Unified Robotics IV: Navigation

Zeynep Seker
Robotics Engineering
Worcester Polytechnic Institute
Worcester,USA
zseker@wpi.edu

Connor Craigie
Robotics Engineering
Worcester Polytechnic Institute
Worcester,USA
cacraigie@wpi.edu

Peter Dentch
Robotics Engineering
Worcester Polytechnic Institute
Worcester,USA
pdentch@wpi.edu

***Abstract*— The TurtleBot Burger is programmed to localize itself in an arbitrary maze and map its environment using a LIDAR. The robot finds the areas of interest and reports back that it has completed exploring the maze. Once the map is complete, it generates optimal paths to drive to given poses using A-star.**

I. INTRODUCTION

In order to navigate the given arbitrary maze the TurtleBot Burger was programmed to get LIDAR readings of its environment. After establishing the obstacles, the system then generates a c-space around them in order to treat the robot as a point particle. This way, the padded map can be used to find the possible frontiers. The robot was sent to the middle of the largest frontier. Each time it reaches the centroid of the frontier, it recalculates the new possible frontiers. Once it runs out of possible frontiers, it is reported that the map is fully explored and is ready to navigate. The robot was sent to its starting position by assigning a 2D Navigation Goal to it. Even when it is kidnapped, it still remembers the explored map and relocates itself.

II. METHODOLOGY

A. Path Planner

The *path_planner* node is a ROS node repurposed from previous exercises performed by the TurtleBot in which it was required to generate a trajectory for navigating a map. This node subscribes to the *turtlebot_slam_gmapping* node from which it initially receives and updates the world map seen by the robot throughout its runtime. This map is a grid of 5 centimeter square cells adequate for representing the world traversed by the approximately 10 centimeter diameter TurtleBot. Each cell has both a 2-dimensional coordinate and a traversability value ranging from zero to 100. Zero being certain of no obstacle to the robot and 100 being a detected wall or obstacle in the map. Additionally, cells which have not yet been seen by the sensor or remain undetermined have a value of negative one for indicating their status.

Once borders of known inaccessible locations to the robot are generated from its accurate 360 degree field of view LIDAR sensor, a padding of two additional grid cells is added to convert the map to an area of possible occupancy. This area of all known traversable locations for the robot to freely and safely move is known as the configuration space or c-space for short. Once complete, the fully-updated padded map is published in order for the *find_frontier* node to parse. This node looks for areas of interest in which the robot should travel in order to continue building the map. The *path_planner* node in return subscribes to *find_frontier* as it publishes a single grid point acting as the goal point to which the robot should navigate. With this information, the A-star search algorithm is then used to find the optimal path for generating a robot motion trajectory for navigating to this point. It achieves this while using its current location as the starting point and the received goal as the end point.

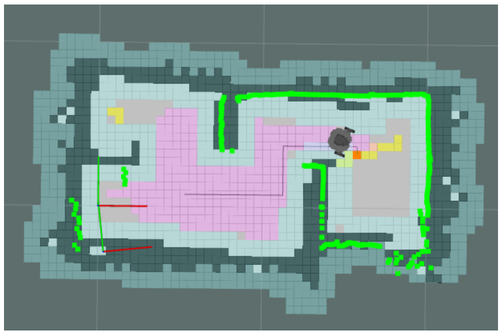

Fig. 1. Example Path and Goal

A-star algorithm finds a path by initially applying a breadth-first search to all neighboring grid cells from the starting point, searching closer cells first as it expands the map. The optimal path is defined as one which reaches the goal point with the least cells traversed, however another cost is added in this algorithm for moving to different cells. This is simply a cost of one for any possible single-cell movement, either straight or diagonally across from the current cell. Finally, a third cost is customly applied for this project such that the paths have a bias to turning as little as possible. This was done so that the robot's believed location on the map has less opportunity to drift as it makes more moves. Once complete, the path is published to the *path_planner/path* topic for the *frontier_finder* node to handle and move the robot accordingly.

B. *Frontier Finder*

The primary goal of *Frontier_Finder* is edge detection. The node subscribes to a constantly updating c_space map. To detect edges, the node checks every index of the c_space mapdata. The large space of data is iterated through very carefully. It is imperative that the c_space map is sent to the *Frontier_Finder*. This assures that frontiers are build between the c_space. If frontiers are built directly adjacent to a wall, there is a chance that frontier centroids could be interpreted incorrectly and calculated within the c_space.

The map image is generated as a large array of 8-bit integers. Using predefined mapdata, the index of a map can be returned as a given (x, y) coordinate. For this exact process, the function *grid_to_index* was developed. Using this function, the mapdata can be indexed cleanly within a double for loop. In such a case, the external loop iterates height and the internal loop iterates width. For each of the indices of the map we determine its associated 8-bit value.

The data structure associates a value of 100 as a wall and -1 and an unknown location. In the case of edge detection the program looks for a value of zero. These zero values are used to represent a traversable path. The edge detection algorithm checks if the current index of the map is zero. If true, it then checks the neighboring indices for their values. If a neighboring value is found to be -1, it is understood that the current index is an edge. In this case the newly found edge location will need to be associated with a frontier.

A frontier is stored as a list of tuples. However, multiple frontiers could exist in a single map at any given time. Therefore, it is important to derive a field of the frontier class called *self.frontiers*. The field *self.frontiers* is built as a list of frontiers. As edge detection occurs the program also detects if there are any adjacent frontiers. If an indexed location is both on an edge and adjacent to another frontier, it is then appended to that frontier. Otherwise, if an edge cell is not surrounded by another frontier, It becomes the first tuple in a new frontier. This new frontier is then added to the global field *self.frontiers*.

Once the entire map is iterated, the node *Frontier_Finder* attempts to derive the centroid of the largest frontier. The program iterates through the list of frontiers and compares them. The function eventually returns the largest frontier measured in array length. Initially, the frontier centroid was calculated as the average of all tuples. However, due to the concavity of some frontiers, the centroid may be developed in and unknown map index. A path into unmapped territory could potentially be hazardous for the robot.

To solve this issue, only points in the frontier are considered. Since a frontier is derived as a list of neighboring points, it can be assumed that the order of the frontier directly

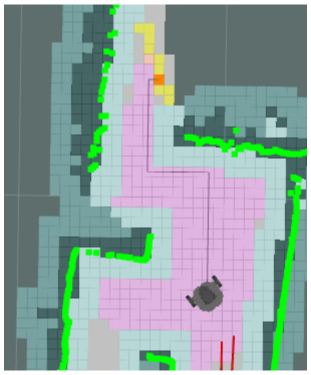

Fig. 2. Example Frontier and Centroid

correlates with the order of the physical points on the map. Therefore, the centroid of any given frontier can be found by indexing halfway through the list of tuple points. This centroid is published as a *PoseStamped* goal to *path_planner* and eventually used to navigate that frontier. Once a path is completed, *Frontier_Finder* will begin searching the map again for new frontiers and the cycle continues. The cycle only terminates when *Frontier_Finder* can no longer distinguish any frontiers. In this case, the program then begins looking for *nav_goals* produced in Rviz.

C. *Robot Control*

A special node called *Robot_Control* was created to handle the actuation of the robot. This node subscribes to the centroid location from the *frontier_finder* as a */goal* topic and an optimal path from the *path_planner* node as a */path_planner/path* topic. The *Robot_Control* node also listens to the *tf.Tranform* to compansate for the map's shift due to the LIDAR's instability. This way, even when the robot's belief of the map's angle is different, the *tf.Transform* rotates it back to the true angle. Finally it publishes a start position received from the odometry which represents the current location of the robot.

The current position of the robot was updated using the *update_odometry* function. These pose values are then used in various functions for driving and rotating to the end goal pose. The drive of the robot was achieved by calculating the euclidean distance between the robot's current position and the goal position. This value is calculated in the *drive* function. The *go_to* function uses the robot's current odometry to obtain the current position, and uses the *drive* function to drive to a specific location.

The moment the robot reaches the location within a certain tolerance, it stops for rotation. There are multiple functions in this node that controls the rotation of the robot. In order to avoid any abundant rotation while reaching the desired pose, the received goal angle was normalized by giving the rotation a specific range. This way, it was assured that the robot could only rotate between 0 and 2 pi radians. Afterwards, the error between the current and the goal angles were calculated and an optimal rotation direction was selected.

Once the robot completes the path, the node publishes a */start_new_cycle* message that repeats the entire process of finding the path. Finally, the *Robot_Control* node includes all the tolerance values for the drive and rotate functions as well as the rotational and linear speeds. *Figure 3* represents the overall flow of the algorithm.

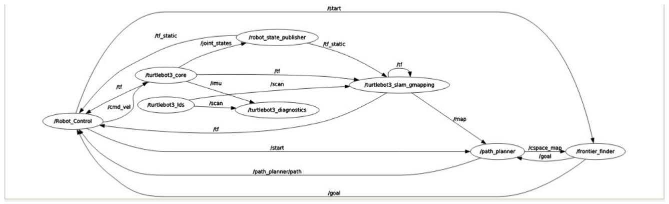

Fig. 3. Overall Flow of the Algorithm

D. Gmapping

Gmapping is a given node. This node is purely implemented by the team and not altered in any way. This node publishes to the */map* topic which is of type Occupancy Grid. An occupancy grid is comprised of a header object and a field called data. The data field is an array of 8-bit integers which is the foundation for all programmed path generation and *c_space* development. Map data is published at a regular rate to the ROS Master through the aforementioned topic. This topic is then subscribed to by the path planner node.

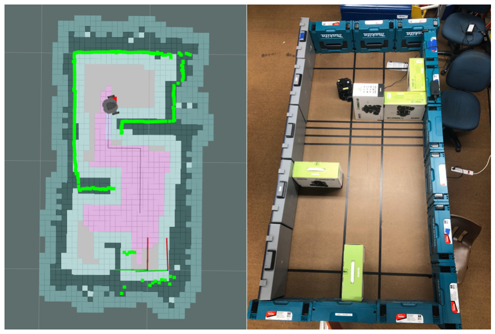

Fig. 4. Comparison Between Generated Map and Real Life

E. Simulation

The simulation of the TurtleBot used for this project was done in Gazebo Simulator, a popular tool for this application. The use of the simulation was specifically for the purpose of running a pre-made launch file for generating a simulated map for the robot to traverse. Many of the deficiencies possessed by the physical robot are taken into account in the simulation such as sensor noise. This makes for an ideal platform for testing and gauging the performance of the real robot on the much simpler test fields it must successfully navigate. While working with a TurtleBot in a simulated environment, the difference in performance quickly became apparent, as the real robot would experience a greater drift of its believed location on the map.

F. Launch File

For ROS programming, launch files are very powerful tools. The nodes require to be executed in a very specific order and in a timely fashion. In order to avoid a runtime error, a launch file named *final_launch* was created for the final demonstration of the robot. This launch file registers the robot model and launches the *gmapping* node, the *Rviz* configuration file that displays the published topics, *Robot_Control* node, *frontier_finder* node, and *path_planner* node respectively.

III. RESULTS

By following the instructions specified in the lab assignment and researching example robots, the Turtlebot Burger was programmed successfully. The robot was tasked to map an arbitrary maze, store the final product, and navigate through it to desired locations. Using ROS publishers and subscribers, messages were shared between nodes. By using each calculation in a separate node, a communication was established between nodes which eliminated the need for service calls. The robot was able to pad the map with a c-space and avoids crashing into the obstacle. Because real life is much different than the simulation due to network and hardware issues, it was decided that having two layers of padding would be ideal to avoid any accidents. It found the future frontiers by using neighbours of 8 and expanded the frontiers using A-star search algorithm. The algorithm has a high success rate at avoiding obstacles and c-space while generating a path. After the map's completion, the robot was able to locate itself in the maze after being forcibly displaced assuring the overall completion of the challenge.

IV. DISCUSSION

Due to significant differences between physical fields and simulation, we found it is important to ensure that ample testing of the robot on the physical mazes. Though the information pulled from sensors added noise and drift while running a simulation possesses discrepancy persisted between it and the actual model to the point of tuning the simulation became a dead end for successful completion of the challenge. The creation and parsing of frontiers by the *frontier_finder* node was an issue which was never fully addressed throughout this project and could have aided in our ability to reliably locate frontier goal points. We frequently experienced discontinuity between relatively large located frontiers which should have been grouped together, as their borders often overlapped or were directly neighboring those of other frontiers. Ideally a fix for this issue is that a function be made to check neighboring frontiers. If frontiers neighbor each other, combine them so that proper centers can be more reliably generated for sending as goal points for the robot.

Map resolution became an integral issue in the development of our system. Creating a proper path required a dense enough *c_space*, yet a fine enough resolution of the map. Late trials of the software focused around balancing the padding of the *c_space* and the resolution of grid size. If either padding or grid size became too large, the *c_space* would unintentionally envelop the robot into small corners and frontier centroids would be unreachable. However, if resolution became too small, then the map would become considerably harder to calculate as its data structure would increase significantly. The robot eventually was able to consistently repeat the task with a grid resolution of 0.05m and a padding value of two. At this point we have come to realize that the overall size of the map data could have been changed within the g mapping launch file. If the overall *OccupancyGrid* size was smaller, calculations could have been made considerably faster on a much more finely resolved map. An increase in resolution would have significantly increased robot pathing and mapping accuracy.

V. Conclusion

The final lab of RBE3002 encourages students to implement a ROS architecture with the use of LIDAR in developing frontier detection and path generation. The lab prepared students for logically high level programming. This lab ambiguously imposes the challenge of navigation. The solution to achieve the final result can be solved in a variety of ways. This lab asks you to exercise technical skill, however it is this ambiguity that exercises a student's complex problem solving ability. This final project well rounds engineers and forces them to approach a problem from multiple angles to achieve an optimal solution.