CS76 Assignment 3: Kinematics

Benjamin Hannam

Fall 2016

NB: I collaborated with Brian Keare

1 The Arm Robot

The arm robot was treated as being an arm with a fixed base at position (0,0) with k arm segments whose direction was defined by an angle, θ, relative to the axis through the previous arm segment or the x-axis if it was the first arm. This meant that the state space consisted of a list of angles $(\theta_1, \theta_2...\theta_n)$. Therefore in order to calculate the x,y positions of the end of each arm segment, trigonometry had to be used. The equations for the x,y positions of the nth arm was:

$$x_n = l_1 cos\theta_1 + l_2 cos(\theta_1 + \theta_2)... + l_n cos(\theta_1 + \theta_2... + \theta_n)$$

$$y_n = l_1 sin\theta_1 + l_2 sin(\theta_1 + \theta_2)... + l_n sin(\theta_1 + \theta_2... + \theta_n)$$

Therefore each configuration of angles corresponded to a corresponding x,y point in 2D space corresponding to the end point of the last arm segment.

2 Probabilistic Roadmap(PRM) Generation

In order to set up the PRM to model the different arm configurations I set up a class called GeneratePRM.java which had the following variables:

```java
public class GeneratePRM extends JPanel{

  //variables
  private int num_samples;
  public int num_arms;
  private int arm_l;
  private int k;
  private Rectangle obstacles[];
  private Graph graph;
  public GraphicsDriver graphics;

  GeneratePRM(int number_arms, int arm_length, Rectangle obs[], int sample_count, int
    num_neighbors){
    //initiate variables
    num_arms = number_arms;
    num_samples = sample_count;
    arm_l = arm_length;
    obstacles = obs;
    k = num_neighbors;
    graph = new Graph(num_samples, num_arms);
  }
```

code/GeneratePRM.java

In here we can see several notable variables such as the Array of Rectangle objects which were used to model obstacles, a GraphicsDriver which was a custom class to draw out the 2D representation and above all the Graph structure. Java did not have a built in graph structure so I had built my own one using three custom classes: Vertex.java, Edge.java and Graph.java.

Vertex Class

The vertex class was the foundation of the graph and consisted of the following components:

```java
// array that holds the angle values for the vertex
private double angle_list[];
private Set<Vertex> adjacent;
private int arm_l;
double depth;

//constructor
public Vertex(double angles[], int arm_length){
  angle_list = angles;
  adjacent = new HashSet<Vertex>();
  arm_l = arm_length;
}
```

code/Vertex.java

As we can see here there is an array list of doubles which are used to keep track of the angles for each arm (and hence the state), a set of other vertices to keep track of all the neighbors which is used in the A*-search. Also inside of the class are several functions that get/set various variable values as well adding neighbors or return the x,y coordinates of an arm segment, the method for which is shown below:

```java
//get the x location
```

```
public double getX(int arm_number){
  //set to 0
  double x = arm_l * Math.cos(Math.toRadians(angle_list[0]));
  double angle = angle_list[0];
  //loop through the angles and compute the new x location
  for(int i = 1; i < arm_number+ 1; i++){
     angle += angle_list[i];
    x += arm_l * Math.cos(Math.toRadians(angle));
  }
  return x;
}

//get the Y location
public double getY(int arm_number){
  //set to 0
  double y = arm_l * -Math.sin(Math.toRadians(angle_list[0]));
  double angle = angle_list[0];
  //loop through the angles and get y
  for(int i = 1; i < arm_number + 1 ; i++){
    angle += angle_list[i];
    y += arm_l * -Math.sin(Math.toRadians(angle));
  }
  return y;
}
```

code/Vertex.java

Edge Class

The Edge class ended up being a fairly straightforward class and was actually mostly used to calculate the total cumulative difference in angles between two vertices. This was stored in the weight variable and was calculated inside the constructor like so:

```
//constructor
public Edge(Vertex first, Vertex second, int num_arms){
  one = first;
  two = second;

  //get the cumultive difference between the angles
  double diff = 0;
  //loop through the arms
  for(int i = 0; i < num_arms; i++){
    //do the bigger angle take away the smaller angle and make sure it is less than 180
    if(one.getAngles()[i] > second.getAngles()[i]){
      double temp = (one.getAngles()[i] - two.getAngles()[i]);
      if(temp > 180){
        temp = 360 - temp;
      }
      diff += temp;
    }
    else{
      double temp = (two.getAngles()[i] - one.getAngles()[i]);
      if(temp > 180){
        temp = 360 - temp;
      }
      diff += temp;
    }
  }
  //set the weight to the cumulutive distance between the angles
  weight = diff;
}
```

code/Edge.java

The method was to check which angle was bigger, subtract the smaller angle from the bigger angle and then check if the difference was greater than 180. If it was this meant that we could go the other way around and so I did 360 minus the difference to get the actual shortest difference between the two (in hindsight I could have just taken off 180). Other than that the Edge class was mainly used to keep track of the k nearest neighbors for each vertex and then used to help insert those neighbors into the adjacent set for each vertex.

Graph Class

The graph class was the simplest of the three and consisted almost entirely of a Set containing all the valid vertices and a HashMap mapping each vertex to a corresponding set of edges. These were mainly used for iterating over the vertices later on.

```
public class Graph {

  private Set<Vertex> vertices;
  private HashMap<Vertex, Set<Edge>> edges;
  private int num_arms;

  //constructor
  public Graph(int num_samples, int number_arms){

    //set up the vertices array and the edge map
    vertices = new HashSet<Vertex>();
    edges = new HashMap<Vertex, Set<Edge>>();
    num_arms = number_arms;
  }
```

code/Graph.java

Random Point Generation

In order to generate the random points I generated arrays of doubles of the same length as the number of arm components and then used the in built random function to generate a random number between 0 and

1 which was then multiplied by 360 to get a random angle for each slot in the array. I then used each new array to create a new vertex which on which I then called a function which checked if any or the arms for that vertex collided with any obstacles. If the vertex was safe it was then added to the Set in the graph.

```java
//Generates the random points
public void GeneratePoints(){

  //for each sample
  for(int i = 0; i < num_samples; i++){
    //get random angles
    double sample_spot[] = new double[num_arms];
    for(int j = 0; j < num_arms; j++){
      sample_spot[j] = Math.random() * 360;
    }
    //create a new vertex
    Vertex new_vertex = new Vertex(sample_spot, arm_l);
    //if it is a safe point
    if(testVertex(new_vertex)){
      //add it to the graph
      graph.addVertex(new_vertex);
    }
  }
}
```

code/GeneratePRM.java

When testing the vertex I made a line corresponding to each arm segment and the checked that it did not intersect any of the obstacles.

```java
//check if a vertex is valid
public boolean testVertex(Vertex v){

  //get each arm
  for(int i = 0; i < num_arms; i++){

    double x1,x2,y1,y2;
    if(i == 0){
      //if its the first arm start from the center
      x1 = 0;
      y1 = 0;
      x2 = v.getX(0);
      y2 = v.getY(0);
    }
    else{
      //otherwise get the segment from each
      x1 = v.getX(i - 1);
      y1 = v.getY(i - 1);
      x2 = v.getX(i);
      y2 = v.getY(i);
    }

    //loop through all the obstacles and see if they contain the x,y point
    for(int o = 0; o < obstacles.length; o++){
      if(obstacles[o].intersectsLine(x1, y1, x2, y2)){
        return false;
      }
    }
  }
  return true;
}
```

code/GeneratePRM.java

getNeighbors

The next step in generating the PRM was to find the k nearest neighbors for each vertex and check that we could safely move to them. To do this I looped through all the vertices in the graph, created an arrayList to hold the nearest neighbors, again looped through the vertices and got the k closest ones by a combination of sorting the array and checking the vertex currently being looked at against the furthest away vertex in the list. Once I had the k vertices I checked that I could move to them safely before creating an edge and inserting the relevant information into the graph.

```
  //A function to generate the k nearest neighbors for each vertex in the graph
  // and insert the edges into the graph
  public void getNeighbors(){
    //loop through each vertex
    for(Vertex v : graph.getVertices()){
      //create a set of ones already seen
      ArrayList<Edge> neighbors = new ArrayList<Edge>();
      for(Vertex u : graph.getVertices()){
        //if not the same vertex
        if(!u.equals(v)){

          //create an edge
          Edge e = new Edge(v, u, num_arms);

          //if we have less than k members automatically add it
          if(neighbors.size() < k){
            neighbors.add(e);
          }
          //otherwise compare it to the largest one
          else{
            if(e.getWeight() < neighbors.get(k - 1).getWeight()){
              neighbors.set(k - 1, e);
            }

          }
          //sort the list
          neighbors.sort(new Comparator<Edge>(){
            public int compare(Edge e1, Edge e2){
              return e1.compareTo(e2);
            }
          });
        }
      }
      //now add all the edges to the graph
      for(int i = 0; i < neighbors.size(); i++){
        //check that we can move from the vertex to the neighbor
        if(moveCheck(v, neighbors.get(i).getAdjacent(v))){
          graph.addEdge(v, neighbors.get(i));
          Vertex adjacent = neighbors.get(i).getAdjacent(v);
          v.addNeighbor(adjacent);
        }
      }
    }
  }
```

code/GeneratePRM.java

I checked that we could move from one vertex to another by sampling ten spots in between the two, creating a vertex out of these spots and testing if it was a safe vertex.

```
// A function to check if you can move safely between two vertices
public boolean moveCheck(Vertex v, Vertex u){

  //check ten spots between the two vertices
  for(int i = 0; i < 10; i++){
    //new vertex array
    double new_vert_array[] = new double[num_arms];
    for(int j = 0; j < num_arms; j ++){
      double angle_diff = (u.getAngles()[j] - v.getAngles()[j])/10.0;;
      new_vert_array[j] = v.getAngles()[j] + i*angle_diff;
    }
    Vertex new_vert = new Vertex(new_vert_array, arm_l);
    if(!testVertex(new_vert)){
      return false;
    }
  }
  return true;
}
```

code/GeneratePRM.java

3 Query Phase

In the query phase I simply ran a modified A*-search through the graph. The search was contained within a class called PRMSearch which allowed it to have access to the graph and it took in a start and end vertex as its parameters. The search also differed slightly from a typical A* search as the start and end vertices did not necessarily exist in the graph. Therefore I had to check if they were in the graph and if they were not use the closest available vertex to it in it's place. This was done like so:

```java
//if the goal is not in the PRM then we need to change that to the nearest
Vertex end_vertex = end;
if(!graph.getVertices().contains(end)){
  System.out.println("end not in PRM");
  Vertex new_end = null;
  double min = 1000;
  //get the closes vertex to the goal and set it
  for(Vertex u : graph.getVertices()){
    Edge temp = new Edge(end, u, num_arms);
    if(temp.getWeight() < min){
      new_end = u;
      min = temp.getWeight();
    }
  }
  end_vertex = new_end;
}
```

code/PRMSearch.java

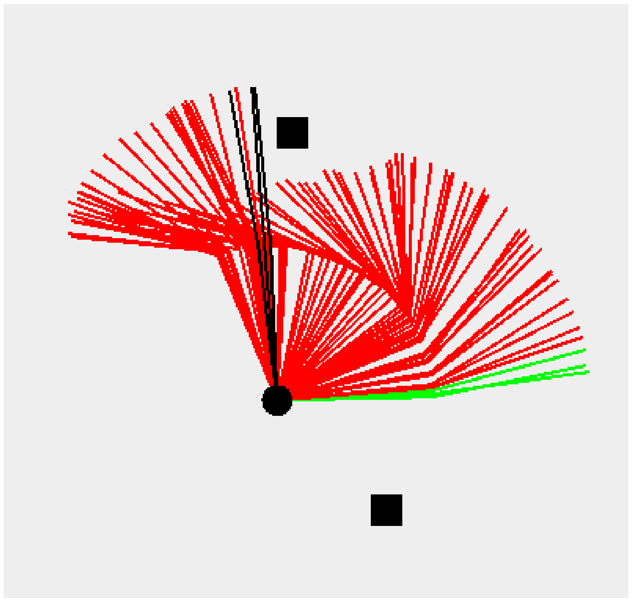
The other main differential was in calculating the priority when comparing two vertices. I did this by taking the depth times a constant and then adding the cumulative difference in the angles between the two vertices being compared and the end vertex. The difference between the end vertex and the two comparing vertices as calculated in the same way as the weight for an edge.

```java
PriorityQueue<Vertex> frontier = new PriorityQueue<Vertex>(10000, new Comparator<Vertex
>(){
  //compare function to place vertices in the right place in the queue
  public int compare(Vertex v1, Vertex v2){
    //initally add the depth
    double diff_one = 50*v1.getDepth();
    double diff_two = 50*v2.getDepth();
    //loop through the arms
    for(int i = 0; i < num_arms; i++){
      //add the difference in the angles for v1
      if(end.getAngles()[i] > v1.getAngles()[i]){
        double temp = (end.getAngles()[i] - v1.getAngles()[i]);
        if(temp > 180){
          temp = 360 - temp;
        }
        diff_one += temp;
      }
      else{
        double temp = (v1.getAngles()[i] - end.getAngles()[i]);
        if(temp > 180){
          temp = 360 - temp;
        }
        diff_one += (v1.getAngles()[i] - end.getAngles()[i]);
      }
      //add the difference in the angles for v2
      if(end.getAngles()[i] > v2.getAngles()[i]){
        double temp = (end.getAngles()[i] - v2.getAngles()[i]);
        if(temp > 180){
          temp = 360 - temp;
        }
        diff_two += temp;
      }
      else{
        double temp = (v2.getAngles()[i] - end.getAngles()[i]);
        if(temp > 180){
          temp = 360 - temp;
        }
        diff_two += temp;
      }
    }
    double pri_one = diff_one;
    double pri_two = diff_two;

    if(pri_one < pri_two){
      return -1;
    }
    else if(pri_one > pri_two){
      return 1;
    }
    else{
      return 0;
    }
  }
});
```

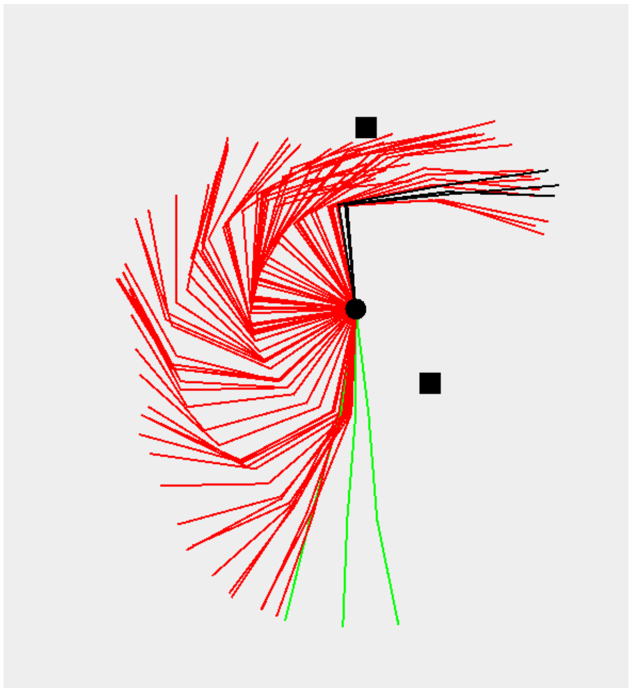
code/PRMSearch.java

Testing

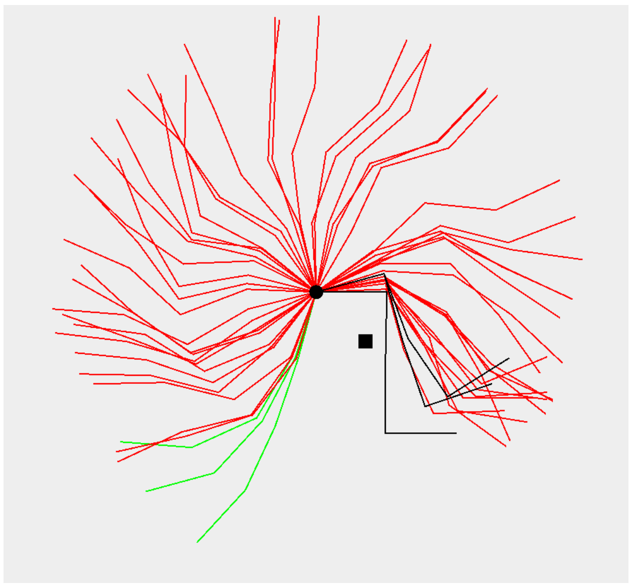
1. Two arms - Here we can see clearly how the arm bends itself to fit around the obstacles, the green lines are the initial few states, the red ones are the transition one and the black lines are the end states. As the arm rotates anti-clockwise it is clear how the second arm component bends to fit underneath the top rectangle before straightening after it has cleared the bottom.



2. Three arms - In this graphic we can once again see how the two outer arm segments fold themselves inward as the arm rotates around towards the obstacles before squeezing under and going to the final position.



3. Fours arms - Due to the increased complexity a lot of samples were required in order to start to get even a reasonable looking path. Without the increased number of samples the gap between the arms would have been too big. In this example we can see how although the most direct path would have been to have gone counter-clockwise from the start state to the goal, the arm recognized that there was an impassable obstacle in the way and went around in the other direction.



4 The Mobile Robot

The next part of the assignment involved a problem where there was a circular shaped robot that had three wheels and was defined by its x,y location and its heading (θ) relative to east. The robot could effectively therefore take six possible moves. It could either go directly forward, directly backwards, curve forwards to the left, forwards to the right or backwards to the left or to the right. The problem was to model the robots behavior and map its path from one point in a graph to another, avoiding obstacles along the way. Its velocity was decided by a set value, v and its angular velocity was decided by a set value ω.

Problem Set Up

In order to implement the problem I created two classes: MobileRobot.java and TreeNode.java. MobileRobot.java's main job was the build the Rapidly Exploring Random Tree(RRT) until it found the goal and the create the path array from the start to the end. It also acted as a structure to hold almost all of the relevant information and had a subclass that ran the graphics.

```
public class MobileRobot {
  public TreeNode goalNode;
  public TreeNode startNode;
  public double radius;
  public double v, omega;
  public Rectangle obstacles[];
  public GraphicsDriver graphics;
  public ArrayList<Line2D> lines;
  public ArrayList<Arc2D> arcs;
  public ArrayList<TreeNode> final_path;

  //constructor
  public MobileRobot(TreeNode start, TreeNode goal, double rad, double velocity, double
    angular_v, Rectangle obs[]){
    startNode = start;
    goalNode = goal;
    radius = rad;
    omega = angular_v;
    v = velocity;
    obstacles = obs;
    lines = new ArrayList<Line2D>();
    arcs = new ArrayList<Arc2D>();
  }
```

code/MobileRobot.java

TreeNode.java on the other hand was a class that as it's name suggests was a class for the node objects in the tree. Within the class it had various methods that got the child leaves, carried out various collision and goal checks as well as checked if it was possible to move from one node to another legally. Each TreeNode also kept track of its parent leaf so that I could backtrack to get the path once we found the goal.

```
public class TreeNode {

  public TreeNode parent;
  private ArrayList<TreeNode> children;
  public double x, y, theta;

  //Constructor
  public TreeNode(TreeNode par, double nx, double ny, double th){
    x = nx;
    y = ny;
    theta = th;
    parent = par;
  }
```

code/TreeNode.java

Robot Motion

Forwards/Backwards

This form of motion was the most straight forward to compute as it simply involved calculation the various x and y components depending on the heading to find the new location like so:

$$newX = currentX \pm velocity \times cos(\theta)$$

$$newY = currentY \pm velocity \times sin(\theta)$$

```java
    //get the forward move
    new_x = this.x + v * Math.cos(Math.toRadians(this.theta));
    new_y = this.y + v * Math.sin(Math.toRadians(this.theta));
```

code/TreeNode.java

Once I had the new point I checked for collisions by creating three line objects: one between the original point and the new point and two parallel ones each a the same distance as the radius above and below the main line. I then checked if any of the three intersected any obstacles and also made a circle at the new point to simulate the robot at that position before also checking that for collisions.

```java
  public boolean straightLineCheck(TreeNode other, Rectangle obs[], double radius, double v,
     double omega, ArrayList<Line2D> lines){

    //are we in bound
    if(other.x < -400 + radius || other.x > 400 - radius || other.y < -400 + radius || other
    .y > 400 - radius ){
      return false;
    }
    //shift the points positive radius
    double pos_x1 = this.x + radius * Math.cos(Math.toRadians(this.theta));
    double pos_y1 = this.y + radius * Math.sin(Math.toRadians(this.theta));
    double pos_x2 = other.x + radius * Math.cos(Math.toRadians(this.theta));
    double pos_y2 = other.y + radius * Math.sin(Math.toRadians(this.theta));

    //shift the points negative radius
    double neg_x1 = this.x - radius * Math.cos(Math.toRadians(this.theta));
    double neg_y1 = this.y - radius * Math.sin(Math.toRadians(this.theta));
    double neg_x2 = other.x - radius * Math.cos(Math.toRadians(this.theta));
    double neg_y2 = other.y - radius * Math.sin(Math.toRadians(this.theta));

    //create a line
    Line2D line = new Line2D.Double(this.x, this.y, other.x, other.y);
    Line2D neg_line = new Line2D.Double(neg_x1, neg_y1, neg_x2, neg_y2);
    Line2D pos_line = new Line2D.Double(pos_x1, pos_y1, pos_x2, pos_y2);

    //create a circle around the end points
    Ellipse2D end_circle = new Ellipse2D.Double(other.x - radius, other.y - radius, radius*
    2, radius*2);

    for(Rectangle ob : obs){
      //check if it intersects the lines
      if(line.intersects(ob) || neg_line.intersects(ob) || pos_line.intersects(ob)){
        return false;
      }
      //the end point circles
      else if(end_circle.intersects(ob)){
        return false;
      }
    }
    lines.add(line);
    return true;
  }
```

code/TreeNode.java

Arc Motion

In order to find the new location for each arc motion a lot of computations had to be done:

1. The rotation center had to be found. This center lied along an axis perpendicular to the heading of the robot and was at a length of $l = v/\omega$ away. In order to do this I had to effectively rotate a point that was l away along the heading through 90 degrees which was done like so for the left rotation center:

$$rotationX = currentX - (v/omega) * sin(\theta)$$
$$rotationY = currentY + (v/omega) * cos(\theta)$$

```
    //variables to hold rotation center
    double rot_x, rot_y;

    //get the left turn rotation center
    rot_x = this.x  - (v / Math.toRadians(omega)) * Math.sin(Math.toRadians(theta));
    rot_y = this.y  + (v / Math.toRadians(omega)) * Math.cos(Math.toRadians(theta));
```

code/TreeNode.java

Once I had the rotation center I had to translate the point that I was rotating by taking away the x and y values of the rotation center to center the rotation around the origin. Once I had carried out the rotation I then had to then translate it back into original coordinate system.

$$newX = (currentX - rotationX) \times cos(\omega) - (currentY - rotationY) \times sin(\omega)$$
$$newY = (currentX - rotationX) \times sin(\omega) + (currentY - rotationY) \times cos(\omega)$$
$$newX = newX + rotationX$$
$$newY = newY + rotationY$$

```
    //rotate around the origin by +omega
    new_x = (this.x - rot_x) *Math.cos(Math.toRadians(omega)) - (this.y - rot_y) * Math.sin(
    Math.toRadians(omega));
    new_y = (this.x - rot_x) *Math.sin(Math.toRadians(omega)) + (this.y - rot_y) * Math.cos(
    Math.toRadians(omega));
    //translate back
    new_x += rot_x;
    new_y += rot_y;
```

code/TreeNode.java

With these new coordinates I now had to check it it was possible to for the robot to move along the arc without any collisions. To do this I built an three arc objects: the actual arc and two translated arcs at a distance equal to the radius of the robot. Each of these arcs was then checked for collisions with any obstacles along with a circle that was built to model the robot at the new x,y point.

```
  public boolean checkArc(TreeNode other, int type, Rectangle obs[], double radius, double v
    , double omega, double rot_x, double rot_y, ArrayList<Arc2D> arcs){

    //are we in bounds
    if(other.x < -400 + radius || other.x > 400 - radius || other.y < -400 + radius || other
    .y > 400 - radius ){
      return false;
    }

    //double start_angle = Math.toDegrees(Math.atan(diff_y/diff_x));
    double start_angle;
    if(type == 0){
       start_angle = 90 - theta;
    }
    else if(type == 1){
       start_angle = 360 - (90 - theta);
    }
    else if(type == 2){
       start_angle = 90 + theta;
    }
    else{
      start_angle = 360 - (90 + theta);
    }


    //get the half the side length of the rectangle
    double length = v / Math.toRadians(omega);
    double top_x = rot_x - length;
    double top_y = rot_y - length;

    //shift the points positive radius
    double pos_x1 = top_x + radius * Math.cos(Math.toRadians(this.theta));
    double pos_y1 = top_y + radius * Math.sin(Math.toRadians(this.theta));

    //shift the points negative radius
    double neg_x1 = top_x - radius * Math.cos(Math.toRadians(this.theta));
    double neg_y1 = top_y - radius * Math.sin(Math.toRadians(this.theta));

    //the arcs to check
    Arc2D new_arc1 = new Arc2D.Double(top_x, top_y, length*2, length*2, start_angle, omega,
    Arc2D.OPEN);
    Arc2D up_arc = new Arc2D.Double(pos_x1, pos_y1, length*2, length*2, start_angle, omega,
    Arc2D.OPEN);
    Arc2D down_arc = new Arc2D.Double(neg_x1, neg_y1, length*2, length*2, start_angle, omega
    , Arc2D.OPEN);

    //create a circle around the end points
    Ellipse2D end_circle = new Ellipse2D.Double(other.x - radius, other.y - radius, radius*
    2, radius*2);

    for(Rectangle ob : obs){
      if(new_arc1.intersects(ob) || up_arc.intersects(ob) || down_arc.intersects(ob) ||
    end_circle.intersects(ob)){
        return false;
      }

      if(ob.contains(other.x, other.y)){
        return false;
      }
    }
    arcs.add(new_arc1);
    //arcs.add(new_arc2);
    return true;
  }
```

code/TreeNode.java

Building the RRT

The buildTree() function in the MobileRobot class handled the actual tree building algorithm. The function made use of two HashSet¡TreeNode¿, one to keep track of all the current leaf nodes and one to keep track of all visited nodes. It the ran a continuous while looped until it found the goalNode. Within the while loop it generated a random point, got the closest unvisited leaf to the point and then got all the child nodes, before removing the current node from the leave set and adding it to the visited set. It then looped through all of the children, checked if they were the goal and then added them to the leaf set if they were not.

If the child was the goalNode then getPath() was called on the child which backtracked up the tree, adding each parent to the path until we reached the startNode which had a null parent. This path was then returned.

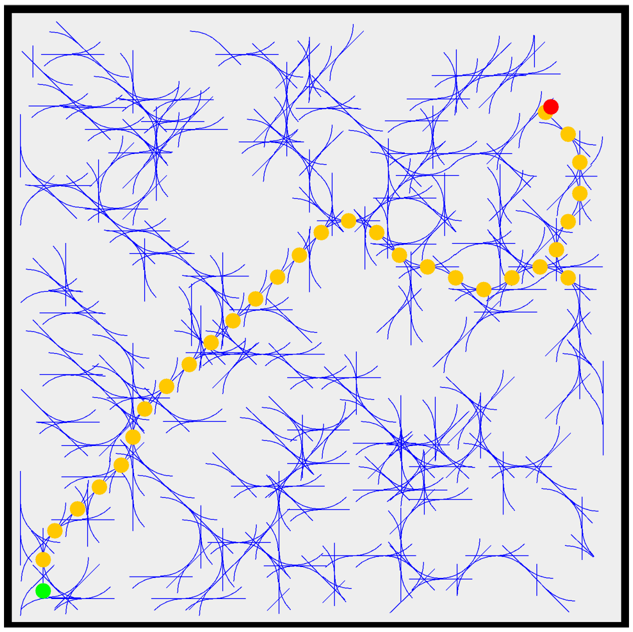
```
  //build the tree and get the path
  public ArrayList<TreeNode> buildTree(){

    //A set of leaf nodes
    Set<TreeNode> leaves = new HashSet<TreeNode>();
    leaves.add(startNode);
    //while we are not at the goal
    while(true){
```

```

    //get the random point
    double rand_x = Math.random();
    double rand_y = Math.random();
    double nx = rand_x * 800 - 400;
    double ny = rand_y * 800 - 400;
    double min = 10000;

    //get the leaf node closest to the random poin
    TreeNode current_smallest = null;
    for(TreeNode leaf : leaves){
      double test_min = Math.abs(leaf.x - nx) + Math.abs(leaf.y - ny);
      if(test_min < min){
        current_smallest = leaf;
        min = test_min;
      }
    }
    //remove the leaf
    leaves.remove(current_smallest);

    //get the children
    ArrayList<TreeNode> children = current_smallest.getSuccessors(v, omega, obstacles,
  radius, lines, arcs);

    //check its children
    for(TreeNode child : children){
      //if they are not the goal test, add them as leave
      if(!child.goalTest(goalNode, radius)){
        leaves.add(child);
      }
      //otherwise get the path
      else{
        final_path = getPath(child);
        return final_path;
      }
    }
  }
}
```

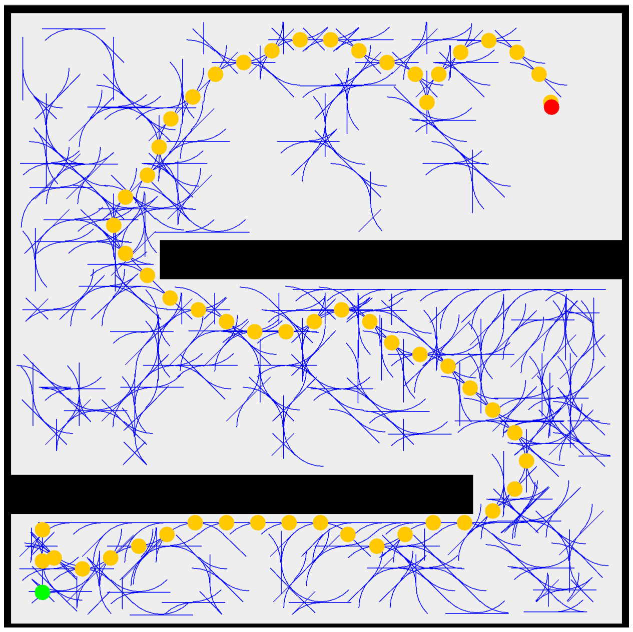
code/MobileRobot.java

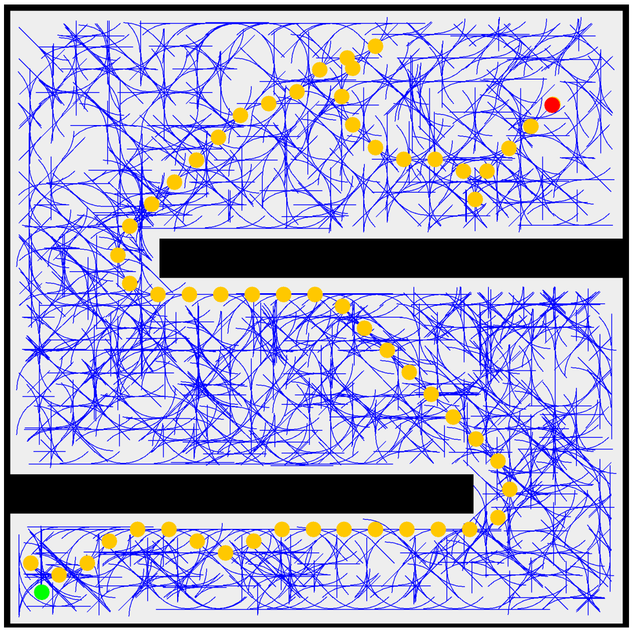
Testing

1. No Walls - In this simple base case we can see how the tree is build and then the path that the robot then takes to get to the goal. The omega was 45 degrees. We can clearly see that the collision detection was working on the side walls as none of the lines or arcs came within a radius distance of the walls.



Walls - these examples demonstrate the random nature of the tree building. In the first one the goal was found actually quite quickly before the tree had expanded out too much, whilst in the second one the tree had expanded a lot more before the goal was eventually found.





4. 90 Degree turns - in this last one I decided to set the turning degree to 90 which was very interesting given that it created an almost grid like structure when building the tree.