Navigating a known Obstacle Course with the Lego Robot

Aman Hogan-Bailey

Sai Karthik Reddy Muddana Laligari

Trieu Nguyen

2238-CSE-4360-001, Professor Manfred Huber

The University of Texas at Arlington

October 23rd, 2023

Contents

Design

The constructed Lego robot is a unicycle type robot with a pivot ball located at the center-front of the robot. There are two wheels, with each wheel having a counterclockwise rotation. The robot has a total diameter of 180 mm, medium sized tires of circumference 178 mm, and uses a flat 400 RPM for the tires. This structural design was chosen for simplicity, and to avoid the more complicated turning radius calculations for 4 wheeled car robots. Additionally, the pivot ball made turning possible for the initial design, so it was a necessary attachment (Figure 1).

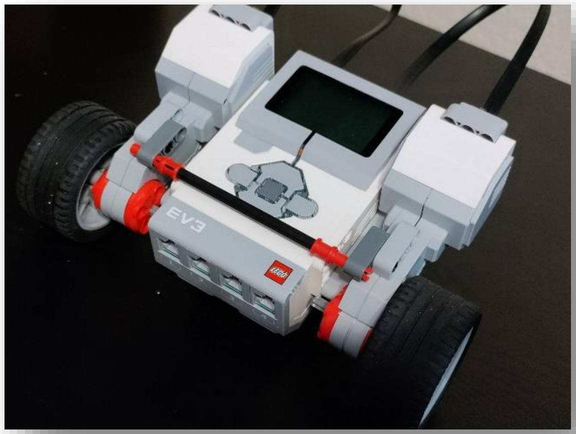

Figure 1: Final Robot Design

Movement

The robot's translational movement was determined by the angle of rotation of the tires. To convert from a desired distance to the angle of rotation of the tires, we used known equations. Taking in a desired distance, we calculated the tire angle and multiplied this by the error factor to get a more accurate angle.

$$\theta = \left(\frac{X}{C}\right)360,$$

$$\theta = \theta\gamma$$

$$\theta = angle\ of\ tire\ rotation\ [deg], \gamma = error\ factor, C = tire\ circumference$$

Turning

The robot's translation movement was determined by the angle of rotation of the tires. To convert from a desired steering angle to a rotation of the tires, we used known equations. First find the arc length of the turning circle of the robot, then divide by the tire circumference, multiply buy the error factor. And then you receive the left and right rotation of the tires

$$s = 2\pi r(\frac{\theta_i}{360})$$

$$\theta_f = \left(\frac{s}{C}\right)360$$

$$\theta_f = \theta_f \gamma$$

$$\theta_R = \theta_f, \theta_L = -\theta_f$$

$$\theta_L = rotation\ of\ left\ tire\ in\ degrees, \theta_R = rotation\ of\ right\ tire\ in\ degrees,$$

$$\theta = angle\ of\ tire\ rotation\ [deg], C = tire\ circumference, \gamma = error\ factor$$

Additional Implementation Designs

1. **Distances List** – This was a data structure used that finds the Euclidean distance between two points.
2. **Angles List** – This data structure is a list of angles using the arc tangent of the position points to find the angles in between points.
3. **Transformations** – This was a data structure used so the robot could know where it was located at. It consists of either a rotation or a translation, and these are then used to derive a transformation matrix. By extracting the rightmost column of the transformation matrix, we then derive the position of the robot. Format: $('T', dx, dy)\ or\ ('R', \theta)$
4. **Commands** – This was a data structure specifically to execute robot commands. Format: $\left('move', \sqrt{dx^2 + dy^2}\right)\ or\ ('turn', \theta)$

Navigation Strategy

Path Planning/Finding Strategy

The specific path finding algorithm used was A* search because of its completeness and optimality. Because we are using an informed search algorithm, a weighted graph needed to be maintained, which was done so using a custom Node class. The specific implementation detail that out admissible heuristic is the Euclidean distance rather than the Manhattan distance. The only downside to our approach is the space complexity, as we needed to store all nodes in memory.

Obstacle Mapping and Detection

To avoid obstacles, we used a grid-based obstacle map. The obstacle map was generated using the positions of the obstacles in the environment. For each node in the graph, we checked whether it was occupied by an obstacle. If it was, we marked the node as blocked.

Once the obstacle map was generated, we used it to guide the A* search algorithm. The A* search algorithm would only consider paths that did not pass through blocked nodes.

This approach to obstacle avoidance is simple and effective, but it does have some limitations. One limitation is that it is only effective for obstacles that are known in advance. Another limitation is that it can be computationally expensive to generate the obstacle map for large environments (Figure 2).

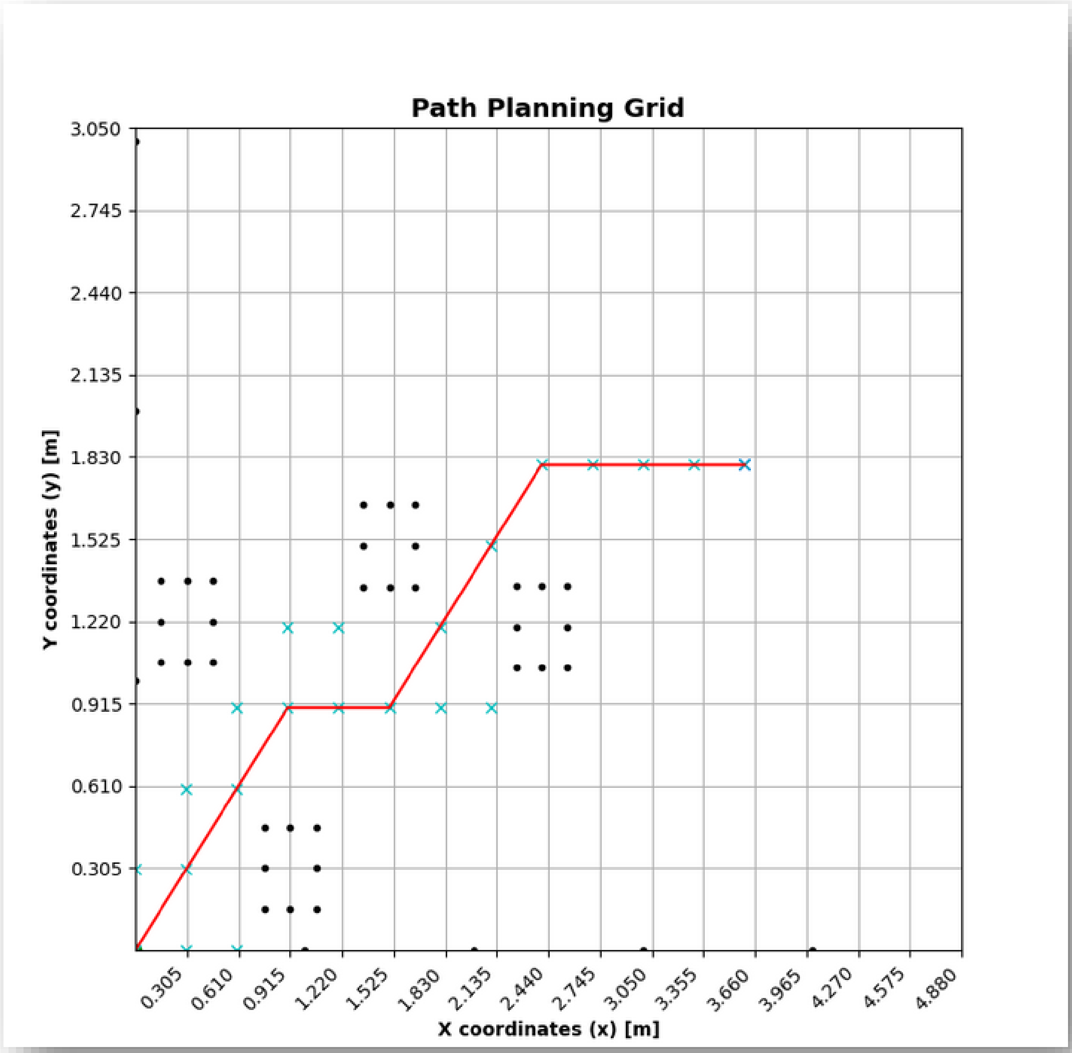

Figure 2: Matplotlib Grid generated via A search and obstacle mapping.*

Obstacle Drawing

For our obstacle drawing algorithm, we take in each point and calculate the distance from that point to each obstacle in the environment. We then draw a box around the given point with a radius equal to the .305/2 (Figure 3).

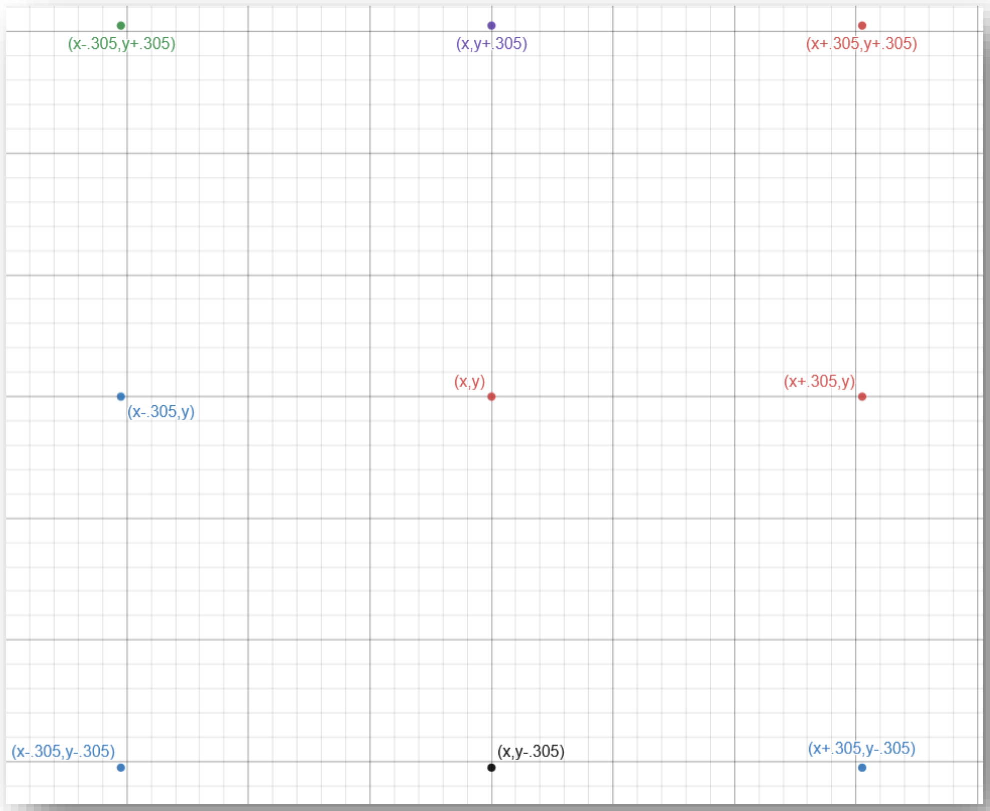

Figure 3: Obstacle visualization

References

Sakai, A. (n.d.). Matplot Cell decomp and Obstacle mapper. Retrieved from https://github.com/sponsors/AtsushiSakai

Hogan, A. (n.d.). A* Search. Retrieved from https://github.com/AmanHogan/AI_StateSpaceSearches/blob/main/StateSpaceSearches/a_star.py

Wikipedia. (2023, October 19). A* search algorithm. Retrieved from https://en.wikipedia.org/wiki/A*_search_algorithm