Motion Planning

Michelle Shu

February 4, 2014

1 Introduction

In this report, I will explore two motion planning algorithms used for search problems on continuous state spaces with obstacles. The Probabilistic Roadmap method

2 Arm Robot Planning with Probabilistic Roadmap

2.1 Arm Robot State Representation

In this section, we will be planning a sequence of actions for a planar robot arm to move between two configurations while avoiding obstacles in its environment. We will consider the robotic arm as a set of rigid segments (links) of fixed and equal length, joined by joints that allow for rotation in a single plane only. We also assume that the base of the arm is fixed at a specific location (x_0, y_0). The state of an arm robot with l links is thus fully represented by a sequence of angles $(\theta_1, \theta_2, ..., \theta_l)$, where θ_m describes the angle between link m and link $m-1$ for $2 \leq m \leq l$ and the angle between link 1 and the x-axis if $m = 1$.

2.2 Probabilistic Roadmaps

A probabilistic roadmap (PRM) is a model that can be generated to subsequently handle multiple queries. When we want to find paths between multiple start-goal pairs, it is logical to put more effort into a pre-processing phase that will enable future queries to be handled more efficiently. The basic idea of the PRM method is to build a topological roadmap of the configuration space in a general way and then use it for specific problems, Hence, we can separate the algorithm into two phases of computation:

1. **Preprocessing Phase** Build a graph by connecting randomly generated (valid) configurations to a subset of their nearest neighbors.
2. **Query Phase** Given a start and goal configuration, connect them to the graph and perform a search of PRM to find a path between them. (Repeat.)

2.3 Preprocessing Step: Generating the PRM

The PRM for the arm robot's motion planner is generated on a configuration space by the procedure `buildPRM` in `ArmPRM.java`. There are two main steps to this process:

1. Create graph vertices by randomly generate a set of arm configurations that are valid (do not collide with obstacles in the world).
2. Sequentially add edges between vertices and those of their k nearest neighbors that are not already in the same connected component.

Here is the top level function `buildPRM`:

```java
public static HashMap<ArmRobot, ArrayList<ArmRobot>> buildPRM(World w) {
  // The graph G is represented as an adjacency list
  HashMap<ArmRobot, ArrayList<ArmRobot>> G =
      new HashMap<ArmRobot, ArrayList<ArmRobot>>();

  // Get random sequence of non-collision configuration states to try
  ArrayList<ArmRobot> randomStates = getRandomConfigs(NUM_ARM_LINKS, w);

  // Add random states to graph
  for (ArmRobot state : randomStates) {
    G.put(state, new ArrayList<ArmRobot>());
  }

  for (ArmRobot state : G.keySet()) {
    PriorityQueue<ArmRobot> KNN = getNearestNeighbors(state, G.keySet());
    for (int i = 0; i < KNN.size(); i++) {
      ArmRobot neighbor = KNN.poll();
      if ((! w.armCollisionPath(new ArmRobot(NUM_ARM_LINKS),
        state.getConfig(), neighbor.getConfig())) &&
        (! inSameComponent(G, state, neighbor))) {
        // If these configs can be connected and are not
        // already connected, connect them.
        G.get(state).add(neighbor);
        G.get(neighbor).add(state);
      }
    }
  }

  return G;
}
```

First, we randomly generate a sequence of points in configuration space This is done in

```java
public static ArrayList<ArmRobot> getRandomConfigs(int links, World w) {
  // Store configurations used to ensure no duplicates
  HashSet<ArmRobot> configsAdded = new HashSet<ArmRobot>();
  Random rand = new Random();
  ArrayList<ArmRobot> randConfigs = new ArrayList<ArmRobot>(NUM_RAND_SAMPLES);

  for (int c = 0; c < NUM_RAND_SAMPLES; c++) {
    ArmRobot newRobot = new ArmRobot(links);
    for (int l = 0; l < links; l++) {
      newRobot.setLinkAngle(l, rand.nextDouble() * (2 * Math.PI));
    }

    if (configsAdded.contains(newRobot) || (w.armCollision(newRobot))) {
      c--; // don't count this one
    } else {
```

```
            randConfigs.add(newRobot);
            configsAdded.add(newRobot);
        }
    }
    return randConfigs;
}
```

2.4 Finding the most efficient direction of rotation

When moving any link of the arm around the joint, we can choose to rotate either in the clockwise or counterclockwise direction. As a result of the choice, the angle of each link will be displaced either θ or $2\pi - \theta$ radians. For the purposes of this search, I explore only the most efficient option for each joint (moving through the smaller angle, so that an arm link angle never changes by more than π radians in one move).

Building this informed choice of direction into the search will make it more flexible and therefore more effective.

3 References