Motion Planning

Mengjia Kong

February 4, 2014

1 Introduction

In this report, I will introduce two problems and implement two motion planners for two systems. The first problem is a planar robot arm with state given by the angles of the joints, which will be dealt with Probabilistic Roadmap(PRM). And the other one is a steered 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, which will be solved by Rapidly Exploring Random Tree(RRT).

In the next section, there is some related work before the experiments. In section 3, I will introduce the arm robot with PRM algorithm. In section 4, RRT will be introduced with a steered car problem.

2 Related work

Probabilistic Roadmaps(PRM) is a new motion panning method for robots in static workspaces. The method proceeds in two phases: a learning phase and a query phase[1]. In the learning phase, PRM constructs a probabilistic roadmap as a collection of random configurations. In the query phase, it uses this roadmap to process path planning queries quickly. But there is an important problem, how PRM scales up when scens with more complicated geometry, due to the cost of collision checking will be much higher.

Rapidly-Exploring Random Tree(RRT) is designed for a broad class of path planning problems as a randomized data structure[2]. While many the beneficial properties of existing randomized planning techniques are shared, RRTs are specifically designed to handle nonholonomic constraints and high degrees of freedom. The key of RRT is to generate a random tree and search it to get the result. However, in random tree generation, efficient nearest-neighbor techniques are needed, which has been a topic of active interest in computational geometry.

3 Probabilistic Roadmap

The *Probabilistic Roadmap*[1] planner is a motion planning algorithm in robotics, which solves the problem of dertermining a path between a starting configuration of the robot and a goal configuration while avoiding collisions.

The basic idea of PRM is to take random samples from the configuration space of the robot, selecting some in free space without collisions. and use a local planner to connect these configurations to other nearby configurations. Then a graph search algorithm is applied to the resulting graph to get a path between the starting and goal configurations.

The probabilistic roadmap planner consists of two phases:

1. Roadmap generation, in which a graph is created.
2. Query phase, in which specific paths from start to goal are created.

3.1 Implementation of PRM

I created a class in PRM.java which extends InformedSearchProblem that is implemented in last report. Here's my code in PRM.java. After giving a big picture, I will explain the code in details.

```
public class PRM extends InformedSearchProblem {
	private World w;
	ArmLocalPlanner ap;

	private final int numofVertices = 500;
	// number of neighbors
	private final int numofNeighbors = 15;
	// this is the initial config;
	private double[] startConfig;
	private double[] goalConfig;

	ArrayList<double[]> configs;
	HashMap<double[], ArrayList<Edge>> roadMap;

	public PRM(double[] c1, double[] c2, World wld) {
		startConfig = c1;
		goalConfig = c2;
		w = wld;
		ap = new ArmLocalPlanner();

		configs = new ArrayList<double[]>();
		configs.add(c1);
		configs.add(c2);

		roadMap = new HashMap<double[], ArrayList<Edge>>();
		roadMap.put(c1, new ArrayList<Edge>());
		roadMap.put(c2, new ArrayList<Edge>());
	}

	public List<SearchNode> PRMPlanner() {
		build_RoadMap();

		startNode = new ConfigNode(startConfig, 0, null);
		return astarSearch();
	}

	public void build_RoadMap() {
		System.out.println("build_RoadMap!!!");

		int i = 2;
		ArmRobot tmpArm = new ArmRobot(startConfig.length/2-1);

		while (i < numofVertices) {
			//generate a random vertex
			double[] v = genVertex();
			tmpArm.set(v);
			//if the vertex is not collided with obstacles.
			//                and not already in roadMap
			if (!w.armCollision(tmpArm) && !roadMap.containsKey(v)) {
				configs.add(v);
				i++;
```

```java
                PriorityQueue<Edge> pq = new PriorityQueue<Edge>();
                for (double[] cfg: configs) {
                    //cannot add a circle v-v
                    if (Arrays.equals(cfg, v))
                        continue;
                    if (!w.armCollisionPath(tmpArm, v, cfg)) {
                        double cost = ap.moveInParallel(v, cfg);
                        pq.add(new Edge(cfg, cost));
                    }
                }
                ArrayList<Edge> list = new ArrayList<Edge>();
                while(list.size() <= numofNeighbors && !pq.isEmpty())
                    list.add(pq.remove());
                roadMap.put(v, list);

                //add symmetrical edge
                for (Edge e: list) {
                    Edge eN = new Edge(v, e.time);
                    ArrayList<Edge> tmpPQ = roadMap.get(e.config);
                    if (!tmpPQ.contains(eN)) {
                        tmpPQ.add(eN);
                    }
                }
            }

        }

    }

    private double[] genVertex() {
        double[] vertex = new double[startConfig.length];
        vertex[0] = startConfig[0];
        vertex[1] = startConfig[1];

        for (int i = 2; i < startConfig.length; i++) {
            if (i % 2 == 0)
                vertex[i] = startConfig[i];
            else
                vertex[i] = Math.random()*2*Math.PI;
        }

        return vertex;
    }

    public ArrayList<double[]> getRoadMap() {
        return configs;
    }

    class Edge implements Comparable<Object> {
        double[] config;
        // time when moving through the Edge
        double time;

        public Edge(double[] cfg, double t) {
            config = cfg;
            time = t;
```

```java
		}


		@Override
		public int compareTo(Object o) {
			return (int) Math.signum(time - ((Edge)o).time);
		}

		@Override
		public boolean equals(Object other) {
			return Arrays.equals(config, ((Edge) other).config);
		}

		@Override
		public int hashCode() {
			int hash = 0;
			for (int i = 3; i < config.length; i = i+2)
				hash = hash* 1000 + (int)(config[i]*100);
			return hash;
		}

	}


	class ConfigNode implements SearchNode{
		double[] config;
		double cost;
		// for backchain
		private SearchNode parent;

		ConfigNode(double[] cfg, double c, SearchNode pa) {
			config = cfg;
			cost = c;
			parent = pa;
		}

		public double[] getCFG() {
			return config;
		}

		@Override
		public int compareTo(SearchNode o) {
			return (int) Math.signum(priority() - ((ConfigNode)o).priority());
		}

		@Override
		public ArrayList<SearchNode> getSuccessors() {

			ArrayList<SearchNode> successors = new ArrayList<SearchNode>();

			for (Edge succ: roadMap.get(config)) {
				SearchNode node = new ConfigNode(succ.config,
						this.cost + succ.time, this);
				successors.add(node);

			}
```

```
                        return successors;
                }

                @Override
                public boolean goalTest() {
                        return Arrays.equals(config, goalConfig);
                }

                @Override
                public boolean equals(Object other) {
                        return Arrays.equals(config, ((ConfigNode) other).config);
                }

                @Override
                public int hashCode() {
                        int hash = 0;
                        for (int i = 3; i < config.length; i = i+2)
                                hash = hash* 1000 + (int)(config[i]*100);
                        return hash;
                }

                @Override
                public String toString() {
                        return new String("Config " + config[3] + ", " + config[5] + " "
                                        + " prior " + priority());
                }

                @Override
                public double getCost() {
                        return cost;
                }

                @Override
                public SearchNode getParent() {
                        return parent;
                }

                @Override
                public double heuristic() {

                        return 0;
                }

                @Override
                public double priority() {
                        return heuristic() + getCost();
                }

        }

}
```

The main approach is in this function:

```
        public List<SearchNode> PRMPlanner() {
                build_RoadMap();
```

```java
3
4         startNode = new ConfigNode(startConfig, 0, null);
5         return astarSearch();
6     }
```

The line 2 is the first phase in PRM, generate a roadmap. the line 5 is the second phase, use a graph search to get the path.

```java
1     public void build_RoadMap() {
2         System.out.println("build_RoadMap!!!");
3
4         int i = 2;
5         ArmRobot tmpArm = new ArmRobot(startConfig.length/2-1);
6
7         while (i < numofVertices) {
8             //generate a random vertex
9             double[] v = genVertex();
10            tmpArm.set(v);
11            //if the vertex is not collided with obstacles.
12            //                and not already in roadMap
13            if (!w.armCollision(tmpArm) && !roadMap.containsKey(v)) {
14                configs.add(v);
15                i++;
16                PriorityQueue<Edge> pq = new PriorityQueue<Edge>();
17                for (double[] cfg: configs) {
18                    //cannot add a circle v-v
19                    if (Arrays.equals(cfg, v))
20                        continue;
21                    if (!w.armCollisionPath(tmpArm, v, cfg)) {
22                        double cost = ap.moveInParallel(v, cfg);
23                        pq.add(new Edge(cfg, cost));
24                    }
25                }
26                ArrayList<Edge> list = new ArrayList<Edge>();
27                while(list.size() <= numofNeighbors && !pq.isEmpty())
28                    list.add(pq.remove());
29                roadMap.put(v, list);
30
31                //add symmetrical edge
32                for (Edge e: list) {
33                    Edge eN = new Edge(v, e.time);
34                    ArrayList<Edge> tmpPQ = roadMap.get(e.config);
35                    if (!tmpPQ.contains(eN)) {
36                        tmpPQ.add(eN);
37                    }
38                }
39            }
40        }
41    }
```

The `build_RoadMap` is the key of the PRM. Before run the function, I have initiate `roadMap`.

1. Use a loop before getting enough vertices for the `roadMap`. I have set the num as `numofVertices` in the method.
2. Generate a random vertex in line 9

3. If the random vertex does not collid with obstacles and has not already in `roadMap`, go to next step; else, go to step 2.

4. Add the vertex into the roadMap.

5. Find K nearest vertices in roadMap of the random vertex. Make sure the path from the random vertex the these vertices will not collid with any obstacles. K is a number in my method as `numofNeighbors`.

6. Add edge from the random vertex to the K nearest vertices in the roadMap. Because the graph is undirected graph, I need to make sure the two vertices in an edge connect to another one.

In my method, I use two variables to represent roadMap. The first one is `ArrayList<double[]> configs;`. It represents all vertices in roadMap. The second one is `HashMap<double[], ArrayList<Edge>> roadMap;`, which represents all edges in roadMap. `roadMap` use a map from a vertex to its adjacent vertices.

To get the K nearest vertices in roadMap. I created a new class `Edge` to store `config` and the length of the edge which is measured by `time`. The `Edge` is `implements Comparable<Object>`. So by overriding a `compareTo` function, I can easily use `priorityQueue` to get the K nearest vertices. Because it is a class, so I need to override `equals` and `hashCode`as well.

About the graph search, I use the `astarSearch` which is implemented in the last assignment.

3.2 Implementation of the model

In the provided codes, `ArmLocalPlanner, ArmRobot` that is about the features of the arm robot and `World, CollisionChecker, Poly` that is about the environment and collision checking have been implemented. So I will not explain this part in this report for the length of the report.

For the model, I only implemented the `ConfigNode` which `implements SearchNode`. This is used for `searchNode` in search algorithm. Like before, I defined state as `double[] config`, `SearchNode parent` and `double cost` which means the cost from starting.

The basic idea of `getSuccessors` is to get the adjacent vertices of this vertex, which is stored in `roadMap`. I only need to created new nodes by configs from the roadMap in Key: config of this vertex and a new cost. Here's my code for `getSuccessors`:

```
	public ArrayList<SearchNode> getSuccessors() {

		ArrayList<SearchNode> successors = new ArrayList<SearchNode>();

		for (Edge succ: roadMap.get(config)) {
			SearchNode node = new ConfigNode(succ.config,
					this.cost + succ.time, this);
			successors.add(node);

		}
		return successors;
	}
```

The`goalTest` here is whether the config is equal.

```
	public boolean goalTest() {
		return Arrays.equals(config, goalConfig);
	}
```

3.3 Experiments and discussion

Obviously, there are two paraments we need to discuss about: the number of random vertices(`numofVertices` in my method, for short, N), the number of neighbors we find for adjacent vertices(`numofNeighbors` in my method, for short, K).

In my experimental results, I will show some pictures. In these pictures, the blue two is starting and goal configurations, the green arms are the vertices in path between starting and goal configurations and the orange arms are all random vertices. The black polygons in the world are obstacles.

3.3.1 N = 50, K = 15

There are some different results when N = 50 and K = 15, as figures 1 2.

In figure 1, the starting configuration is the up one and the goal configuration is the down one. The selected path is counter clockwise.

In figure 2, the starting configuration is the down one and the goal configuration is the up one. The selected path is clockwise.

Because there is a obstacle in the right side, the shortest path is always in the left side.

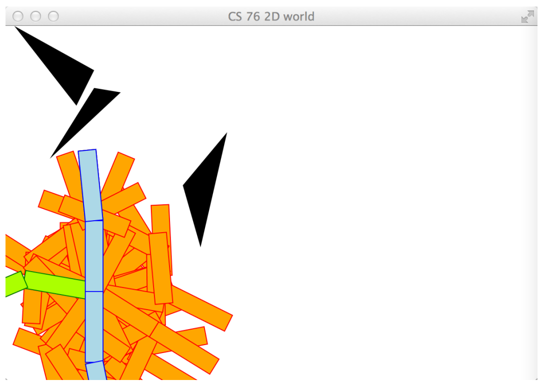

Figure 1: N = 50, K = 15, result 1

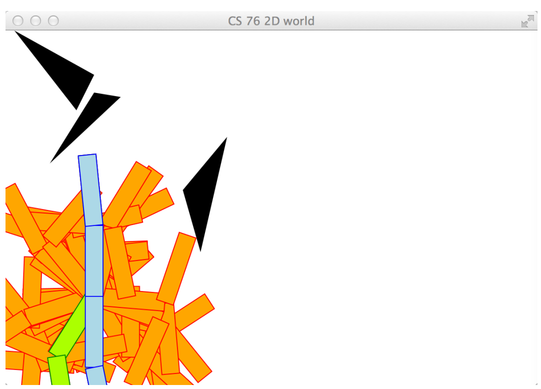

Figure 2: N = 50, K = 15, result 2

3.3.2 N = 500, K = 15

There are some different results when N = 500 and K = 15, as figures 3 4 5.

In figure 3, the starting configuration is the right one and the goal configuration is the left one. The selected path is counter clockwise. In figure 3, there are more searchNode in the path. It is the complicated goal configuration, fold arm robot, that leads to more searchNode is needed for the shortest path.

In figure 4, the starting configuration is the down one and the goal configuration is the up one. The selected path is clockwise. The figure 4 also shows that for the shortest path, the algorithm will select to hide from the obstacle.

In figure 5, the starting configuration is right one and the goal configuration is the left one.

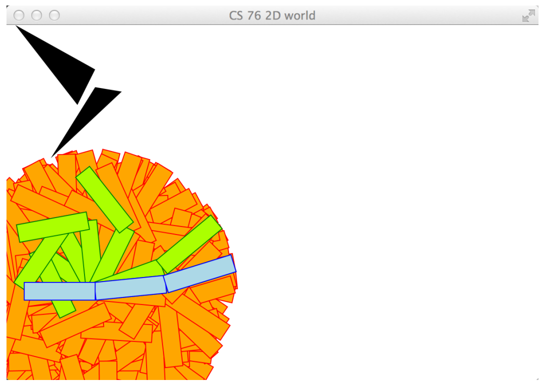

Figure 3: N = 500, K = 15, result 1

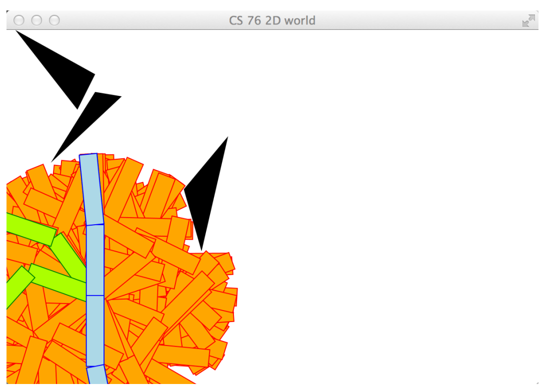

Figure 4: N = 500, K = 15, result 2

3.3.3 N = 1000, K = 15

There are some different results when N = 1000 and K = 15, as figures 6 7 8. In all figures, the starting configuration is the up one and the goal configuration is the down one.

In a conlusion, when N is increasing, there are more and more random vertices in roadMap. But actually, the increasing N has no relation with the number of searchNodes in the resulting path. Because the shortest path has nothing with the number of searchNodes in the path. The measurement of the cost of the path is the time, the sum of time between two adjacent vertices in the path. For example, we get a path which has 3 nodes including start and goal. The cost of the path is the sum of time between start and vertex and time between vertex and goal. If there are another path, there are 4 vertices in the path. But the whole moving path is the same with the previous one with 3 nodes. Then they have the same cost. If the path with 4 vertices has a different moving path with the previous one. It may cost more time than the path with fewer nodes.

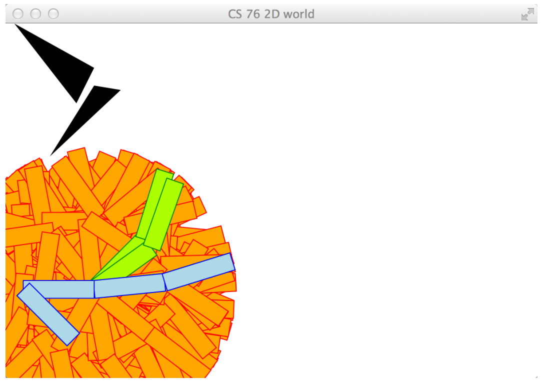

Figure 5: N = 500, K = 15, result 3

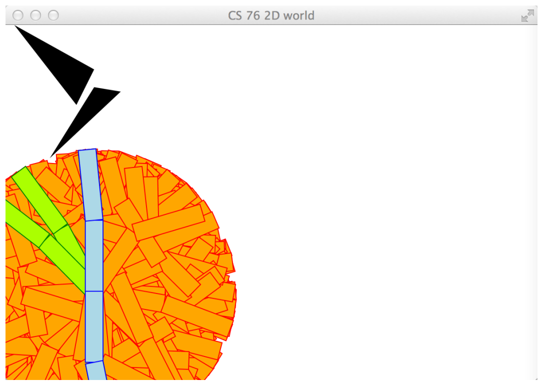

Figure 6: N = 1000, K = 15, result 1

Moreover, when N is increasing, the path are more likely shortest. Because we get vertices in roadmap randomly. If we get more vertices, it is more likely to get the vertice in the shortest moving path.

3.3.4 N = 500, K = 50

There are some different results when N = 500 and K = 50, as figures 9 10. In all figures, the starting configuration is the up one and the goal configuration is the down one.

When K is increasing, it is more likely to get fewer nodes in the resulting path. Because when K is greater, every vertex will have more adjacent vertices. Then we will get the goal with going through fewer nodes.

4 Rapidly Exploring Random Tree

A *Rapidly-exploring RandomTree(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.

The probabilistic roadmap planner consists of two phases:

1. Random Tree generation, in which a graph is created.

2. Query phase, in which specific paths from start to goal are created.

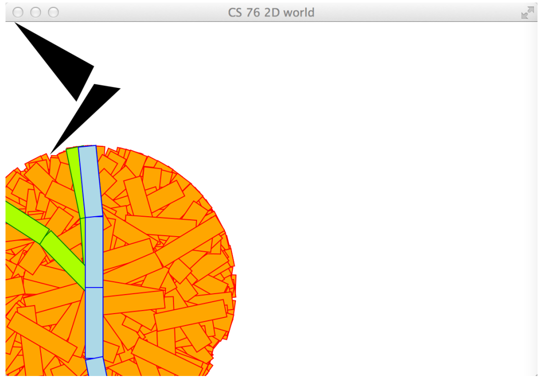

Figure 7: N = 1000, K = 15, result 2

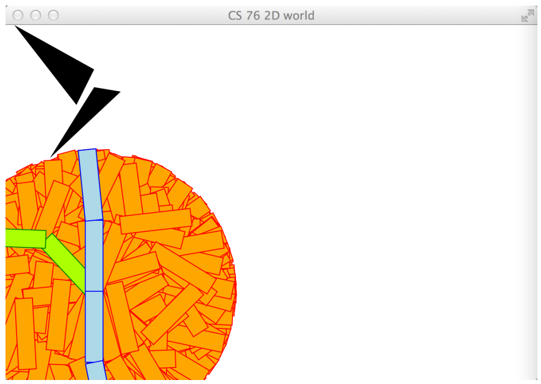

Figure 8: N = 1000, K = 15, result 3

4.1 Implementation of RRT

I created a class in `RRT.java` which `extends InformedSearchProblem` that is implemented in last report. Here's my code in `RRT.java`. After giving a big picture, I will explain the code in details.

```
public class RRT extends InformedSearchProblem{
	private World w;
	private SteeredCar scar;
	private CarRobot cr;

	private final int numOfVertices = 500;
	private final int MIN = 10;
	private ArrayList<CarState> carSts;
	private HashMap<CarState, ArrayList<CarState>> randomTree;

	private CarState startSt, goalSt;

	public RRT(CarState sCSt, CarState gCSt, World wld) {
		w = wld;
		scar = new SteeredCar();
		cr = new CarRobot();
		startSt = sCSt;
		goalSt = gCSt;
		carSts = new ArrayList<CarState>();
```

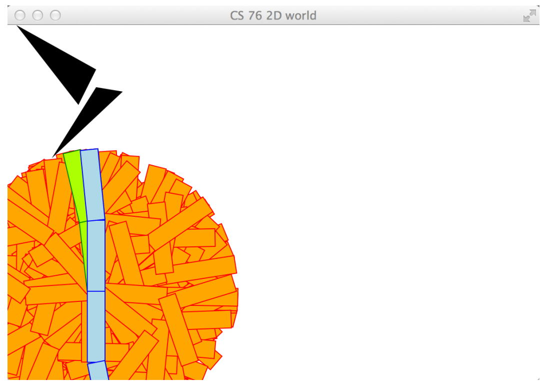

Figure 9: N = 500, K = 50, result 1

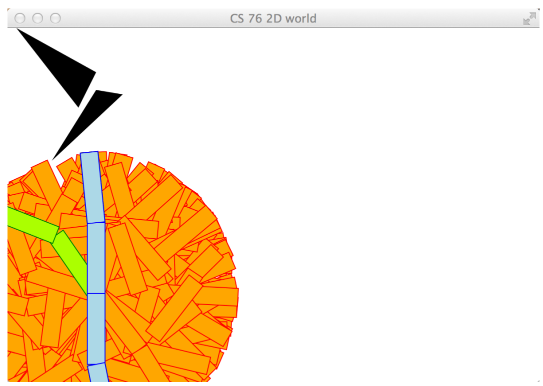

Figure 10: N = 500, K = 50, result 2

```
            carSts.add(sCSt);
            randomTree = new HashMap<CarState, ArrayList<CarState>>();
            randomTree.put(startSt, new ArrayList<CarState>());
    }

    public List<SearchNode> RRTPlanner() {
            Build_RRT();

            startNode = new CarStNode(startSt, 0, null);
            return astarSearch();
    }

    public ArrayList<CarState> getRRT() {
            return carSts;
    }

    public void Build_RRT() {
            for(int i = 0; i < numOfVertices; i++) {
                    CarState carst = genCarSt();
                    CarState nearest = findMin(carst, carSts);

                    // 6 controls
                    ArrayList<CarState> list = new ArrayList<CarState>();
```

```
                        double time = 1;
                        for (int j = 0; j < 6; j++) {
                                CarState carstNew = scar.move(nearest, j, time);
                                cr.set(carstNew);
                                if (!w.carCollision(cr) && !w.carCollisionPath(cr, nearest, j, time))
                                        list.add(carstNew);
                        }
                        CarState nearestNew = findMin(carst, list);
                        carSts.add(nearestNew);

                        //add edge
                        list = randomTree.get(nearest);
                        if (!list.contains(nearestNew))
                                list.add(nearestNew);
                        if (randomTree.containsKey(nearestNew)) {
                                list = randomTree.get(nearestNew);
                                if (!list.contains(nearest))
                                        list.add(nearest);
                        }
                        else {
                                list = new ArrayList<CarState>();
                                list.add(nearest);
                                randomTree.put(nearestNew, list);
                        }
                }
        }

        private CarState genCarSt() {
                CarRobot cr = new CarRobot();
                CarState carst = new CarState();

                while(w.carCollision(cr)) {
                        double x = Math.random()*600;
                        double y = Math.random()*400;
                        double tt = Math.random()*Math.PI;

                        carst.set(x, y, tt);
                        cr.set(carst);
                }

                return carst;
        }

        private CarState findMin(CarState cSt, ArrayList<CarState> list) {
                double minD = 10000;
                CarState minCST = null;
                for (CarState cst: list) {
                        double tmpD = Math.sqrt(Math.pow(cSt.getX() - cst.getX(),2) +
                            Math.pow(cSt.getY() - cst.getY(),2));
                        if (tmpD < minD ) {
                                minD = tmpD;
                                minCST = cst;
                        }
                }
                return minCST;
        }
```

```
	class CarStNode implements SearchNode{
		CarState cSt;
		double cost;
		// for backchain
		private SearchNode parent;

		CarStNode(CarState cst, double c, SearchNode pa) {
			cSt = new CarState(cst.getX(), cst.getY(), cst.getTheta());
			cost = c;
			parent = pa;
		}

		public CarState getCST() {
			return cSt;
		}

		@Override
		public int compareTo(SearchNode o) {
			return (int) Math.signum(priority() - ((CarStNode)o).priority());
		}

		@Override
		public ArrayList<SearchNode> getSuccessors() {

			ArrayList<SearchNode> successors = new ArrayList<SearchNode>();

			for (CarState cst: randomTree.get(cSt)) {
				double tmpD = Math.sqrt(Math.pow(cSt.getX() - cst.getX(),2) +
				    Math.pow(cSt.getY() - cst.getY(),2));
				SearchNode node = new CarStNode(cst, cost + tmpD, this);
				successors.add(node);

			}
			return successors;
		}

		@Override
		public boolean goalTest() {
			double d = Math.sqrt(Math.pow(cSt.getX() - goalSt.getX(),2) +
			    Math.pow(cSt.getY() - goalSt.getY(),2));
			return (d <= 10);
		}

		// an equality test is required so that visited sets in searches
		// can check for containment of states
		@Override
		public boolean equals(Object other) {
			return Arrays.equals(cSt.get(), ((CarStNode) other).cSt.get());
		}

		@Override
		public int hashCode() {
			return cSt.hashCode();
		}
```

```java

        @Override
        public String toString() {
            return new String("CarSt " + cSt.getX() + ", " + cSt.getY() + ", " +
                cSt.getTheta()
                        + " prior " + priority());
        }

        @Override
        public double getCost() {
            return cost;
        }

        @Override
        public SearchNode getParent() {
            return parent;
        }

        @Override
        public double heuristic() {

            return 0;
        }

        @Override
        public double priority() {
            return heuristic() + getCost();
        }
    }
}
```

The main approach is in this function:

```java
1       public List<SearchNode> RRTPlanner() {
2               Build_RRT();
3
4               startNode = new CarStNode(startSt, 0, null);
5               return astarSearch();
6       }
```

The line 2 is the first phase in RRT, generate a randomTree. the line 5 is the second phase, use a graph search to get the path.

```java
    public void Build_RRT() {
        for(int i = 0; i < numOfVertices; i++) {
            CarState carst = genCarSt();
            CarState nearest = findMin(carst, carSts);

            // 6 controls
            ArrayList<CarState> list = new ArrayList<CarState>();
            double time = 1;
            for (int j = 0; j < 6; j++) {
                CarState carstNew = scar.move(nearest, j, time);
                cr.set(carstNew);
                if (!w.carCollision(cr) && !w.carCollisionPath(cr, nearest, j, time))
                    list.add(carstNew);
```

```
14              }
15              CarState nearestNew = findMin(carst, list);
16              carSts.add(nearestNew);
17              System.out.println("new vertex " + nearestNew.getX() + ", " +
                    nearestNew.getY());
18
19              //add edge
20              list = randomTree.get(nearest);
21              if (!list.contains(nearestNew))
22                  list.add(nearestNew);
23              if (randomTree.containsKey(nearestNew)) {
24                  list = randomTree.get(nearestNew);
25                  if (!list.contains(nearest))
26                      list.add(nearest);
27              }
28              else {
29                  list = new ArrayList<CarState>();
30                  list.add(nearest);
31                  randomTree.put(nearestNew, list);
32              }
33          }
34      }
```

he `Build_RRT` is the key of the RRT. Before run the function, I have initiate `randomTree`.

1. Use a loop before getting enough vertices for the `randomTree`. I have set the num as `numofVertices` in the method.
2. Generate a random vertex in line 3. I have checked whether the vertex will collided with obstacles in the function of generate a random vertex `genCarSt`.
3. Find the nearest vertex of the random vertex in `randomTree`.
4. The nearest vertex executes the actions, in the car robot problem, it is steering the car in 6 directions. Then get 6 new vertices. Find the nearest vertex of the random vertex from the 6 vertices.
5. Add the vertex into the randomTree.
6. Add edge from the random vertex to the new vertex in the randomTree. Because the graph is undirected graph, I need to make sure the two vertices in an edge connect to another one.

In my method, I use two variables to represent randomTree like roadMap in the last section. The first one is `ArrayList<CarState> carSts;`. It represents all vertices in randomTree. The second one is `HashMap<CarState, ArrayList<CarState>> randomTree;`, which represents all edges in randomTree. `randomTree` use a map from a vertex to its adjacent vertices.

To get the nearest vertex in randomTree. I created a new function `findMin`.

About the graph search, I use the `astarSearch` which is implemented in the last assignment.

4.2 Implementation of model

In the provided codes, `CarRobot, CarState` that is about the features of the arm robot and `World, CollisionChecker, Poly` that is about the environment and collision checking have been implemented. So I will not explain this part in this report for the length of the report.

For the model, I implemented the `CarStNode` which `implements SearchNode`. This is used for `searchNode` in search algorithm. Like before, I defined state as `CarState cSt`, `SearchNode parent` and `double cost` which means the cost from starting.

The basic idea of `getSuccessors` is to get the adjacent vertices of this vertex, which is stored in `randomTree`. I only need to created new nodes by CarState from the randomTree in Key: carState of this node and a new cost. Here's my code for `getSuccessors`:

```
		public ArrayList<SearchNode> getSuccessors() {

			ArrayList<SearchNode> successors = new ArrayList<SearchNode>();

			for (CarState cst: randomTree.get(cSt)) {
				double tmpD = Math.sqrt(Math.pow(cSt.getX() - cst.getX(),2) +
				    Math.pow(cSt.getY() - cst.getY(),2));
				SearchNode node = new CarStNode(cst, cost + tmpD, this);
				successors.add(node);
			}

			return successors;
		}
```

The`goalTest` here is whether the distance between this carState and goal is less than a fixed number.

```
		public boolean goalTest() {
			double d = Math.sqrt(Math.pow(cSt.getX() - goalSt.getX(),2) +
			    Math.pow(cSt.getY() - goalSt.getY(),2));
			return (d <= MIN);
		}
```

Because I want to use CarState in HashMap, so I added some functions in `CarState` as below:

```
	@Override
	public int hashCode() {
		return (int)(s[2] * 10000000 + s[1]*1000 + s[0]);
	}

	@Override
	public boolean equals(Object other) {
		return Arrays.equals(s, ((CarState) other).get());
	}

	@Override
	public String toString() {
		return new String("CarSt " + s);
	}
```

4.3 Experiments and discussion

Obviously, there are two paraments we need to discuss about: the number of random vertices(`numofVertices` in my method, for short, N), the limited distance between the goal node in the path and the real node(`MIN` in my method).

In my experimental results, I will show some pictures. In these pictures, the red two is starting and goal carStates, the blue cars are the vertices in path between starting and goal configurations and the green arms are all random vertices. The black polygons in the world are obstacles. The starting carState is always the one in the middle of bottom.

4.3.1 N = 50, MIN = 10

There are some different results when N = 50 and MIN = 10, as figures 11 12.

Because both N and MIN are too small. I always cannot find the path.

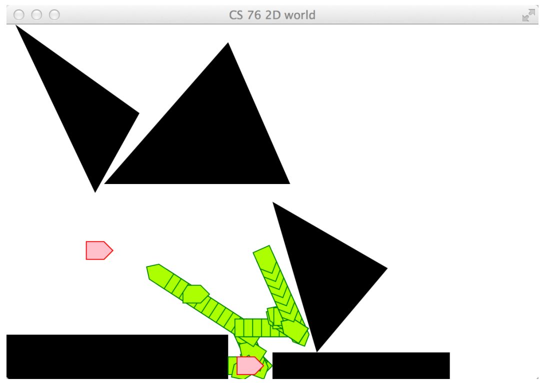

Figure 11: N = 50, MIN = 10, result 1

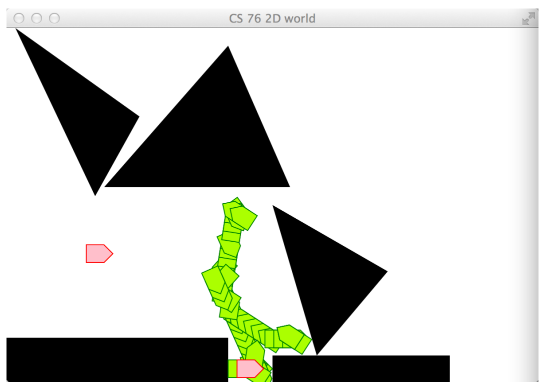

Figure 12: N = 50, MIN = 10, result 2

4.3.2 N = 500, MIN = 10

There are some different results when N = 500 and MIN = 10, as figures 13 14 15 16.

I can find that when the goal is close to the start, I always can find the path, but when I move the goal to far away, I also can find the path, but it is seldom. I tried many times to find the path like Figure 16. Most time, I cannot find the path like Figure 15. Obviously, the random Tree does not have a vertex close to the goal. But according to the člosеї̈s defined by MIN, if I increase the MIN, it will be more possible to find a path.

4.4 N = 1000, MIN = 10

There are some different results when N = 1000 and MIN = 10, as figures 17 18.

Obviously, when N is increasing, there are more vertices in the randomTree. So it is more possible to find a path, even the goal is very far away from the start.

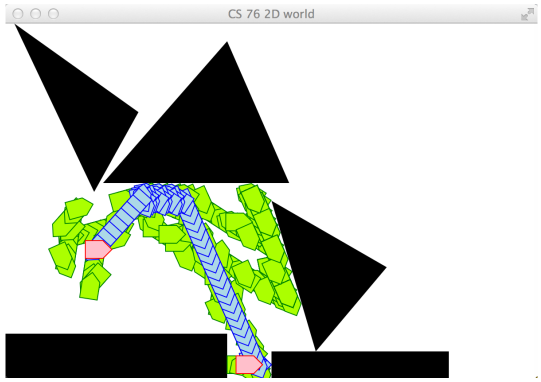

Figure 13: N = 500, MIN = 10, result 1

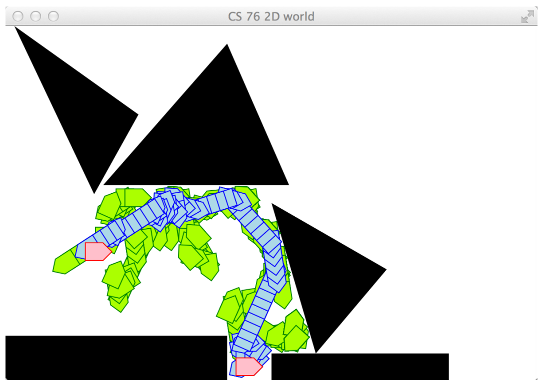

Figure 14: N = 500, MIN = 10, result 2

4.5 N = 500, MIN = 50

There are some different results when N = 500 and MIN = 50, as figures 19 20.

when MIN is increasing, it is more possible to find a path, even the goal is very far away from the start. You can see in Figure 19. The goal node in the path is not so close to the goal carState. But that's what I defined.

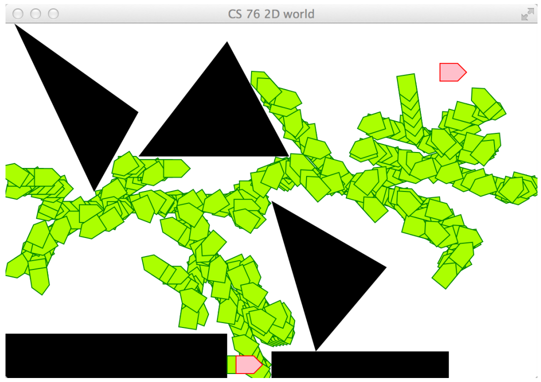

Figure 15: N = 500, MIN = 10, result 1

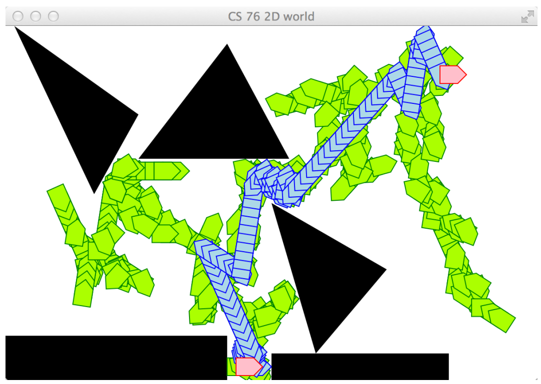

Figure 16: N = 500, MIN = 10, result 2

References

[1] Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; Overmars, M. H. (1996), P̈robabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Transactions on Robotics and Automation 12 (4): 566580, doi:10.1109/70.508439

[2] LaValle, Steven M. (October 1998). R̈apidly-exploring random trees: A new tool for path planning. Technical Report (Computer Science Deptartment, Iowa State University) (TR 98-11).

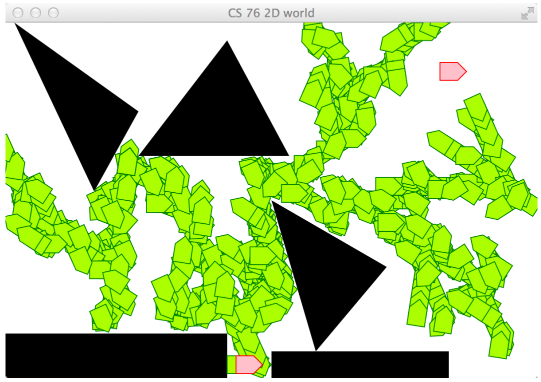

Figure 17: N = 1000, MIN = 10, result 1

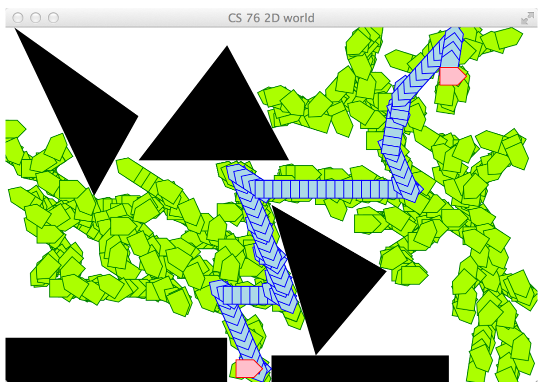

Figure 18: N = 1000, MIN = 10, result 2

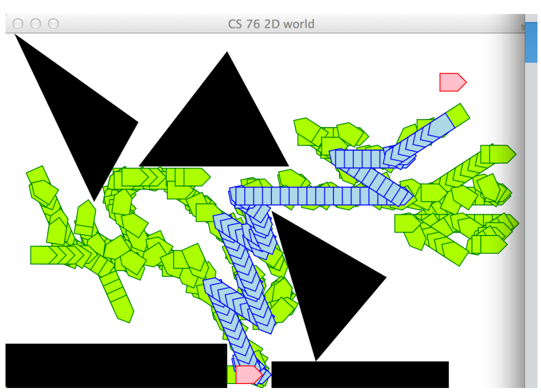

Figure 19: N = 500, MIN = 50, result 1

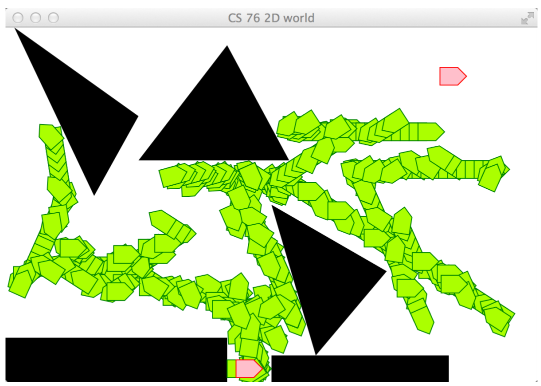

Figure 20: N = 500, MIN = 50, result 2