Autonomous exploration, mapping, and planning using ROS

Erasmus Mundus Master in Computer Vision and Robotics (VIBOT)

Autonomous Robotics final project

Hassan ZAAL
enghassanzaal@gmail.com

Yu LIU
alphadadajuju@gmail.com

Abdelrahman G. ABUBAKR
eng.abdelrahman.gaber@gmail.com

***Abstract*— In this project, we develop a ROS-based system to autonomously navigate the turtlebot in the Gazebo's willow garage environment. Open-source libraries (OMPL and Octomap)are applied to complete the project. The turtlebot explores the unknown environment utilizing brushfire as point-to-go selection, RRT path-planning algorithm with collision detection, as well as self-implemented driver to execute movement. A similar framework is applied to the final path-planning to return the robot back to its initial position.**

I. INTRODUCTION

The aim of this project is to design an autonomous robotic system to explore and map an unknown environment starting off at an initial position. Then the robot has to find a path to return back to this initial position. The map used is the willow garage map in the Gazebo environment, and the robot used in this project is the simulated turtlebot.

In order to implement this system, a hierarchy of control levels are implemented: high level, medium level, and low level control. The high level control concerns assigning the general tasks to the lower level, such as "rotate 360 degrees" and "find best points to explore next". These commands then are processed by the medium and low level controls, for example the rotation command will be executed by the low level driver node, and find best points will be managed by the medium level exploration node.

The workflow of our system can be summarized as follows:

- First, initialize the map by spinning the turtlebot (no translation) for 360 degrees.
- Continuously process the latest 2D map to decide the best point to go next for exploration:
  - Configuration space is considered.
  - Apply Brushfire on the existing 2D map to locate best points to visit.
  - Find N best points to go next and choose the best.
  - Move turtlebot to the point (RRT Planner and motion driver).
  - Update the 2D map, and then re-iterate,.
- Use the latest 2D map to find a path back to the initial robot position.

The explanation of all these steps, in addition to the details of their implementation will be discussed in this report. The rest of the report is organized as follows: first the exploration and planning will be explained with their implementation details, such as configuration space, brushfire algorithm, finding best points, and how to reach the best point for exploration. After that the path planning algorithm used will be shown, and finally the low level driver and obstacle avoidance will be discussed.

The result of our proposed framework can be seen in the video link (Youtube link). All source codes are available on our github repository (github repo).

II. EXPLORATION

For a mobile robot to engage in exploration of a priori-unknown environment, it must be able to identify locations which will yield new information when visited (in this case, map expansion). Initially only a small fraction of the workspace is visible and plans must carried out for the robot to move further to increase the extent of the map. To initialize exploration, a command is first sent to the low level controller to make the robot spin 360 degrees around its initial position. This would begin building the 2D map (utilizing "octomap_server" to be explained in the next section) and give the robot some base areas to find the best -promising- point to go next.

After that, the obtained 2D map is read from the "projected_map" topic, and morphology preprocessing is done on it to take into account the configuration space. Then, the brushfire algorithm is applied to the preprocessed 2D map to have insight about how far each discrete grid in the map is from the walls. With this "potential" map, the N best points are selected by the algorithm to decide where to go next. In following sections (3 to 7), all these steps are presented in addition to the implementation details.

III. PROJECTED MAP

To access the 2D map explored by turtltbot we use the package "octomap_server". The "Octomap_server" builds and distributes volumetric 3D occupancy maps as OctoMap binary stream and in various ROS-compatible

formats e.g. for obstacle avoidance or visualization. The map can be incrementally built from incoming range data (as PointCloud2).

The ”projected_map” topic publishes down-projected 2D occupancy map from the 3D map. This map is given as row data so we reshape it by the given height and width. Each cell in the resulted map has length equivalent to 0.05m in Gazebo. Obstacles, free space and unknown area are represented by different values in the 2D map(100, 0 and -1 respectively). In addition, the following equations are used to convert metrics from 2D map's cell position (integer grid) to actual position in the Gazebo's world frame (meters). Both information of ”resolution” and ”origin” can be obtained directly from the projected_map topic

Xgazebo = Xcell * resolution + Xorigin ;
Ygazebo = Ycell * resolution + Yorigin

IV. Configuration Space

Although the motion planning problem is defined in the regular world, it lives in another space: the configuration space. The configuration space (C-space) is the space of all possible configurations. A robot configuration is a specification of the positions of all robot points relative to a fixed coordinate system. By considering configuration space we can treat the robot as a point object and hence can omit collision resulted from the size of the robot.

In our approach, we address robot's configuration space by performing morphological operation on the subscribed 2D map. Specifically we dilated obstacles and unknown area. Size of dilation is equal to turtlebots radius. Figure 1 shows an example of free space before considering the configuration space and figure 1 shows it after considering the configuration space. Figure 3 shows part of the real map of our problem before and after dilation to consider the configuration space.

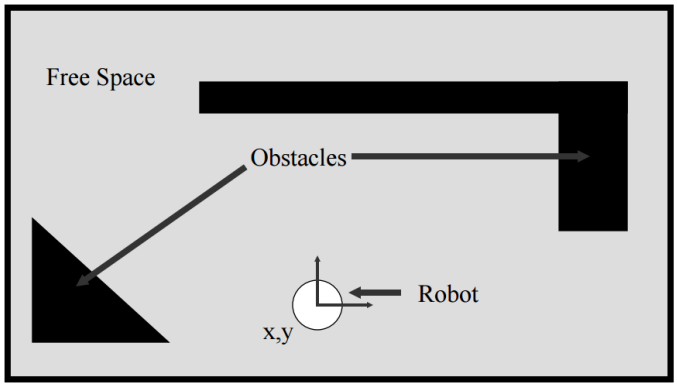

Fig. 1. sample map before configuration space

V. Brushfire Algorithm

Autonomous navigation of the robot relies on the ability of the robot to intelligently reach its goal while avoiding the obstacles in the environment. In some cases the robot has a complete knowledge of its environment, and plans its movement based on it. However, in our case, the robot only has an idea about the goal position, and should reach it using its sensors to gather information about the environment.

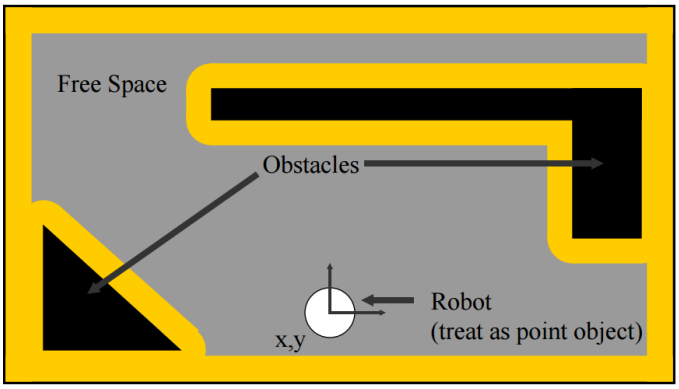

Fig. 2. sample map dilated to take effect of configuration space

To deal with robot position relative to the obstacles we use brushfire algorithm. Brushfire algorithm depends on the idea of simply making each obstacle generate a repulsive field around it. If the robot approaches closer to the obstacle, a stronger repulsive force will act on it, pushing it away from the obstacle. In our case we used brushfire to obtain a potential map for knowing how far a cell in the map is with respect to the surrounding obstacles. We use it to put weights on cells. This weight is the minimum distance to an obstacle. The algorithm implementation works as follows:

- Until all cells are not equal to zero:
- Assign to ”zeros” cells the weight of their smallest nonzero-neighbours + 1. In our case we did brushfire with nonzero-neighbours (unexplored) only and not equal to -1 (unknown area).

Figure 4 shows result of the implementation of brushfire on a simple example. Figure 5 shows result of the implementation of it on a real part of our map.

VI. Finding N best points to go next

After processing the map with space configuration and brushfire algorithm, we have now some knowledge about current map. The brushfire algorithm shows how far each point is from walls and if it is free-known or unknown area. Using this information, and knowing that the robot should choose next point to be the best-promising-point in the map, we put criteria for which point should be chosen next. Our approach is to check all the points near the walls with brushfire weight of 106 (experimentally obtained), then check a window around this point and count how many unknown points in this area; the best point would be the point with more unknown areas inside the window.

However, because a single best-point may be unreachable by path-planning algorithm (since we sampling-based algorithm), or very far away from the robot, we depend on first

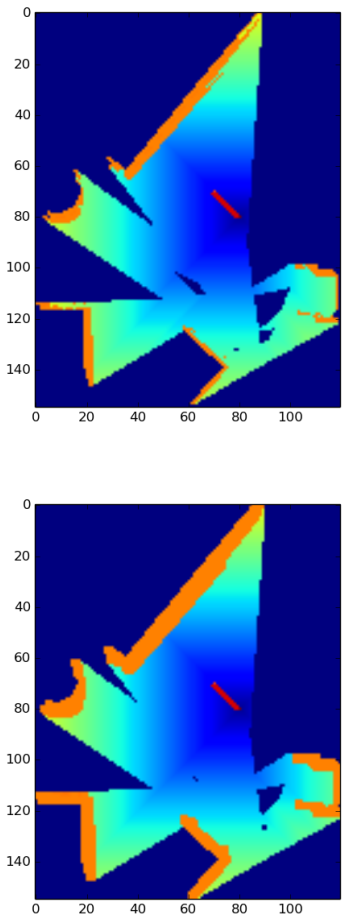

Fig. 3. First image shows the original 2D map before dilation. Second one shows the dilated map to take into account configuration space

N best points, and choose the nearest one to the robot, and if it is unreachable, then the robot will try to go to the next best point, and so on. In our proposed code, we chose N to be 20-30 points, from which the algorithm tries to choose the nearest one to go. Figure 6 shows an example of possible best-points in a map gathered by our algorithm.

VII. RRT TO VISIT TARGET POINT

Once a grid position in the known-free space is determined by our brushfire-based point selection, the robot will advance to this position while before reaching, continues to sense its surrounding to expand the known map. In the exploration stage, the path generated to reach a determined point needs not to be short / optimized, as the wandering, explorative behavior of the turtlebot can facilitate building the known map more thoroughly. On the other hand, it is not ideal if the path requires a significant amount of time to compute as the turtlebot would always be static before trying to reach the next exploration point. These two rule-of-thumbs lead to the use of RRT algorithm provided by the OMPL Library.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	2	2	2	2	2	1	1	2	2	2	2	2	2	1	1	2	2	2	1
1	2	3	3	3	2	1	1	2	3	3	3	3	2	1	1	2	3	2	1
1	2	3	4	3	2	2	2	2	3	4	4	3	2	1	1	2	3	2	1
1	2	3	3	3	3	3	3	3	3	3	4	3	2	1	1	2	3	2	1
1	2	3	2	2	2	2	2	2	3	4	3	2	1	1	2	3	2	1	
1	2	3	2	1	1	1	1	1	2	3	4	3	2	1	1	2	3	2	1
1	2	3	2	1	1	1	1	1	2	3	4	3	2	2	2	2	3	2	1
1	2	3	2	2	2	2	1	1	2	3	4	3	3	3	3	3	3	2	1
1	2	3	3	3	3	2	1	1	2	3	3	3	2	2	2	3	3	2	1
1	2	3	4	4	3	2	2	2	2	3	3	2	2	1	2	3	3	2	1
1	2	3	3	3	3	3	3	3	3	3	2	2	1	1	2	3	3	2	1
1	2	2	2	2	2	2	2	2	2	2	2	1	1	1	2	2	2	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 4. Example of brushfire algorithm on simplified map

Fig. 5. Example of brushfire algorithm on a real map

The provided framework allows us to easily apply OMPLs path planning algorithm from the turtlebots current position to a goal position in the known map. The ”offline_planner” node governs using a host of sampling-based path planning algorithms, returning a sequence of position values (x and y) for the turtlebot to follow. However, a major limitation lies in the fact that even though we specify the goal position in the known-free map, paths generated by OMPL do not necessarily follow the known-free map (in fact, they usually fall within the unknown area). The paths in unknown map by RRT hence do not concern presence of obstacles. It is very likely that unknown areas hide obstacles, and encountering any obstacle would completely mess up the entire mapping behavior.

To address this issue, we evaluate the generated paths by RRT and check if any obstacle is present along the path. This is done in an incremental approach, where we divide a line of path into segments of grids (each has length 0.1m). We then incrementally evaluate each grid

Fig. 6. Example of regions of the N best candidate points chosed by our algorithms

at a specified frequency to check if it corresponds to an obstacle in the respective 2d map subscribed from the topic ”projected_map”. A conversion between the gazebo world and occupancy grids metrics (meter and grid pixel respectively) is required. When the turtlebot continues to move toward the destination of a path, areas faced by the robot would be sensed and eventually become known areas. Any grid along the path detected as occupied in the corresponding 2D map indicates that this path is not valid and rerouting is needed.

Specifically in our implementation, a threshold distance between the turtlebot and the detected obstacle dictates the robots reaction. If the distance exceeds 2 meters, turtlebot would continue moving until the distance no longer exceeds the threshold (this is simply to make the robot go further while avoiding hitting the wall). In addition, it was observed that even when a line of generated path completely falls onto known-free map, it may be very near to an obstacle (for instance, a path passing through a narrow door or passing by a wall corner). Turtlebot would not be able to properly follow this path without hitting obstacles on the side. To avoid collision in this situation, instead of checking the line of path (single grid along the generated path) the presence of obstacles, the area around the path (five-grid width orthogonal and along the generated path) is checked.

VIII. Path Planning

Once the exploration stage is finished, it is required that the turtlebot returns from its current position to the initial position. For returning the robot to the initial position, we in fact apply the same technique as used during exploration: utilizing OMPLs RRT along with our own collision checking mechanism. For our implementation, we have two separate nodes (turtlebot_drive and turtlebot_return) which function similarly but each is dedicated to handling the exploration and returning respectively. In addition, an additional ”offline_planner” node (offline_planner_R2_2) is created to pair with the ”turtlebot_return” node. In this new ”offline_planner”, parameters for OMPL algorithm were altered (specifically, giving more time to compute for paths to return to the initial position).

In fact, initially the OMPLs RRT* was employed for the returned trip, as it is established that RRT* offers more optimized path toward the goal. However in our case, the RRT* did not generate correct paths, as the tree branches were observed crossing over known-occupied spaces. On the other hand, RRT provides consistent result as what we saw in the exploration stage.

IX. Low Level Control - Driver

Motions of the turtlebot is implemented with the concept of proportional control. In other words, robots linear velocity is greater if it is further away from where it is going, and the speed drops when it approaches near the target. Likewise, robots angular velocity is greater when its orientation (along the z axis) is more different from the target direction (to summarize, the difference in the goal position/orientation from the robots current and goal position is positively correlated with the linear/angular velocity).

Specifically in our implementation, we control the robot first by adjusting its orientation toward the goal position (linear velocity set as 0). Once the robot is rotated and positioned correctly in terms of orientation with respect to the goal, linear velocity increments by 10 percent for each iteration. While the robot takes linear displacement, its orientation with respect to the goal is constantly checked; linear velocity

would continue to increment if the robots orientation is still correct. On the other hand, if its orientation begins to deviate (by a specified threshold), linear velocity would decrease by 25 percent every iteration. We only increase/decrease linear velocity by a percentage of the previous velocity to avoid drastic changes which would lead to the drifting effect. The drifting effect can cause obstacles having multiple layers and appear larger than their actual size. As our environment is bounded with narrow corridor and doors, this consequently would lead to narrow passages potentially being completely shutted down and hinder exploration. Lastly, we set an upper bound velocity (both linear and angular) of 0.8 m/s to further prevent uncontrollable scenarios caused by high speed.

X. CONCLUSION

In this report, we presented our approach to solve the problem of autonomous exploration, mapping, and planning for turtlebot using ROS. The exploration part is done using a frontier-based approach depending on a processed map with brushfire algorithm, while the map is continuously updated and represented using octomap, then processed again to choose the next best point to go. The planning is done using RRT algorithm from OMPL library, which is used to visit target point in exploration, and planning the path back to starting point.