RBE 3002 Final Project: Navigation

Bhon Bunnag
Yil Verdeja
May 1, 2018

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

With the increasing demand of autonomous vehicular systems, navigation has become one of the most important field in robotics. With a focus on navigation, position estimation and mapping, the function of this project is to help in further understanding the ROS (Robot Operating System) structure and implement:

1. Forward and inverse robot kinematics
2. Path planning algorithms
3. Noise filtering
4. And simultaneous localization and mapping algorithms

From the start of the project, a kinematics model for the robot was implemented (using differential drive motion, trajectory generation and odometry reporting), as well as a real-time A* path planning algorithm for constant replanning and for optimizing navigation through a dynamic environment. This section of the project focuses on combining every aspect from previous sections, and developing ROS packages and nodes for the turtlebot3 burger to allow it to navigate in an unknown maze from a specified corner to its diagonally opposite corner in the shortest path. From the project specifications, the objectives of this project are as follows:

- ❏ Through frontier exploration and the use of the ROS navigation stack, explore the entire maze and save a 2D occupancy grid in a map file
- ❏ Modify the A* heuristic to penalize the number of turns in a path
- ❏ Find the lowest cost path from start to goal, generate waypoints and navigate autonomously through the map using the optimum path (ideally the fastest route)
- ❏ Find the best turning radius and add additional waypoints to turn smoothly instead of stopping and turning.

1.1 Equipment

To complete this project, the following are necessary:

- ❏ A computer running Linux on the Ubuntu OS
- ❏ Turtlebot3 Burger Robot and corresponding ROS (kinetic) packages

1.2 Turtlebot3 Specification[1]

With a maximum translational velocity of 0.22 m/s, and a maximum rotational velocity of 2.84 rad/s, the turtlebot3 burger weighs roughly 1 kg and has a size of 138mm × 178 mm × 192 mm (L × W × H).

Figure 1. TurtleBot3 Burger Model

To sense its environment, the turtlebot3 burger is equipped with a 360 Laser Distance Sensor (LIDAR), which is a 2D laser scanner that collects a set of data around the robot to use for SLAM (Simultaneous Localization and Mapping)[2].

1.3 Maze Description

The maze that the robot will travel through is described below:

1. It is composed of walls such that the minimum distance between 2 adjacent walls on at least one path from the start to goal is of 40 cm (enough space for the robot to traverse around it)
2. The walls are high enough to be detected by the LDS sensor on the turtlebot 3 (therefore the walls are much higher than 192mm)
3. The start of the maze is located at one of the four corners and the goal is at the diagonally opposite corner. The time starts when the robot leaves the first cell.

[1] (n.d.). TurtleBot3 - ROBOTIS e-Manual. Retrieved April 25, 2018, from http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/

[2] (n.d.). 360 Laser Distance Sensor LDS-01 (LIDAR) - Robotis.us. Retrieved April 25, 2018, from http://www.robotis.us/360-laser-distance-sensor-lds-01-lidar/

4. Multiple paths to the destination may exist (its depends on the robot to choose the most optimal path)

2 METHODOLOGY

This section will go through the steps that were taken to complete this project. It is built on previously completed material such as the implementation of a kinematics model for the robot, obstacle expansion, re-planning, and obstacle avoidance - which will be explored further in *section 2.1*. This final project was broken up into two tasks: exploration and navigation. The exploration task shows how the objective of exploring an unknown environment and creating a map from its sensor readings is completed. Similarly, the steps taken to complete the navigation task verify how the robot will traverse from one corner of the map to the opposite corner in the shortest amount of time.

2.1 Preliminary Material

This subsection will summarize the preliminary steps that were taken in order to complete this project. For a more in depth description, see previous reports.

2.1.1 Turtlebot3 Kinematics

The script *robot.py* handles all of the robots movements and kinematics. The main function that will be called will be the *navToPose* function that calls both the *driveStraight* and *rotate* functions. As the name suggests, the *navToPose* function navigates from an original pose to a goal pose. It does this by first rotating the robot towards the direction of the goal, then driving straight towards it, and finally rotating to its goal orientation. The *navToPose* function is iteratively called for every intermediate waypoint before the goal point that is generated by A*.

Another function that can come to use for better maneuverability of the robot is the *driveArc* function which taking a radius, a speed, and a final angle, drives a specified arc. If implemented correctly in the project, this function may help the robot turn without stopping.

2.1.2 Obstacle Expansion and Grid Size

The script *getGridCellData.py* focuses on processing all the environmental data collected by the Lidar sensor. Two of its main functions are to expand the obstacles and affect the grid size. The *C-Expansion* function grabs every node in the occupancy grid that is an obstacle (~100), and depending on the size (i.e. size *n*) of the expansion, it will make its neighbors (i.e. *n* neighbors away from origin node) and set them as obstacles on the occupancy grid.

The second function is the *resizeMap* function which changes the size of the grid by a certain interval. Increasing the interval, increases the size of each grid cell. In order to effectively get all the nodes within this newly transformed grid, the cells around them are chosen to cluster into one whole cell depending on the *interval* factor. A grid size that is too big results in very precise movements that the robots odometry errors is not able to handle. However, making a grid size too small, allows for an increase in errors as it over generalizes the map.

2.1.3 A* Algorithm Implementation

The final script is the *searching.py* script which handles the A* algorithm and its heuristic for optimally traversing the map from origin to goal cell. The preliminary heuristic of the searching algorithm is defined by the sum of g(n), the manhattan distance (with diagonals) from the start node, and h(n), the euclidean distance to the goal node. If utilizing a cost map, it takes into account the probabilistic cost of traversing to a node that is open or blocked (0 - 100). By searching the map with the provided heuristic, the A* algorithm retrieves the most optimal path from start to finish. From the final path, intermediate waypoints are generated at every node where a turn occurs.

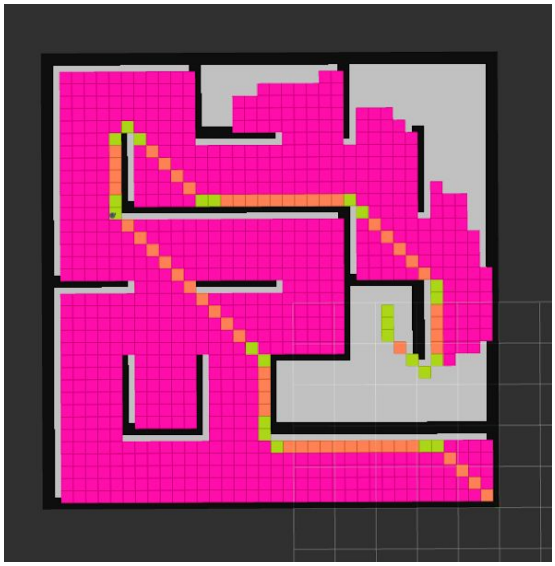

Figure 2. A Search Algorithm with Search (Pink), Final Path (Orange) and Intermediate Waypoints (Green)*

2.1.4 Obstacle Avoidance

The use of a service was implemented to continuously call the A* algorithm after the robot finished traversing an intermediate waypoint. For every waypoint, the robot would search again for a new path as changes to the map may have been made.

2.2 Exploration

With the help of the ROS navigation stack library, frontier exploration was implemented in order to explore an unknown map. This section describes how mapping of the environment with the laser scan took place.

2.2.1 SLAM and Gmapping[3]

SLAM (Simultaneous Localization and Mapping) refers to the problem of trying to simultaneously localize (i.e. find the pose of) the robot using sensor readings of its surroundings, while at the same time mapping the structure of that environment.[4] In ROS, there exists a gmapping package that provides laser-based SLAM as a ROS node called *slam_gmapping*. With *slam_gmapping*, a 2D occupancy grid map is created from the laser and pose data collected from the robot.

An occupancy grid represents a 2D grid map where each cell represents the probability of occupancy. The probabilities range between 0 and 100, where 0 defines an open space, while 100 defines a blocked space. Unknown cells are denoted with a -1.

2.2.2 Frontier Exploration

A frontier is defined as the border where an open node and an unknown node meet. These are areas that must be explored in order completely map the environment that the robot is located in. In the *searching.py* script, by subscribing to the *map* topic which returned the *slam_gmapping* occupancy grid, the automated search was implemented within the *automatedSearch* function.

As seen by the function, if a node in the occupancy grid had a probabilistic cost between 0 to 30, and neighbored an unknown node of value -1, then it was considered a frontier. However, a single node on the map could not be considered a frontier as the *slam_gmapping* node is not accurate enough to get to that degree of certainty, but also misreadings or noise could appear that would misguide the robot.

To fix this issue, the frontier had to be of a certain size. Too small would cause the robot to travel to fake frontier borders, while making the frontier size limit too big would stop the robot from exploring actual frontiers. Considering the size of the grid, the frontier size limit had to be greater than 2 bordering nodes in order for it to be considered a frontier. Once found, the robot

[3] (n.d.). gmapping - ROS Wiki - ROS.org. Retrieved April 29, 2018, from http://wiki.ros.org/gmapping
[4] (2016, May 13). An Introduction to Simultaneous Localisation and Mapping | Kudan. Retrieved April 29, 2018, from https://www.kudan.eu/kudan-news/an-introduction-to-slam/

would navigate to it. After mapping the whole unknown map, the search would stop including the robot.

The heuristic that was implemented for exploring frontiers was a greedy one. It would explore the closest one to the robot at that instance. Considering the map was small and compact, this method was justified.

```
def automatedSearch(self):
        goalPub = rospy.Publisher('move_base_simple/goal', PoseStamped, queue_size=10)
        data = self.slamData.data
        resolution = self.slamData.info.resolution
        width =  self.slamData.info.width
        height = self.slamData.info.height
        origin = self.slamData.info.origin
        # Traverses the SLAM occupancy grid
        for c in range(80,height-80):
                for r in range(80, width-80):
                        # If the node is open (prob = 0 to 30)
                        if(0 <= data[c*height + r] <= 30):
                                knownCount = 0
                                goToCount = 0

                                # Check the neighbors of the "frontier" node
                                for e in self.getNeighbors((c,r),data, height):
                                        if(data[e[0]*height+e[1]] == -1):
                                                knownCount += 1
                                        if(0 <= data[e[0]*height+e[1]] <= 30):
                                                goToCount += 1

                                # If there are more than two open nodes and two unexplored node
                                # Then this is an explorable frontier
                                if(knownCount > 2 and goToCount > 2 and (c,r) not in self.goalsTried):
                                        # Places a frontier goal within a list of goals to attempt
                                        self.goalsTried.append((c,r))
                                        point = Point()
                                        point.x = (r+0.5)*resolution + origin.position.x
                                        point.y = (c+0.5)*resolution + origin.position.y
                                        goal = PoseStamped()
                                        goal.header =  self.slamData.header
                                        goal.header.frame_id = "map"
                                        goal.pose.position = point
                                        goal.pose.orientation =self.orientation
                                        print(goal)
                                        goalPub.publish(goal)
                                        return

        self.searchComplete = True
        print("Search Complete")
        return
```

Figure 3. Automating Exploration through Frontier Exploration

2.3 Changing the Map

After mapping the whole closed environment with the *slam_gmapping* node and frontier exploration, the map needed to be changed in order to allow for better maneuverability when navigating through it. As mentioned previously in the preliminary steps prior to this project, a

grid size and an obstacle expansion were implemented. In this lab, since the robot had to navigate the real-world rather than the simulated world, some changes had to be made.

2.3.1 Grid Size

For occupancy grid used in the prior lab was with an interval size of 4. The reason for this was because this interval divides without remainder for the 384x384 grid (into a 96x96 grid) and reduces the grid size so that A* can be called regularly without taking too much time. However, this large interval was implemented in a simulation that had large open spaces for the robot to navigate through. Since the real-map of the robot consisted in walls that were 40 cm apart, it would be much more compact, thus the grid size had to decrease to an interval of 2. This change not only would stop obstacles from expanding too much, but also it would allow the robot to have more precise movements. Nonetheless, if the grid size is not large enough to make it represent a point on the simulation, then errors of the robot meeting the surrounding can occur.

2.3.2 Obstacle Expansion

Expanding the obstacles too much could result in blocked off open pathways. Therefore the obstacle expansion algorithm was kept to expand the first neighboring nodes of occupied cells.

2.4 Navigation

After SLAM was performed to obtain a map, the map was saved via `rosrun map_server map_saver`. The *2d pose estimate* command was used to provide an initial guess in order to help the robot localize, and navigation begins after AMCL is activated. The navigation was broken down into two different stages: A* pathway mapping and robot kinematics, as explained in 2.1.3 and 2.1.1 respectively.

2.4.1 Modifying A* Heuristic

For the final lab, the robot was occasionally providing sub-optimal paths. This was because the cost g(n) was equivalent for traveling straight and diagonals.

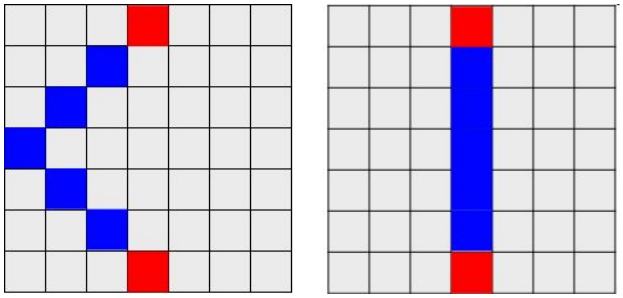

Figure 4. Different paths with equal costs[5]

This was fixed by changing the weight of traveling through a diagonal. Traveling in a straight line still costs 1. However, traveling diagonally now will costs $\sqrt{2}$. In other words, the cost g(n) is determined by the Euclidean distance from the current node.

2.4.2 Localization

Localization was performed using AMCL via *roslaunch rbe3002_d2018_final_gazebo final_run.launch map_file:=./map.yaml*, as provided. AMCL uses a particle filter to determine its location, which will eventually converge on a location and make /odom more accurate over time.

3 RESULTS

This section serves to describe how the robot did in the real-world to complete its tasks. The program on the turtlebot was implemented to both do autonomous exploration of an unknown environment through the use of the Monte Carlo Localization and to autonomously explore the map using the A* algorithm.

3.1 Overall Findings

3.1.1 Exploration

By using the gmapping package libraries included in ROS, and through frontier exploration, the robot was successful in mapping its environment. The figure below shows the returned map file from the demo, prior to inflation of its obstacles. As seen, the white portrays the open space (0), the black portrays occupied space (100), while they grey portrays unknown space (-1).

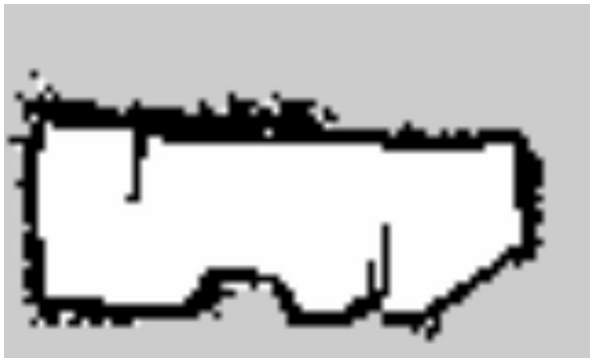

Figure 5. Map from environment in demo run

[5] In the original A* algorithm, these two different paths are evaluated as having equal costs.

Although this was deemed a success, as seen in the image, there exists nodes that are on the outside border of the occupied cells (perimeter) which are considered as *known* or open spaces rather than unknown areas. Nonetheless this noise was minimal in even affecting the results of the exploration. After the inflation of obstacles, most of those noisy nodes will be covered with a *known* cell.

3.1.2 Navigation

After receiving the map from the exploration phase, and adjusting it to its corresponding grid size and obstacle expansion, the second phase of the demo began. Through localization, the robot was able to find its relative position to the map. From there, using Rviz, a pose on the opposite side of the map was selected for the robot to navigate to. With the implemented A* algorithm and localization, the robot was able to find the optimal path and start navigating to its intermediate waypoints.

In various runs beforehand, the robot would navigate effectively through the map to the desired pose. However, at times, it would skim certain walls. During the demo, the turtlebot kept coming in contact with the same wall which hindered its navigation to the goal pose.

4 DISCUSSION

From the results of the demo, the purpose of this section is to state any observations, and analyze what went wrong and how they could be avoided/improved with different implementations. The observations include experiments on localization methods, as well as false sensor readings from the robot to the map. This section also describes means to improve on certain aspects, such as: smoother navigation and the removal of sensor noise.

4.1 Observations

4.1.1 AMCL vs SLAM

Initially, SLAM was used to perform localization after mapping is complete. We believed this method would be more accurate, since further mapping (while localizing) would correct errors from the initial mapping. When we ran experiments with this method, we found that although initial errors are corrected sometimes, more errors are also mapped. We determined in the end that using AMCL on the saved map was more accurate.

4.1.2 Detecting Fake Frontiers

Although we do have a method in differentiating between real frontiers and fake frontiers as mentioned in 2.2.2, it is still not perfect. As a result, the robot occasionally tries to navigate to these fake frontiers, and wastes a lot of a time as a result. We relied mainly on the fact that the navigation will eventually abort this impossible navigation and move on to the next one.

4.2 Improvements

4.2.1 Failure of Navigation

Although this was successful in prior runs [see video], in the actual PDR it failed. We have a few hypotheses to why it failed. Firstly, we believe there may be a mapping error due to noise. The mapping snipped out the edge of one specific border, so it appeared shorter than it really is. The A* algorithm, in its attempt at finding the optimal path, made very sharp turns. These two issues combined caused the robot to keep crashing of that border, and could not go further.

Though the former error could not be trivially fixed, the latter can. We believe that by either using a wider width for C-Expansion or changing the interval of grid clustering from 2 to 3, the border would have expanded and force the A* algorithm to take a wider turn. This would ultimately make the robot avoid the wall.

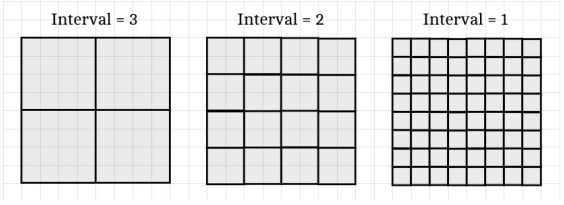

Figure 6. Different intervals subject to different grid sizes[6]

4.1.3 Detecting Fake Frontiers

The most obvious way to solve this problem is to determine if the frontier could be reached before calling the Navigation Stack. We would use the A* algorithm to determine whether the path is reachable, and if not it would move onto the next candidate frontier. This idea was conceived of before the demonstration, was not implemented due to time constraints.

[6] The width and height of the map is an integer value that is depicted by the sum of the remainder and the quotient of the previous width and height

5 CONCLUSION

This lab provided a further experience of using ROS to control the robot in a known and unknown environment, in both simulation and the physical world. The robot used SLAM in order to localize and map an unknown environment. AMCL was then used to localize in a known environment.

5.1 Future Implementations

The following are future implementations that can provide significant benefits to the robotic system, specifically in limiting waiting periods on the robot through rotations and/or actual stops.

1. **Turning without stopping:**
   In order to increase the speed of the robot in general, the robot should be able to perform arc-drives (moving forward while rotating). This was implemented in Lab 2 via the *driveArc* function, and we would need to add an intermediate function that takes in as an input the desired destination and executes this function correctly.

2. **Finding the minimum angle rotation:**
   In many instances, the robot did not rotate optimally to the desired angle. This was due to the overflow of angle values (it changes from 180 degrees to -179 degrees). In order to fix this, a more precise mathematical algorithm would be needed in order to detect this overflow.

3. **Backward drive to reduce rotation:**
   We observe many cases in which the robot would need to rotate more than 90 degrees before moving straight. A more efficient method would be to allow the robot to drive backwards as well. This would mean that the robot will never have to rotate more than 90 degrees.This is because a rotation of θ and a linear movement of d is equivalent to a rotation of $180 - \theta$ and a linear movement of $-d$.