CSC595 Final Project Report

Group 4

Luigi Alecce

Pete Gillespie

Arturo Granados

**Environmental Model**

In order to model the entire field, obstacles and the robot we created a class called Point. The Point class works that it holds doubles that represent an x and a y value of the cartesian plane. It is able to calculate the distance between the calling point and any other point. The Point class also implements a compareTo method that allows us to order the points by first ordering by x-values than y-values. Where the obstacles get created is in the RectInsersect class. RectIntersects holds nine global variables. We created three ArrayLists called invalid, which contains a list of all points that are being covered by an obstacle, toVisit  is a list of up to three additional points that we must visit and finally freeSpace is a list that contains all the Points that are not being covered by an obstacle.

Next, we have global variables for a start and ending points called start and end. Our next variable is called PADDING. We used this to add additional padding to all our rectangles. By adding padding to our rectangles this reduced our robot to be a single point and not have to be worried about the robot crashing since that was already taken into account. Our next variable is called CM, which converts the input of meters into 10cm. Lastly our max x and max y to ensure the points are within bounds. The function createRec creates the rectangles. It prompts the user to enter two points regardless if they are the upper or lower corners. Afterwards it gets the minimum of both x and y from both points and applies the floor function and saves this point. That point becomes the lower left corner. To find the upper right-hand corner the maximum function is applied to x and y of both points and ceiling function is applied. The function returns a list containing these two points. After the points are created, createOccupied and iterates through the points that are defined by createRec and populates invalid with points.

Lastly, RectIntersect creates all the free space. It iterates through the max x and y values and ensures the points are not in invalid then it populates freeSpace. The final class created to model our environment was the PointGraph class which created three additional classes called Edge which represents the link between the nodes, Node which is the vertex and Graph which is the accumulation of both Node and Edge. The graph is populated using the data generated in RectIntersect and creates edges based on if two points have a difference of one between their x and y coordinates.

**Path Planning**

At first, to plan a path from the origin location to the destination we used BFS. Having created a graph with all the available points in the first part of the project, we were able to create an adjacency list which just contains the entire list of points that are available and all the neighboring points for them. We then created a static class that runs BFS. Within the class, we start at the source node and add it to the queue. Then we visit all the adjacent nodes and add them to the queue as well. Eventually, we visit all nodes and get the shortest path to every node. We use a hashmap called edgeTo to keep track of the path to each edge. For example, if we want to know how to get to an edge, we simply follow all the values and keys in the hashmap until we get to the source. Keeping track of all those steps will give us a path from the origin to the destination. In this case, the algorithm does not take into account sharp turns. It does not factor that a turn might be too tight for the robot. However, we do know that the robot will not crash into any of the obstacles in the space because all those points that are not available have been removed from the map.

In order to make sure that the robot chooses a smooth path that will prevent it from making impossible turns, we used Dijkstra’s algorithm. When creating the graph, we’re using the edges to represent how difficult it would be for the robot to go in that direction. Then, we use Dijkstra to determine the shortest path based on the edge weights that we pass on for the robot.

**Obstacle Avoidance**

As mentioned previously, we applied the floor and ceiling functions to the corners when we created the rectangles and then applied the padding as a safety buffer. From the dimensions given, we calculated that if the robot were to spin in a circle on its origin, the resulting circle would have a max radius of 113mm.  We used this dimension in our padding and added to the obstacle size so that we don’t have to worry about the robot turning at large angles as it maneuvers through the course. Essentially, we then are considering the robot as just a point about its origin. From here, the extra padding is removed from the free space and the graphs will not contain these points. So, after we run Dijkstra, the resulting path can be considered free of collision. The paths have an option of being either shortest path or most energy efficient, which is input from the user. The weight in Dijkstra would be either Euclidean distance from point to point or the amount of wheel rotation plus translation required to get from point to point (in radians). Larger rotations get penalized by having larger weights, so the robot will prioritize moving forward.

This path is saved as an array list of Point objects and then passed to our classes that first calculate the trajectories to get from point to point and then the wheel displacements. The trajectories are saved as a nx3 matrix, where n is the number of points in the path. These are all calculated as relative moves. Once this matrix is calculated in the CalculateTranslations class, it is passed to the CalculateWheelDisplacements. Here, the matrix is dissected and the trajectories are split into first, the rotation required to turn the robot towards the next point from where it’s at, and then next, the x and y movement required to get it to the next point. These movements are then calculated into pure wheel rotations based on the dimensions of the robot.

The wheel displacements are saved again in a matrix with size (2n)x2. We first calculated a factor that is:

Axle length / (2 \* wheel radius) = 0.12 / (2\* 0.028)

The alpha factor can then be applied to each rotation in the matrix to get the wheel displacements, one positive and one negative - depending on left or right turn. We noticed the right wheel will be the same sign as the rotation sign and the left wheel will be the opposite sign and of the same magnitude. The last step then writes the matrix to a file where it will be used as input to the simulator.

**Testing**

To test the translations and wheel displacements we mainly used the simulator. We first manually tested the calculations our program came up with by creating lists of points in the configuration space like (1,1,0) or (–1,-1,pi) for example. We created many points to try to test edge cases. We then verified the wheel displacements by checking in the sim if the robot moved to where we wanted it to move.

Once everything was pieced together in the code, we tested the whole program by coming up with different configurations of obstacles, start points, and end points. We manually entered those and wrote the final wheel displacements file to the sim folder. We then used the same inputs in the sim configuration and verified the robot took the correct path. We could verify the path by printing it out to the console (or file) and watching the simulator if it reached the goal or hit an obstacle.

A sample video is also included in the submission folder showing test results and verification of running the path in the simulator.