Motion Planning Solution

Kevin Farmer

October 12, 2016

1 Introduction

For this assignment, I implemented motion planners for two systems: the first is a planar robot arm anchored at the point $(0, 0)$ with n arm segments, and the second is a steered car given by a location (x, y) of the center of the car and a heading Θ. In both cases we use a motion-planning algorithm to get from some start state to some goal state, and we do this be discretizing the continuous search space. In order to do this for the robot arm, we randomly generate points within the configuration space consisting of an array of the angles of the arms. We then attempt to connect that point to its k nearest neighbors, and use the resulting graph to search the space. For the steered car, we limit its movement from a specific position to six possible movements: forward, backward, forward turning left, forward turning right, backward turning left, and backward turning right. We then use a rapidly explored random tree to seach for some path from the start to the goal.

2 Robot Arm Problem

The model is implemented in `RobotArmProblem.java`, and required a few major parts for the implementation: kinematics to compute the arm positon given a specific configuration, and a local planner which attempts to connect two nodes.

The kinematics are calculated in a method `getPoints`, which first calculates the angle of an arm relative to the space itself rather than relative to the previous segment. It then uses this to find x and y values relative to the previous point, and then adds the previous values to get the endpoint for the given arm segment.

```
1     public Point[] getPoints() {
2       Point[] points = new Point[numSegments];
3
4       double sumTheta = 0;
5       for (int i = 0; i < numSegments; i++) {
6         int x, y;
7         sumTheta += theta[i];
8
9         x = (int) (length[i] * Math.cos(sumTheta));
10        y = (int) (length[i] * Math.sin(sumTheta));
11
12        if (i != 0) {
13          x += points[i-1].x;
14          y += points[i-1].y;
15        }
16
17        points[i] = new Point(x, y);
18      }
19
20      return points;
21    }
```

Next, we use a local planner to attempt to connect two vertices together. I implemented this in `connectNode`, which goes through each arm segment in order and determines whether it would be closer to rotate the current arm segment clockwise or counterclockwise in order to reach the angle of the other arm segment. It will then rotate that segment in `RESOLUTION` increments, which is currently set to 2 degrees but could be raised in order to decrease calculation times. It will rotate every segment this way until the current configuration has reached the configuration specified by `other`. If at any point the current configuration intersects with an obstacles then we return failure and do not connect the two nodes.

```
1       public boolean connectNode(RobotArmNode other) {
2         double[] currTheta = Arrays.copyOf(theta, theta.length);
3         RobotArmNode currNode;
4
5         //Rotate in closest direction
6         for (int i = 0; i < theta.length; i++) {
7           int rotateDir;
8           double diff = other.theta[i] - theta[i];
9
10          double absDiff = Math.abs(diff);
11
12          if (absDiff < Math.PI && diff > 0)
13            rotateDir = 1; //Rotate curr counterclockwise
14          else if (absDiff < Math.PI && diff < 0)
15            rotateDir = -1; //Rotate curr clockwise
16          else if (absDiff > Math.PI && diff > 0)
17            rotateDir = -1; //Rotate counterclockwise
18          else //If absDiff > Math.PI && diff < 0
19            rotateDir = 1;
20
21          while (currTheta[i] != other.theta[i]) {
22
23            double temp = (Math.abs(other.theta[i] - currTheta[i]) % (2*Math.PI));
24
25            if (temp > RESOLUTION ) {
26              currTheta[i] += rotateDir*RESOLUTION;
27            } else {
28              currTheta[i] = other.theta[i];
29            }
30
31            currTheta[i] = currTheta[i] % (2*Math.PI);
32            currNode = new RobotArmNode(currTheta);
33            if (currNode.isIntersect()) {
34              return false;
35            }
36          }
37        }
38
39        return true;
40      }
```

For the collision detection itself, this is done using the Graphics2D library. I use `Rectangle` objects as the obstacles, and create `Line2D` objects to represent the arm, and there is a built-in method `Rectangle.intersectsLine(Line2D)` that returns whether some pair intersects each other. Therefore, it is just a simple matter of looping through all of the obstacles and arm segments and checking for any intersections between each.

3 k-PRM

The PRM consists of a roadmap generation phase, and a query phase. In order to generate the graph, we need a sampling method which creates points within the configuration space, and we must attempt to connect a node to its k nearest neighbors.

We must sample the configuration space in order to create nodes for the graph. This is done in `genRobotGraph`, which creates an array of random `double`s between 0 and 2π, checks if this node is a repeat within the graph and whether it is intersecting one of the obstacles, and if not then adds it to the graph.

```
1     //Generates a graph with no edges
2     private RobotArmGraph genRobotGraph() {
3       RobotArmGraph graph = new RobotArmGraph();
4
5       //Randomly generate totalSamples points in configuration space
6       int samp = 0;
7       while (samp < totalSamples) {
8
9         double[] randTheta = new double[numSegments];
10        double x;
11        Random r = new Random();
12
13        for (int i = 0; i < numSegments; i++) {
14          x = (2*Math.PI) * r.nextDouble();
15          randTheta[i] = x;
16        }
17
18        RobotArmNode randNode = new RobotArmNode(randTheta);
19
20        if (!graph.containsNode(randNode) && !randNode.isIntersect()){
21          graph.addNode(randNode);
22          samp++;
23        }
24      }
25
26      return graph;
27    }
```

Now we attempt to create edges between a given node and it's k nearest neighbors. We do this in `getNeighbors` by adding every node in the graph to a list except the current node, and then sorting that list by the distance to the current node which is given by the sum of the differences in the angles. We then loop through the first k nodes in the list and use the local planner to check whether the current node can connect to that node. If so, then we add that edge to the graph.

```
1     //Adds neighbors to graph
2   private void getNeighbors(RobotArmNode node, RobotArmGraph graph, int k) {
3     List<RobotArmNode> successors = new ArrayList<RobotArmNode>();
4
5       for (RobotArmNode key : graph.getKeySet()) {
6         if (!node.equals(key)) //Don't link to self
7           successors.add(key);
8       }
9
10      // Sorting
11      Collections.sort(successors, new Comparator<RobotArmNode>() {
12              @Override
13              public int compare(RobotArmNode node1, RobotArmNode node2) {
14                  return  node.getDist(node1).compareTo(node.getDist(node2));
15              }
16          });
17
```

```
    int i = 0;
    while (i < k) {
      //Add edge if needed
      RobotArmNode adjNode = successors.get(i);
      if (node.connectNode(adjNode)) {
        graph.addEdge(node, adjNode);
      }
      i++;
    }
  }
```

In order to actually query a path from a start to a goal node, we merely add both nodes to the graph and run `getNeighbors` in order to connect both to the graph, and then run a graph search algorithm from the start to finish. I implemented this first with breadth-first search for simplicity and the results are similar to what one might expect, however this does not actually give a shortest path within the graph because it does not take into account edge weights. I then attempted to implement A* search as well, which I have mostly done but ran into issues when having `RobotArmNode` implement Comparable. Some of the library functions used as part of the Graph would call Comparable if I overrode it, but there is no way to know what the distances are for the nodes until actually running the search algorithm. I did not have time to implement the queue in another way, otherwise I would be using A* search in the PRM rather than breadth-first search.

4 Robot Car Problem

We now look at the steered robotic car problem, in which we want a car to drive from some starting position to some ending position. We do this by limiting it to six possible movements from any given position. In order to implement this in `RobotCarProblem.java`, I have a method `rotatePos(int v, int w)` which takes a forward and angluar velocity and calculates the resulting position after this movement.

```
    //Returns a node for the successor postion based on v and w
    private RobotCarNode rotatePos(int v, int w) {

      //Not turning
      if (w == 0) {
        int newX = (int) (x+(v* Math.cos(theta)));
        int newY = (int) (y+(v* Math.sin(theta)));

        RobotCarNode newNode = new RobotCarNode(newX, newY, theta);
        if (!newNode.isIntersect()) {
          //Doing graphics stuff here
          return newNode;
        } else {
          return null;
        }
      }

      double rotateAngle = Math.PI/4; //The amount to rotate around the arc
      double rotateDir;   //The direction that the point of rotation is in
      double rotateTheta; //The angle to rotate around the point by

      //determine direction of the rotation point
      if (Math.signum(w) == 1) { //To the left of current heading
        rotateDir = (theta + Math.PI/2) % (2*Math.PI);
      } else {         //To the right of current heading
        rotateDir = (theta + Math.PI*3/2) % (2*Math.PI);
      }

      //Determine angle to rotate by
      if (Math.signum(w) != Math.signum(v)) {
        rotateTheta = (2*Math.PI - rotateAngle) % (2*Math.PI);
```

```
32        } else {
33          rotateTheta = rotateAngle;
34        }
35
36        double newTheta = (theta + rotateTheta) % (2*Math.PI);
37
38        double l = Math.abs(v/w);
39        int rotateX = (int) (x+ (l*Math.cos(rotateDir))); //Point to rotate around
40        int rotateY = (int) (y+ (l*Math.sin(rotateDir)));
41        int relX = x - rotateX; //Coords relative to rotation point
42        int relY = y - rotateY;
43
44        //New coords relative to rotation point
45        int newRelX = (int) (relX*Math.cos(rotateTheta) - relY*Math.sin(rotateTheta));
46        int newRelY = (int) (relY*Math.cos(rotateTheta) + relX*Math.sin(rotateTheta));
47
48        int newX = rotateX + newRelX;
49        int newY = rotateY + newRelY;
50
51        RobotCarNode newNode = new RobotCarNode(newX, newY, newTheta);
52        if (!newNode.isIntersect()) {
53          //Doing graphics stuff here
54          return newNode;
55        } else {
56          return null;
57        }
58
59      }
```

The methods `getSuccessors` merely calls `rotatePos` with the six different possible configurations of v and w, and returns a list of the results excluding any that collide with any of the obstacles. Getting the successors of the current position is the primary challenge of implementing the model. The collision checking is again done with the `Graphics2D` library. When drawing the tree created, I ended up using straight lines even for movements which involve turning for simplicities sake, and the end result still looks reasonable.

As part of the implementation of the RRT, we also need to randomly sample nodes while doing the exploration. So, I created a method `getRandomSample` which randomly creates a point within the coordinates $-dim/2$ and $dim/2$ and returns it.

```
1   protected RRTNode getRandomSample() {
2     Random r = new Random();
3     int x = r.nextInt(panelDim/2);
4     int y = r.nextInt(panelDim/2);
5
6     if (0 == r.nextInt(2))
7       x *= -1;
8
9     if (0 == r.nextInt(2))
10      y*= -1;
11
12    return new RobotCarNode(x, y, 0);
13  }
```

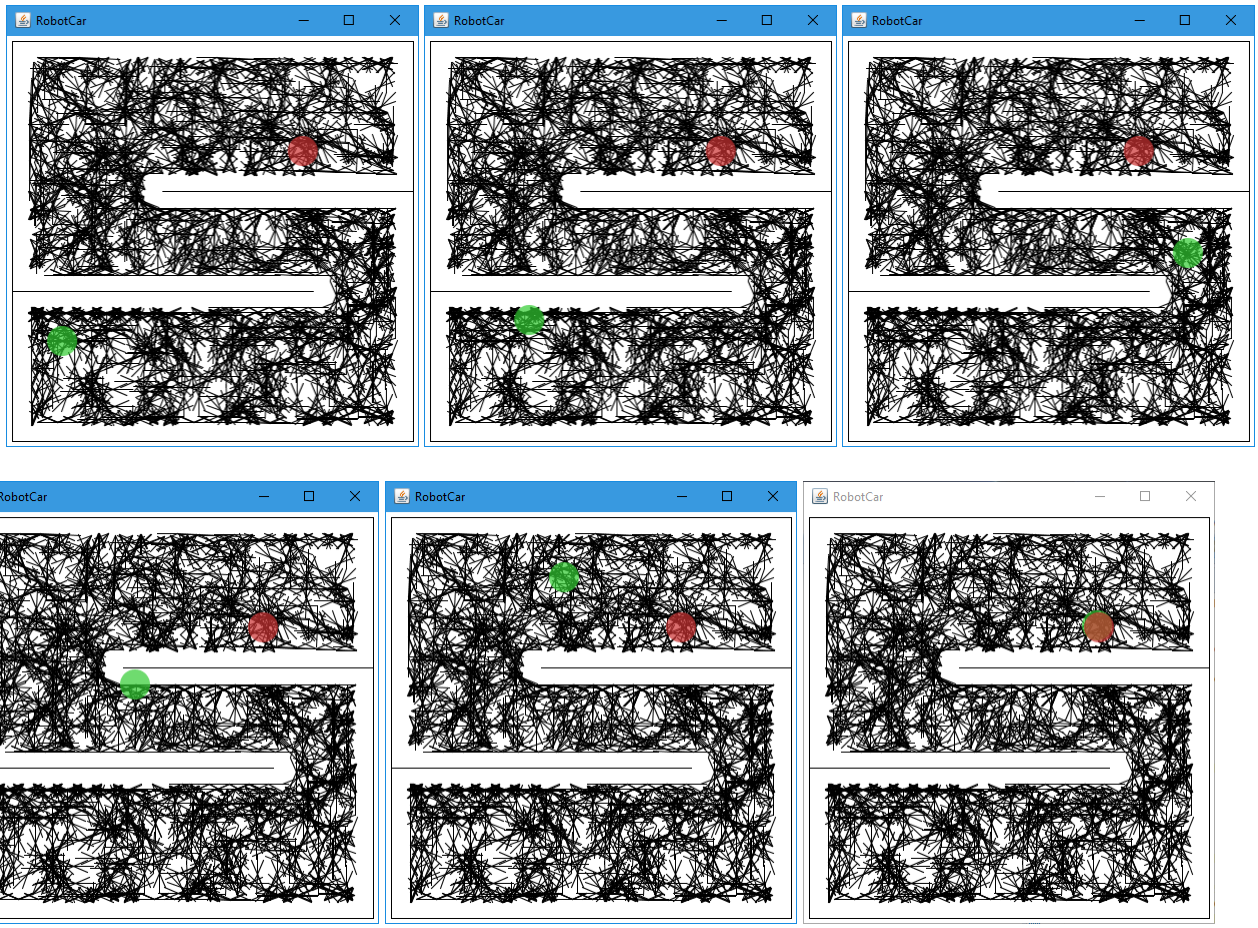
5 RRT

The RRT itself is implemented in `RRTProblem.java` and is actually fairly simple to implement. I implemented it in the method `rrtExploration` which keeps track of the tree used in the exploration and also the set of nodes that are leaves on the tree.

```
1  //Explores a search space using a Rapidly Exploring Random Tree
2  public List<RRTNode> rrtExploration() {
3    Tree<RRTNode> rootTree = new Tree<RRTNode>(startNode);
4    List<Tree<RRTNode>> leaves = new ArrayList<Tree<RRTNode>>();
5
6    for (RRTNode n : startNode.getSuccessors()) {
7      Tree<RRTNode> child = rootTree.addLeaf(n);
8      leaves.add(child);
9    }
10
11   //Continually loop
12   while (leaves.size() > 0) {
13     RRTNode rand = getRandomSample();
14
15     //Find nearest leaf to rand
16     Tree<RRTNode> leafTree = rand.getNearestLeaf(leaves);
17
18     //Expand that leaf
19     for (RRTNode n : (leafTree.getHead()).getSuccessors()) {
20       if (n.goalTest()) {
21         Tree<RRTNode> child = leafTree.addLeaf(n);
22         leaves.add(child);
23         return backtrack(rootTree, child);
24       }
25
26       Tree<RRTNode> child = leafTree.addLeaf(n);
27       leaves.add(child);
28     }
29     leaves.remove(leafTree);
30   }
31
32   return null;
33 }
```

The algorithm generates some random node within the space, and then determines which leaf node is the closest to that node. We then expand that node, and for each successors check if we have reached the goal. If not then we merely add this node as a new leaf to the tree and continue. After finishing expanding that node, we remove it from the set of leaves. This process continues indefinitely until eventually the goal is found. This does assume that the goal is reachable from the start position, otherwise the algorithm will endlessly loop. This could be eliminated by checking whether a given node is already in the tree, but it will be rapidly apparent if the goal is not reachable from the start and so the user can end the program on their own anyway.

Below you can see images of the car (green) moving towards the goal (red).


RobotCar
RobotCar
RobotCar
RobotCar
RobotCar
RobotCar