Robot Motion Planning

Niranjan Ramanand

September 2016

1 Introduction

In this report, we explore two different probabilistic approaches to graph construction: Probabilistic Roadmaps (PRMs) and Rapidly Explored Random Trees (RRTs). We use the former to guide a robot arm to a desired configuration, and we use the second to guide a moving robot to a desired location in the cartesian plane. There are several constraints in both cases. In general probabilistic algorithms will not find definitive results. That is, there is not usually a guarantee that, after a certain amount of finite time, a path is found.

2 Setup

Both algorithms have their own simulators, which must be run. Within each simulator, there are different things one must set up (initial/final configurations, link lengths, etc). For the actual object in question (arm/disk robot), there are Arm and MobileRobot classes respectively. Within each is paint and collision handling. To try different configurations, look no further than init() in the Simulator and MobileSimulator classes for what to change. The two techniques also have their own class (similar to how we had MultiRobotProblem and BlindRobotProblem before). To change the properties of the car, omega, phi, and v values are local variables of the RRT class.

3 Algorithms

3.1 Probabilistic Roadmaps

In essence, a PRM works by randomly picking points in the configuration space and storing them. Then, by navigating back through them, we can find the k nearest neighbors of each point. This is complicated by the fact that we must check that the linear route between the two points in configuration space is free (no obstacles). One way to do this, as is done in this implementation, is to take different interval steps along the line from one point to another. We then check if each point is in collision with an obstacle or not. Of course, this naturally means that smaller step sizes are both more correct and more expensive. Furthermore, it is possible to come up with a clever algorithm to find the k nearest neighbors in logarithmic/constant amortized time, but this was not done here.

An interesting question was raised on Piazza and might be worth addressing here. Why exactly does the probabilistic way PRM works triumph over a more deterministic way of doing it? Does is it even? After all, we can imagine that we construct a lattice that covers all the open configuration space and then we move around in it. This has the benefit that we know exactly where the k nearest neighbors are (I.e. easy to identify 4 neighbors in a square grid), which would save a lot of computational time. I speculate that, as ever more points are introduced to the map, the time taken for probabilistic roadmaps to find a close-to-optimal path is lower than its deterministic counterpart. This is because adding nodes to the map takes time, and the deterministic algorithm is effectively packing the entire configuration space, even where its not necessary. Granted, the PRM also puts points in places that do not matter. However, for the deterministic way to successfully improve its shortest path, it would need to introduce points everywhere in the grid, as there is no knowledge of which points may be better than others for the path. This means that it must add a lot more points to get perhaps a slightly better path. The probabilistic roadmap will likely take less random nodes to achieve a similar optimality.

Although not implemented, PRMs can be queried multiple times after construction.

All of the nodes in the graph are contained in an ArrayList object. We further have a HashMap data structure that keys indices in this list for ArrayLists of indices in this ArrayList. This takes care of the various connections between nodes. We can thus access the edges of any node in O(1) time, similar to if we had made an actual class for the state.

The robot arm is animated. We use thread sleeping to slow down its movements, and every time the arm is painted, the Arm class will provide the next Arm to be painted, which carries it through the animation. We make use of several shape classes to check for collisions.

```

public boolean inCollision(Rectangle [] obstacles){
        int [] prevCoords;
        Line2D currLink;

        for(Rectangle obs : obstacles){
                for(int i = 0; i < cartesianCoords.length; i++){
                        prevCoords = (i == 0) ? new int [] {0,0} : cartesianCoords[i−1];
                        currLink = new Line2D.Double(prevCoords[0], prevCoords[1], cartesianCoords[i][0],


                        if(obs.intersectsLine(currLink)){
                                return true;
                        }
                }
        }
```

```
            return false;
}
```

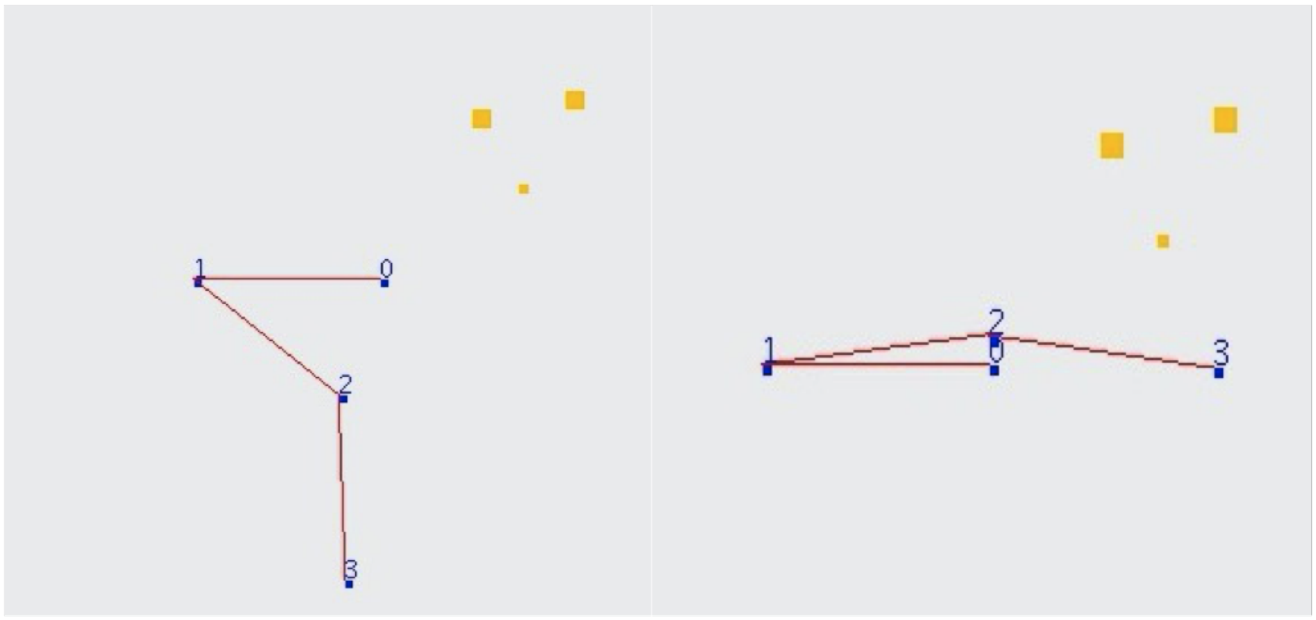
To make it easier to check for collisions with our objects that are defined in cartesian coordinates, we find the cartesian coordinates of the links of the robot arm as well.

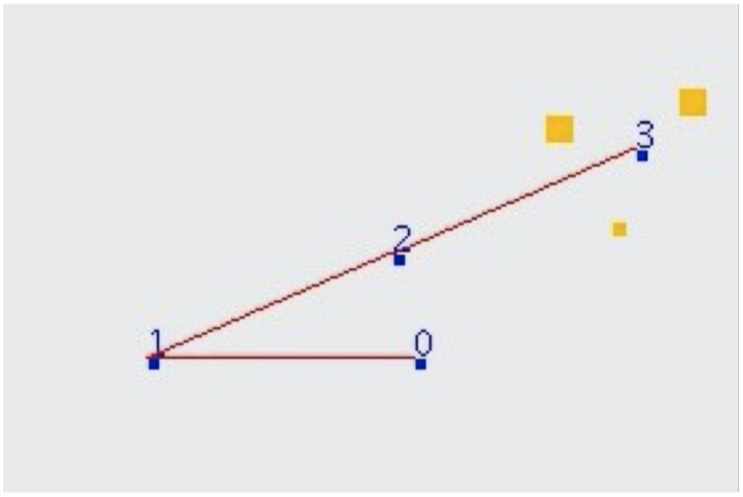
```

private int [][] getArmState(double [] thetas, int [] linkLengths){
                int [][] armCoords = new int[thetas.length][2];
                double prevAngle = 0;
                double prevX = 0, prevY = 0;

                for(int i = 0; i < thetas.length; i++){
                        prevAngle = (i == 0) ? 0 : thetas[i − 1];
                        prevX = (i == 0) ? 0 : armCoords[i − 1][0];
                        prevY = (i == 0) ? 0 : armCoords[i − 1][1];

                        armCoords[i][0] += prevX + Math.cos(thetas[i] + prevAngle) ∗ linkLengths[i];
                        armCoords[i][1] += prevY + Math.sin(thetas[i] + prevAngle) ∗ linkLengths[i];
                }
                return armCoords;
}
```





3.2 Rapidly Explored Random Trees

This is very similar in its probabilistic nature to the first. The difference, however, is that what is formed is a tree. The algorithm was implemented here by first inserting a starting root into what will form into a tree. We then sample the configuration space (like before) for a random point. Because the car has 6 ways it can move for any fixed omega, velocity, and phi (chosen here to be how much the car rotates), the way we implement the RRT is by finding the closest point in the tree and checking all 6 directions it could go in. One of these 6 points will be closest to the random point we sampled, and this we add to the tree. We further know exactly what this is connected to. The last observation may seem trivial, but it has the emergent property that when we are at a node that is near the goal, we know that there must be some path from the start to the goal because the whole thing is a tree. Notice that we need to be sufficiently close to the goal, which we determine as being lower than a certain distance squared away, as we can never be ”at” the goal in continuous space. It make sense to set this squared distance to a combination of phi, omega, and v.

There is quite a bit of overlap between this and PRMs. Both make use of the A* search we had developed some time ago. Furthermore, we use the same methods to check for connections/collisions between nodes.

There is some trigonometry involved in the actual computation of the 6 successor states, but the essential idea is that we can translate the center of the circle that the car may be traversing in its left/right turns to the origin. Once at the origin, we can use properties of the unit circle (I.e. that (x, y) points are described by (cos(Θ), sin(Θ)) to actually rotate the car by ϕ, and the proceed to translate the center of the circle back to where it originally was.

RRT’s can also be queried for different goals once constructed to some degree. This is because much of the expansion is random and so will be helpful towards finding any node. In order words, everything that incorporate into a graph that is used to find a certain node will be helpful if we decide to simply change the goal.

Further implementation details include that the way we implemented this is similar to how we did before for PRMs. Particularly, all of the nodes in the tree are contained in an ArrayList object. We further have a HashMap data structure that keys indices in this list for ArrayLists of indices in this ArrayList. This takes care of the various connections between nodes. We can thus access the edges of any node in O(1) time, similar to if we had made an actual class for the state.

```
public void populate(int n){
        RRTNode node, closest, next;
        double curr = vertices.get(0).heuristic();

        while(vertices.size() < n){
                node = randomNode(-1);
                closest = getClosest(node);
                next = random(closest);
                if(closest != null && next != null){
                        if(isConnectable(closest.index, next, 10)){
                                vertices.add(next);

                                ArrayList<Integer> list = new ArrayList<>();

                                list.add(closest.getIndex());
```

```
                                        edges.put(vertices.size() - 1, list);
                                        if(edges.containsKey(closest.index)){
                                                edges.get(closest.index).add(vertices.size() - 1);
                                        } else {
                                                ArrayList<Integer> list2 = new ArrayList<>();
                                                list2.add(vertices.size() - 1);
                                                edges.put(closest.index, list2);
                                        }

                                        if(next.heuristic() < curr){
                                                curr = next.heuristic();
                                                System.out.println(curr);
                                        }

                                        if(curr < (phi*v/omega)*phi*v/omega)
                                                n = 0;
                                }
                        }
                }

        }
```

There are several helper methods used in the method that populated the rest of the tree. First a random node is created. Then getClosest(node) will return the closest node in the tree to the random node that was picked. Finally, random(node) picks randomly one of the six positions the robot can go in. Notice here that we could have made the choice to pick the position that is the closest to the random node, but I have found that it works better not to do this, although they appear to be very similar.

The second implementation choice we make is to terminate the population once we have a node whose A* heuristic is less than a certain value. This will guarantee that the tree will be able to get sufficiently close to the goal node. If this is taken away however, we will find ever-increasingly optimal solution paths.

What follows is the search result for a population of 10 and 20 respectively. It does not seem to be a very fluid motion because of the few positions of the robot we have decided to display (more nodes require much more time to process). If we shrunk phi enough, together with a high population, we would get the very