Motion Planning Solution

Boying Shi

February 2, 2014

1 Introduction

Motion planning is a term used in robotics for the process of breaking down a desired movement task into discrete motions that satisfy movement constraints and possibly optimize some aspect of the movement. A basic motion planning problem is to produce a continuous motion that connects a start configuration S and a goal configuration G, while avoiding collision with known obstacles.

Motion planning is a very practical aspect which can help to solve many real world problem, like furniture move problem or car parking problem. It helps to find a path without touching known obstacles in 2D or 3D world.

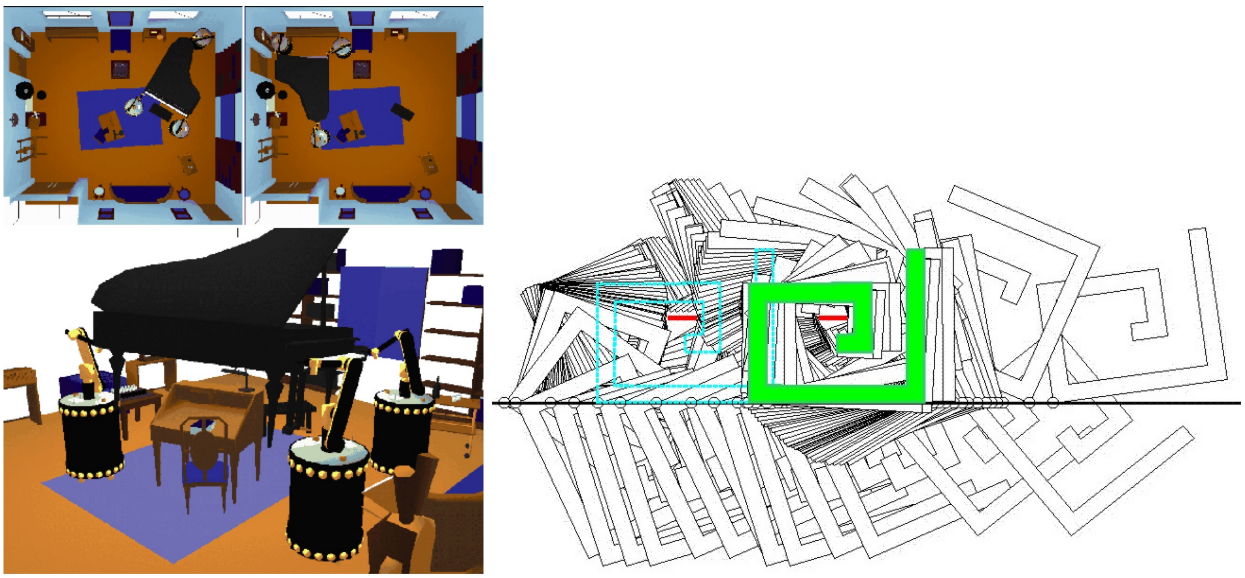

Figure 1: motion plan example

This report contains two main implementations of motion planners, one is using k-PRM to find a road map and a path for a arm robot. The other one is using RRT to expand a tree road map for a car robot. For the arm robot problem, I assumed that the base of the arm is fixed which means that its x coordinate and y coordinate will not change. Then it can avoid touching obstacles on the map while moving from start state to goal state. For car robot problem, the road map will be created by RRT algorithm and the result of road map is shown on the map.

2 Previous work

*As my status is graduate student, this part is required.

2.1 PRM

The probabilistic roadmap approach(PRM) is one of the leading motion planning techniques in recent years it gain the success to plan path for holonomic problem efficiently by using samples and local planner. PRM is a relatively new approach to motion planning among many other approaches and it turns out be very efficient and easy to implement and applicable for many different types of motion panning problem.

The basic PRM approach is very simple, two phases: generate graph by sampling and find neighbors and query phase. However, there are still many details to be discussed like how to perform collision checks of the paths produced by the local planner, and different sampling strategies and different strategies for choosing promising pairs of nodes to connect.

Thanks to other researchers' works and study, it concludes that a binary approach performs best when performing collision checks of the paths for local planner. And except for very special cases, a best way for sampling is to use a deterministic approach based on Halton points, with a small amount of added randomness. Also, a best way to choose nodes to connect is to pick a few nodes in each connected component of the roadmap.

2.2 RRT

While we have already have a very efficient motion planning method, PRM, to help us solve path planning problem and it is simply to implement and use. There still exist a problem that for non-holonomic problem, connecting each k neighbor of current state by using local planner is inefficient and need to connect thousands of edges. Therefore, RRT is a very practical method to solve non-holonomic problem and contain small number of edges but they are all connected. RRT is very efficient and easy to implement. The idea is to iteratively find the nearest control state to a random position and expand the tree. Nowadays, people have already successfully applied RRT to holonomic, non-holonomic and kinodynamic motion planning problems of up to twelve degrees of freedom.

3 k-PRM Planner

The probabilistic road map planner is a motion panning algorithm which solving the path finding problem for finding a path without touching obstacles on the map from the start position and goal position. The basic idea is to take random samples from the configuration space of the robot, testing whether they are legal and keep those legal space and then using a local planner to find the k nearest configuration space regrading to the current position and create a graph by exploring every space for those samples. And then, by using path finding algorithm, a path could be found for the start position to the goal position.

3.1 Implementation

The probabilistic roadmap planner consists of two phases: a **construction phase** and a **query phase**. In the construction phase, a roadmap (graph) is built, approximating the motions that can be made in the environment. First, a random configuration is created. Then, it is connected to some neighbors, typically either the k nearest neighbors or all neighbors less than some predetermined distance. My implementation of this graph construction phase is shown below:

Listing 1: Construction Phase

```
//k-PRM construction phase
public Graph CreateGraph(World check){
  double[] theta1 = new double[this.sampleNum];
  double[] theta2 = new double[this.sampleNum];

  Graph graph = new Graph();

  //Get random samples of configuration space
  for(int i = 0; i<this.sampleNum; i++){

    Random rand1 = new Random();
    int temp1 = rand1.nextInt(314);
    theta1[i] = (double)temp1/100;

    Random rand2 = new Random();
    int temp2 = rand2.nextInt(628);
    theta2[i] = (double)temp2/100;

    //Add these sample vertices into graph
    double[] temp_config = {x,y,armLength,theta1[i],armLength,theta2[i]};
    ArmRobot temp_arm = new ArmRobot(ArmRobot.LinkNum);
    temp_arm.set(temp_config);
    graph.addVertex(temp_arm);
    }

  //for each sample position find its k neighbors
  for(int j = 0; j<this.sampleNum; j++){
    //arm_list is used to store and sort all other samples, regarding to time
    ArrayList<ArmRobot> arm_list = new ArrayList<ArmRobot>();

    //list is used to store and sort the k nearest neighbor, regarding to time
    ArrayList<ArmRobot> list = new ArrayList<ArmRobot>();

    double[] current_config = {x,y,armLength,theta1[j],armLength,theta2[j]};
```

```
    ArmRobot current_arm = new ArmRobot(ArmRobot.LinkNum);
    current_arm.set(current_config);

    //find all neighbors and sort them in arm_list
    for(int i = 0; i<this.sampleNum;i++){
      if(j!=i){
        double[] new_config = {x,y,armLength, theta1[i], armLength,
            theta2[i]};
        ArmRobot new_arm = new ArmRobot(ArmRobot.LinkNum);
        ArmRobot new_arm_fake = new ArmRobot(ArmRobot.LinkNum);
        new_arm.set(new_config);

        //since armCollisionPath will change value, use a fake one to check
        new_arm_fake.set(new_config);

     if(check.armCollision(new_arm)==false &&
        check.armCollisionPath(new_arm,current_arm.config,
                                              new_arm.config)==false){
           ArmLocalPlanner ap = new ArmLocalPlanner();
           double time = ap.moveInParallel(current_arm.config,
               new_arm_fake.config);
           new_arm_fake.setTime(time);
           arm_list.add(new_arm_fake);
           arm_list.sort(comparator);
           }
             }else{
               continue;
                 }
               }

      //find the nearest k neighbors and add them into graph
      for(int i = 0 ; i<k; i++){
        ArmRobot temp_arm = new ArmRobot(ArmRobot.LinkNum);
        temp_arm = arm_list.get(i);
        list.add(temp_arm);
        try {
          graph.addEdge(current_arm, temp_arm);
            } catch (Exception e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
                }
               }
               }
        return graph;
      }



//sort array list by override compare function regarding to time
private Comparator<ArmRobot> comparator =  new Comparator<ArmRobot>(){
  @Override
```

```
  public int compare(ArmRobot a, ArmRobot b) {
    if(a.getTime() < b.getTime())
      return -1;
    else if(a.getTime() > b.getTime())
      return 1;
    return 0;
    }
  };
```

In the query phase, the start and goal configurations are connected to the graph, and the path is obtained by A* algorithm. I reusing the implementation of A* algorithm of my homework 2 and simply change its heuristic() function to calculate the time to move to the next move plus the time of moving from the next position to goal position. My code is here:

Listing 2: Query Phase

```
public List<ArmRobot> QPhase(Graph graph){
  //reusing A* algorithm
  PriorityQueue <ArmRobot> OpenList = new PriorityQueue<ArmRobot>();

  HashMap<ArmRobot, ArmRobot> visited = new HashMap<ArmRobot, ArmRobot>();

  ArmRobot startNode = new ArmRobot(ArmRobot.LinkNum);
  ArmRobot currentNode = new ArmRobot(ArmRobot.LinkNum);
  ArmRobot goalNode = new ArmRobot(ArmRobot.LinkNum);

  //set start position and goal position
  startNode = graph.mVertices.get(0);
  System.out.println("Start config "+startNode);
  goalNode = graph.mVertices.get(30);
  System.out.println("Goal config "+goalNode);

  OpenList.add(startNode);
  visited.put(startNode,null);

  while(!OpenList.isEmpty()){
    currentNode = OpenList.poll();

    if (currentNode.equals(goalNode)) {
      return backchain(currentNode, visited);
        }

    ArrayList<ArmRobot> successors = graph.getSuccessor(currentNode);

      for (ArmRobot child : successors){
        if(!visited.containsKey(child)){
          OpenList.add(child);
          visited.put(child,currentNode);
          }else{
            if(!child.equals(startNode) &&
               visited.get(child).getTime() > child.getTime()){
               visited.remove(child);
               visited.put(child,currentNode);
```

```
                }
              }
            }
          }
     OpenList.clear();
     visited.clear();
     return null;
    }


protected List<ArmRobot> backchain(ArmRobot node,
          HashMap<ArmRobot, ArmRobot> visited) {
  LinkedList<ArmRobot> solution = new LinkedList<ArmRobot>();
  // chain through the visited hashmap to find each previous node,
  // add to the solution

  while (node != null) {
    solution.addFirst(node);
    node = visited.get(node);
    }
    return solution;
}
```

After finishing these two main part of k-PRM planner, the job is done and I can test with my map. Here, I have personally implemented a `Graph.class` to help me encapsulate all graph operations and data structure. I used `HashMap` to represent a graph.

3.2 Test Result

3.2.1 Two links

Here in `Figure 2: Results of 2 links` shows my test result of two links arm robots and these 4 pictures are interesting from my point of view. The arm can find a path to avoid collision with known obstacles on the map and like a real word robot arm.

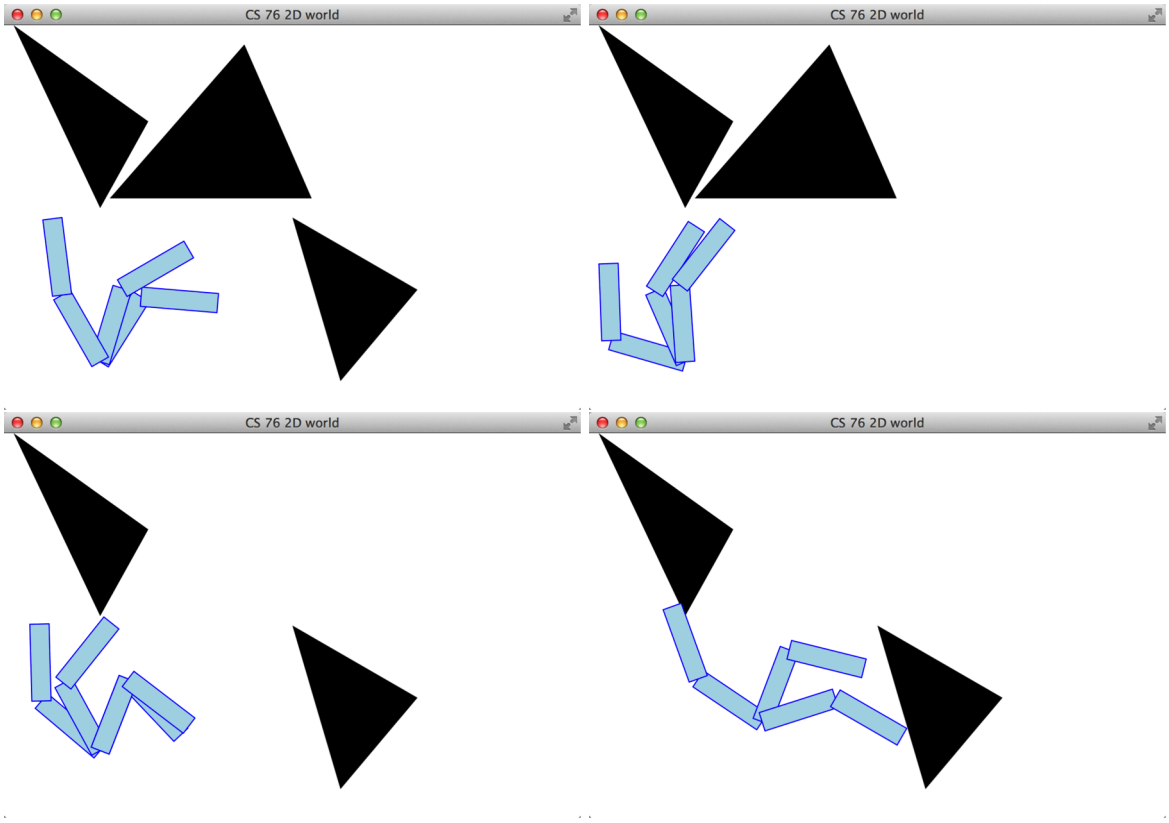

Figure 2: Results of 2 links

3.2.2 Three links

Here in `Figure 3: Results of 3 links` shows my test result of three links arm robots. And the the arm can avoid collision with known obstacles.

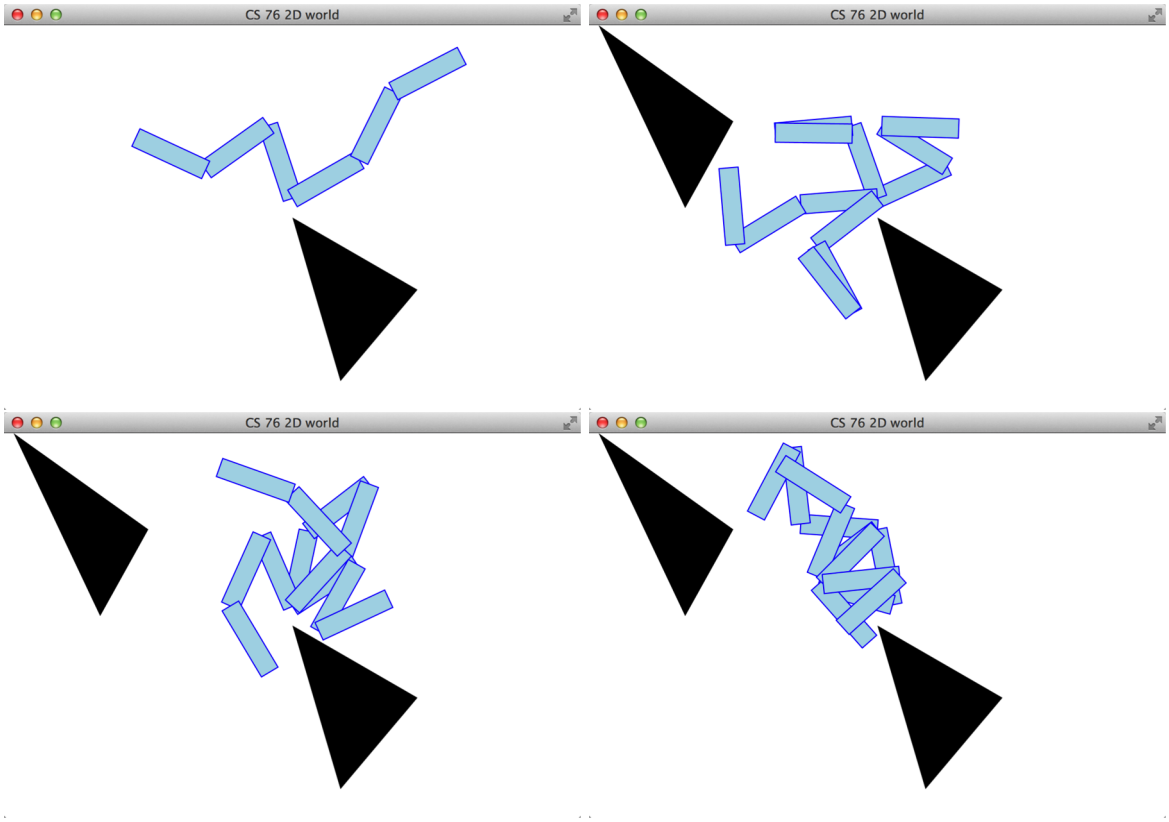

Figure 3: Results of 3 links

From my test results, I can clearly see that by using k-PRM algorithm, the arm robots can find a path to move with its arm and can avoid touching the obstacles by rotating its arm.

3.3 Analysis

k-PRM algorithm is a complete planner which means that if a solution exists and the planner will eventually find it by using random sampling. k-PRM planner is a sampling Based-planners and it can create a topological graph with samples and accept single-query and multi-query of a path. However, there are many variables like k neighbors and number of samples and different methods of collision detector to choose for a k-PRM planner, how do they affect the algorithm is need to be discussed.

How do k neighbors affect k-PRM?

k neighbors are chosen by local planner that they are the nearest legal configuration of current position. As is known, by sampling m configuration spaces ($k < m$), for the current position, we will choose k nearest legal configurations among those m configurations. Then, I will add edges from the current position to those k neighbors. This process will iterate for m times and finally I will get a graph with edges linked to configurations. Therefore, a larger k will result in a denser graph and will provide more choices for a path finding problem.

Also, it depends on how large the m is. When the number of simples is very big, and the current position can just get k nearest configurations. While the number of simples is not very big and k is not small, which I mean k and m is close to each other, then it cannot find the real "nearest" configurations and then the result will be not very proper. Also, if m is very large, but k is small, then it cannot get enough neighbors thus the graph will be loose.

Therefore, I think a proper k should have some relationships with how to choose m. And then a good k will result in a good graph which is neither too dense nor too loose and can give a good preprocessing for the query phase.

How are random configurations chosen?

As I have discussed previously, a good choice of the number of samples will affect a good graph. Also, it is very important to think about how to choose those random configurations. In my implementation, I just using Math.random(width) and Math.random(height) to choose those random configurations. It is the most simple way, though, a better way is to think about how to choose those configurations which are well-distributed on the map. My method will have a problem that those samples will be unevenly chosen on the map.

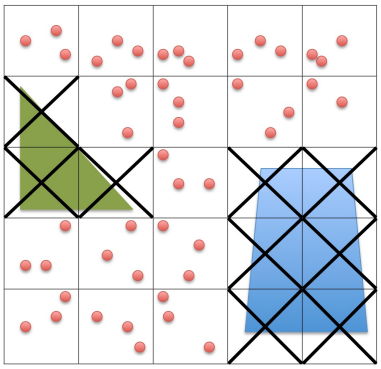

Figure 4: how to random configurations

In my opinion, I think a better way to choose a more well-distributed sample set is to separate the entirely map into more cells. By using Math.random(width) and Math.random(height) to choose random configurations, I just simply treated the entire map as a big cell and get random configurations in it. Now, I can cut the big cell into more pieces. I can cut the big cell into 5*5 grids map, and therefore there will have

25 cells. If I want to choose m samples, then I can just choose $\frac{m}{25}$ samples for each cell. Then it will get a better distribution of random configurations.

Moreover, since obstacles are all known, when cut the big map into small cells, I can try to exclude those cells on the obstacles and just choose those cells are legal, then I can get a better random configurations.

How to choose distance function?

After solving the problem of choosing random configurations and the number of k neighbors, it comes to the problem that how to choose a distance function which means how do we define the nearest. In my implementation, I choose feature *time* to be a standard for calculating the distance function. As is mention in class, the distance function can also be chosen as *energy* or *manhattandistance*.

In my opinion, the standard of the nearest neighbors should be chosen regarding to the specific situation. If the feature *time* can better describe the problem then *time* should be the standard for distance function. Or if it just a map that a robot want to move from one place to another place, then a *manhattandistance* is much better.

Therefore, I think distance function should be determined by a certain problem need to be solved, after analyzing the problem, and then a distance function can be chosen.

Other problems

Also, there still exist some other problems need to be discuss, like how to choose the nearest neighbor (the local planner) and how to detect collisions. Since for this homework, these parts are provided by instructor, I will not discuss them here but I still think they are interesting.

4 RRT Planner

A Rapidly-exploring Random Tree (RRT) is an algorithm designed to efficiently search nonconvex, high-dimensional spaces by randomly building a space-filling tree. The tree is constructed incrementally from samples drawn randomly from the search space and is inherently biased to grow towards large unsearched areas of the problem.

4.1 Implementation

An RRT grows a tree rooted at the starting configuration by using random samples from the search space. As each sample is drawn, a connection is attempted between it and the nearest state in the tree. If the connection is feasible (passes entirely through free space and obeys any constraints), this results in the addition of the new state to the tree.

My implementation of it is shown below:

Listing 3: Query Phase

```
public class RRTPlanner {
  private int k;
  private int x;
  private int y;
  private double theta;
  CarRobot startnode = new CarRobot();

  public RRTPlanner(int sk, int sx, int sy, double theta) {
    this.k = sk;
    this.x = sx;
    this.y = sy;
    this.theta = theta;
    CarState state = new CarState(sx, sy, theta);
    startnode.set(state); //set start position
    }

  public Tree CreateTree(World check){
    Tree tree = new Tree();
    tree.addVertex(startnode);

    //for store and sort
    ArrayList<CarRobot> car_list = new ArrayList<CarRobot>();

    //for choose different control
    SteeredCar control = new SteeredCar();

    //k steps
    for(int i = 0 ; i<k;i++){

      Random rand1 = new Random();
      double rx = (double)rand1.nextInt(600);

      Random rand2 = new Random();
      double ry = (double)rand2.nextInt(400);

      Random rand3 = new Random();
```

```
int temp = rand3.nextInt(628);
double rtheta = (double)temp/100;

//get and set random position
CarRobot carrand = new CarRobot();
CarState carrand_state = new CarState(rx,ry,rtheta);
carrand.set(carrand_state);


//calculate the distance from tree members to the random position
//sort and store them in car_list
for (int j = 0; j < tree.Vertices.size(); j ++){
  double tx = tree.Vertices.get(j).getCarState().getX();
  double ty = tree.Vertices.get(j).getCarState().getY();
  double distance = LocalDistance(tx,rx,ty,ry);
  tree.Vertices.get(j).setDistance(distance);
  car_list.add(tree.Vertices.get(j));
  car_list.sort(comparator);
  }

//choose the nearest position as current position
//clear car_list
CarRobot current = new CarRobot();
current = car_list.get(0);
car_list.clear();

//calculate the distance from each move to the random position
//sort and store them in car_list
for(int j = 0; j < control.control.length;j ++){
  CarRobot newleaf = new CarRobot();
  CarState newleaf_state = control.move(current.s, j, 1);
  newleaf.set(newleaf_state);
  double leafdistance =
     LocalDistance(newleaf_state.getX(),rx,newleaf_state.getY(),ry);
  newleaf.setDistance(leafdistance);

  if(check.carCollision(newleaf)==false &&
     check.carCollisionPath(newleaf, newleaf_state,j,1)==false){
       car_list.add(newleaf);
       car_list.sort(comparator);
       }
     }

  //choose the nearest move from car_list
  //add it to the tree
  //clear car_list
  CarRobot child = new CarRobot();
  child = car_list.get(0);
  tree.addVertex(child);
  try {
     tree.addEdge(current, child);
```

```
            } catch (Exception e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
              }
              car_list.clear();
       }
         return tree;
     }

//calcuate the distance between to position
private double LocalDistance(double cx, double rx, double cy, double ry){
      return Math.sqrt((cx-rx)*(cx-rx)+(cy-ry)*(cy-ry));
      }


 private Comparator<CarRobot> comparator =  new Comparator<CarRobot>(){
 @Override
 public int compare(CarRobot a, CarRobot b) {
 // TODO Auto-generated method stub
 if(a.getDistance() < b.getDistance())
   return -1;
 else if(a.getDistance() > b.getDistance())
   return 1;
 return 0;
     }
   };
 }
```

In my implementation, I will each time random a position on the map and get a position form my tree data structure which is closest to the random position. And from this current position, I will do six moves (6 controls) and find the move that the distance is shorts to the random position. Then, store this new position into my tree and add edge between the current position and the new position. After iterating k times, then it will take k steps. Therefore, the RRT is created.

4.2 Test Result

From *Figure*5 :RRT of different k, it shows that different results of different k values. The k value are 300, 500, 700, 900 in order. The picture clearly shows that k affects the density of the tree, the larger k is, the denser the tree is.

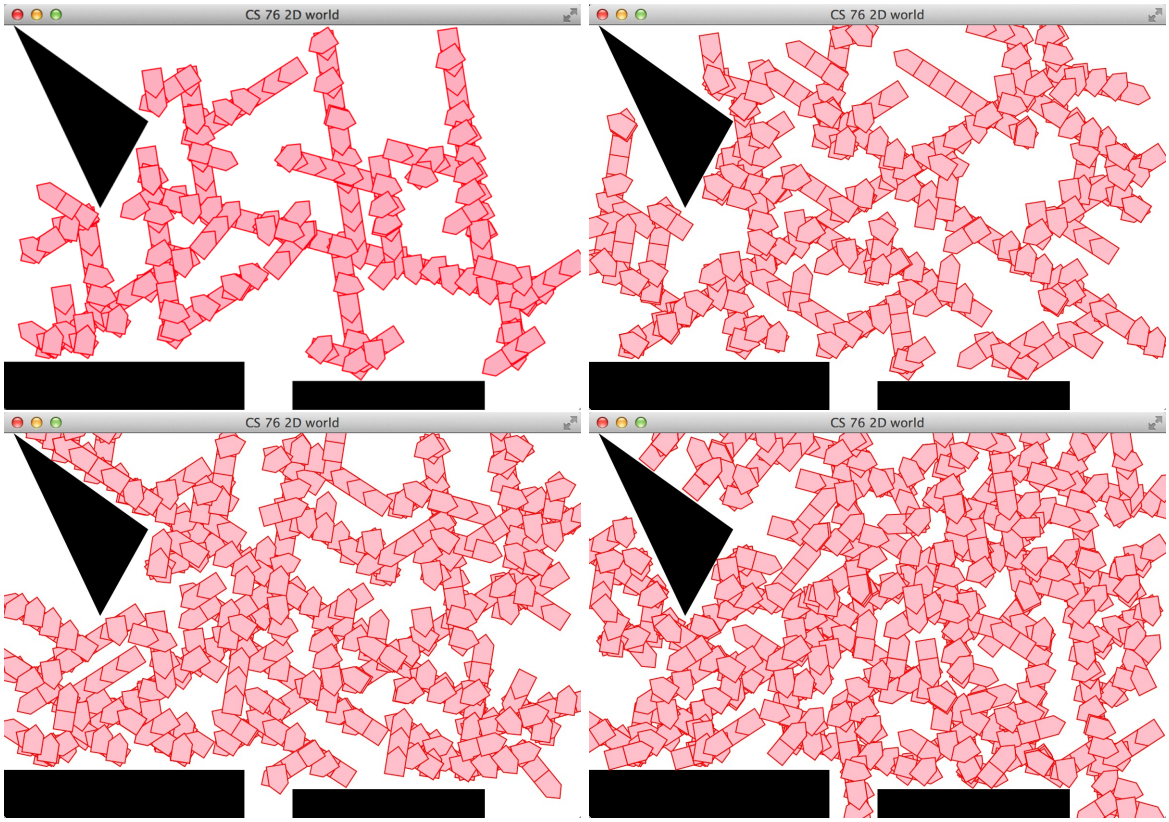

Figure 5: RRT of different k

From *Figure*6 :RRT of different map, I change the position of obstacles and create a different map. I run RRT algorithm with unchanged k value, and the result shows that the car robot and detect collision with different obstacles and create tree road map.

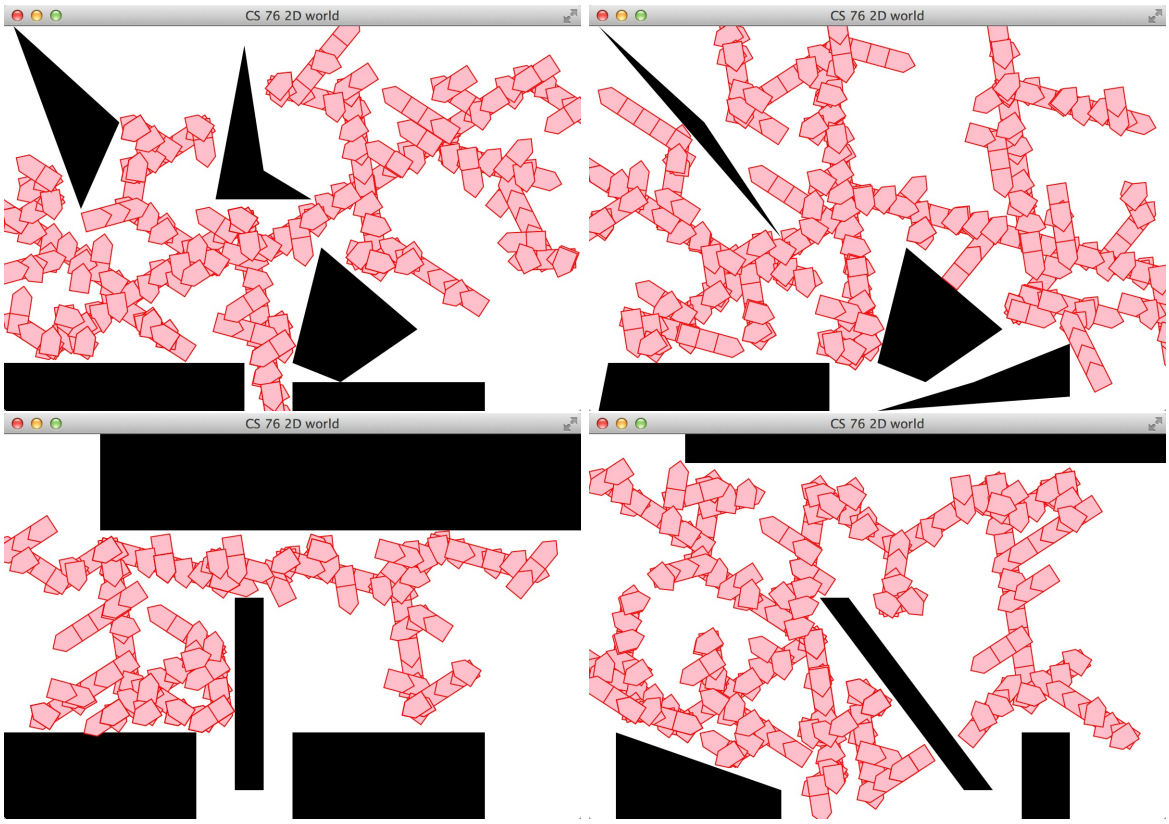

Figure 6: RRT of different map

4.3 Analysis

An RRT is iteratively expand by applying control inputs that drive the system slightly toward randomly-selected points, as opposed to requiring point-to-point convergence, as in the probabilistic roadmap approach. As trees grow, the eventually share a common node, and are merged into a path. Therefore, a road map has been created.

Why do we need RRT?

As is known, k-PRM and many other randomized potential field algorithm have been solved path planning problem and they are efficient enough. And why do we still need another randomized path planning technique. The reason is that there still exist difficulty to general non-holonomic path planning problem. In k-PRM, we can simply use local panning to help find the nearest neighbors and connect configurations. While for steerable non-holonomic systems, like what we deal with in this assignment, the car robot problem, the connection problem can be complicated since the connections of local planner might need to connect thousands of configurations to find a solution. And therefore, it seems impractical for non-holonomic problems to use k-PRM algorithm. Thus, we need to apply RRT to solve these problems, since RRT can be directly applied to non-holonomic planning and it do not need any connections to be made between pairs of configurations and it is very efficient.

The advantage of RRT

As I have mentioned in last section, RRT is useful to solve non-holonomic planning problem and do not need to connect all nearby neighbors which is very practical. Also, RRT still has other advantages to use. First, the expansion of an RRT is heavily biased toward unexplored portions of the configuration space. Since RRT will random a position each time from the configuration space, and then calculate the distance between each move to this random position, it will lead to a consistent behavior. Second, RRT will always be connected although the number of edges are small. RRT will choose the nearest one to connect at each step and therefore the tree will be always connected and good for path searching problem. Moreover, RRT is a simple and short algorithm that is very easy to understand and implement. Also, it can be applied to many different applications.

Explore different environments for RRT

As I have shown in my test results, I change the position of obstacles and create different map to test an unchanged k value of RRT. The result shows that RRT can adjust to different environments and expand the tree. When there is no obstacle on the way of a tree branch, then it will continue its way to the same direction until it finally reach an obstacle then it will come back to find another path which can be seen from the picture that it will find another branch of the tree. iteratively, the tree form road map will be generated after taking k steps. And the density of the tree will increase by increasing the value of k. If k is big, then the tree will be denser and results in a more complicated connected road map.

5 Citation

For picture in introduction:
http://planning.cs.uiuc.edu/node10.html
http://en.wikipedia.org/wiki/File:Graph2displaycolor.jpg
For introduction part:
http://en.wikipedia.org/wiki/Motion_planning
http://planning.cs.uiuc.edu/node4.html
For previous work part:
http://www.staff.science.uu.nl/~gerae101/pdf/compare.pdf
http://msl.cs.uiuc.edu/~lavalle/papers/Lav98c.pdf
For k-PRM part:
http://en.wikipedia.org/wiki/Probabilistic_roadmap
http://www.isi.edu/robots/CS561/Lectures/Probabilisticpath.pdf
For RRT part:
http://en.wikipedia.org/wiki/Rapidly_exploring_random_tree
http://www.isi.edu/robots/CS561/Lectures/Probabilisticpath.pdf
http://msl.cs.uiuc.edu/~lavalle/papers/Lav98c.pdf
For code part:
Using provided code by Weifu Wang.
Discussed with TA Yinan Zhang, Yuhan Lyu.
Got ideas from discussion on Piazza.