PRM and RRT

Robin Wang

February 16, 2014

1 Introduction

This report focuses on motion planners for two different systems. One system is a planar robot arm, which has its state given by the angle of the joints. The other is a steered car with its state given by its x and y coordinates and the angle the car makes with the horizontal axis. The two approaches to the motion planners are the Probabilistic Roadmap (PRM) and the Rapidly Exploring Random Tree (RRT).

2 Probabilistic Roadmap (PRM)

We are implementing the k-PRM version of the Probabilistic Roadmap Planner. In this planner, we have a local planner that attempts to connect a vertex to its nearest k neighbors. There are two phases to this k-PRM planner: roadmap generation and query phase.

2.1 Roadmap Generation

The first phase of the k-PRM planner is to generate a roadmap with all of the sampled vertices and their k-nearest neighbors. We initially start by sampling a random N configurations across our window. Using these N configurations, we then connect each vertex to its closest k neighbors. While doing this, we need to check for collisions between the ArmRobot and the obstacles in the world. With a finished roadmap, we can then begin the query phase. The implementation of the roadmap generation is as follows.

We created a new class `PRM.java` to implement both phases of the k-PRM planner. The constructor where I initialize the declared class variables for this class is:

```
public PRM(int num1, int num2, int num3, double[] sConfig,
                        double[] gConfig, World world, double wid, double hei) {
                    // ArmPlanner
                    planner = new ArmLocalPlanner();
                    // The World objected created in the Driver
                    theWorld = world;
                    // Number of random configurations
                    N = num1;
                    // Number of neighbors
                    k = num2;
                    // Number of links for each ArmRobot
                    links = num3;
                    // Start ArmRobot
                    start = sConfig;
                    // Goal ArmRobot
                    goal = gConfig;
                    // Width and Height of the window
```

```
        width = wid;
        height = hei;
    }
```

We generate the roadmap in the `generateMap()` function:

```
    public HashMap<ArmRobot, Set<ArmRobot>> generateMap() {
        // Hashmap to hold the vertices and their k neighbors
        HashMap<ArmRobot, Set<ArmRobot>> roadmap = new HashMap<ArmRobot, Set<ArmRobot>>();
        // Initialize start and end ArmRobots
        sRobot = new ArmRobot(links);
        gRobot = new ArmRobot(links);
        sRobot.set(start);
        gRobot.set(goal);
        gRobot.setGoal(gRobot);
        sRobot.setGoal(gRobot);
        // Add in start and end ArmRobots to the roadmap
        roadmap.put(sRobot, null);
        roadmap.put(gRobot, null);
        // Generate N random configurations
        while (roadmap.size() < N + 2) {
            double[] configs = { Math.random() * width, Math.random() * height,
                    80, Math.random() * 2 * Math.PI, 80,
                    Math.random() * 2 * Math.PI };
            ArmRobot robot1 = new ArmRobot(links);
            robot1.set(configs);
            // Set the goal state for each generated ArmRobots
            robot1.setGoal(gRobot);
            // Check to see if the generated ArmRobot collides with any obstacle
            if (!theWorld.armCollision(robot1)) {
                // Add the configuration to the roadmap
                roadmap.put(robot1, null);
            }
        }
        // Getting the k nearest neighbors
        for (ArmRobot key1 : roadmap.keySet()) {
            // Used my own comparator for the priority queue
            RobotComparator comparator = new RobotComparator();
            comparator.setRobot(key1);
            // Priority queue to hold the closest neighbors
            PriorityQueue<ArmRobot> queue = new PriorityQueue<ArmRobot>(k,
                    comparator);
            // Set of k neighbors
            Set<ArmRobot> neighbors = new HashSet<ArmRobot>();
            for (ArmRobot key2 : roadmap.keySet()) {
                // Don't want to have a node be its own neighbor
                if (key2.equals(key1))
                    continue;
                // Check if the neighbor is reachable
                ArmRobot roboTest = new ArmRobot(links);
                if (!theWorld.armCollisionPath(roboTest, key1.get(), key2.get()))
                    queue.add(key2);
```

```
                        }
                        // Add all into queue; no for loop
                        for (int i = 0; i < k; i++) {
                                // Add the k neighbors
                                if(queue.peek() != null)
                                neighbors.add(queue.poll());
                        }
                        roadmap.put(key1, neighbors);
                }
                return roadmap;
        }
```

The roadmap that is generated in this function is stored in a hash map. We create the N random configurations and for each of them, we find the k nearest neighbors by using a priority queue. The comparator that I used for the priority queue is as follows:

```
public class RobotComparator implements Comparator<ArmRobot> {
                private ArmRobot mainRobot;
                public void setRobot(ArmRobot a) {
                        mainRobot = a;
                }
                @Override
                public int compare(ArmRobot x, ArmRobot y) {
                        ArmLocalPlanner ap = new ArmLocalPlanner();
                        double xx, yy;
                        // Time as a measure of closeness
                        xx = ap.moveInParallel(mainRobot.get(), x.get());
                        yy = ap.moveInParallel(mainRobot.get(), y.get());
                        if (xx > yy)
                                return 1;
                        if (xx < yy)
                                return -1;
                        return 0;
                }
        }
```

This comparator will compare the distance between two configurations as a measure of time that is defined in `moveInParallel()` function in the provided `ArmLocalPlanner.java` file. The priority queue will thus hold the closest neighbors for each of the vertices we check. This eliminates a costly sorting algorithm each time we need to search for nearby neighbors. Once we retrieve the top k nearest neighbors from the priority queue and add them into a hash set, which is added into the hash map under the appropriate key, we are finished with the roadmap generation and can begin with the query.

2.2 Query Phase

In the query phase, we search through the roadmap to find our goal and then return the path from the goal to our start node. The search can be breadth-first search or A* search. We will use both implementations. The breadth-first implementation is as follows:

```
        public LinkedList<ArmRobot> BFS(HashMap<ArmRobot, Set<ArmRobot>> map) {
                // Fringe
                Queue<ArmRobot> fringe = new LinkedList<ArmRobot>();
                // Map to store backchain information
```

```
        HashMap<ArmRobot, ArmRobot> reachedFrom = new HashMap<ArmRobot, ArmRobot>();
        // Start node has no parent, add to fringe
        reachedFrom.put(sRobot, null);
        fringe.add(sRobot);
        while (!fringe.isEmpty()) {
                // Get node from fringe
                ArmRobot currentNode = fringe.remove();
                // Goal test
                if (currentNode.equals(gRobot)) {
                        return backchainz(currentNode, reachedFrom);
                }
                // Get the set of neighbors of the current node
                Set<ArmRobot> successors = map.get(currentNode);
                //Check each neighbor
                for (ArmRobot node : successors) {
                        // If not visited
                        if (!reachedFrom.containsKey(node)) {
                                reachedFrom.put(node, currentNode);
                                fringe.add(node);
                        }
                }
        }
        //No solution
        return null;
}
```

2.3 Results and Testing

An initial test to see if this implementation works was to remove one of the obstacles and use

```
double[] start = {10, 20, 80, Math.PI/4, 80, Math.PI/4};
double[] goal = {500, 300, 80, .1, 80, .2};
```

as my start and goal configurations for the ArmRobot. At N = 50, we get the following image in Figure 1.

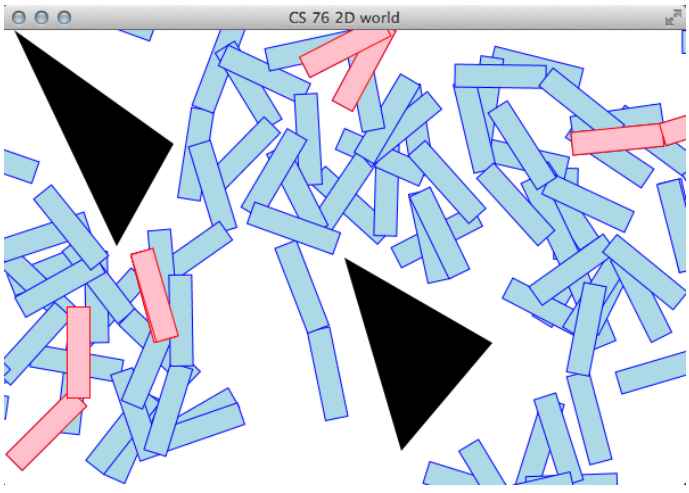
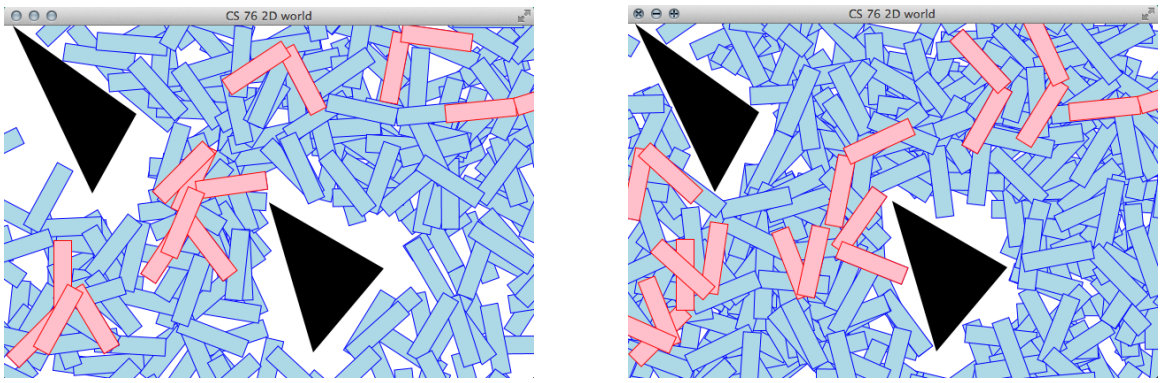


Figure 1: Solution Path with N = 50

The blue ArmRobots are a lll the random configurations that were generated. The ArmRobots that are colored red are the ones that are on the solution path from the starting configuration to the end goal configuration. In order to see the difference that a change in N does, the following images of the same map and same k value, but with N = 150 and N = 250, are presented in Figure 2.



(a) N = 150 (b) N = 250

Figure 2: Solution Paths with Different N values and k = 15

The biggest differences in all of these figures is, of course, the number of configurations, as well as the number of configurations on the path from the start to the goal node returned on the solution path. So it seems that increasing N, the number of sampled configurations, will result in a larger path, which is reasonable as you have many more candidates for your k nearest neighbors. We also want to check what would happen when k is changed.

To see the k-PRM work for the sample graph provided, refer to Figure 3.

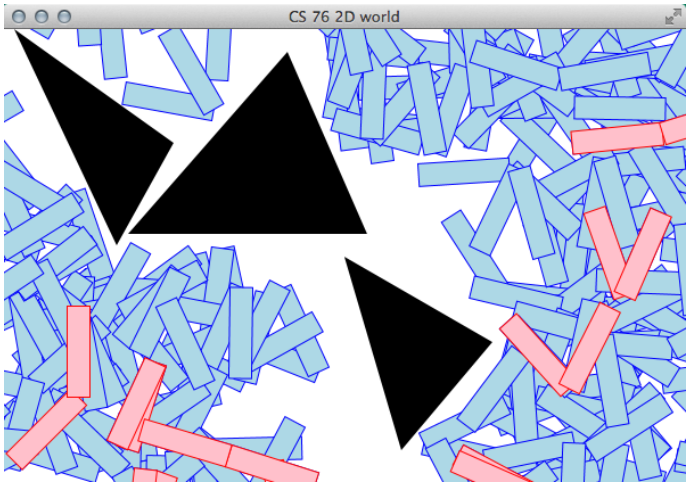


Figure 3: Sample Map - Solution Path with N = 150 and k = 15

While the ArmRobot appears to move outside of the boundary, the base vertices are still within the boundaries of the defined world. While it seems that the ArmRobot could fit between the narrow gap between the middle two triangles, it was more likely for the k-PRM planner to move the ArmRobot across the large space in the bottom of the map and up towards the goal configuration.

I changed the map to be more interesting by creating a vertical line of blocks in the middle of the map with a narrow gap. The start and goal are still the same:

```
double[] start = {10, 20, 80, Math.PI/4, 80, Math.PI/4};
double[] goal = {500, 300, 80, .1, 80, .2};
```

The solution path returned by breadth-first search for this map with the same N and k as before is shown in Figure 4.

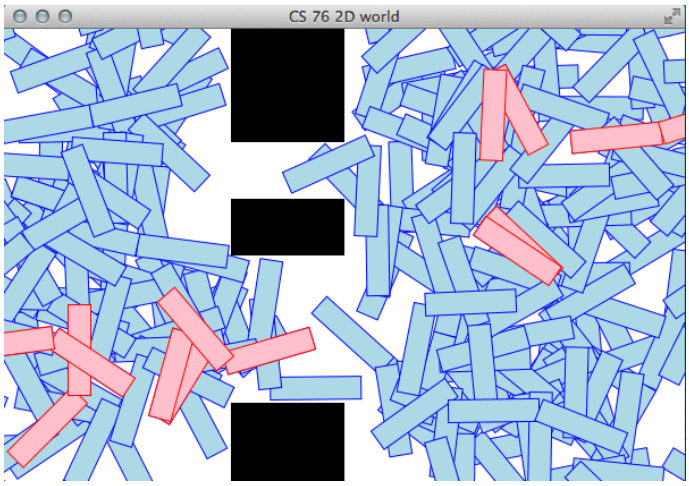
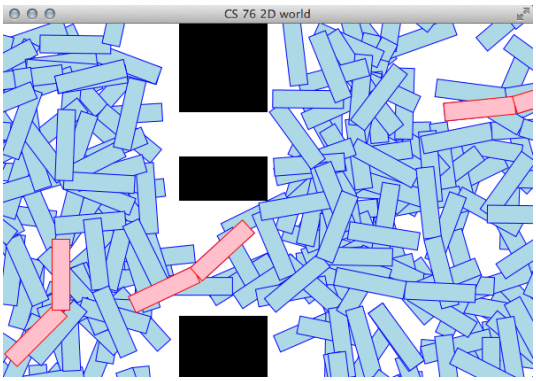
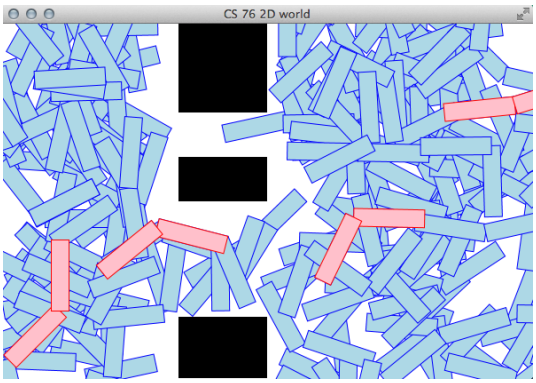


Figure 4: New Map with N = 150 and k = 15

In Figure 4, we have N = 150 and k = 15. The solution path is correct as it goes from the start configuration on the left, through the narrow opening, and to the goal configuration on the top right. If we want to see the change in the solution path when we increase the number of nearest neighbors, we need to change the k value. When we increase k to 25 and 45, but keeping N = 150, we get the following results in the two comparison figures in Figure 5.



(a) k = 25



(b) k = 45

Figure 5: New Map - Solution Paths with Different k values and N = 150

Notice in Figure 5 that the path is still correct, but the number of configurations on the path does not necessarily correlate to the increase in k. Of course, we need to keep in mind that the N randomly sampled configuration could have an impact on the neighbors of each of the configuration.

2.3.1 Breadth-first search vs A* search

While the breadth-first search returned the shortest path from the starting state to the goal state, we could have used A* search to create a solution path. The implementation of A* search that is used in this report is similar to the A* search implemented in the MazeProblem from the previous assignment:

```
public LinkedList<ArmRobot> astar(HashMap<ArmRobot, Set<ArmRobot>> map) {
                // Priority queue and hash maps to help with our search
                PriorityQueue<ArmRobot> frontier = new PriorityQueue<ArmRobot>();
                HashMap<ArmRobot, Double> explored = new HashMap<ArmRobot, Double>();
                HashMap<ArmRobot, ArmRobot> parents = new HashMap<ArmRobot, ArmRobot>();
                // Add the start robot to the frontier
                frontier.add(sRobot);
                sRobot.setCost(0);
                explored.put(sRobot, sRobot.priority());
                parents.put(sRobot, null);
                while (!frontier.isEmpty()) {
                        // Get the front of the priority queue
                        ArmRobot blah = frontier.poll();
                        // Get rid of duplicates
                        if (explored.containsKey(blah)) {
                                if (explored.get(blah) < blah.priority())
                                        continue;
                        }
                        // Goal test
                        if (blah.equals(gRobot)) {
                                return backchainz(blah, parents);
                        }
                        //Get neighboring configurations
                        Set<ArmRobot> successors = map.get(blah);
                        for (ArmRobot node : successors) {
                                // Set goal to be our goal
                                node.setGoal(gRobot);
                                // Set the cost of the configuration
                                node.setCost(blah.getCost() + planner.moveInParallel(blah.get(), node.g
                                // Check and mark duplicate
                                if (!explored.containsKey(node)) {
                                        frontier.add(node);
                                        explored.put(node, node.priority());
                                        parents.put(node, blah);
                                } else if (explored.containsKey(node)) {
                                        if (explored.get(node) > node.priority()) {
                                                explored.put(node, node.priority());
                                                parents.put(node, blah);
                                                frontier.add(node);
                                        }
                                }

                        }

                }
                return null;
```

```
}
```

In this A* search, the edges each has an weight, which are denoted by the time it takes for one configuration to reach another configuration as defined in the `moveInParallel()` method in the `ArmLocalPlanner.java` file. This represents the cost, which we set as we go through the set of neighboring configurations for each of the configurations that is popped from the frontier. The heuristic function utilizes the Euclidean distance and is implemented in the `ArmRobot.java` file:

```
public double heuristic() {
        double dist = 0.0;
        double diffX = this.get()[0] - compareG.get()[0];
        double diffY = this.get()[1] - compareG.get()[1];
        dist = Math.sqrt(diffX * diffX + diffY * diffY);
        return dist;
}
```

Along with this change to the `ArmRobot` class, we need to add and modify the `equals()`, `hashCode()`, `compareTo()`, and other methods so that we can utilize the ArmRobots in our A*Search. All of these changes were based on a similar implementation for the SearchNode in the previous assignment:

```
@Override
public int compareTo(ArmRobot o) {
        return (int) Math.signum(priority() - o.priority());
}
@Override
public boolean equals(Object other) {
        return Arrays.equals(config, ((ArmRobot) other).config);
}
@Override
public int hashCode() {
        return config.hashCode();
}
public double priority() {
        return heuristic() + getCost();
}
// Set goal ArmRobot configuration
public void setGoal(ArmRobot comp){
        compareG = comp;
}
public double getCost(){
        return cost;
}
public void setCost(double newCost){
        cost = newCost;
}
```

The result of this implementation of the A* search as our query on the roadmap generated is shown in Figure 6. From this figure, we note that the solution path is very similar to the solution path returned by the breadth-first search. However, there are certainly more configurations on the solution path. This is most likely due to the fact that the A* search attempts to find the path that has the least cost, or in our case, the least time from one configuration to another. The breadth-first search simply returned the shortest path, which may or may not be the most optimal path in terms of the cost in time.

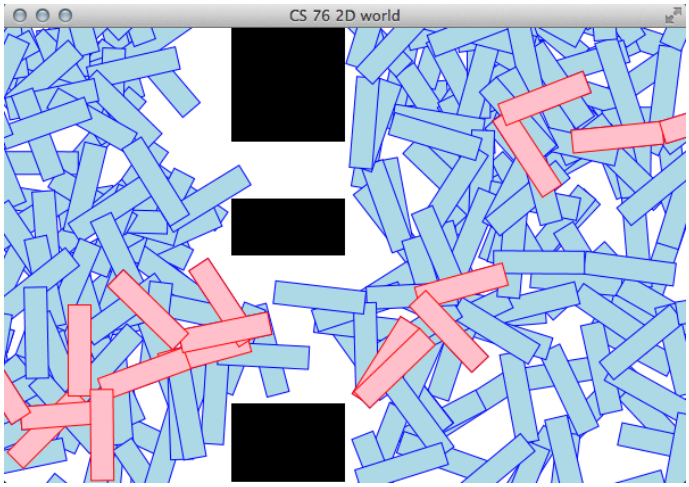


Figure 6: A*Search with N = 150 and k = 15

3 Rapidly Exploring Random Tree (RRT)

The second approach is the Rapidly Exploring Random Tree (RRT). We are building this tree in order to plan the motions for a car-like robot. The purpose of using RRT is because it is designed for efficiently searching non-convex high-dimensional spaces; they are created incrementally in a way that quickly reduces the expected distance of a randomly-chosen point to the tree and are suited for path planning problems that involve obstacles. However, note that RRT do not solve problems completely. There is no way of reaching a goal exactly as we will discover in this report.

3.1 Implementation

The RRT implementation is mostly in the `RRT` class. The constructor for this class is as follows:

```
public RRT(int num2, CarState sConfig, CarState gConfig, World world,
                    double wid, double hei) {
                // The world
                theWorld = world;
                // Number of vertices
                k = num2;
                // Width and height of map
                width = wid;
                height = hei;
                // Start and goal states
                start = sConfig;
                goal = gConfig;
                // Steered car
                sc = new SteeredCar();
                time = 1.0;
        }
```

The central method in this class is to generate the tree, which is completed by the `generateTree()` method.

```
        public LinkedList<CarState> generateTree() {
                // Hash map to hold the generated tree
```

```
tree = new HashMap<CarState, CarState>();
// Counter
int kk = 0;
// Put in the start node
tree.put(start, null);
// Before the counter hits k
while (kk < k) {
        // Generate random CarState
        // A Probability of choosing the goal state
        CarState rando = new CarState();
        if (Math.random() > 0.75) {
                rando = goal;
        } else {
                rando = new CarState((Math.random() * width),
                                (Math.random() * height), (Math.random() * Math.PI));
        }
        // Find closest vertex in tree to the generated state
        CarState current = new CarState();
        double closestN = Double.MAX_VALUE;
        for (CarState node : tree.keySet()) {
                if (euclid(node, rando) < closestN) {
                        current = node;
                        closestN = euclid(node, rando);
                }
        }
        // Out of the 6 ctrls, find one that is closest to the generated
        // state
        CarState closeCS = new CarState();
        double closest = Double.MAX_VALUE;
        CarRobot p = new CarRobot();
        // Generate the 6 ctrls
        for (int i = 0; i < 6; i++) {
                CarState next = sc.move(current, i, time);
                p.set(next);
                // Make sure there is no collision
                if (!theWorld.carCollision(p)) {
                        if (!theWorld.carCollisionPath(current, i, time)) {
                                if (euclid(next, rando) < closest) {
                                        closest = euclid(next, rando);
                                        closeCS = next;
                                }
                        }
                }
        }
        // Add in the new state to tree
        tree.put(closeCS, current);
        // Increment the counter
        kk++;
}
// Backchaining Portion
LinkedList<CarState> solution = new LinkedList<CarState>();
```

```
                // Find node closest to goal
                CarState Cgoal = null;
                double closestG = Double.MAX_VALUE;
                for (CarState node : tree.keySet()) {
                        if (euclid(node, goal) < closestG) {
                                Cgoal = node;
                                closestG = euclid(node, goal);
                        }
                }
                // Node closest to goal
                CarState node = Cgoal;

                // Back chain from this node
                while (node != null) {
                        solution.addFirst(node);
                        node = tree.get(node);
                }
                // Solution path
                return solution;
        }
```

In this method, we generate a random state on the map, with a probability that this random state is the goal state. This is so that we are always somehow attempting to reach our goal node. After we generate the random state, we need to find the a vertex in our tree that is closest to the generated node. This vertex, A, is the basis for our six controls: forward, backwards, left turn, right turn, backward right turn, and backward left turn. We have a constant, `time` which denotes how long the car is do the control movement. We set this constant to 1. After deterring the resulting state from moving the vertex A in each of the 6 controls, we check to see which of the resulting states is the closest to the randomly generated CarState in the beginning. We add the closest state to our street after checking for collisions. We continue this process until there are k vertices in our tree. At that point, our tree has been built and we need to search for a path from the starting initial node, which is also the first node to be inserted in our tree, to the goal CarState node. To do this, we iterate through the tree and check the distance between each vertex and our goal node. This, and checking which of the six controls is closest to the randomly generated state, is accomplished by the `euclid()` method:

```
public double euclid(CarState a, CarState b) {
                double dist = 0.0;
                // Constant multipler for the delta theta value
                double constant = 3.0;
                double diffX = a.getX() - b.getX();
                double diffY = a.getY() - b.getY();
                double diffT = a.getTheta() - b.getTheta();
                // Distance in x and y
                dist = Math.sqrt(diffX * diffX + diffY * diffY);
                // Add in the angular difference
                dist = dist + Math.abs(diffT) * constant;
                return dist;
        }
```

This `euclid()` method finds the Euclidean distance between two CarStates and also factors in the angular difference multiplied by a constant factor. Once we have the node that is closet to our goal, we can simply back chain from that node to the initial starting node using the hash map's structure. This gives us our solution path from our starting vertex to the vertex that is closest to the goal. Since we never add our goal

to the tree, we can never reach the goal, only the nodes that are close to the goal.

3.2 Testing and Results

In the sample map, we run the RRT with a N, number of vertices to be generated for our tree, value of 1000. With the starting and goal states being:

```
CarState state1 = new CarState(270, 15, 0);
CarState state2 = new CarState(400, 315, 65);
```

we achieve the following result in Figure 7.

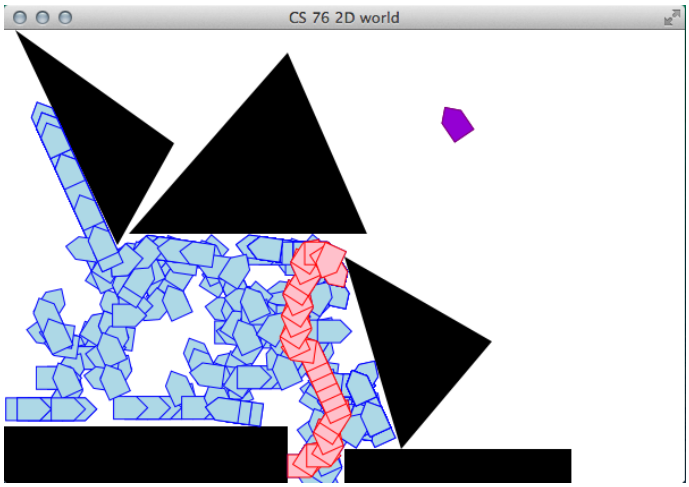


Figure 7: RRT for Sample Map

The blue cars that are plotted are all the vertices on the tree and the red cars are the ones that are on the path from the initial starting node to the goal node. The purple car is the goal state. Note that in this case, the car does not reach the goal because the path is blocked by one of the polygons.

A map in which the car successfully traverses the tree and reaches a node that is very close to the end goal is in Figure 8, on a custom created map that involves the car traveling around a wall.

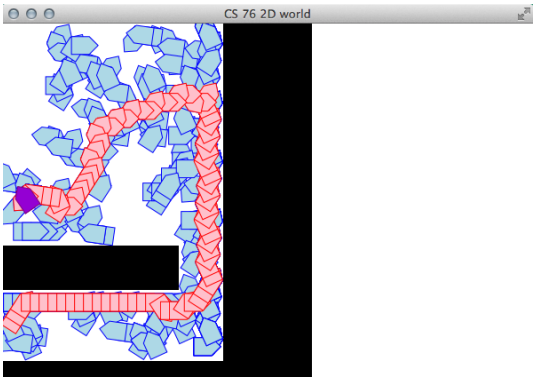
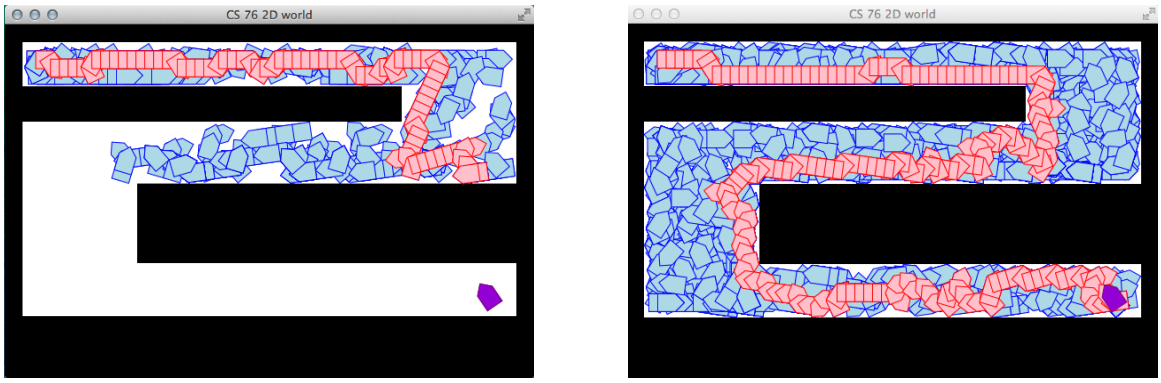


Figure 8: RRT for Custom Map with N = 1000

Another map that I generated involves the car traversing a miniature maze in the shape of an inverted 'S' shown in both images in Figure 9. Initially, the car does not reach the goal when N was set to 1000.

However, when N was increase to 5000, the car reaches the goal easily as there were more configurations generated so the car can actually reach the goal with the time constant being set to 1.



(a) N = 1000 (b) N = 5000

Figure 9: New Map - Solution Paths with Different N Values

4 Concluding Thoughts

It seems that while PRM and RRT both plan out motions for our robots, there are some major differences. PRM initially prepares a roadmap before it searches through the roadmap to find the goal. For PRM, it can accept any start and goal states even after it creates the roadmap. RRT creates a tree on the fly and adds vertices onto the initial start state and continues to add vertices to the tree until N vertices have been reached or a state that fulfills all stopping criteria set by the user is added to the tree. This means that while PRM can accept multiple queries, RRT is a one-shot planner.

There is also the difference in the structure that is being generated. PRM generates a roadmap while RRT generates a tree, which is why a search such as breadth-first search or A* search is necessary in order to generate the solution path in PRM while the tree generated by the RRT can be back chained easily from the node that is closest to the goal, which brings up another point of interest: the goal. In PRM, the solution path returns the path that will reach the goal exactly since the initial and goal state are added onto the roadmap. In RRT, however, only the initial state is inserted into the tree, and the solution path returns the path from the initial starting node to the node that is closest to the goal state in the tree. Also, the tree generated by RRT seems to form a pattern, while PRM generates a much more random set of configurations.

PRM and RRT, both of which plan motions for robots, vary in their implementation and results.

5 Previous Work - Undergraduate

5.1 PRM Paper

In the paper titled "Asymptotically Near Optimal Planning with Probabilistic Roadmap Spanners" by James D. Marble and Kostas E. Bekris, the focus is on a near-optimal option planning solution that greatly reduces construction time and roadmap density through a small reduction in path quality using roadmap spanners. The k-PRM method, which is implemented in our assignment, creates a very dense roadmap, which results in high query times (measure of efficiency for online queries which is utilizing the motion planners described in this paper). The authors proposes two solutions: to build a roadmap spanner in a sequential manner given an asymptotically optimal roadmap (Sequential Roadmap Spanner) or to interleave roadmap construction with a spanner preserving filter to directly construct a roadmap spanner in a more efficient

manner (Incremental Roadmap Spanner); both methods provide asymptotic near-optimality. The Sequential Roadmap Spanner constructs a sparse graph given a dense roadmap, provides asymptotic guarantees on the number of edges and has the same asymptotic time complexity as PRM. The Incremental Roadmap Spanner directly constructs sparse roadmaps using a sampling and filtering process and since collision detection is expensive, this constructs roadmaps in a computationally efficient way while providing asymptotically near-optimal solutions and having an asymptotic time complexity close to that of PRM.

5.2 RRT Paper

In the paper titled "Improving the Efficiency of Rapidly-exploring Random Trees Using a Potential Function Planner" by Ian Garcia and Jonathan P. How, the focus is on improving the standard randomized path planning algorithm, which is implemented in this report. The key idea presented is to change the way two vertices or points are connected. Instead of using the standard approach of using a simple and fast function to link two points, the authors propose using a new connection function that finds a link between the points by minimizing a distance function to the target point with a feasible optimizer that accounts for the constraints. However, this is most certainly a more computationally expensive connection function compared to the standard function, but it does result in a significant overall speed improvement of the planner. The standard function is a simple function that tries to connect two points, but stops as soon as there is an obstacle, thus generating as many connections as possible. The paper proposes to increase the computational cost of this function in order by using some information about the obstacles in order to increase the chance of reaching the target point. To accomplish this, artificial potential functions consisting of a continuous function that decreases monotonically in the direction of the goal are used to include information about the obstacles.

6 Acknowledgments

This report was completed with the help from the wonderful TAs and other students through the discussions on the Piazza forum. This report also utilized the information presented in the links provided on the assignment page, courtesy of Professor Balkcom.