Motion Planning

Vivek Sah

January 28 2016

1 Introduction

In this paper, we will implement motion planners for two systems: a planar robot arm (with state given by the angles of the joints), and a kinematic car (with state given by the x,y location of a point on the robot, and the angle that the robot makes with the horizontal). We will apply and implement two algorithms: Probabilistic Roadmap and Rapidly Exploring Random Tree(RRT). With Probabilistic Roadmap (PRM) , we will build a graph for configuration space and try to find a collision free path from start position to end position for a robot arm. With RRT, we will build a rapidly exploring tree to represent configuration space for a car-like robot and try to find a path from a start position to a position nearest to goal.

2 Overview of Project Structure

The main files are *MotionPlannerDriver.java*, *MotionPlanner.java, PRMPlanner.java, RRTPlanner.java.* In MotionPlannerDriver Class, we will define and setup the environment . We will also make a initialize the View and the motion planner (PRM or RRT). We will then call *solve* function of the MotionPlanner Class. The solve function will setup and build configuration space and call findpath() function which will find and return path from start to goal. PRMPlanner/RRTPlanner classes extend motionPlanner and have their own implementation of setup(), growmap() and findpath() function along with other functions. The returned path will be used to build animation.

3 PRM

The algorithm for PRM is implemented in *PRMPlanner.java* . This class extends the abstract MotionPlanner class and implements these abstract functions: *getSize, setup, reset, getSuccessors, generateFreeConfiguration, addVertex, and growMap.* Among them, *getSize, setup, reset and getSuccessors* are very straightforward. So, we will only look at implementation of *generateFreeConfiguration, addVertex,* and *growMap.*
Let's review each function closely.

generateFreeConfiguration

We basically generate a random configuration and check if the new random configuration is valid. If it is valid, we return it or we keep trying.

```
private Vector generateFreeConfiguration() {
    for (int i = 0; i < numberOfAttempts; i++) {
        Vector randomConfiguration =
            getRobot().getRandomConfiguration(getEnvironment(),this.random);
        if (this.getEnvironment().isValidConfiguration(getRobot(),
            randomConfiguration)){
             return randomConfiguration;
        }
    }
```

```
        return null;
    }
```

addVertex()

In this method, we will first check if the vertex to be added is in the graph. if it is not there already, we will get K nearest neighbor(where K is predefined to be 15). We will then check if there is a path without collision between the vertex and its neighbor. if there is such path, we will create edges between them and add them to our *roadmapGraph*.

```
    private void addVertex(Vector free) {
        if (! (roadMapGraph.keySet().contains(free))) { //check to see if
            the graph contains vertex
            List<Vector> neighbors =
                nearestKNeighbors(roadMapGraph.keySet(), free, kValue());
                //get nearest neighbors
            roadMapGraph.put(free, new HashMap<Vector, Double>()); //make
                a new entry in roadmap braph
            for (Vector neighbor : neighbors) {
                if (getEnvironment().isSteerable(getRobot(),
                    free,neighbor, this.RESOLUTION)) {
                    roadMapGraph.get(free).put(neighbor,
                        getRobot().getMetric(free, neighbor));//put an
                        edge between both vertices
                    roadMapGraph.get(neighbor).put(free,
                        getRobot().getMetric(neighbor, free));
                }
            }
        }
    }
```

growMap()

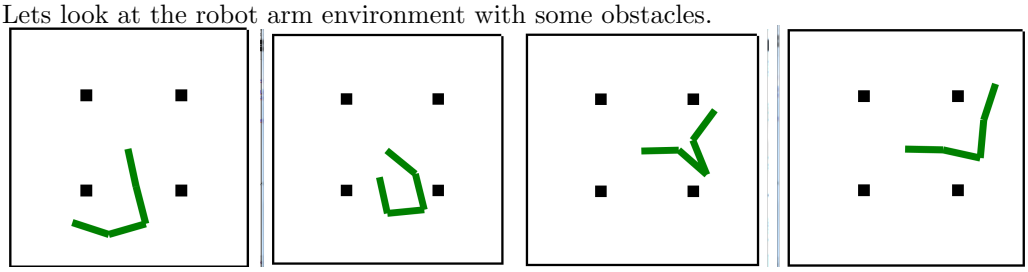
In this method, we will use our earlier function *generateRandomConfiguration* to create a random configuration and call addVertex method to add it to the graph.

```
    protected void growMap(int K) {
        for (int i = 0; i < K ; i++) {
            Vector v = generateFreeConfiguration();
            if (v != null){
                addVertex(v);

            }
        }
    }
```

Testing for PRM

Lets look at the robot arm environment with some obstacles.



It took 69.278000 seconds to growing 1000 nodes , 14.434597 nodes per second. The error was 0.

4 Variation for PRM

: Benchmark results for basic setup:
Error: 0.000000 Sample rate: NaN Length: 6.581595 Collision Detection: 85914
INFO: Growing 100 nodes takes 2.176000 seconds. 45.955882 nodes per second.

Variation 1: **Instead of getting K neighbors, get 2*K neighbors and add the first K nodes(if possible).**

Results: The length is similar, but the speed is slower and collision detection is higher Examples:
INFO: Growing 100 nodes takes 3.868000 seconds. 25.853154 nodes per second. Error: 0.000000 Sample rate: NaN Length: 6.695929 Collision Detection: 133111

It takes longer to grow nodes because it probably has to check for more neighbors for each random configuration)

***Variation 2*: Connect to all vertices within a certain distance**
It takes a lot of time to grow a node (average speed 0.45 nodes/sec). The change was discarded. Some distances resulted a null pointer because there were no vertices within that range.

5 RRT

This algorithm is implemented in *RRTPlanner.java*. RRTPlanner Class extends the abtract class MotionPlanner.java and build configuration space using tree structure. We will use Map data structure to store the tree. It will be in the form: *Map¡Vector, MyEdge¿ parents* where parents.keySet will return all the vertices. Each vertices will be mapped to an ***Edge***. We have defined *MyEdge* class which stores information regarding two connected vertices such as their positions, control vector and duration.

In RRTPlanner, we will also need to implement the following methods: getEdges, getSize, setup, growMap, newConf, findPath, and reset. Among them getSize, setup, and reset are very straightforward. Let us look at the implementation of these functions: newConf, findPath, growMap and getEdges.

newconf()

In this method, we will add a new configuration to the tree. The new configuration is generated by applying a random control from qnear for duration units. We will first get a random control variable using an already defined function getRandomControl from Robot class. We then use Robot's move method to get a new configuration using that random control. Then, we check if there is valid motion from from qnear to the new configuration; if there is, we add it to the tree.

```
private boolean newConf(Vector qnear, double duration) {
        Vector randomControl =
           this.getRobot().getRandomControl(this.random);
        Vector newConfig = this.getRobot().move(qnear,
           randomControl,duration);
```

```
        Trajectory t = new Trajectory(randomControl, duration);
        if(this.getEnvironment().isValidMotion(this.getRobot(),qnear,t,this.RESOLUTION
            if(!parents.containsKey(newConfig)){
                parents.put(newConfig,new MyEdge(newConfig,qnear,
                    randomControl, duration));
                return true;
            }
        }

        return false;
}
```

growMap()

We will generate random Configurations and find its nearest neighbors. for each neighbor, we will invoke newConf to add new vertices to the tree. We keep adding to the tree until the size of the tree is at least a predetermined number. The code snippet is below. Observe that we have changed the basic algorithm where we generate the nearest neighbor. Here, we can determine how many neighbors are to be considered for each *randomConfig*

```
    protected void growMap(int K) {
        while (getSize() < K){
            Vector randomConfig =
                getRobot().getRandomConfiguration(getEnvironment(),this.random);
            if (randomConfig != null) {
                List<Vector> qnears = nearestKNeighbors(parents.keySet(),
                    randomConfig,rateofMapGrow);
                for (Vector qnear: qnears){
                    newConf(qnear, DEFAULT_DELTA);
                }
            }
        }
    }
```

findPath()

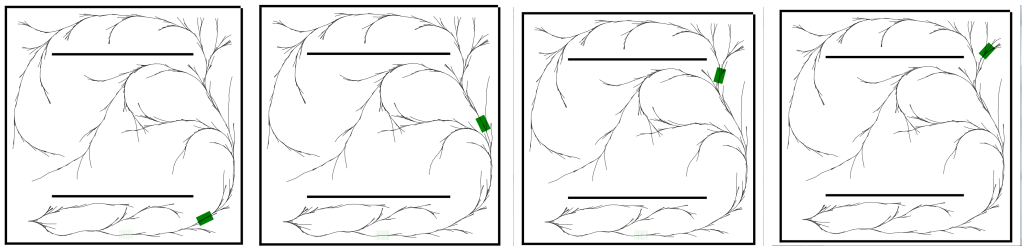
This algorithm is not guaranteed to find the exact path from start to goal. So, we get the nearest vector(qnear) to the goal and find path from start to that vector. We will just backtrack and add all the vectors from the qnear to the start to an arrayList. Then, we traverse that arrayList in reverse order and add controls of each edge to a trajectory and return it.

```
protected Trajectory findPath() {
        Vector qnear = nearestNeighbor(parents.keySet(),getGoal());
        Trajectory path = new Trajectory();
        ArrayList<MyEdge> pathNodes = new ArrayList<>(); //make a new
            arraylist to hold the nodes
        for (Vector current = qnear; current != getStart(); current =
            parents.get(current).getParent()) {
            pathNodes.add(parents.get(current));
        }
        for (int i = pathNodes.size()-1; i >= 0 ; i--) {
            path.addControl(pathNodes.get(i).getControl(),
                pathNodes.get(i).getDuration());
```

```
        }
        return path;
    }
```

6 Testing for RRT

With dubins car, start as(-4,-4,0) and goal as (4,4,0)

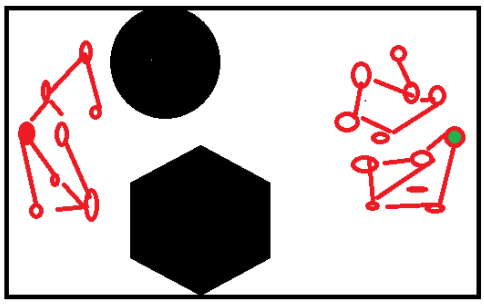


7 Variation for RRT

We can add k newConf are to be added for each qnear in each step of growMap. Here are the results(for 2000 nodes).
k Error time(sec)
1 0.6 4.65
4 0.6 2.8
10 0.4 1.9
15 1.7 2.64

8 Questions for Discussions

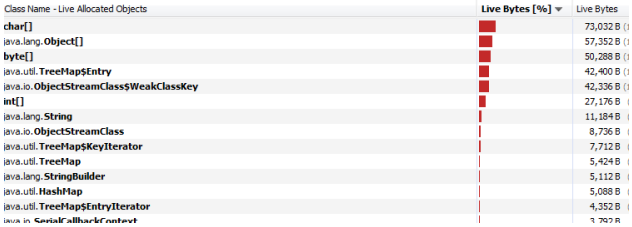
- If there is a feasible path, PRM or RRT is not guaranteed to find it in finite time. For PRM, it adds a random point in the configuration space and draws edges with the nearest vertices. This way, we might get stuck when there is a very narrow corridor. for example, in the picture below, we can see that most of the nodes will get concentrated around start and goal node and there is a low probability that there will be a random vector in the narrow passage. The probability will be even more less when we consider a point robot and a infinitely small narrow passage which is the only feasible path between start and goal position. So, in finite time, it will be hard for PRM to find feasible path.



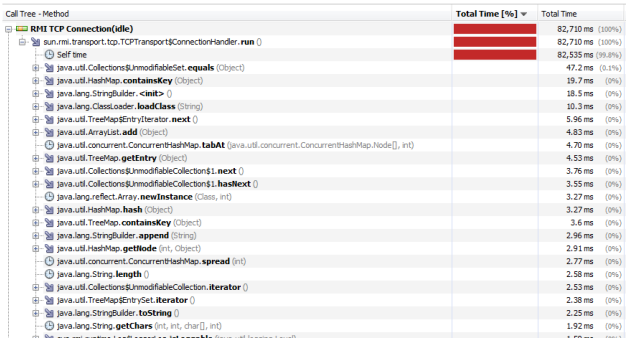
For RRT, we can guarantee that it will return a feasible path either. by design, it is not guaranteed to find a path from start to goal but expected to find a path from start to a point very close to the goal. We will have to grow exponentially large number of nodes as the configuration space gets more obstacles.

- **Pros and Cons of PRM**
  a) It is problematically complete. That means given enough time, it will find a feasible time if there exists one.
  b) It builds a graph to represent a configuration space and depending on the nature of configuration space, the graph may become very large requiring high computational power and memory.
  Pros and Cons of RRT a) It focuses more on getting towards the goal rather than constructing a configuration space. b) It uses a lot of nearestNeighbor calculation and is not guaranteed to find the path from start to goal.
  When designing a specific system, we can use a variant a either of these depending on the system. For example, if we need to find a guaranteed path from start to goal and we do not care about time, we can use PRM.

- From visualVM profiling, I deduced that treeMaps and hashMaps take much memory which is expected considering that we are storing the graph using maps. They are used to generate visual graphics for the animation. Here is a snapshot from my profiler.



  Among methods creating bottlenecks, comparison methods take the most processing power which is expected as they are used whenever things are added to the graph and also during path search.
  her is a snapshot depicting that.



- In PRM, I introduced some variation(such as adding nodes within a certain distance) which have already been discussed earlier and benchmarks results show that they did not help improve the performance. IN RRT, I introduced a variation such that instead of adding one new configuration to map, it adds K new configuration to the map. The results were not consistent as discussed earlier. The main reason is that nodes grow quickly and can accumulated near the start position whereas in some cases they can spread pretty far within a short time.

9 Previous work

I reveiwed a paper by Sertac Karaman and Emilio Frazzoli who wrote about sampling-based algorithms for optimal motion planning. In this paper, they talked about the need to do formal analysis of the quality of the solution returned by algorithms such as PRM and RRT as a function of the number of samples. They

actually proceed to show that these algorithms are not asymptotically optimal, i.e. they will return a solution to the path planning problem with high probability if one exists, but the cost of the solution returned by the algorithm will not converge to the optimal cost as the number of samples increases. Moreover, this paper also introduces new algorithms which are variation of PRM and RRT which are called PRM* and RRT*. They also show that computational complexity of the new algorithms is within a constant factor of that of their probabilistically complete counterparts.

Citations:

Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research, 30(7), 846-894.