Introduction

This lab implements a bug 0 algorithm to navigate a robot to a specified goal. It utilizes its camera and LiDAR scanners to move towards the goal. If the goal is not visible or blocked by an obstacle, it will perform the necessary rotations and wall following procedures until it is visible again. This is accomplished in a series of stages. The tasks that were performed in this lab are defined as follows:

Task 1 – Motion to Goal

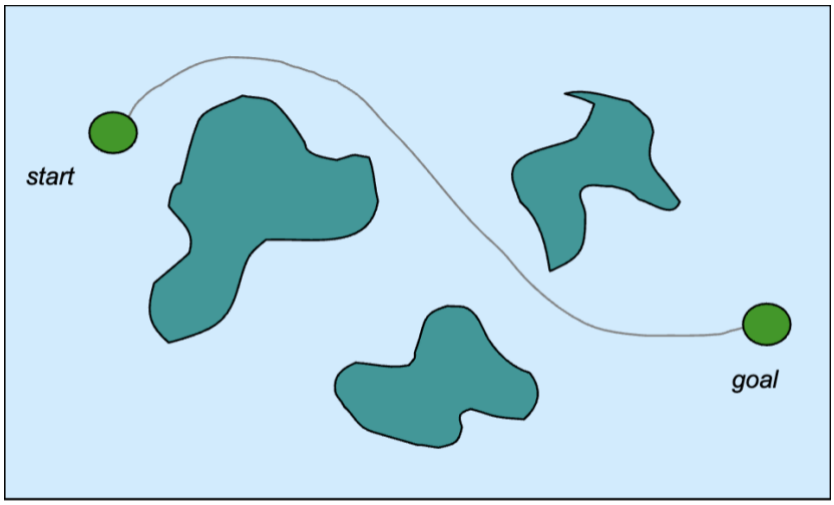

Figure 1: A path that describes the motion to goal.

The first stage to be considered in the bug 0 algorithm is a straight-line motion to goal. This is satisfied when the robot's on-board camera detects the specified object and is not blocked by any obstacles.

To detect the object, we must figure out if the array created by

```
rec_objects = robot.rgb_camera.getRecognitionObjects()
```

holds the specified object using this if-statement:

```
if len(rec_objects) > 0:
        landmark = rec_objects[0]      # specified object stored in the first index of the array
```

Once the object is detected, the robot must "lock in" directly towards it by using the camera's y-position functions that determine the position of the object in the camera view.

```
If 0.5 > landmark.getPosition()[1] > -0.5:
```

The above if-statement line determines if the object is in the center of the frame ± 0.5 meters. The ± 0.5 acts like an error range since the robot will miss the exact y = 0 (center of the frame) so there must be a range to "catch" the robot.

Once the object is centered in the camera view, this means the robot is directly facing it which means stage 1, motion to goal, can be initiated.

However, to detect the object and make it centered in the camera view, the robot must find the object. This is done using a simple rotation.

To find out if the camera does not detect the object, this if-statement can be used:

```
If len(rec_objects) == 0:
        # Rotate robot until object is in view
```

A PID controller is used in this task to slow down the robot as it gets closer to the object. The PID is implemented by calculating error and returning a saturated velocity.

Error is calculated by the difference of target distance and current distance:

$$e(t) = y(t) - r(t)$$

The control signal is calculated by the product of the error and a specified KP value ran through a saturation function that keeps the speeds within limits.

$$u(t) = f_{sat}[K_p(r(t) - y(t))]$$

Task 2 – The Bug Zero Algorithm

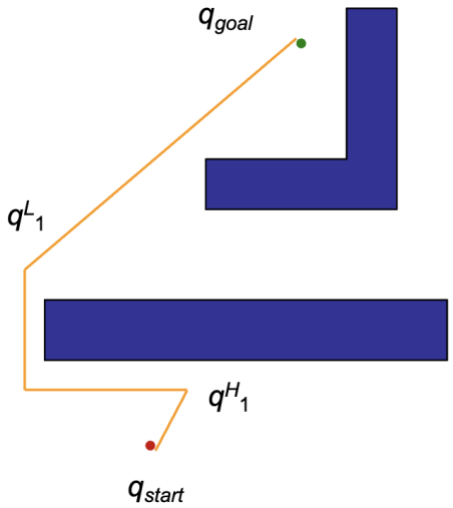

Figure 2: A path that is defined using bug zero.

The Bug Zero algorithm works by performing stage 1 – motion to goal until it encounters an obstacle.

This can be seen in Figure 2 from $q_{start} \rightarrow q^{H_1}$

q^{H_1} is described as a "hit point" in which case it has encountered an obstacle (Stage 2) and must rotate and follow the obstacle (Stage 3). The robot performs wall following until the goal is visible in the camera view again.

This can be seen in Figure 2 from $q^{H_1} \rightarrow q^{L_1}$

From the hit point, the robot can perform wall following on either the left or right side. The wall following functions using a PID controller like motion to goal. Wall following is accomplished by maintaining a set distance from the wall. If the robot gets too close, the robot turns away. If the robot gets too far, it turns back towards the wall.

Once the goal is visible in the camera view again, it initiates stage 1 – motion to goal.

This can be seen in Figure 2 from $q^{L_1} \rightarrow q_{goal}$

This process repeats until the robot reaches the goal. A simple state diagram that describes the bug zero algorithm is shown in Figure 3 below:

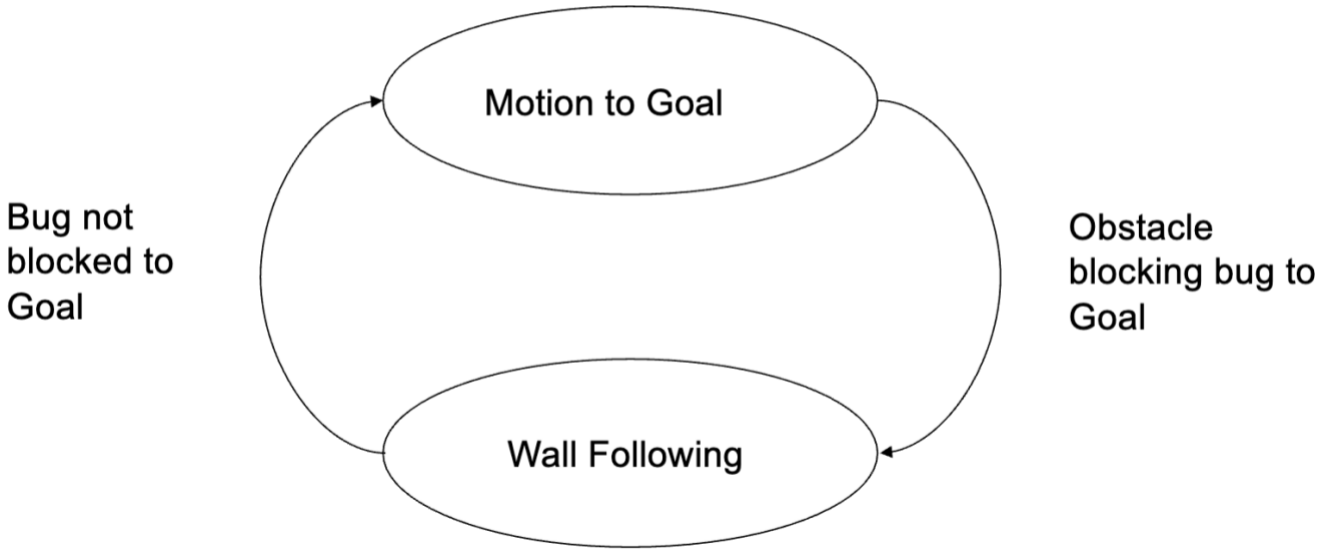

Figure 3: State diagram that describes the Bug Zero Algorithm

Conclusion

In conclusion, this lab effectively uses the robot's camera functions to perform the Bug Zero algorithm while avoiding obstacles using its LiDAR sensor. It implements previous concepts such as PID and wall-following to accomplish this. The areas I found difficult are getting the right KP values, front distance sensor ranges, and other metrics to make the robot navigate both mazes smoothly. Although I was able to get the core logic correct, most of my time went into figuring out these correct values so the robot doesn't crash into walls or run out of bounds. Additionally, an issue fixed for one maze file creates an issue in the other. This required finding sensor values that worked for both maze files and changing class definitions. Upon finding the correct values, the tasks performed as expected.