Robot Motion Planning Solution

Nicholas Golini

October 12, 2016

1 Introduction

In this lab, we must solve for a motion plan for 2-4R robot arms to get from an initial configuration to a final configuration. We also must solve for a motion plan for a mobile robot moving through a world from an initial location to a final location.

1.1 Robot Arm

The state system for the robot arm is a set of angles for each link of the robot arm. We assume that the robot arm is fixed at the origin, $(0, 0)$. For example, if the robot arm is made up of 3 links, each link will be oriented at an angle, where the first angle is measured from the horizontal, and each successive angle is measured counter-clockwise from the straight line created by the previous link. These angles simulate the degrees of freedom of the arm, where there can be rotation around each joint. Using these angles, we can calculate for the endpoints of each link in the 2D space. The initial state of the most basic, 2R, robot arm is $(\theta_0, \theta_1) = (270, 270)$, meaning the first link is along the vertical going down from the origin, and the second link lies on the horizontal to the left of the first link. The final state for the robot arm is $(\theta_0, \theta_1) = (0, 90)$, where the arm extends to the right on the horizontal from the origin, and the next link extends upwards on the vertical. In addition to the robot arm, there are obstacles in the space where the arm is moving, indicating areas in which the arm cannot cross. In order to find a path from the initial state to the goal state, we must create a random sampling of possible angles for the bot arms (all of which do not obstruct the obstacles), create a $k-$nearest neighbor graph of the samplings, and search the graph for a path of robot arm configurations to take it from its starting state to its goal state, making sure that each arm movement between the points does not pass an obstacle.

1.2 Mobile Robot

The state system for the mobile robot is an x, y coordinate of its location, θ, the angle the front wheels are making counterclockwise from the horizontal, and another CarNode named parent, which will be used for backtracking. In this problem, since we are trying to get the robot to a final goal x,y coordinate, distance will always be considered as the euclidean distance between two nodes' x, y coordinates. The world is created by the four outside walls which keeps the robot within a set boundary, in addition to interior walls within the boundaries making it more difficult for the bot to find its way to the goal. The initial node is in the bottom left corner of the world facing straight ahead (positive x direction on the horizontal), and the goal state is in the upper right corner of the world, each shown by a square in their respective corners. Because the CarNodes are defined by doubles for their x, y, and θ values and the orientation of the car depends on random samplings and rotations around different points each movement, it would be very improbable that the RRT produces a Node exactly equal to the goal state. Thus instead of checking if the current node in the RRT is exactly the same as the goal node, I check to see if its x, y coordinate lies in a 20 by 20 pixel range around the goal node. This simplifies the goal checking, and makes the run time much faster, since it would take a very long time to reach the exact same node as the goal node in the RRT. The main method of finding the path for the mobile robot from start to goal is by generating an RRT until the tree reaches

within acceptable range of the goal node, then backtracks its parent nodes to give a final path for the robot to take.

2 The Arm Robot

2.1 Kinematics and Drawing

An ArmNode is my class to represent each arm configuration. It takes in a list of thetas, and calculates the endpoints of each arm link in the arm. In this example, each arm link will be the same length. Each link of the bot is then converted into a java Line2D object, where the calculated endpoints of each link are the endpoints of a line. Likewise, the obstacles are represented by the java Rectangle2D class. This decision was made because java Shape classes have simple intersection calls, so finding an obstruction between an arm link and an obstacle was easy to implement. These Shape classes were also easy to draw on the GUI of the system, which will now be explained. The blue lines represent the robot arm along the path, the starting arm configuration is in RED, and the goal arm configuration is DARK GREEN. The lime green line represents the path the end of the robot arm will take in the space. A robot arm has been drawn for every other arm configuration along the solution path to pseudo-animate the motion of the arm through the space. This is kind of a tricky way to see the robot arm movement, but if you follow the lime green line from the end of the red arm till the dark green arm, you can follow where the end of the arm will be in the movements, and you can start matching the blue lines to make up arms in the space.

2.2 Probabilistic Roadmap (PRM)

The first part of creating the PRM is linking each node with its k nearest neighbor. This algorithm is the brute force way of finding k nearest neighbors. The code is shown below:

```
1 public HashMap<ArmNode, ArrayList<ArmNode>>  createNeighborGraph(int k){
2  Comparator<PQNode> comparator = new maxPQComparator();
3  HashMap<ArmNode, PriorityQueue<PQNode>> neighborhoodTemp = new HashMap<>();
4  HashMap<ArmNode, ArrayList<ArmNode>> neighborhood = new HashMap<>();

5  for(int i = 0; i < alpha.size(); i++){ // looking at all of the points
6    ArmNode currPoint = alpha.get(i);
7    PriorityQueue<PQNode> kNeighbors = new PriorityQueue(k, comparator);
8    for(int j = 0; j < alpha.size(); j++){
9      ArmNode currNeighbor = alpha.get(j);
10      if (currNeighbor.thetas != currPoint.thetas){
11        if(kNeighbors.size() < k) {
12          if(collision(currPoint, currNeighbor)) {
13            PQNode currNeighborNode = new PQNode(currNeighbor,
                angleDistance(currPoint.thetas, currNeighbor.thetas));
14            kNeighbors.add(currNeighborNode);
15          }
16        } else{
17          if(collision(currPoint, currNeighbor)) {
            double furthestCurrNeighbor = angleDistance(currPoint.thetas,
18                kNeighbors.peek().node.thetas);
19            if (angleDistance(currPoint.thetas, currNeighbor.thetas) <
                  furthestCurrNeighbor) {
20              kNeighbors.remove();
21              PQNode currNeighborNode = new PQNode(currNeighbor,
                  angleDistance(currPoint.thetas, currNeighbor.thetas));
```

```
22                kNeighbors.add(currNeighborNode);
23              }
24            }
25          }
26        }
27      }
28    neighborhoodTemp.put(currPoint, kNeighbors);
29  }
30    for(ArmNode curr : neighborhoodTemp.keySet()){
31      ArrayList<ArmNode> currNeighbors = new ArrayList<>();
32      for(PQNode n: neighborhoodTemp.get(curr)){
33        ArmNode neighbor = n.node;
34        currNeighbors.add(neighbor);
35      }
36      neighborhood.put(curr, currNeighbors);
37    }
38    for(ArmNode curr : neighborhood.keySet()){
39      for(ArmNode neighbor : neighborhood.get(curr)){
40        ArrayList<ArmNode> n = neighborhood.get(neighbor);
41        if(!n.contains(curr))
42          n.add(curr);
43          neighborhood.put(neighbor, n);
44        }
45     }
46     return neighborhood;
47  }
```

The graph is implemented by a HashMap, called neighborhood, with keys being ArmNodes, and values being an ArrayList of its k nearest neighboring ArmNodes and any other Node whose k nearest neighbors include the current node. To get the k nearest neighbors of a node, I used another HashMap with keys as ArmNodes, and the values as a max PQ of its closest k neighbors. The max PQ is implemented with a custom comparator that returns opposite values of a min PQ (making it a max queue), and holds PQNodes, which is just a wrapper class that holds a an ArmNode and its priority (distance to the key). The algorithm starts by a double for loop, going over each node in the sampling for each node in the sampling. A new max PQ is initialized for the node that the neighbors are being found for, and until the PQ is full (to size k), the ArmNodes being considered to be one of the current's $k-$nearest neighbors are added into the PQ. Once the PQ is full and we get to another node to considered to be a close neighbor, we check to see if that node is closer than the furthest current k closest neighbor, which is at the top of the PQ. If it is, we remove the head of the PQ and add the new close node. If it is not, we move on to the next node. Once all of the possible neighbors are considered, the priority queue is added into the HashMap to represent edges between every node in the ArrayList and the node whose neighbors they are. Now that every 'curr' node has a hash mapping to its k nearest neighbors, we must add these k neighbors to the neighborhood map with the 'curr' node as the key, and appending each k nearest neighbor to the key's ArrayList value of ArmNodes. Then we want to go through the neighborhood and make a connection backwards for each node, where a mapping is added to neighborhood between every node in the values set to its respective key. This makes the edges in the map bidirectional. This is necessary because if the robot arm needs to backtrack the map to get around collisions, it can. Without doing this final step, the graph's edges remain single direction, thus can not backtrack out of imminent collisions.

This method of finding k nearest neighbors brings up an interesting discussion on what it means to be "nearest" in this case. When using the robot arm, we want to simulate movements that have small arm swings, i.e. changes in thetas, since the goal state is represented by bot link angles, not x, y coordinates.

For example, it does not help for the robot arm to be at a position right above the goal state if but with completely different angles as the goal state because the arm would have to make a large rotation all the way around to the goal state. On the other hand, if we have a node with angles close to those of the goal state, even if it may have a farther euclidean distance from where the arm would lie in the goal state, this node would be considered closer because it will take fewer rotations to get to the goal. Thus when calculating distance from a node in the k nearest neighbor algorithm, I use a method called angleDistance(), which takes the sum of the differences between each nodes corresponding thetas $\theta_1 - \theta'_1 + ... + \theta_n - \theta'_n$.

An important note to add to this discussion is how I am finding if a collision would occur between the arm and any obstacles in the movement between two nodes. When I am making the random sampling of nodes, I am checking to see if the robot arm would collide with any of the obstacles at the random arm configuration, and if there would be a collision, I find a new node that will not collide with the obstacles. Since each node in the random sampling of nodes is a viable arm configuration in the $C-$space, it is vital to check in between nodes if the robot arm would collide with an obstacle. To do this, I create an even distribution of nodes along a line from one point a possible neighbor, and check for a collision at each point in the distribution. This collision check is happening when considering a neighbor relationship between the current node and a potential neighbor. Thus every node can rotate to one of its neighbors without collision. This collision method is shown in code below this paragraph. If a movement from a current node to the potential neighbor is going to cause a collision, then it is not taken to be a neighbor. By making a collision check here in the neighborhood creation, we do not have to worry about collisions in the search of the tree.

This is not a perfect check, because there may be a very slim portion of that line where the robot would collide with an obstacle, and if the distribution size is not large enough, then the even distribution may skip over that collision. This raises the interesting point that with more random sample nodes created, comes a more dense sampling thus shorter distances between sample nodes. Since the distribution size is a constant in my code, the smaller the distance between two nodes, the smaller the steps are in the collision checking, thus a more accurate collision check. In addition, a larger distribution size would also be better for checking for collisions, but a large constant in this algorithm will really slow down the creation of the PRM. The same goes for increasing the size of the random sampling. Since this brute force method of finding k nearest neighbors is $O(n^2)$, n being the number of random samples, n cannot be too large or else the code will take too long to run. It cannot be too small, though, because if the sampling is too small then a path for the robot arm may not be found. A reasonable sample size for my lab was in the range of 1000-5000, with k being between 10-20.

```
1  public boolean collision(ArmNode currNode, ArmNode neighbor) {
2    int n = 50; // number of points to look at between two points
3    double t1 = 0;
4    ArrayList<ArmNode> checkNodes = new ArrayList<>();
5    for(int i = 1; i < n; i++){
6      ArrayList<Double> t = new ArrayList<>();
7      for(int j = 0; j < currNode.thetas.size(); j++) {
8        t1 = ((neighbor.thetas.get(j) - currNode.thetas.get(j)) * (i)) / (n)
               + currNode.thetas.get(j);
9        t.add(t1);
10     }
11     ArmNode checkNode = new ArmNode(t);
12     checkNodes.add(checkNode);
13   }
14   for(int i = 0; i < checkNodes.size(); i++){
15     ArmNode curr = checkNodes.get(i);
16     for(int j = 0; j < curr.botLinks.size(); j++){
17       for(int k = 0; k < obstacles.size(); k++){
```

```
18              if(obstacles.get(k).intersectsLine(curr.botLinks.get(j))) {
19                  return false;
20              }
21          }
22      }
23  }
24  return true;
25 }
```

This creation of neighborhood is through brute force, requires a lot of time to run, and is not ideal. To make this $k-$nearest neighbor map more efficiently, one can use an Approximate Nearest Neighbor algorithm, that approximates each node's k nearest neighbors rather than looping through each node for each node and finding its exact k nearest neighbors. In addition, grid hashing could be used, in which the panel is split up into an $n \times m$ grid, and each randomly sampled node is placed in a hashmap corresponding to the section of the grid it is in. Once the samples have been hashed into the grid, then each node's k nearest neighbors are found within the node's grid. Again, this would be an approximation function, but would reduce the brute force run time by a factor of mn. In addition, the value of k is very important to this problem. A low number for k means that each node is connected with only a few other nodes, very limiting its movement choices. Having too low of a k value could also leave the PRM unconnected, leaving the start node and goal node in different connected components of the map, which would make it impossible to find a path form the start to the goal. A larger value for k gives each node more options for possible movements, and would thus create a smoother path for the robot arm to get to the final configuration. However, too large of a k value will make the run-time of this algorithm extremely slow, so there is definitely a trade-off with increasing the k value for the problem.

Once we have this graph made for every node in the random sampling, we can then easily search the graph and find a path of arm configurations to get from its start to goal state. BFS was the path finding algorithm of choice due to its short run time and simplicity to implement. I reused my BFS algorithm made from the Missionaries and Cannibals problem, which is explained in depth in my report for that problem. Since collision checking is happening while creating the neighborhood graph, there is no need to check for collisions in the BFS, because every node should already be able to get to each of its neighbors from the graph without making any collisions.

One note I would like to make for the code is that if the robot arm planner is run and the path appears to have a collision in it, this may be due to the sample space being too small, too small of a k value, or error from the collision method, which is described above in the collision discussion.

3 The Mobile Robot

3.1 Kinematics and Drawing

The trickiest part of the mobile robot problem was calculating the kinematics for each successor node in the RRT, which will be explained more in depth below. After creating a random sample node and finding the closest node in the tree, we create 6 successor nodes simulating the actions as follows: forward movement, backwards movement, turn forward clockwise, forward counter-clockwise, backward clockwise, and backward counter-clockwise. Nodes to replicate these actions are created in a method called, getSuccessorMoves. The forward and backwards nodes are easy to calculate, where the newNodes x value is simply $x = x + cos(\theta)$, the y value is $y = y + sin(\theta)$, and θ stays the same since the bot is moving in a straight direction. For backwards motion, it is the same as forwards, but instead of adding $cos(\theta)$ and $sin(\theta)$ to x and y respectively, they are subtracted. For the remaining four actions, more complicated kinematics are necessary to finding the resultant node's x, y, and θ value, which are calculated in a function called makeMove(). It takes in a velocity, an angular velocity, a node to be have the action simulated form, and an angle of how far the robot moves

along a path (rotAngle). A move forward counter-clockwise requires: $v = 1, \omega = 1, \theta = rotAngle$, backwards counter-clockwise: $v = 1, \omega = -1, \theta = rotAngle$, forwards clockwise: $v = 1, \omega = 1, \theta = 360 - rotAngle$, and backwards clockwise: $v = 1, \omega = -1, \theta = 360 - rotAngle$. Given this information, we can calculate the point to rotate around: $x = curr.x + (-v/w)sin(curr.theta)$ and $y = curr.y + (v/w)cos(qn.theta)$. Then this point must be translated to rotate around the origin. Once it is translated, the x coordinate of the node after the rotation can be calculated by: $x = xcos(rotAngle) - ysin(rotAngle)$, and y can be calculated by: $y = ycos(rotAngle) + xsin(rotAngle)$. Once this rotation calculation occurs, the x and y must be translated back to represent a rotation from the original starting node. Then, the newNode's theta value ends up being $theta = theta + rotAngle$. Once these nodes are all created, they are then checked for collisions. If they do not represent a collision, then the nodes are added to the RRT.

Collision checking in this RRT method is simplified by making a line between the current node and its successor, found by the methods above, and seeing if this line intersects with any of the walls. Since the step size for the movements are so small, where the bot moves somewhere between a fraction of a pixel to a few pixels for most movements, thus this simplification is sufficient for collision checking. Another simplification I made in this assignment was to show any of the rotating movements by straight lines. I was unsure how to draw arcs in Java to show exactly what was happening with the RRT, so I figured since the movements are all so small, this simplification would not be a huge deal. My graphic still shows the lattice created by the RRT, and it still successfully finds and displays a path for the mobile bot. The path the bot will take is shown in yellow, and the green lines represent all of the nodes in the RRT. The red lines represent the walls in the world. Another simplification I made was to make the robot 1 pixel large, so that it can just be represented solely as a x, y coordinate. While this is definitely an oversimplification, it still successfully finds a path that a small bot could take in a large world. It also shows the creation of the RRT and how it can wrap around obstacles, which was the main goal of this assignment.

3.2 Rapidly Exploring Random Tree

Given that the movements are all being created successfully and correctly, the generation of the RRT is not too complicated. The RRT structure is represented by a HashMap, mapping each CarNode to the successor action nodes from it. The RRT starts out by adding the start node to the tree, and assigning the start node's parent as null (the parent variable will be discussed later on in this report). The algorithm then goes into a while loop, that keeps going until a node is found within the given goal range, and the RRT can stop expanding. With each iteration of the loop, a random node is created, and the node closest to this random node in the RRT is found to be the current node. The current node then simulates the 6 actions, creates nodes for each action, and puts the new nodes into the tree if they do not collide with a wall. If one of the successor nodes is within acceptable range of the goal node, then we say the goal has been reached and the nodes are backtracked to get the path form the tree from start to goal. In order to backtrack properly, each node contains a CarNode parent, which represents the node that was used to create the current node. This parent assignment happens for each node before adding them into the tree. So to backtrack, you just have to follow a path from the goal node to its parent, then to its parent's parent, etc. until you reach the start node, who's parent is null.

The random sampling is key to creating the RRT. By creating a random node and finding the nearest node to it from the RRT, we are expanding the RRT in random directions each time, with the goal of expanding out evenly into the C space while still expanding out to finding the goal. This is required when finding a path through a complex maze with many walls or obstacles. If the random sample was actually just the goal node, the nearest neighbor to the goal node will always be the same node, and if there is an obstacle in between that node and the goal node, there would be no way for the RRT to get around the obstacle. Thus, the randomization is key to creating a successful RRT. The full algorithm is in code below:

```
1 public HashMap<CarNode, ArrayList<CarNode>> buildRRT(CarNode start, int k, int s){
2  int step = s; // max step for the RRT
```

```
3  boolean goal = false;
4  HashMap<CarNode, ArrayList<CarNode>> tree = new HashMap<>();
5  ArrayList<CarNode> startList = new ArrayList<>();
6  startList.add(startNode);
7  startNode.parent = null;
8  tree.put(start, startList);
9  int i = 0;
10 while(!goal){
11    CarNode qs = getRandomNode();
12    CarNode qn = getClosestNeighbor(qs, tree);
13    ArrayList<CarNode> newNodes;
14    newNodes = getSuccessorMoves(qn, step);

15    for(CarNode curr : newNodes) {
16      curr.parent = qn;
17      neighbor to its value list
18      if (tree.get(qn) != null) {
19        tree.get(qn).add(curr);
20      } else {
21        ArrayList<CarNode> newList = new ArrayList<>();
22        newList.add(curr);
23        predMap.put(curr, qn);
24        tree.put(qn, newList);
25      }
26      if(curr.nearGoal()){
27        goalNode = curr;
28        goal = true;
29      }
30    }
31 }
32 return tree;
33}
```

4 Interesting Examples

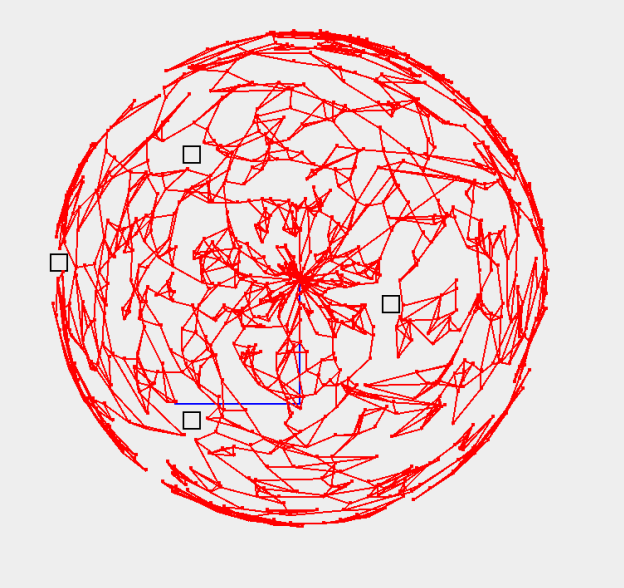

Figure 1: This is a visual representation of the k nearest neighbor graph for a system. This specific map is a 2-nearest neighbor map with N = 1000

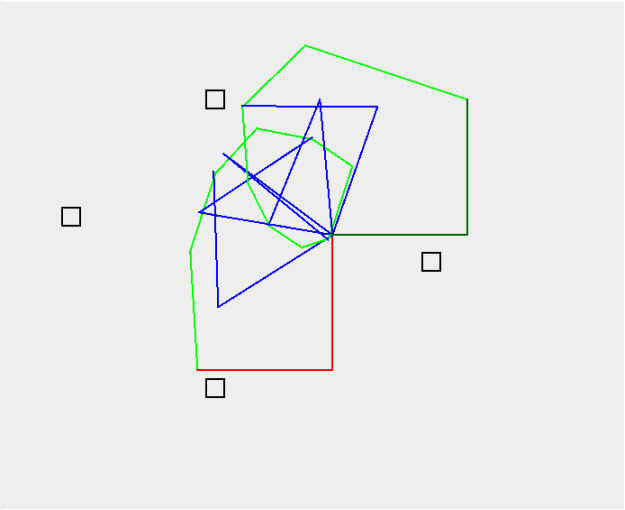

Figure 2: This is an example of a 2R robot moving through the C space

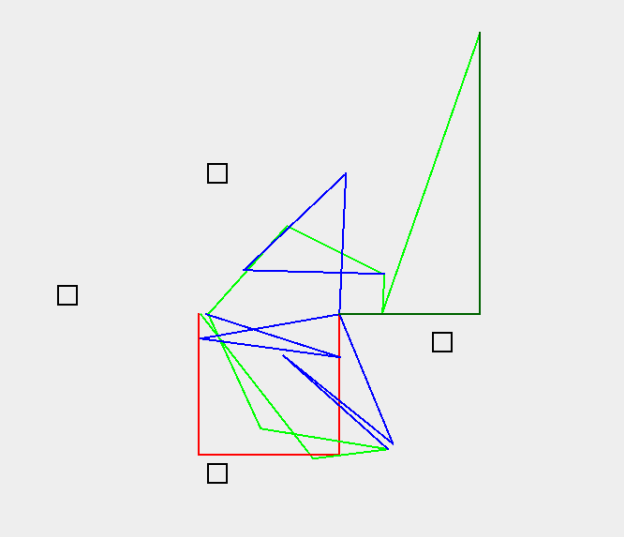

Figure 3: This is a 3R arm moving through the same C space. These examples show the complexity of the system, and how the arm has to really contort itself to get between the obstacles.

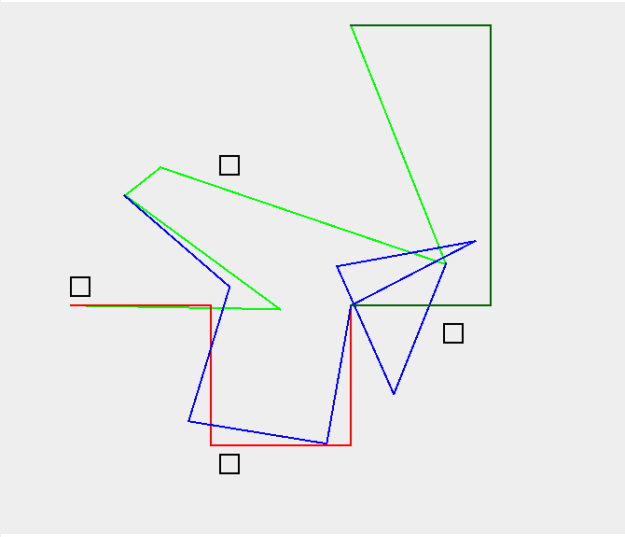

Figure 4: This is a 4R robot arm int he same world. This is the most interesting example of the robot arm because it shows how the arm moves away from two of the obstacles (the left and top ones).

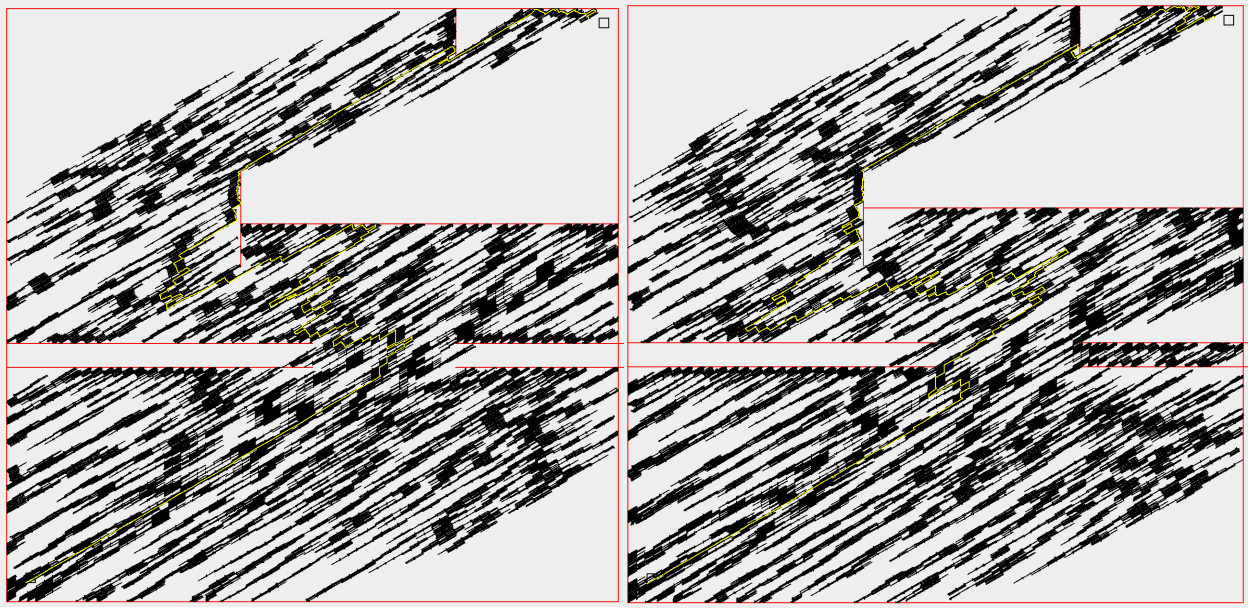

Figure 5: If you cannot see the yellow line, zoom in on your screen because the yellow line shows the path the bot would take in the C space. There are many obstructing walls in this world making it difficult.

Figure 6: This is a more interesting example because it is showing how the RRT can and will explore a corridor. It validates that wherever the goal state would be, the tree would eventually find it.