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 preprocessing 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 Phase: 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`:

```
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;
}
```

In the first step, the helper function `getRandomConfigs` is used to generate a set of random configurations. Each configuration in the set is a tuple of random angle sizes. Before adding a configuration, I check that it is valid with the collision detector function `armCollision`, which will be discussed later in this section.

```
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;
}
```

In the next step, I iterate through all the randomly sampled configurations. For each configuration a, I find a's k nearest neighbors among the other samples. I accomplish this in $O(nlogk)$ time and using $O(k)$ additional space by using a max-heap data structure (Java's PriorityQueue) and the following algorithm:

- Insert any of the k configurations in the graph to the heap. The root of the heap is now the maximum of the k elements inserted.
- For all other configurations in the graph, compare to the current root of the heap. If the current configuration is closer to a than the heap's root, replace the root with it.
- Return all heap entries. These are the k nearest neighbor's of a.

Here is my implementation of k-nearest neighbors (also used in `buildPRM`) in the function `getNearestNeighbors`:

```
public static PriorityQueue<ArmRobot> getNearestNeighbors(ArmRobot a,
    Set<ArmRobot> neighbors) {
  PriorityQueue<ArmRobot> kNearest = new PriorityQueue<ArmRobot>(
      new KNNComparator(a));

  // Start priority queue by adding first K nodes in graph
  Iterator<ArmRobot> iter = neighbors.iterator();
  for (int i = 0; i < K; i++) {
    ArmRobot n = iter.next();
    if (! n.equals(a)) {
      kNearest.add(n);
    } else { i--; }
  }
  // For remaining nodes, add only if they are possibly one of the K
  // shortest distances to the source node. (i.e. < max of our heap)
  while (iter.hasNext()) {
    ArmRobot next = iter.next();
    if (distance(a, next) < distance(a, kNearest.peek())) {
      kNearest.poll();
      kNearest.add(next);
    }
  }
  return kNearest;
}
```

`getNearestNeighbors` uses a `KNNComparator` to order configurations in the heap on the basis of their distance from source configuration (a):

```
public static class KNNComparator implements Comparator<ArmRobot> {
  public ArmRobot source;

  public KNNComparator(ArmRobot s) { // compare by distance to source robot
    this.source = s;
  }

  // Prioritize robot with larger distance to source for max heap
  // So return negative number if dist(a, s) > dist(b, s).
  public int compare(ArmRobot a, ArmRobot b) {
    double diff = distance(b, this.source) - distance(a, this.source);
    if (diff < 0) return -1;
    else if (diff == 0) return 0;
    else return 1;
  }
}
```

Let the nearest neighbors of configuration a be $c_1, c_2, ..., c_k$. Then, after finding $c_1, c_2, ..., c_k$ via the k-nearest neighbors algorithm, I attempt to connect a to each of its neighbors by adding edge (a, c_i) to the PRM, which represents an arm robot's movement between a and c_i in physical space. This edge is only added if two conditions are fulfilled: (1) a and c_i are not already in the same connected component and (2) I find a local path from a to c_i that avoids collisions with all obstacles in the configuration space. The first condition is checked by a simple depth-first search that finds whether or not c_i is reachable from a in the current PRM. For the second condition, a local planner is used to generate a path (linear in configuration space) from a and c_i and check intermediate states along the path for collisions.

I will now proceed to discuss the choice of direction in which to move the arm robot as it transitions between states and then explain how the local planner is used to check for a collision-free path between two adjacent vertices in the PRM.

2.3.1 Choosing 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 $min(\theta, 2\pi - \theta)$, 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 thereby more accurate. To see why, consider the alternative: If we arbitrarily select a direction of rotation, we risk selecting the larger angle on roughly half of the trials. This means we will choose a longer path and traverse a larger portion of the configuration space, which means a higher probability of collisions.

2.4 Query Phase:

2.5 Results

(1.7009102896360668, 3.6707274655809954, 2.6300602749681725, 3.4162156691330847)
(1.4744951187473923, 4.473464160943738, 2.658124479247245, 1.7982481607093652)
(1.5762953748817836, 4.501727980052681, 2.9239471648095505, 0.7669045445348482)
(1.9463712956055843, 5.370673739912323, 3.809665481993138, 5.936886883537734)
(1.5447266785586244, 0.5544523113231762, 3.555028420847484, 5.797065856470821)

(1.5094313371146866, 6.238824453727329, 3.059431764508278, 0.8264163462037615)
(1.727301607335237, 0.0828188127134834, 2.420004991404254, 1.8499480318200519)
(1.1422410168934505, 0.9809766363890837, 1.2262685353929836, 2.3625457125192795)
(1.8058969494686439, 0.8925137240409543, 6.157838791898388, 1.7565930315374954)
(1.001162816273552, 1.2032099720783773, 0.5218715654199031, 2.823140326152687)
(1.355000116455186, 1.6103832163355014, 5.374089774809896, 2.3735140498250717)
(1.4489780627361928, 0.9957763492947765, 0.7114846759950995, 3.148473850997469)
(1.9698460022257698, 0.5814551835918683, 1.4926171421957293, 4.329163209139811)
(1.6637025120012237, 0.21979247748289016, 1.506779558348471, 5.8883639154476155)

2.5.1 Effect of Parameter Choice

3 Car Robot Planning with Rapidly-Exploring Random Tree

4 References