Report

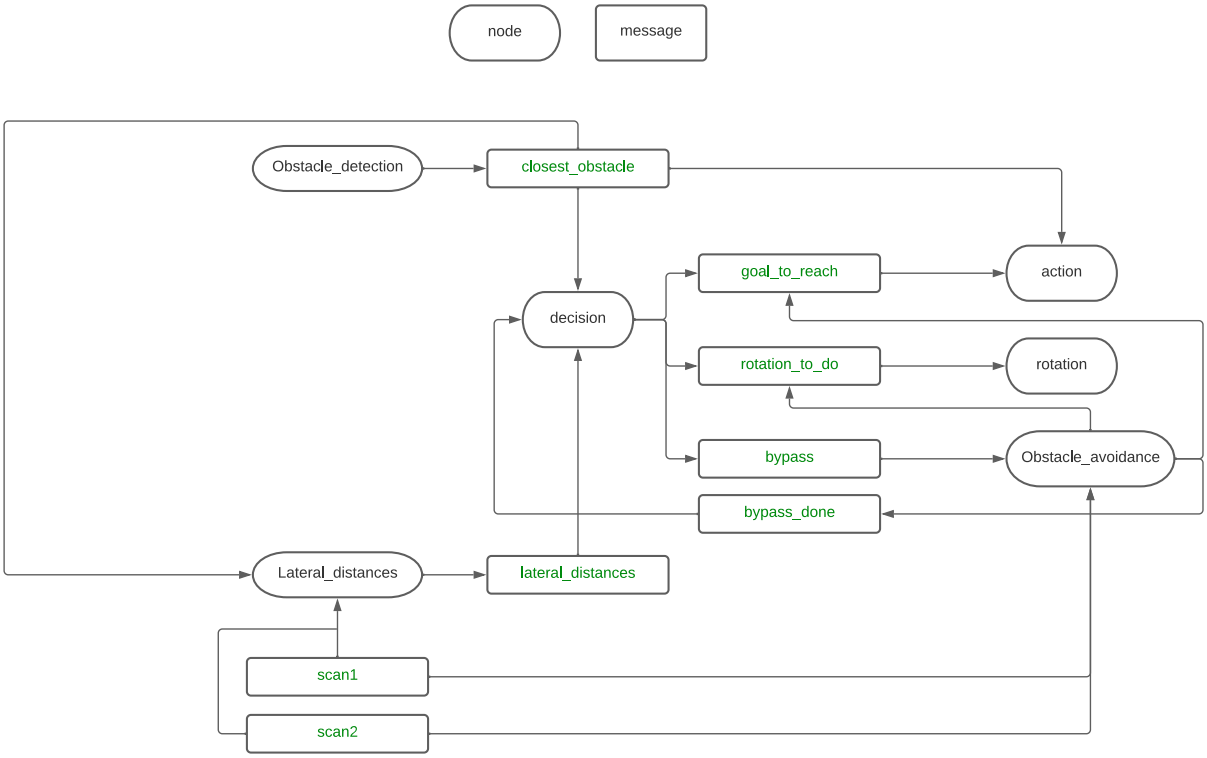
Internshsip - RobAIR

Name: Vinicius Gabriel Angelozzi Verona de Resende

Date: June/July 2023

GitHub: RobAIR

Main Architecture



Nodes

Decision Node

The decision node is, like in the lab during class, the main node. In order to execute it, it is necessary to define a parameter `map_name`, this parameter is used to load the graph of a given map (check below for an example). As it is at this moment, the node does not look for aruco marker or person, all

it does is going to the target and bypass obstacle in the away.

```
rosrun patrol_robot_development decision_patrol_robot_development_node _map_name:="320_plus_hall"
```

> WARNING: The bypass has been disabled since the Potential Field Algorithm has a bug I was unable to identify. Further information regarding the bug will be discussed in the Obstacle Avoidance Node section.

When the graph is loaded and the path is to be created, there is two possibilities for finding the path. Both use the BFS algorithm (in order to look for the path with the least amount of vertices) but differs when finding the vertices close to the robot.

- In the first one (currently being used), as soon as the method starts running, we divide the map in 4 quadrants (where the position for the division is the one robot position). Then, it looks for two vertices surrounding the robot in two opposite quadrants (1-3, 2-4) and at least 1 surrounding the target point. The reason for the quadrant choice was an attempt to find more straight forward paths once both points are opposite to each other.
- The second approach is simpler. It only look for the two of the closest vertices, regardless of where they are.

Obstacle Avoidance Node

This node has as input both lidar scans, the odometry, the robot_moving and a `bypass` message. The bypass message has the following structure:

```
geometry_msgs/Point front_obstacle    # front closest point
geometry_msgs/Point lt_obstacle_point # left closest point
geometry_msgs/Point rt_obstacle_point # right closest point
```

```
geometry_msgs/Point goal_to_reach     # target poistion
bool enable_apf
```

As output, it publishes a message in the topic `bypass_done` with the following structure:

```
geometry_msgs/Point goal_to_reach     # target position
bool apf_in_execution                 # Tells if APF algorithm is being executed
                                      # if true, the next messages should be provided
                                      # to obstacle_avoidance and decision node should
lock the process
```

In short, this node aims to implement the artificial potential field algorithm as described in this article. One point worthy of mention is: the APF algorithm will not be executed during the whole path planning. As it requires a lot of computation, it only executes if an obstacle is found in the way to the target. A small simulation of the computations were executed in python with pyplot to simulate the robot and better understand the algorithm. It works in the simulations, however, when executing the node, a stranger behavior appears. In the plateau, when executing the tests, the RobAIR would start going around the obstacle and suddenly stop. As it wasn't the main goal of the internship, I made the decision to stop the tests and proceed with the localization. Even so, after discussion, some hypothesis for the bug are:

- Shape of the used obstacle
- Unseen mistake in the implementation of the algorithm in C++
- Early exit of the algorithm (returning the `apf_in_execution` as false too early)

The logic behind the communication with the decision node is: as soon as the APF node is enabled, the decision node would send every single message to the obstacle avoidance until it receives from the `obstacle_avoidance_node` the response of completion. The target position in the output of the node is used to keep track of the original target, once the node publishes new `goal_to_reach` points.

Generate Map End-points Node

This node is not necessary for the other ones, it was created as an easier way to generate the graph. All it does is, identify an aruco code, walk until the code and register the location in the map frame in a file. To execute the node, a parameter must be set during execution (as in the decision node). The parameter is `map_name`, and for the same reason as in the decision node, it is used to load the map and create the graph file with the same name.

This node is supposed to also generate the edges in the file, however this has not been implemented yet. One idea is to check if between two nodes exists any obstacle in the way (using the scan values). If there is none, add the edge between both vertices. This may be a naive and slow way, however, as this should be executed only once, the problem is not that significant. Having said that, the edges section of the graph must be done by hand. The output file has the following format:

```
#X      #Y      #Z11.96   5.97249 011.8395 3.4305  09.97737 2.82132 07.65372 2.13215 0
6.85869 6.14042 04.63425 1.20686 0-0.310606 -0.171949 0# The next line is required####
############### edges ###################0 1 # the numbers are IDs for the vertices 0 is
the first and n-1 is the last.    # Where n is the number of vertices1 22 33 43 55 6
```

Lateral Distances Node

This node is a simple implementation of a lateral crash avoidance algorithm. All it does is, based on the lidar scans, check for the closest points in the left and right of the RobAIR withing a 180 degrees range (-90 to +90) and in less then 1.5 meter away from the robot in the X axis. With this information, it returns as an output in the topic `lateral_distances` the following message:

```
geometry_msgs/Point lt_obstacle_point # location of the closest point
geometry_msgs/Point rt_obstacle_point # location of the closest point
float32 lt_obstacle_distance # distance to the closest point
float32 rt_obstacle_distance # distance to the closest point
```

Then the decision node should use this information to avoid the lateral crash. So far, the decision node implements a simple algorithm that computes the middle point between both points and publish a new goal a bit ahead and in the middle between both points. If one of the points is not returned (there is no obstacle to that side), it publishes a goal 1 meter away (in the y axis) to the closest obstacle.

There is a limitation with this idea, if the goal is within a narrow space, we might always end overshooting the position of the goal. One way to overcome this is by using the current localization in the map frame to stop the RobAIR as soon as the position of the robot is within a threshold of the target position.

To integrate this node with the decision node, a small modification must be done in the decision node. We should keep the original target location stored somewhere in order not to loose it.

Localization Node

The localization node is the same used in the lab class. As the tests execution were done in a place where the location can be easily mismatched due to similar scans, the localization node still requires a initial position to be set in the RVIZ.

Future Work

For future work, besides checking the points mentioned, something worthy to work on is using the aruco IDs to improve the localization in case of maps such as the 3rd floor of IMAG, where every hall is similar to each other. In this cases, the robot can easily mismatch its location, so using the aruco ID fixed to each intersection in the map, it is possible to set the correct location of the robot in case of possible mismatch. One other possible improvement for localization would be using both lidars when locating the robair. As I used the one from the class, it had only one lidar, the bottom one. For using both, i thought about

comparing the hits on both lidars, the one with the best result would be the one setting the position in the map.