(m_MOTSDK_rob);
LocalizeEstStateWithParticleFilter(m_MOTSDK_rob);


// If using the point tracker, call the function
if (controllerType == CONTROLLERTYPE_POINTTRACKER)
{

    // Check if we need to create a new trajectory
    if (motionPlanRequired){
        MotionPlanner(m_MOTSDK_rob);
        motionPlanRequired = false;
    }

    // Follow the trajectory
    TrackTrajectory(m_MOTSDK_rob);

}
```

As in lab 4, the localization is done in 3 functions. We use the function `LocalizeRealStateWithOdometry` to determine the actual state `[ x y t ]` of the robot in simulations. Make sure that `x`, `y`, and `t` are set in this function.

To estimate what this actual position is, we use a particle filter localization algorithm with known start location. This will be implemented in the function `LocalizeEstStateWithParticleFilter`. Make sure that you set `x_est`, `y_est`, and `t_est` in this function.

Make sure to remember your wall distance checking code from `Map.cpp`. Note that your trajectory tracking code isn't required but your point tracking code is required. **Copy all code for these functions from the previous lab 4 to your new lab 5 code. **

When a user presses the *Track Point* button, the controller type is set to `CONTROLLERTYPE_POINTTRACKER` and sets the `motionPlanRequired` flag to `true`. The main control loop will then call the `MotionPlanner` function once and set the `motionPlanRequired` flag to `false`.

Once a trajectory is constructed by the `MotionPlanner` function, the `TrackTrajectory` function will set the desired points to be tracked by the point tracker to be those nodes of the newly constructed trajectory.

**Use all code for point tracking from the previous labs to your new lab 8 code.

For this lab, the code you will modify is located in `Robot.cpp`:

1. **Create the start and goal nodes**
    Using the constructor `Node(double x, double y, int nodeIndex, int lastNode)`, create start and goal nodes in the `MotionPlanner` function. The position of the nodes should be (`x_est`, `y_est`) and (`x_goal`, `y_goal`) respectively. Set the `nodeIndex` and `lastNode` values to 0. These are used later when constructing the trajectory from the PRM.

    Use the `AddNode` function to add the start node to the PRM.

2. **Random Node Selection**
    A while loop has been created for you that iterates over possible node expansions. This loop will terminate if the maximum number of iterations is exceeded, or a path was found, (i.e. the PRM successfully connected to the goal node).

    Within the while loop, the first step is the random selection of a node to expand from. First, randomly select a cell from all those occupied cells. Use the variable `numOccupiedCells`, and pick an integer `randCellNumber` between 0 and `numOccupiedCells`.

    Next, randomly pick a node within the cell. Use the variable `numNodesInCell`, and pick an integer `randNodeNumber` between 0 and `numNodesInCell[occupiedCellsList[randCellNumber]]`.

    The node `NodesInCells[occupiedCellsList[randCellNumber]][randNodeNumber]` is your randomly selected node. Name it `randExpansionNode`. This is the node you can expand from!

3. **Node Expansion**
    For the node expansion, we will use straight line segments. That is, all edges in the PRM will be straight, thereby ignoring and dynamic or kinematic constraints.

    First, randomly select a distance and orientation. You can play with the ranges of these random numbers later and see how they affect planner performance.
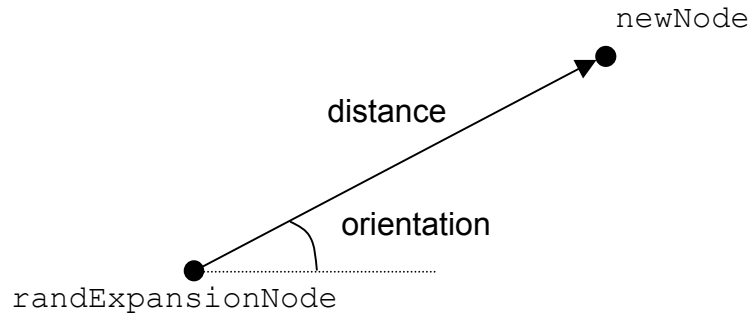
Figure 1: Random expansion to create a new node

Use the distance and orientation, along with the position of the parent node (`randExpansionNode.x, randExpansionNode.y`) to determine the location of the new node `newX` and `newY`. Using the `Node` constructor, create the `newNode` with this position. Set the `nodeIndex` to be the `numNodes`, and the `lastNode` to be the `nodeIndex` of the parent node.

4. Add new node to PRM
Create an `if` statement that calls `robotMap.CollisionFound(Node n1, Node n2, double tol)` to determine if the edge connecting the `newNode` to its parent is collision-free. The variable `tol` is short for tolerance, the allowable distance between the center of the robot and the wall, (perhaps `robotWidth/2`).

If no collision was found, add the `newNode` to the PRM using `AddNode()`.

5. Check for connection to goal
After adding the `newNode` to the PRM, check if the new node connects to the goal node using the collision checker again. If there is no collision between `newNode` and `goalNode`, then set the `goalNode.nodeIndex` to `numNodes` and set the `goalNode.lastNode` to be the parent's `nodeIndex`. Finally, add the `goalNode` to the PRM and set the `pathFound` flag to be `true`. Setting this flag will terminate the while loop.

REMOVE the forward slashes "//" to uncomment the `BuildTraj(goalNode);` call!

6. Optimize Trajectory Tracker (Optional)

At this point, test your planner on many start/goal locations. Enter a desired goal location in the *Point Tracker* text boxes, and then click *Track Point*. You should see the trajectory connect your estimated position with the goal point on the screen. If the screen fills up with red lines, your planner didn't find a solution (maybe your goal destination is on a wall!) Once the trajectory is constructed, the robot will track each node in the trajectory using your Trackpoint function.

Feel free to modify the planner for different orientation tracking schemes or ways to follow edges. For example, one could implement a turn on the spot scheme at every node so that the robot always faces towards the next node before following the edge that connects the current and next nodes. Or, port your traj tracking code from the previous lab 3.

Also, feel free to make several plans and pick the shortest plan before following it. Another good idea is to check if the startnode directly connects with the goalnode.

DELIVERABLES

1. DEMO!!!

You must demo your working code to the instructor or TA by noon on Friday Dec. 16.