Introduction

The purpose of this project is to create a simulated mobile robot capable of navigating a maze-like environment to reach three separate, predetermined goals. This robot is a differential drive robot with an onboard LIDAR sensor for obstacle detection. The path planning algorithm employed is a version of the A* algorithm.

A template containing launch files for Gazebo and ROS, default maps, a default differential drive robot, and a default 2D LIDAR sensor programs were included for use. It is the assumption of this report that the reader is familiar with the contents of these files and the meaning of their applicable terms and concepts.

The robot

The robot used in this project was largely based on the default differential drive robot with the only modifications being the addition of inertial tags and slight management of the LIDAR sensor. The LIDAR was shortened to 180° from side to side and extend 22.5° back in each direction. This is to allow for potential obstruction on the physical robot.

Mapping

The majority of issues arose in the generation of sufficient Occupancy Map and will be discussed in the "Major Hurdles" section. Within the ROS environment, all mapping related files were maintained in the 'mapping' package with the two scripts being map_node.py and mapping.py. map_node.py was a tertiary default script provided during the Module covering mapping and provided the framework for subscribing to LIDAR data under the /scan topic, and publishing a map under the /map topic. It also generated the 2D map plot (Fig 01).

Mapping.py was used to analyze the /scan data and produce an occupancy map. The nodes of the occupancy map had one of three conditions, Unknown, Unoccupied, or Occupied. These values were assigned via a confidence score. This confidence score would increase for any sweep that determined the node unoccupied, decrease for any node found to be occupied during any sweep. The concept is that successive accurate sweeps would outnumber inaccurate sweeps so the confidence score would move towards their respective maxima. Node state would be assigned based on the confidence score, rather than the results of one sweep, making the map less susceptible to noise.

Nodes would be made up of successive breaking up of the map area, dividing each layer into four 'children' of the 'parent' node. These children would be further broken down until a maximum depth of layers was achieved. Further, if a lower level has all four children of the same value, they would combine back into a larger parent node. The center of these nodes would become a point through which a path planning algorithm could operate.

Path Planning

The path planning portion was housed in the path_planning package. Inside the path planning package was an A* planner (a_star_planner.py) and a path planning ROS node file, path_planner_node.py. The path planning node handled subscribing to the topics /map, and /goal_pose_{num} where num is 1-3. It published to /goal_pose, and /path_msg, which contained the planned path.

A_star_planner.py handled the logic for determining the path to the goal via the A* algorithm. Some important concepts here are that the planned path will need to traverse areas that are currently unknown. Paths will be frequently updated. Another key detail is that, unlike a typical A* algorithm with a weight and a heuristic, this A* also required a 'penalty' value. The weight value was based on the distance between points in the path and made the shortest path preferential. The heuristic was based on the direct path from the robot's location and the desired

goal (chosen based on shortest linear distance from problem start). The penalty value was an addition based on the state of the nodes traveled through. Obstacles were prescreened to not be considered in the selected path. In no condition should the robot select a path through an obstacle. Nodes that were known to be unoccupied but were adjacent to an obstacle was assigned a penalty value of 10. Unknown nodes were assigned a penalty of 5.0, to influence the robot to select known paths wherever possible. Unknown nodes that were adjacent to occupied nodes were assigned an infinite penalty score to prevent a robot from going near an obstacle if the state of the node isn't known. Unoccupied nodes not near an obstacle had a penalty of 0, to influence selection of paths that use unoccupied space away from an obstacle.

Major Hurdles

The most significant hurdle was the creation of the occupancy map. For some reason, there was a node per parent node that was not updating the state to unoccupied. This prevented the occupancy map from ever combining and created a combination of grid and column/row patterns (see Figure 1).

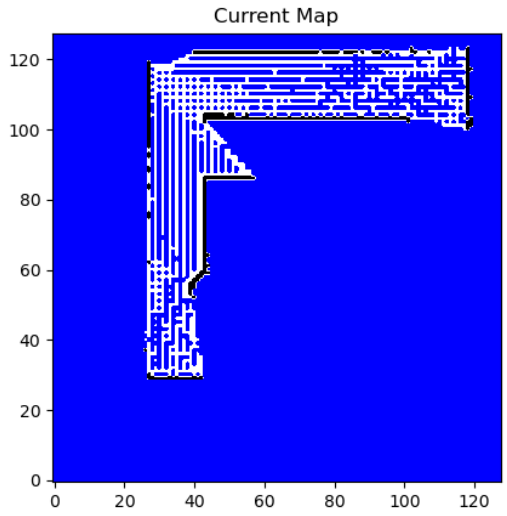

Figure 1 Occupancy Map displaying grid or line pattern. White is Unoccupied, black is occupied, and blue is unknown. Note all nodes are in their smallest leaf node size.

This means path planning would step through the center of a series of very small steps. Major attempts at correcting this include specifically targeting the center of each child node for assessment and inferring the value of one unknown node based on the confidence scores of all surrounding nodes being sufficiently high. Unfortunately, this was very computer intensive and did not sufficiently run on my computer. This was the most significant problem in this project and took the majority of time to attempt to debug.

With more time I would like to fully implement my state inference function, or more properly correct the bug that is leaving the occupancy map incorrectly represented. My next major steps were to have my map_node.py file subscribe to /path and represent the planned path in the plot display. Finally, implementation of the diffdrive_pid.py to subscribe to the /path and /goal_pose_{num} topics and drive the robot through the intended path.

Running the Code

There are multiple launch files within this project. To begin the ROS simulation and the Gazebo environment, launch final_gazebo_spawn.launch.py in the robot_spawner_pkg package. Each package has it's own launch file, with robot_control being teleop_test.launch.py. In the mapping package is the mapper.launch.py. Path_planning has the launch file path_planner.launch.py. Teleop_twist_keyboard is required for each of these files. Also within the path_planner package is the full_system.launch.py, which launches all required packages although diffdrive_pid.py was commented out for debugging. Teleop_twist_keyboard remains the best way to operate the robot for any package configuration at this time.