Motion Planning

Seyed Mahdi Basiri Azad (Erfan)

January 29, 2016

1 PRM: Probabilistic Road Map

The robot-arm in this project uses the PRM algorithm to transform its configuration from the start configuration to the goal configuration if there is a feasible path from the start to the goal configurations. Each configuration is in a form of a vector containing an angle in radians for each of the robot-arm links.
PRM algorithm is probabilistic in the sense that it creates a random configuration in the configuration space and if the configuration is not colliding with any obstacle it tries to connect that configuration to a number of its neighbors. Depending on the configuration space we may get lumps of connected configurations graphs that are developed separate from each other but given enough number of random configurations for growing the map the separate lumps will be eventually connected to each other. The cool property here is that since the random configurations are getting connected to their K nearest neighbors if a path exists and the graph has grown enough the start configuration **will** find a path to the final goal. So there is no distance error for the found goal and the actual goal configuration.
In order to create a graph from the random configurations I used a function makeNeighbors() that would put the a and b configurations inside each other's adjacency list so the graph will become direction less.

```
private void makeNeighbors(Vector a, Vector b){
  if(getEnvironment().isSteerable(getRobot(),a, b, RESOLUTION)){
    roadMap.get(a).put(b, getRobot().getMetric(a,b));
  }
  if(getEnvironment().isSteerable(getRobot(),b,a, RESOLUTION)){
    roadMap.get(b).put(a,getRobot().getMetric(b,a));
  }
  }
```

The movie, **PRM-hard.mov** shows the robot-arm animation for the environment with multiple obstacles. I have changed the height and width of the robot-arm and increased the number of links.

2 RRT: Rapidly Exploring Random Trees

RRT is yet another fascinating algorithm for path finding. Although RRT, like PRM, uses randomly generated configurations its has more parameters to play with and makes this algorithm unique and more flexible. RRT algorithm generates a random configuration in the configuration space and then find the closest existing configuration to it and starting from that existing configuration it moves a distance d in a "random" direction. The direction does not have to be fully random as it can be chosen randomly from a given list of directions, just like we are doing with different robots in our code. if you run RRT in a narrow corridor (the environment such as "planar-robot-environment-hard") you can see that the tree grows from the starting configuration toward the goal configuration. This is because the new configuration is made by moving slightly away from an existing configuration. Hence in comparison with PRM there is no chance of having separate lumps growing separate from each other. One disadvantage of RRT compared to PRM is the fact that RRT will probably not be connected to the exact "goal configuration" however it will cover the area such that the distance error between the exact goal and the founded goal state will decrease as the graph expands more and more.
In my code i created my own "Edge" class to connect the new configuration with the rest of the expanding graph.

```
private class Edge{
    Vector parentVector = null; //by default
    Trajectory parentTrajectory = null;
    public Edge(Vector P, Trajectory T){
      parentVector = P;
      parentTrajectory = T;
    }
  }
```

There are two movie showing the RRT algorithm at work!
1) **planar-hard-DubinsCar.mov**: This shows the "palanar-robot-environment-hard". I have used the DubinsCar as my robot and has set the goal to the top-right corner. Notice that I have increased the

distance (DEFAULT-DELTA) by 5 folds and instead decreased the number of new configuration needed to reach the goal approximately. Although this practice will increase the distance error to the final goal configuration but it is a much faster way of getting pretty close to the goal!

2) **ant-hanover.mov**: I have changed the boolean field "IS-ANT-MAN" to true for the Hanover map in order for the car to be able to find a path among the Haover buildings. In the video I have pointed to the position of the ant-man-robot with the mouse so you can see it! the start and the goal configurations are same as the video above.

3 Experiments and Founding

As mentioned before the RRT and PRM are both fascinating algorithms but their major difference is the fact that PRM will definitely find the path from start to goal if it exists (given that the graph grows enough so that there are no disconnected graph lumps). However RRT does not necessarily connect the start configuration to the exact goal configuration although its error will decrease as the the tree grows further.

The difference between RRT and PRM makes them appealing for different kinds of applications. For example it is necessary for a surgical robot to find the exact path during the operation on the patients body but an approximate will be good enough for a drone to land some part of a field using RRT! Also varying the "Duration" variable and number of new configurations to grow for the RRT we can change the speed of the algorithm as well as how find or coarse we would want to approach the goal configuration. We can even combine the two by using a coarse RRT to an approximate location near the goal and use PRM to fully reach the goal configuration. This will speed up the path finding process and will definitely reach the goal if there is a feasible path. although the path will probably not be the most optimal path due to the coarse RRT.

While using the profiler in **VisualVM** I realized that the grow() method is the one that takes the most amount of time, this was expected! for the PRM the aStar search can be performed on the graph which given the right heuristics will produce the optimal path given the graph. And in RRT the back chain function is pretty straight forward, it simple follows the links back to the start node. In order to make the path finding faster I played with the "DEAFULT-DELTA" variable and the controls. As I have mentioned before by increasing the "DEAFULT-DELTA" I managed to reduce the number of node needed to get a good approximation to the goal. Hence I decreased the number of iterations inside of grow() function which reduced the time in exchange for increase in error. Also using different robots proved the fact that limiting

the randomness of the new control direction (think DubinsCar vs. OmniDirectional as an example) can help with taking longer paths due to taking the wring direction. The image below is using OmniDirectional robot(Left). See the difference between between the clearness of the paths compared to DubinsCar (right) from the movie (both graphs are growing 20000 new configurations)