Motion Planning
Probabilistic Roadmap & Rapidly Exploring Random Tree

Xinqi Li

February 4, 2014

Abstract

The report discusses the implementations of motion planners for two systems: a robot arm and a steered car. The first part shows the implementation of Probabilistic Roadmap algorithm and it use A* search to find the path of the robot arm moving from one configuration to another. The second part is about Rapidly Exploring Random Tree algorithm. It shows all the nodes in the tree and plans a path using Bread First Search. Lastly, it present the main idea of a recent paper concerning about the optimal path planning problem using efficient PRM* and RRT* algorithms.

1 Probabilistic Roadmap (PRM)

1.1 Introduction

Probabilistic roadmap is a method from the family of sampling-based roadmaps[1]. To plan for the motion path of the arm robot, there are two phases. One is generation of the roadmap which contains the problem of sampling and getting the K-nearest neighbors (K-NN). The other is query phase which search the path for the robot. Here I modify the A* search from last assignment.

1.2 Code Design

For the arm robot problem, I design a class `PRMPlanner` in addition to the provided skeleton code. Besides the construction functions, there are two public functions in this class: `prmGenerator` which generates the roadmap and `astarSearch` which plan the path from start configuration to goal configuration. Moreover, it contains two comparators inner classes to control the function of priority queue and the process of sorting and finding the KNN. I use `ArrayList<Double>` instead of double array to store configurations due to the problem of key losing in `Hashmap`. To make it works a function of `convertd2D` is needed. This function converts the double array to Double array which will be used to convert double array to `ArrayList<Double>`. Maybe you think a better way is to use a class to encapsulate the configuration array, I think so too, but when I encounter the `hashmap` key missing problem, I just want to try something Ive never tried before. To tell the truth, using `ArrayList` is really suffered.

1.3 Code and Explanation

The key part of it is the `prmGenerator`.
Firstly, it creates N collision free samples in the world. Here I used the class Random to implement the random number generator; it generates a number from 0 to 2 pi every time it is called. To be emphasized is that the start and goal configurations are put in the samples to ensure a path can be found between them in the query phase.
Here is the code for `prmGenerator`:
Then it finds the KNN. The intuition is to use a priority queue but it need to be sorted every time when

it gets a new configuration. Hence I use a List and the sort method on it. To make the comparator works every time it gets a new configuration, a private attribute `currentconfig` should be defined.
I implement the query phase by changing the search node type to `ArrayList<Double>` which lead to adding the comparator for the `priorityqueue` is needed.

The attributes and constructor of the class `PRMPlanner`:

```
public class PRMPlanner {
  private World world;
  private ArrayList<Double> IniConfig;
  private ArrayList<Double> goalConfig;
  private ArrayList<Double> currentconfig;
  private ArmRobot arm;
  private int K=15;//K nearest
  private int N=300;//#of samples

  //The constructor
  public PRMPlanner(World W, ArmRobot a, double[] i, double[] g) {
    // TODO Auto-generated constructor stub
    world=W;
    arm=a;
    //reference http://stackoverflow.com/questions/5178854/convert-a-double-
        array-to-double-arraylist
    IniConfig= new ArrayList<Double>(Arrays.asList(this.convertd2D(i)));
    goalConfig= new ArrayList<Double>(Arrays.asList(this.convertd2D(g)));
  }

public HashMap<ArrayList<Double>,List<ArrayList<Double>>> prmGenerator() {
    HashMap<ArrayList<Double>,List<ArrayList<Double>>> adjlist=new HashMap<
        ArrayList<Double>, List<ArrayList<Double>>>(N);
    ArmLocalPlanner ap = new ArmLocalPlanner();
    int linknum=arm.getLinks();
    currentconfig=new ArrayList<Double>();
    for(int l=0;l<IniConfig.size();l++)
      currentconfig.add(IniConfig.get(l));
    Random rng=new Random();
    //will remove node while finding road
    List<ArrayList<Double>> samples=new ArrayList<ArrayList<Double>>(N);
    //use to sort and find the knn, will not change while finding road
    List<ArrayList<Double>> samplessort=new ArrayList<ArrayList<Double>>(N);
    int n=0;
    samples.add(IniConfig);
    samples.add(goalConfig);
    samplessort.add(IniConfig);
    samplessort.add(goalConfig);
    while(n<N){
      ArrayList<Double> config=new ArrayList<Double>();
      config.add(IniConfig.get(0));
      config.add(IniConfig.get(1));
      //creat configs
```

```
      for(int ii=2;ii<IniConfig.size();ii+=2){
        config.add(IniConfig.get(ii));
        config.add(rng.nextDouble() * Math.PI * 2);
      }
      ArmRobot a=new ArmRobot(2);
      double[] c= new double[linknum*2+2];
      for(int t=0;t<linknum*2+2;t++)
        c[t]=config.get(t);

      a.set(c);
      if(!world.armCollision(a)){
        samples.add(config);
        samplessort.add(config);
        n++;
      }
    }

    int i=0;
    while((i<N+2) && (samples.size()!=0)){
      List<ArrayList<Double>> neighbors=new ArrayList<ArrayList<Double>>(N);
      ArrayList<Double> configs=new ArrayList<Double>();
      for(int l=0;l<samples.get(0).size();l++){
        configs.add(samples.get(0).get(l));
        currentconfig.set(l, samples.get(0).get(l));
      }
      samples.remove(0);
      Collections.sort(samplessort, new comparator());
      double[] g= new double[configs.size()];
      for(int t=0;t<configs.size();t++)
        g[t]=configs.get(t);
      int k=1;
      while(k<N+2 && neighbors.size()<=K){
        ArrayList<Double> nb=new ArrayList<Double>();
        double[] p= new double[samplessort.get(k).size()];
        for(int t=0;t<samplessort.get(k).size();t++){
          nb.add(samplessort.get(k).get(t));
          p[t]=samplessort.get(k).get(t);
        }
        if(!world.armCollisionPath(arm, g, p))
          neighbors.add(nb);
        k++;
      }

      adjlist.put(configs, neighbors);
    }
    return adjlist;
  }
```

Following is the code for `comparator`which used to sort and find the KNN. It compares the time to move from currenconfig.

```
public class comparator implements java.util.Comparator<ArrayList<Double>>{
  @Override
  public int compare(ArrayList<Double> o1, ArrayList<Double> o2) {
    // TODO Auto-generated method stub
    ArmLocalPlanner ap = new ArmLocalPlanner();
    // get the time to move from currenconfig to config;
    double[] d= new double[o1.size()];
    double[] e= new double[o1.size()];
    double[] f= new double[o1.size()];
    for(int t=0;t<o1.size();t++){
      d[t]=currentconfig.get(t);
      e[t]=o1.get(t);
      f[t]=o2.get(t);
    }
    double time1 = ap.moveInParallel(d, e);
    double time2 = ap.moveInParallel(d, f);
    return (int) Math.signum(time1 - time2);
  }
}
```

Following is the `convertd2D`. It converts the double array to Double array [2]

```
private Double[] convertd2D(double[] array){
  Double[] narray = new Double[array.length];
  for(int i = 0; i < array.length; i++){
    narray[i] = array[i];
  }
  return narray;
}
```

Here is the code concerning the path planning. `astarSearch` and `backchain` search the path. `comparator2` is use to compare the states in priority queue.

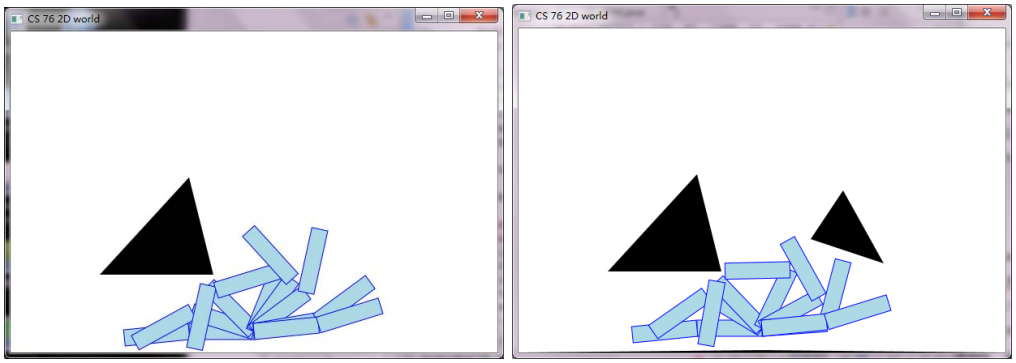
```
public List<ArrayList<Double>> astarSearch(HashMap<ArrayList<Double>,List<
    ArrayList<Double>>> prm) {

  PriorityQueue<ArrayList<Double>> frontier = new PriorityQueue<ArrayList<
      Double>>(N*N,new comparator2());

  // map to store backchaining information
  HashMap<ArrayList<Double>, ArrayList<Double>> explored = new HashMap<
      ArrayList<Double>, ArrayList<Double>>();
  //HashSet<SearchNode> nodecost=new HashSet<SearchNode>();
  HashMap<ArrayList<Double>,Double> nodecost=new HashMap<ArrayList<Double>,
      Double>();
```

```
    // startNode must be set by the constructor of the particular
    // search problem, since a UUSearchNode is an interface and can't
    // be instantiated directly by the search
    double[] h= new double[IniConfig.size()];
    double[] i= new double[IniConfig.size()];
    for(int t=0;t<IniConfig.size();t++){
      h[t]=IniConfig.get(t);
      i[t]=goalConfig.get(t);
    }
    explored.put(IniConfig, null); // startNode was not reached from any other
        node
    ArmLocalPlanner ap = new ArmLocalPlanner();
    // get the time to move from currenconfig to config;

    double totalcost=0;
    nodecost.put(IniConfig,totalcost);//nodecost store the explored nodes
    frontier.add(IniConfig);

    while (!frontier.isEmpty()) {

      ArrayList<Double> currentNode = new  ArrayList<Double>() ;
      currentNode=frontier.remove();
      if (currentNode.equals(goalConfig)) {
        return backchain(currentNode, explored);
      }

      List<ArrayList<Double>> successors = prm.get(currentNode);

      for (ArrayList<Double> node : successors) {
        double[] j= new double[IniConfig.size()];
        for(int t=0;t<node.size();t++)
          j[t]=node.get(t);
        totalcost = ap.moveInParallel(i, j);
        // if not visited
        if (!explored.containsKey(node)
            ||(totalcost<nodecost.get(node))){
          explored.put(node, currentNode);
          nodecost.put(node, totalcost);
          frontier.add(node);
        }
      }
    }
    return null;
}
public class comparator2 implements java.util.Comparator<ArrayList<Double>>{
    @Override
    public int compare(ArrayList<Double> o1, ArrayList<Double> o2) {
      // TODO Auto-generated method stub
      ArmLocalPlanner ap = new ArmLocalPlanner();
      // get the time to move from currenconfig to config;
      double[] d= new double[o1.size()];
```

```
        double[]  e= new double[o1.size()];
        double[]  f= new double[o1.size()];
        double[]  j= new double[IniConfig.size()];

        for(int t=0;t<o1.size();t++){
          d[t]=IniConfig.get(t);
          e[t]=goalConfig.get(t);
          f[t]=o2.get(t);
          j[t]=o1.get(t);
        }
        double totalcost1 = ap.moveInParallel(e, j);
        double totalcost2 = ap.moveInParallel(e, f);
        return (int) Math.signum(totalcost1 - totalcost2);
      }
    }
    protected List<ArrayList<Double>> backchain(ArrayList<Double> node,
        HashMap<ArrayList<Double>, ArrayList<Double>> visited) {
      LinkedList<ArrayList<Double>> solution = new LinkedList<ArrayList<Double
          >>();
      while (node != null) {
        solution.addFirst(node);
        node = visited.get(node);
      }
      return solution;
    }
```

1.4 Testing

Firstly, I test the robot with some tricky world then observe its motion.

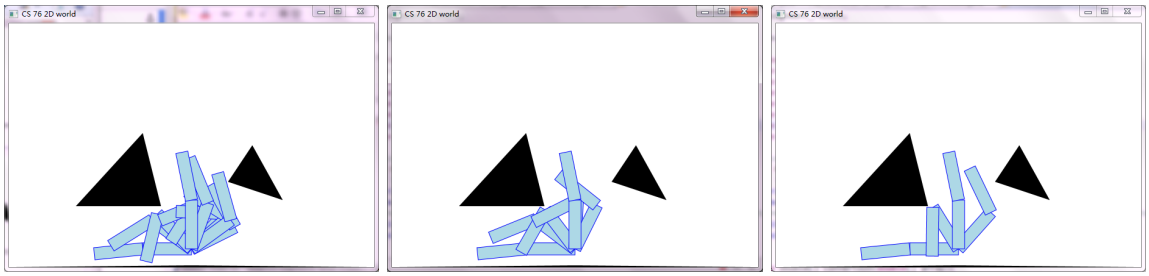


(a) k=15 with 100 samples (b) k=15 with 100 samples

Figure 1: robot arm in different worlds

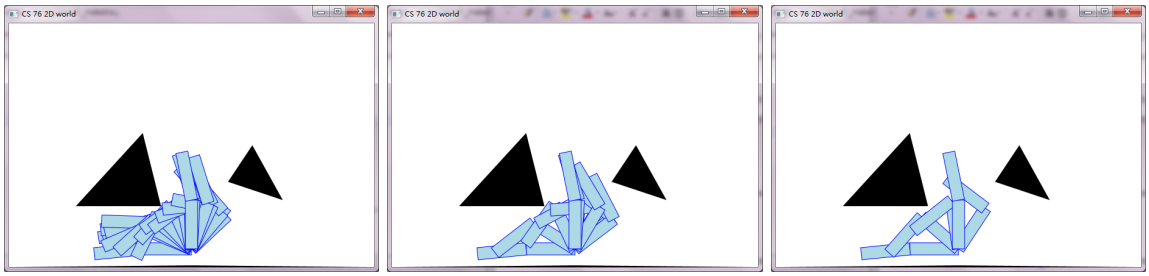
I find out that it wont collide any obstacle or world edge but its path is not optimal. It is because the graph may not have an edge between vertices on the optimal path.

Then I compare the different performance of different K. Here I chose K=5, 15, 55 for 100 and 300 samples.



(a) k=5 with 100 samples (b) k=15 with 100 samples (c) k=55 with 100 samples

Figure 2: robot arm move from pi to pi/2(100 samples with different k)



(a) k=5 with 300 samples (b) k=15 with 300 samples (c) k=55 with 300 samples

Figure 3: robot arm move from pi to pi/2(300 samples with different k)

I find out that the larger the K, the fewer the motion showed on path. Since each vertex has more neighbors, it will find more optimal and concise path. When I have more samples the path may tend to be more specific which means more robot arms will be showed.

1.5 Discussion

Why we choose K = 15?
According to the paper Sampling-based Algorithms for Optimal Motion Planning [4], it shows a figure about the cost of the best path in the k-nearest PRM algorithm, k=15 has the lower cost.

2 Rapidly Exploring Random Tree (RRT)– Steered Car

2.1 Introduction

This problem is about steered car motion planning be generating the RRT. The car can move in 6 directions: forwards, backwards, forwards turning counter-clockwise, backwards turning counter-clockwise, forwards turning clockwise, backwards turning clockwise.
There are many problems concerning the RRT generation, especially problem with choosing the nearest vertex in a world with tricky obstacles. The Planning Algorithm mentions[1] many scenarios but the algorithm it provides is too simple. Hence, I reference the Wikipedias version. [3] In algorithm, it first put the start configuration in the hashmap which represents an adjacent tree. Then it iterates K times, each time it adds a new node in hashmap. To add new node, it first randomly chooses a collision free node, and then find a nearest node to the chosen node, lastly it moves from the nearest node in a near distance to get to the new node. Here I choose a random direction to move since I can consider the random node I choose is in that

direction.
The query phase, I use the Bread First Search to find a path from goal; it may or may not get to the goal. If the graph doesnt contain the goal, it will move until there is no path to go. (Here I just want to plot a path). I also plot all the node in graph to show the result of RRT generator.

2.2 Code Design

For the steered car problem, I design a class `RRTPlanner` in addition to the provided skeleton code. Besides the construction functions, there are two public functions in this class: `prmGenerator` which generates the roadmap and `breadthFirstSearch` which plan the path starts from configuration. Moreover, it contains a comparators inner class to control the process of sorting and finding the nearest vertex. Furthermore, to make the `Hashmap` work, I override the equals and `hashcode` function in `CarState` class.

2.3 Implementation and How it Works

2.3.1 RRTPlanner

The implementation is based on the algorithm I mentioned above. Here I use the similar sort method as finding KNN did to find the nearest vertex in graph. I use `List<CarState> nodes`, an `ArrayList<CarState>` to store the vertices in graph. Every time I find a new node I should add it to the `ArrayList` of the father node.
The distance is defined as the Euclidian distance between two nodes which means it has no relationship with the theta.

Here is the code for the attributes and constructor of the class`RRTPlanner`.

```
public class RRTPlanner {
  public World world;
  public CarRobot car;
  public CarState start;
  public CarState goal;
  public int w_width;
  public int w_height;
  public CarState randstate = new CarState();
  CarState current_node=new CarState();

  public double step=2;
  public int K=2000;//K iteration

  public RRTPlanner(World W, CarRobot c, CarState s, CarState g, int w, int h)
      {
    world=W;
    car=c;
    start=s;
    goal=g;
    w_width=w;
    w_height=h;
  }
```

Here is the code for the attributes and constructor of the class`RRTGenerator`and its `comparator`.

```
public HashMap<CarState,ArrayList<CarState>> RRTGenerator(){
  HashMap<CarState,ArrayList<CarState>> adjlist = new HashMap<CarState,
      ArrayList<CarState>>();
  Random rng=new Random();//random number generator
  CarState state=new CarState();//get the random state before collision
      checking
  CarRobot carrobot=new CarRobot();
  //nodes is used to sort and find the nearest node.
  List<CarState> nodes = new ArrayList<CarState>();
  SteeredCar planner = new SteeredCar();

  nodes.add(start);
  nodes.add(goal);
  for(int iter=0; iter<K;){
    state.set(rng.nextDouble() *w_width, rng.nextDouble() *w_height, rng.
        nextDouble() * Math.PI * 2);
    carrobot.set(state);
    //check the random state is collision free
    if(!world.carCollision(carrobot)){
      CarState near=new CarState();
      CarState newstate=new CarState();

      randstate.set(state.getX(), state.getY(), state.getTheta());
      Collections.sort(nodes, new comparator());
      near.set(nodes.get(0).getX(), nodes.get(0).getY(), nodes.get(0).
          getTheta());

      int ctl=rng.nextInt(5);
      carrobot.set(near);
      //check the path collision
      if(!world.carCollisionPath(carrobot, near, ctl, step)){
        newstate= planner.move(near, ctl, step);

        carrobot.set(newstate);
        ArrayList<CarState> adjs = new ArrayList<CarState>();
        if(adjlist.containsKey(near)){
          adjs = adjlist.get(near);
          adjs.add(newstate);
          adjlist.remove(near);
          System.out.println(near+"has more than one neighbor");
        }
        else
          adjs.add(newstate);
        adjlist.put(near, adjs);
        nodes.add(newstate);
        iter++;
      }
    }
  }
  return adjlist;
```

```
}

public class comparator implements java.util.Comparator<CarState>{
  @Override
  public int compare(CarState c1, CarState c2) {
    // TODO Auto-generated method stub

    double d1 = Math.sqrt(Math.pow(c1.getX()-randstate.getX(), 2)+Math.pow(
        c1.getY()-randstate.getY(), 2));
    double d2 = Math.sqrt(Math.pow(c2.getX()-randstate.getX(), 2)+Math.pow(
        c2.getY()-randstate.getY(), 2));
    return (int) Math.signum(d1 - d2);
  }
}
```

Here is the code for the attributes and constructor of the class`breadthFirstSearch`and `backchain`.

```
public List<CarState> breadthFirstSearch(HashMap<CarState,ArrayList<
    CarState>> adjList){

  int i;
  HashMap<CarState,CarState> explored=new HashMap<CarState,CarState>();//
      explored:tracking the explored nodes
  List<CarState> path=new ArrayList<CarState>();//the path to goal
  Queue<CarState> queue = new LinkedList<CarState>();

  queue.add(start);//initializing, add start node in queue
  explored.put(start,null);

  while(!queue.isEmpty()) {
    //dequeue the head of queue to be the current node
    current_node = (CarState)queue.remove();

    if(current_node.equals(goal)){
      path=backchain(current_node, explored);
      System.out.println("get to the goal");
    }
    else{
      //add unexplored successors to fringe
      ArrayList<CarState> successors= new ArrayList<CarState>();
      successors=adjList.get(current_node);
      if(successors!=null){
        for(i=0;i<successors.size();i++) {

          if(explored.containsKey(successors.get(i))==false){
            explored.put(successors.get(i),current_node);
            queue.add(successors.get(i));
          }
```

```
            }
          }
        }
      }
      path=backchain(current_node, explored);
      return path;
    }

    // backchain should only be used by bfs, not the recursive dfs
    private List<CarState> backchain(CarState node,
        HashMap<CarState, CarState> visited) {
      // you will write this method
      List<CarState> path=new ArrayList<CarState>();

      while(node!=null){
        path.add(node);
        node=visited.get(node);
      }
      Collections.reverse(path);
      return path;
    }
```

2.3.2 equals and hashcode function in CarState class

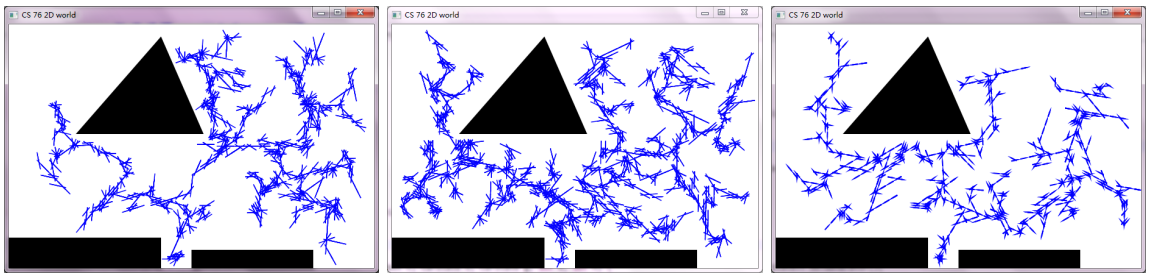
The original equals and `hashcode` function will cause the missing key in `hashmap` since either the array or the `CarState` class is a mutable key for the `hashmap`. Hence, I override the equals method to compare each configuration in the `CarState`. The `hashcode` is the linear equation of three configurations: s[0]*10+s[1]*1+s[2]*100. I also override the toString method which is used in debugging.

Here is the code for the `toString`, `equals` and `hashcode`function in `CarState` class

```
  @Override
  public String toString() {
    // TODO Auto-generated method stub
    String str=s[0]+" "+s[1]+" "+s[2]+" ";
    return str;
  }
  @Override
  public boolean equals(Object obj) {
    // TODO Auto-generated method stub
    return s[0]== ((CarState)obj).getX()&&
          s[1]== ((CarState)obj).getY()&&
          s[2]== ((CarState)obj).getTheta();
  }
  @Override
  public int hashCode() {
    // TODO Auto-generated method stub

    return (int) (s[0]*10+s[1]*1+s[2]*100);
  }
```

2.4 Testing and Discussion

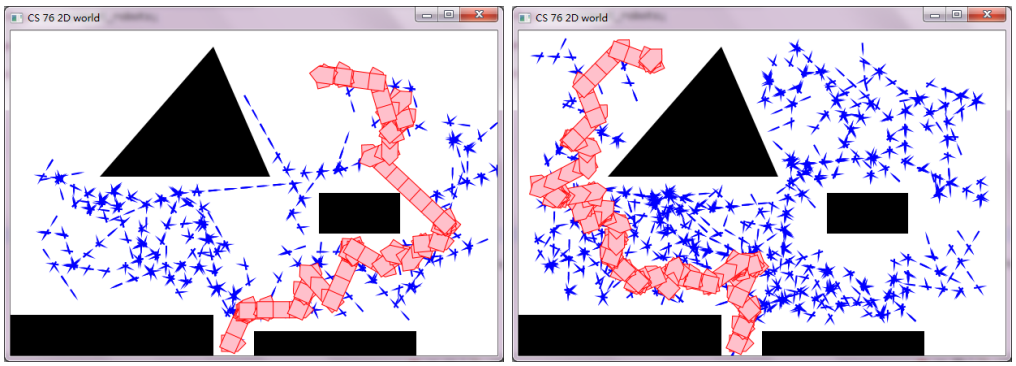
Here I draw all the nodes in RRT by drawing a smaller needle with its head pointing to its direction. It shows clearly what the tree looks like. Following are the RRTs with different duration time the car move from nearest node and different K iterations.



(a) time=1 K=2000 (b) time=1 K=3000 (c) time=1.5 K=1500

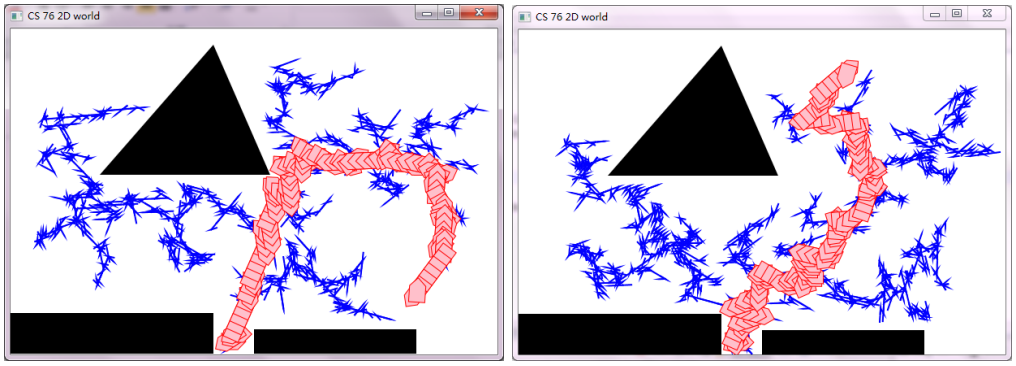
Figure 4: RRTs with different durations time and iterations K

It shows that the larger the duration, the sparser the tree will be. And the larger the iteration K, the more details will show on tree.
Then I compare the performance in different world, in both case the car have no collision with obstacles and edges and it works well in finding a path.



(a) K=1000 (b) K=2000

Figure 5: RRTs with time=2 and different iterations K in world 1



(a) K=2000 (b) K=3000

Figure 6: RRTs with time=1 and different iterations K in world 2

The path it moves is not optimal since there might not be an edge between two nodes in the tree.

3 Related Work(I am a graduate student)

I read the sampling-based algorithms for optimal motion planning [4]
The paper mainly analyzes asymptotic behavior of the cost of the sampling-based algorithms and introducing two optimal algorithms PRM* and RRT*. It first introduces the sampling- based algorithms PRM and RRT and their extension algorithms. It then compares the cost and complexity of different algorithms. Proving that the PRM* and RRT* works better than the traditional algorithms. Following that it shows many theorems and proves the optimality of the new algorithms.
PRM* and RRT* both modify the origin algorithms by using a variable connection radius that scale as log(n)/n, where n is the number of the samples. Hence, the average number of connection made at each iteration is proportional to log(n).

References

[1] sampling-based roadmaps, Planning Algorithm. `http://planning.cs.uiuc.edu/node239.html`.

[2] convert double array to list `http://stackoverflow.com/questions/5178854/convert-a-double-array-to-double-arraylist`

[3] Rapidly exploring random tree, Wikipedia. `http://en.wikipedia.org/wiki/Rapidly_exploring_random_tree`.

[4] Sertac Karaman and Emilio Frazzoli, "sampling-based algorithms for optimal motion planning" `http://sertac.scripts.mit.edu/web/wp-content/papercite-data/pdf/karaman.frazzoli-ijrr11.pdf`

[5] iterate hashmap `http://stackoverflow.com/questions/1066589/java-iterate-through-hashmap`