Introduction to Robotics - Mini Project 2

Duarte Meneses Maria Duarte Vasco Silva

Abstract

The main goal of this project is to understand the mobile robot path planning and guidance to solve the navigation problem.

1 Introduction

We started by trying to understand what we were working with. For that we studied the references and tried to connect everything together.

After understanding how everything worked, we tried to start working with the packages, searching for how to use them and what parameters we could tweak in order to make the algorithms work well with our particular setup.

We could not show every result in this report so we arranged a Google Drive Folder with videos that we think are explicitly named. Through the videos it is possible to see the path planned by the robot (in black) and path the robot actually took (in blue).

2 Background

In this section we will present all the theoretical background needed to understand the topics present in this project. For each of the mobile robot challenges (Motion Planning and Guidance) we will present the main algorithms used and establish a notation. This challenges are part of the Navigation Problem, the main problem of this mini-project. Localization is also part of the navigation problem because the robot has to know where it is in order to know if it reached its goal. For this mini project we used the algorithm (AMCL) of the previous mini project to deal with that because it was already integrated in the launch file used for navigation and we also got good results with it in the last project. We only adjusted the number of particles to be smaller because processing all those particles was being heavy on the system.

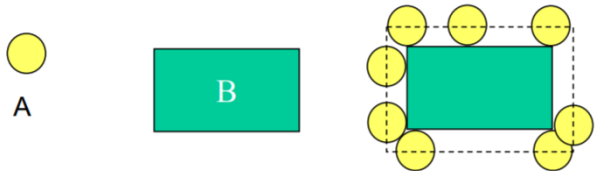

Figure 1: Transformation into a C-Obstacle

3 Robot Motion Planning

Robot Motion Planning corresponds to the problem of finding a collision free path between an initial pose and a goal pose, taking into account the different constraints (geometrical, physical and temporal).

3.1 Configuration Space

For motion planning, we use configuration space to make our life easier. So let's consider that we have our robot, A, in an Euclidean space, W, where usually ($W = R^2$ or $W = R^3$), where we have n known fixed obstacles (let's represent them with B_1, ... , B_n). We assume that: the geometry of A and B_i is known, the localization of B_i in W is accurately known, and that there are no kinematic constraints in the motion of A (A is a free-flying object).

A has multiple configurations, a configuration being it's pose (position and orientation) in F_A (robot frame, a moving frame) in relation to F_W (world frame, this is a static frame). So the configuration space, corresponds to the space occupied by all the configurations of A. The configuration space is important because we say that in a world W, we have a configuration space C, and that for every B_i, we have a CB_i. With this, we are saying that we have C-obstacles (CB_i), these correspond to the union of the space of the object i, with the space of the configurations of A that intersect that obstacle. We can see this in Figure 1. Configuration space then allows us to plan paths considering the robot as a single point in the world frame.

3.2 Dijkstra

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph. The algorithm works in the following way:

- Dijkstra first marks all nodes as unvisited. Then creates a set of all the unvisited nodes called the unvisited set and assigns to every node a initial distance value (zero for our initial node and infinity for all other nodes). Then it sets the initial node as current node.

- For the current node, it considers all of its unvisited neighbours and computes the distances from the current node. It compares the new calculated distance with the current assigned distance and assigns the smaller one.

- When it's done considering all of the unvisited neighbours of the current node, it marks the current node as visited and removes it from the unvisited set (a visited node will never be checked again).

- If the destination node has been marked visited , then it stops. The algorithm has finished. Otherwise, it selects the unvisited node that is marked with the smallest distance, sets it as the new "current node", and goes back to step 2.

It is important to refer that Dijkstra is **complete** and **optimal**. Complete means that it will always finish running and we will always have a path to follow. Optimal means that the path returned by this algorithm will always be the shortest there is between the start and end point.

4 Guidance

Guidance is a closed loop control problem where the mobile robot control variables are controlled to make it follow a reference trajectory while avoiding obstacles. For this problem Dynamic Window Approach (DWA) was used.

4.1 Dynamic Window Approach Algorithm

In the dynamic window approach the kinematics of the robot are taken into account by searching a well-chosen velocity space. DWA assumes the robot moves in circular arcs. That makes the velocity space all possible sets of tuples (ν, ω) where ν is the linear velocity and ω is the angular velocity. Given the current robot speed, the algorithm first selects a dynamic window of all tuples (ν, ω) that can be reached within the next sample period, taking into account the acceleration capabilities of the robot and the cycle time. The next step is to reduce the dynamic window by keeping only those tuples that ensure that the vehicle can come to a stop before hitting an obstacle. The remaining velocities are called admissible velocities. In Figure 2, a typical dynamic window is represented. Note that the shape of the dynamic window is rectangular, which follows from the approximation that the dynamic capabilities for translation and rotation are independent. A new pair (ν, ω) is chosen by applying an objective function to all the admissible velocity tuples in the dynamic window. The objective function prefers fast forward motion, maintenance of large distances to obstacles and alignment to the goal heading. The objective function has the form:

$$O = a.heading(\nu, \omega) + b.velocity(\nu, \omega) + c.dist(\nu, \omega)$$

where **heading** is the measure of progress toward the goal location, **velocity** is the forward velocity of the robot, and **dist** is the distance to the closest obstacle in the trajectory.

In this version of the algorithm it can also be used as a way of planning in unknown environments. In our project, however it is used just as a guidance algorithm. In practice this implies that the objective function will consider different things, but the idea remains the same. The way this is implemented in ROS is explained in 5.3.

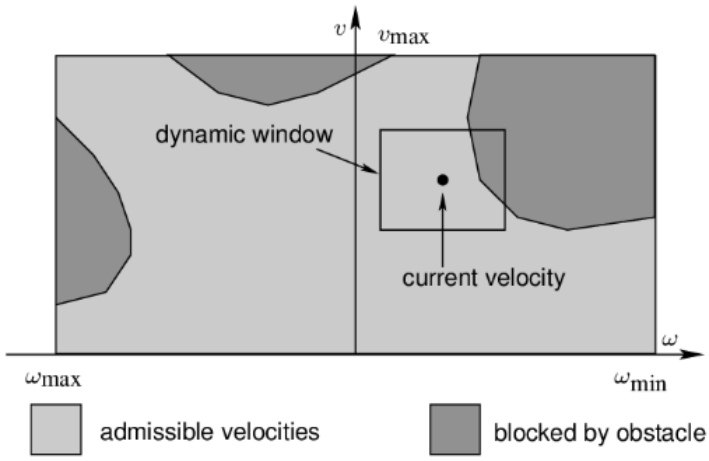

Figure 2: Dynamic Window

5 Implementation

5.1 Move Base

The ROS move_base package provides a central node of the navigation stack. The overall architecture can be seen in Figure 3. This node takes information from the sensors of the robot, maintains some structures based on that information and then, based on the present goal, outputs velocity commands to the robot. Laser scanner readings are used in pair with location information provided by AMCL to maintain cost maps (Section 5.2). Because the global cost map needs information from the actual occupancy grid map, move_base also retrieves it from the map_server node. The location is also used to plan paths in the global planner and to make small adjustments to that path on the local planner.

5.2 Cost Maps

Cost maps are ROS way of considering the Configuration Space. They are grid maps in which each cell has an associated cost. Paths are then planed so that the cumulative cost of the robot being in each cell is minimal. The global cost map takes into consideration the map and the laser scanner readings from the robot to be maintained while the local cost map only needs the

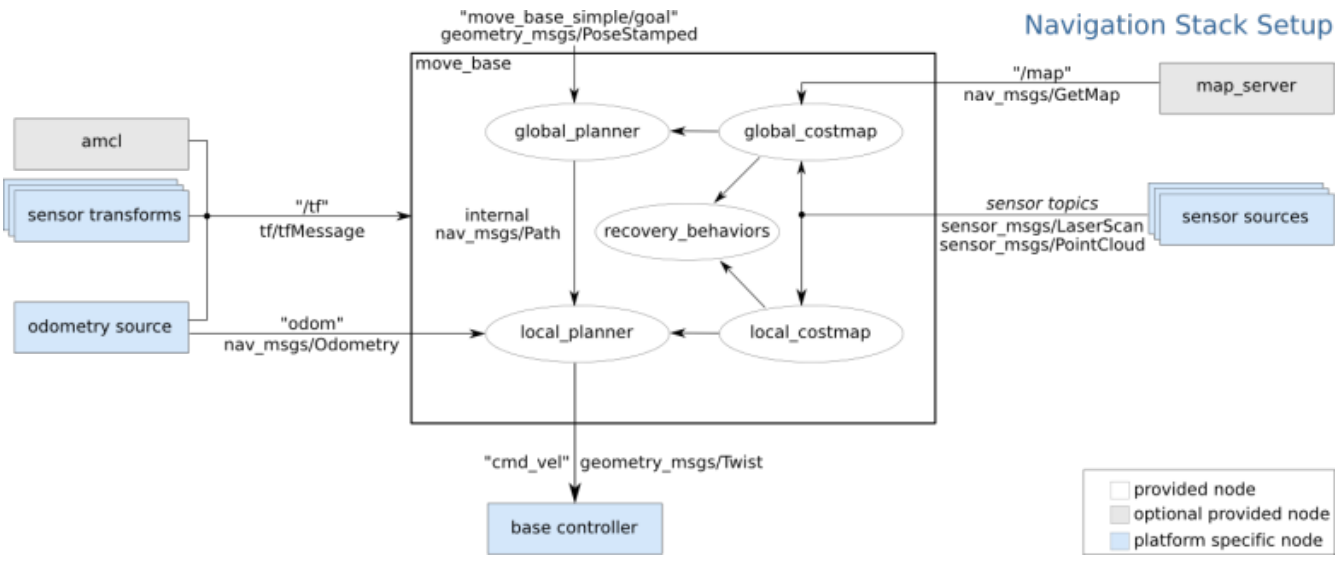

Figure 3: move_base architecture

information coming from the laser. Initially, all the cells in the cost maps are attributed one of three different costs. Lethal cost cells are the ones that the robot knows for sure are occupied, free cost cells are the ones that the robot knows are free and then there are unknown cost cells. Then, based on the footprint of the robot it increases the cost of cells around the lethal cells to have inscribed cost cells, circumscribed cost cells and some cells that have a cost which is between the circumscribed cost and the free cost, lets call them inflated cost cells. Inscribed cost cells are those that have a distance to lethal cells smaller than the inscribed radius of the robot so, if the robot ever gets to one of those cells it will crash for sure. Circumscribed cost cells are those that have a distance to lethal cells bigger than the inscribed radius of the robot, but smaller then the circumscribed radius of the robot. This means that, depending on orientation, if the robot ever gets to one of this cells it may crash, but it may also be fine and keep on its way. The inflated cells have a distance to lethal objects bigger than the circumscribed radius of the robot but smaller then a user defined parameter called **inflation radius**. If this parameter is big, the robot will take safe ways in the sense that it will stay very far from obstacles. If this parameter is small then the robot will plan paths in which it passes very close to the obstacles. We actually changed this parameter from 1.0 to 0.25. We will talk about the implications this add in Section 6.

Regarding cost maps, the only other parameter we changed was size of the local cost map. We increased it a whole lot so that the robot could make adjustments more informed. This helped a lot because of the constant changing of the environment the robot is in. Going back to when we captured the map, the pioneer robots had a very different layout and that has some implications while planning a path based only on the map we captured before.

5.3 DWA Local Planner

Regarding the DWA Local Planner, we changed some parameters. First, we highly increased (0.05 to 5) the yaw_goal_tolerance and the xy_goal_tolerance just a little bit (0.05 to 0.2) which helped a lot because this way we avoided micro-movements when the robot gets close to the goal's surroundings.

Changing the heading_scoring to true could be helpful in order to achieve the goal faster because as the robot avoided obstacles it would be oriented to the goal. We have tried this and we indeed saw some improvements. But this was not the goal of this project, the robot was supposed to follow the planned path, so we kept this parameter as default.

ROS uses a cost function instead of the objective function presented in Section 4.1, this cost function is defined as:

$$cost = path\_distance\_bias * \beta 1 + goal\_distance\_bias * \beta 2 + occdist\_scale * \beta 3$$

where $\beta 1$ is the distance to path from the endpoint of the trajectory in meters, $\beta 2$ the distance to local goal from the endpoint of the trajectory in meters and and $\beta 3$ the maximum obstacle cost along the trajectory in obstacle cost.

We changed two of these parameters: the path_distance_bias (from 32 to 40) and the goal_distance_bias (from 20 to 1). The first one is the weighting for how much the controller should stay close to the global path it was given and that is exactly what we want so we increased the value. The second one is the weighting for how much the controller should attempt to reach its local goal and since is not the goal of this project we highly decreased it.

6 Results

Initially, with the default parameters, the robot was able to go from one waypoint to another quite well. But there were two things that we felt the need to correct. First, the robot was planning paths that were very conservative in the sense that it would plan paths to pass very far from obstacles. Then the robot was not avoiding obstacles well and in many cases it would get stuck trying to go around an obstacle and it would give up on reaching the goal even with a lot of open space behind him. The big parameter that allowed us to correct both of this problems was the **inflation radius** (explained in Section 5.2). With an inflation radius of 0.25 we got very good results both in path planning and obstacle avoidance. The only problem with this approach were the chairs in the room. They are even present in the cost map and they can be seen in the right side of the robot in Figure 4 but because they are not perpendicular with the floor, because the robot's laser is $14cm$ from the ground and because the robot was risking a lot it would sometimes bump into the chairs and it would get lost because the odometry readings were reporting movement while the robot was not moving at all.

Also with this change we got one result that we were not expecting. We placed a couple of open umbrellas in the middle of the room as an obstacle for the robot but without noting we left a small opening between the two. In one run we did not place any further obstacle next to the umbrellas so the robot went around it as the cost of going through the small opening was too big and so the robot did what we expected. In one other run, we placed a foldable panel next to the umbrellas to make it harder for the robot to deal with and this time the robot disappeared between the umbrellas. In the moment it was very impressive to watch because from our perspective there was not even enough space for the robot. This happened because the cost to go around the panel was actually bigger than the cost of going through that small opening. This result can be seen in our Google Drive Folder. The one where the robot goes around the umbrellas is under the name "umbrella_goes_around.mp4" and the one where the robot went through the umbrellas is under "umbrella_goes_through.mp4".

Regarding obstacle avoidance, before changing the parameters, the robot would go backwards for quite a while before trying to go around the obstacle and would also go back and

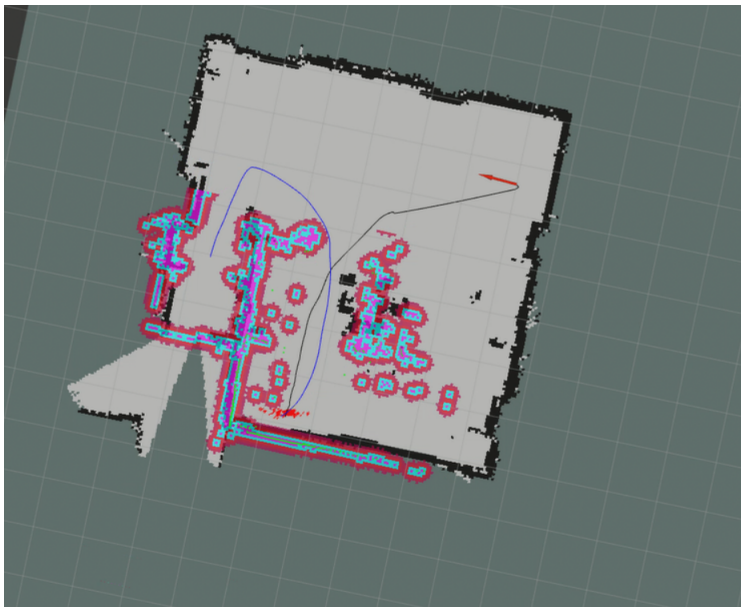

Figure 4: Cost maps with small inflation radius. Planned path (black); Actual path (blue)

forth many times before making a decision about how to go around the obstacle. This is valid for obstacles we suddenly placed in front of the robot because if the obstacle is in the way from the beginning of the run then that information is present in the global cost map and the robot will just plan around it since we plan a path from waypoint to waypoint, not a unique path that passes through the four waypoints. In our Drive we have two videos showing the good results we have here. In the video "bag_throwing.mp4" we threw a bag in front of the robot and it is able to go around it quite smoothly. In the video "narrow_passage.mp4" we force the robot to go through a narrow passage and the robot does not hesitate.

When we changed the DWA parameters we noticed the difference in the adjustments the robot made in the planned path. The difference is visible in the videos in our Drive. In video "hard_path_without_obstacles.mp4" we can clearly see the big changes in the black line in front of the robot while in video "hard_path_with_obstacles.mp4" the black line does not change much. This means that the robot is trying to follow the path initially planned, not making adjustments to reach the goal faster.

7 Conclusions

Initially we had some issues regarding the laser scanner readings that delayed our experiments. We were able to solve them by changing the laser max beams back again to the default value. We believe that it was the source of the problem reported in the previous report, related to kidnapping.

Considering what we said before about the Dijkstra's algorithm, we decided to use it instead of implementing the A* algorithm for two reasons: first because we did not had enough time to do it, but also because we believe the it's impact would not justify its implementation. Probably it would be a little faster than Dijkstra. A* is also complete and optimal (as long as the heuristic is admissible), the only difference would be the number of expanded nodes, which, because of the heuristic, should be smaller in A*.

About following a single path with different waypoints, we were not able to do this. Instead, our robot plans a path between waypoints individually. This was not what was asked but move_base can only receive one point at a time and we did not have time to hack something together that would do what was asked of us.

The fact that the robot cannot detect the feet of the tables ruined some runs for us because they would sometimes be in the way of the robot and he could not avoid it, that's why some backpacks can be seen around the room covering those problematic spots.

Besides that, everything went well. We got the robot to navigate well on the map with and without unexpected obstacles in a smooth and safe way (except for the chairs). And we could see the impact that the different parameters did on the navigation.

References

[1] *ROS Robot Programming (English).* ROBOTIS, Dec. 2017. ISBN: 9791196230715.

[2] Roland Siegwart. *Introduction to autonomous mobile robots.* Cambridge, Mass: MIT Press, 2004. ISBN: 0-262-19502-X.