Motion Planning

Lexin Tang

October 13, 2016

1 Introduction

In this report, I will introduce solutions of motion planing problem for the arm robot and the mobile robot. Probabilistic Roadmap, A star search algorithm and Rapidly Exploring Random Tree are used in solving this kind of motion planing problem.

2 The Arm Robot

2.1 Implementation of Problem Model

The arm robot problem is a search problem but we need to construct the graph for searching at first. Once we generate enough random samples from PRMplanner and we believe they have covered most of the configuration space, we start to search a path from start node to goal node just like what we do in our previous assignment. The problem model is implemented in `ArmRobotProblem.java`, which is an extended class from `UUSearchProblem`. The configuration state of `ArmRobotNode` is an array containing all the angles of arm joints. The heuristic used in A start search is Euclidean distance of two states.

$$h(state) = \sqrt{\sum_{i=1}^{k}(\theta_i - goal_i)^2}$$

2.2 Probabilistic Roadmap (PRM)

PRM is implemented in `PRMplanner`. The procedure of PRM can be described in the following steps:
1. Generate a set of random samples in collision-free space and add them to the graph, which is a HashMap;
2. Generate a new sample, and find the k nearest neighbors of it in the graph.
3. Check the path between the newly generated node and its neighbor. If there is no collision, connect them;
4. Loop from step 2 to step 3 until we get enough graph nodes;

`HashMap<GraphNode, ArrayList<GraphNode>>` is used as the data structure for generated roadmap. There are several important functions implemented in `PRMplanner.java`.
`localPlanner` is responsible for checking collisions and connecting two GraphNodes if they are closed to each other.

```
1  private void localPlanner(GraphNode newNode){
2    ArrayList<GraphNode> nodeLists = new ArrayList<GraphNode>();
3    for (GraphNode key : graph.keySet()){
4      nodeLists.add(key);
5    }
6
```

```
    // sort the nodes in graph based on their distance towards newNode
    Collections.sort(nodeLists, new Comparator<GraphNode>(){
      public int compare(GraphNode a, GraphNode b){
        int dist_a = 0;
        int dist_b = 0;
        for (int i = 0; i < dim; i++){

          dist_a += (a.state[i] - newNode.state[i]) * (a.state[i] - newNode.state[i]);  //
    use Euclidean distance
          dist_b += (b.state[i] - newNode.state[i]) * (b.state[i] - newNode.state[i]);
        }
        return dist_a - dist_b;
      }
    });

    // find K-NN
    ArrayList<GraphNode> neighbors = new ArrayList<GraphNode>();
    for (int i = 0; i < k; i++){
      // check along the path to ensure no collision between two nodes
      if (isSafeAlongPath(newNode, nodeLists.get(i), 50)){
        System.out.println("connect two nodes: " + nodeLists.get(i) + " and " + newNode);
        connectTwoNodes(newNode, nodeLists.get(i));
        neighbors.add(nodeLists.get(i));
      }
    }

    graph.put(newNode, neighbors);
  }
```

`isSafeAlongPath` returns true if there is no collision along the path between two nodes, which are closed to each other.

```
  private boolean isSafeAlongPath(GraphNode a, GraphNode b, int sampleRate){
    GraphNode start, end;
    if (a.compareTo(b) < 0){
      start = new GraphNode(a.state);
      end = new GraphNode(b.state);
    }
    else{
      start = new GraphNode(b.state);
      end = new GraphNode(a.state);
    }

    double[] gap = new double[dim];
    for (int j = 0; j < dim; j++){
      gap[j] = (double)(end.state[j] - start.state[j]);
    }

    for (int i = 0; i < sampleRate; i++){
      GraphNode tempNode = new GraphNode(start.state);
      for (int j = 0; j < dim; j++){
        tempNode.state[j] +=  (int)(gap[j] * i / (sampleRate - 1));
      }

      if (!isSafeState(tempNode)){
        return false;
      }
    }
    return true;
  }
```

2.3 Collision Detection

Collision detection is done by Java built-in function `Area.intersect()`.

```
private boolean testCollision(Area areaA, Area areaB) {
  areaA.intersect(areaB);
  return !areaA.isEmpty();
}
```

To enable area collision check, I implement a function `moveTo(GraphNode node)` in `ArmRobot.java`, which is used to calculate the rectangle area of arm links.

```
public void moveTo(GraphNode node){
  int x = base.x;
  int y = base.y;
  int theta = 0;
  for (int i = 0; i < dimension; i++){
    theta += node.state[i];
    x += (int)(linkLength[i] * Math.cos(Math.toRadians(theta)));
    y += (int)(linkLength[i] * Math.sin(Math.toRadians(theta)));
    this.endpoints[i] = new Position(x, y);
  }


  Shape[] links = new Shape[dimension];
  Shape rect = new Rectangle2D.Float(base.x, base.y - linkRadius, linkLength[0],
  linkRadius * 2);
  AffineTransform at = new AffineTransform();
  at.rotate(Math.toRadians(node.state[0]), base.x, base.y + linkRadius);  // rotate is
  clockwise
  links[0] = at.createTransformedShape(rect);
  theta = node.state[0];
  linkArea = new Area(transformCoordinates(links[0]));
  for (int i = 1; i < dimension; i++){
    theta += node.state[i];
    //System.out.println("link " + i + " theta = " + theta);
    //System.out.println("endpoint " + i + ": " + endpoints[i - 1].x + " " + endpoints[i -
   1].y);
    rect = new Rectangle2D.Float(endpoints[i - 1].x, endpoints[i - 1].y - linkRadius,
  linkLength[i], linkRadius * 2); // x1, y1, x2, y2
    AffineTransform newAt = new AffineTransform();
    newAt.rotate(Math.toRadians(theta), endpoints[i - 1].x, endpoints[i - 1].y +
  linkRadius);
    links[i] = newAt.createTransformedShape(rect);
    linkArea.add(new Area(transformCoordinates(links[i])));
  }
}
```

2.4 Query phase / Testing

I generated a simple environment for testing the PRM algorithm. The results for 2R, 3R, 4R arm robots are shown below.

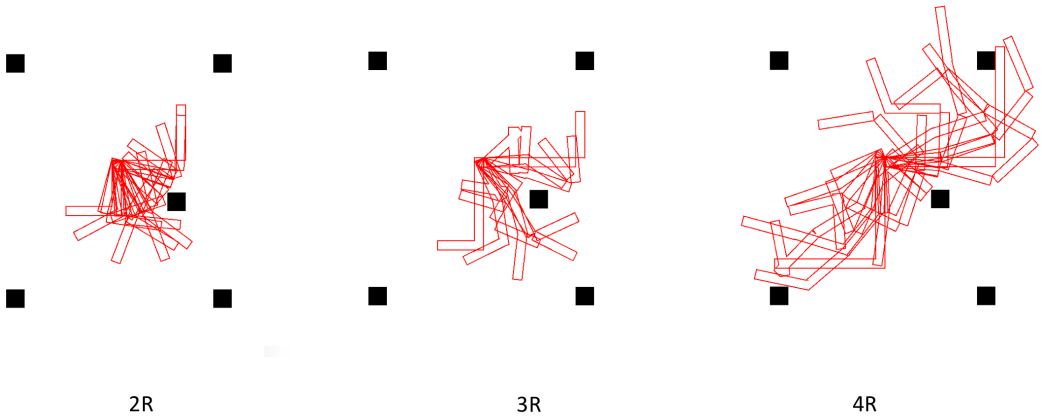

Figure 1: Examples of motion planning for the arm robot

3 Mobile Robot Problem

3.1 Implementation of Problem Model

The problem is not a traditional BFS / DFS search problem we solved in the previous assignments, so we do not need any BFS-like search algorithm for finding a path. Instead, we grow a random tree from the start node. Once the leaves of the tree reach the goal region, we backtrace from that leave and generate a path from start node to the goal in the end.

3.2 Rapidly Exploring Random Tree (RRT)

The basic idea of a Rapidly Exploring Random Tree (RRT) can be described in the following steps:
1. Grow a tree rooted at the start state;
2. Randomly generate a position in the map and find a nearest node in the tree;
3. Branching out 6 neighbors of the nearest node based on steer-car control rules and find 1 from the 6 which is to the direction of the random generated position.
4. Connect the nearest and its selected neighbor if it is collision-free.
5. Loop from step 2 to step 4 until the newly added neighbor reaches the goal.

The RRT algorithm is implemented as a class in `RRTplanner.java`. The key part of the class is the function expanding the random tree. `expandRRT()` generates 6 successors of the parent node and choose the one with minimal distance towards the ramdom generated position.

```
1   private CarNode expandRRT(CarNode parent, CarNode newNode){
2     //System.out.println("expanding RRT from " + parent);
3     double dist = Double.MAX_VALUE;
4     CarNode nearest = null;
5     // forwards, backwards, forwards turning counter-clockwise, backwards turning counter-
      clockwise, forwards turning clockwise, backwards turning clockwise
6     double[] v = {1, -1, 1, -1, 1, -1};
7     double[] w = {0, 0, 1, 1, -1, -1};
8     double time = 5;
9
10    double[] state = new double[3];
11    CarNode node = null;
12
13    // generate 6 candidates based on v & w control
14    for (int i = 0; i < 6; i++){
15      double theta = w[i] * time;
```

```
      double newX = parent.state[0] + time * v[i] * Math.cos(Math.toRadians(parent.state[2]));
      double newY = parent.state[1] + time * v[i] * Math.sin(Math.toRadians(parent.state[2]));
      double centerX = parent.state[0] - (v[i] / w[i]) * Math.sin(Math.toRadians(theta));
      double centerY = parent.state[1] + (v[i] / w[i]) * Math.cos(Math.toRadians(theta));

      if (theta == 0){  // forwards, backwards
        state[0] = newX;
        state[1] = newY;
        state[2] = parent.state[2];
      }
      else {  // turning
        double vector_origin_x = parent.state[0] - centerX;
        double vector_origin_y = parent.state[1] - centerY;
        double vector_new_x = Math.cos(Math.toRadians(theta)) * vector_origin_x - Math.sin(Math.toRadians(theta)) * vector_origin_y;
        double vector_new_y = Math.sin(Math.toRadians(theta)) * vector_origin_x + Math.cos(Math.toRadians(theta)) * vector_origin_y;
        state[0] = centerX + vector_new_x;
        state[1] = centerY + vector_new_y;
        state[2] = (parent.state[2] + theta) % 360;
      }

      node = new CarNode(state, parent);

      if (!isSafeState(node)){
        continue;
      }
      double computDist = node.eucliDistance(newNode);
      if (computDist < dist){
        dist = computDist;
        nearest = node;
      }
    }

    return nearest;
  }
```

4 Extension

4.1 Rectangle Arm Link

Since the coordinate system defined in our algorithm and Java drawing functions are different, we need to do coordinate transformation before drawing.

```
    private Shape transformCoordinates(Shape a){
      AffineTransform at = new AffineTransform();
      at.translate(250, 250);
      at.scale(1, -1);  // last in, first out order!
      return at.createTransformedShape(a);
    }
```

4.2 Purely Random Tree v.s Goal-Biased Random Tree

If we add goal node multiple times to the random generated sampel set, RRT will be more likely to expand the tree towards the goal so that the algorithm will find a path more quickly.

```
Position pos;
if (iter % 10 == 0){
  iter++;
  pos = new Position((int)(goalNode.state[0]), (int)(goalNode.state[1]));
}
else{
  pos = randomGeneratePosition();
}
```

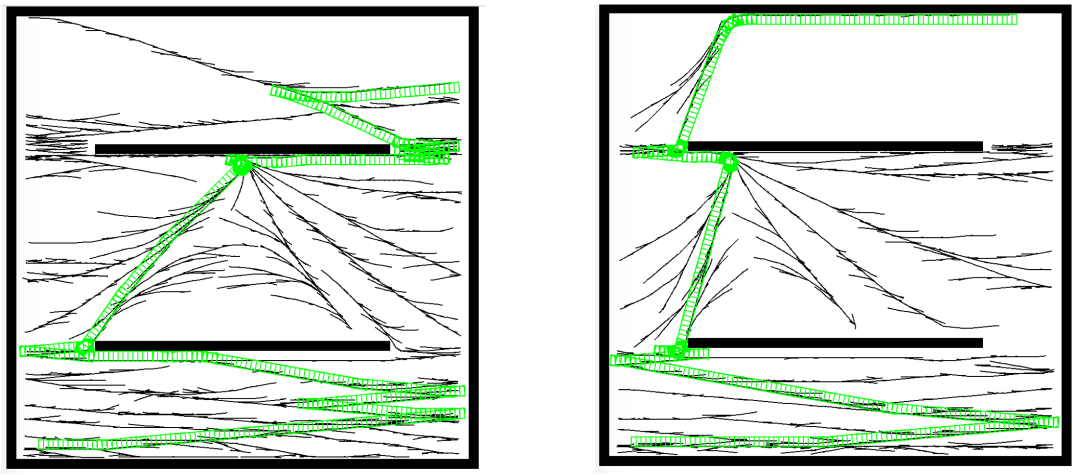

Figure 2: Left: without adding goal node to the random sample set; Right: add goal node to the set while generating random positions

4.3 Step Size in Generating RRT

Different time duration used in generating neighbors in 6 directions may affect time cost for finding a path. Here are some examples of choosing large time duration and small time duration in Figure 3 and Figure 4.

5 Previous Work

"Brandon D. Luders, Mangal Kothari and Jonathan P. How: Chance Constrained RRT for Probabilistic Robustness to Environmental Uncertainty"

The main idea is to compute the probability of collision for each newly generated tree node based on its Voronoi region. A nodes likelihood of being selected to grow the tree is proportional to its Voronoi region for a uniform sampling distribution. If the probability of collision is too high, the node will not be added to the tree; otherwise the node is kept and may be used to grow future trajectories.

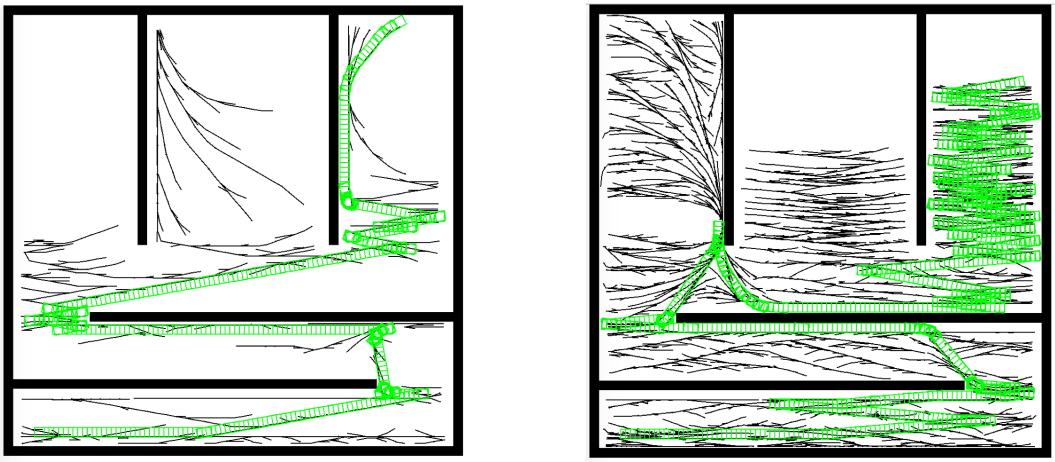

Figure 3: Left: time set to 10; Right: time set to 5

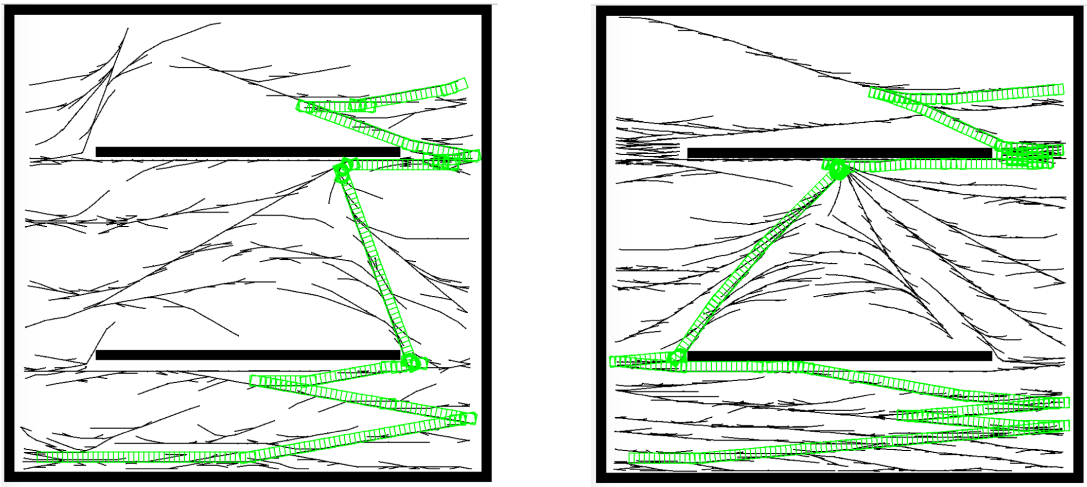

Figure 4: Left: time set to 10; Right: time set to 5