**RBE 3002 - D16**

**William Hunt**

**Annie Hernandez**

**Trevor Rocks**

**Final Project - Autonomous Mapping**

**Introduction**

The objective of this laboratory project was to survey an indoor environment of limited complexity using an autonomous mobile robot platform. The purpose of the exercise was to demonstrate the following principles:

* Integration of custom code with existing application nodes using the ROS meta-operating system environment.
* Rapid frontier analysis in a two-dimensional map.
* Frontier exploration with decision making activities.

The formal problem definition for this project was “to act as a building surveyor, generating a map of a closed space autonomously, in no more than twenty minutes”. The room for testing was guaranteed to be free of moving obstacles.

**Methods**

Our procedure for tackling the challenges of this project was to break each problem down into a collection of isolated small tasks. We first recognized that there were three necessary component of functionality to satisfy the project requirements:

* Simultaneous Localization and Mapping (SLAM)
* Path planning
* Frontier management (goal selection)

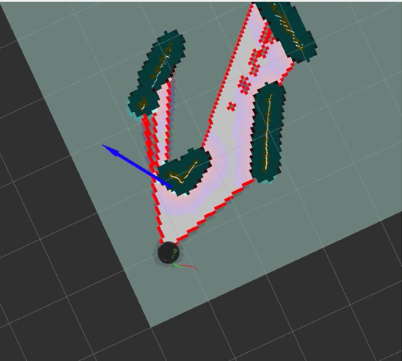
Early-on, we identified the GMapping software included with ROS as a potential solution for SLAM and path planning, provided that the Turtlebot platforms in the laboratory were used. We elected to proceed with the Turtlebot platform and to rely on the demo launch configuration turtlebot\_navigation/gmapping\_demp.launch for movement and map construction. This decision left only a single problem: frontier management.

We broke the challenge of frontier management down into five sub-problems:

* Frontier detection
* Frontier grouping (blobbing)
* Goal selection
* Completion detection
* Fault detection and recovery

We implemented solutions for each of these sub-tasks within a single ROS node, with a helper library handling some map-based mathematics.

For frontier detection, we implemented an algorithm roughly equivalent to Wavefront Frontier Detection (WFD) in reference [2].Our algorithm was a breadth first search utilizing graph-search optimization, which entails the tracking of visited nodes. We initially tracked visited nodes as a list of points, iterating on that list to determine visitation status. We found that this approach was extremely inefficient as the map size grew large. To improve performance, we switched to a sparse lookup array containing visitation flags for each map element. This gave the algorithm a lookup resource with O(n), and led to dramatically increased algorithm speed. See Figure 1 for an example of frontier detection, with frontier nodes illustrated in red.



*Figure 1: Frontier Detection*

For frontier grouping, we chose to consider adjacent (diagonal and Manhattan) frontier points to belong to the same frontiers. This is the least zealous blob detection function available, and it yields reasonable results at low cost. The blobbing algorithm iterated over the points in the frontier, detecting neighbors’ memberships in existing frontiers. Again, for fast lookup, an array was maintained in the shape of the map. Because of the small amount of memory required to store references, we chose to have the frontier lookup table provide a reference to the frontier attached at any given point, as applicable. This allowed the blobbing algorithm to quickly determine whether a given node was in the frontier, and the frontier list to which any node had already been added.

In certain cases, the blobbing algorithm happened upon a frontier node which “attached” two existing frontiers together. Our algorithm included a special detector for this situation, and could properly merge the two existing frontiers without creating duplicates. Our algorithm also included a feature for “escaping” from an obstacle, if the robot was initialized “inside” an obstacle.

Our frontier selection algorithm was intrinsically tied up with our methods for completion detection, fault detection and fault recovery, so these elements of the problem will be addressed together here. In normal operation, the frontier selection algorithm computed frontiers from a map with obstacles expanded by the radius of the robot. It then simply chose the largest (by point-count) frontier, computed its centroid, and set that as the goal. We subscribed to the move\_base node’s *result* topic to determine whether the robot reached the goal setpoint. If the goal was reached, we allowed the robot to attempt one more goal from the same frontier set, without re-analyzing the map. We repeated this for the first three frontiers, ordered by size. Once two goals were successfully reached, the map was requested again and the frontier analysis repeated.

In normal operation, if the robot failed to reach two goals successfully out of the first three, we assumed that a fault condition had been reached. The most common fault condition occurred because of overzealous goal selection, a solvable problem. Thus, we chose to attempt fault recovery when this condition was detected.

For recovery mode, we switched to a different goal selection algorithm. In this mode, we first planned frontiers using the normal algorithm, but with obstacles expanded only a small amount. We then selected a frontier centroid, and searched the global two-dimensional costmap for the closest point to that centroid *likely to be reachable* based on costmap values. In this mode, we allowed the robot to attempt up to three solutions, but terminated and returned to normal mode after a single goal was successfully reached.

Note that, in both the recovery and normal operating modes, our robot selected a random angular heading target at each goal, to promote stochastic map growth and avoid repetitive fault conditions.

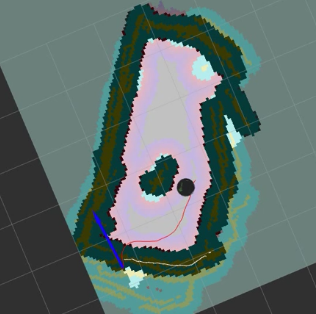
To detect completion, our algorithm simply checked whether the largest frontier in the normal operating mode (zealous obstacle expansion) was smaller than a threshold value. In this condition, we considered the map to be completed, and terminated execution. Note that we did *not* permit the robot to terminate mapping during the fault recovery mode; in general, if recovery mode produced a completion state, that state would be detected on the next pass through the normal operating mode.

**Results**

Our mapping algorithm was able to consistently map small areas while moving about and marking floating obstacles. The system was able to look at an initial frontier and find centroids from which it would best be able to continue to map the given area. Using the built in path planning and following functions from GMapping, the Turtlebot could navigate to the desired point, given that the point is not in an unreachable space.

Because of the map-expansion motif developed within the frontier detection algorithm, any gap in the frontier that is large enough for the robot to pass through will be explored, while gaps that cannot be traversed will be either ignored or closed by the GMapping nodes. We observed that our fault detection mechanism ran frequently, and dramatically increased the reliability of the algorithm.

Figure 2 shows the robot resting in a completed map of an environment, alongside the UI callout “Map Complete” indicating the terminal condition.



terminal complete.PNG

*Figure 2: Completed Map w/ UI Callout*

As shown the robot has completed the outer frontier of the map, finding no gaps large enough to explore, and detected an obstacle floating within the navigable space. It has also delivered a message to the user through the terminal, to indicate that the map is complete.

For a video of our solution operating, please visit <https://www.youtube.com/watch?v=V5FJzfqFjpA>.

**Discussion**

To the aim of mapping an unknown finite-territory environment, our robot performed well. Frontiers are found reliably, and unhandled fault conditions are encountered quite rarely. We found that poorly calibrated robots (subject to gyroscope drift) could not operate properly using our algorithm, which relies on a strong SLAM underpinning.

We found that setting reachable navigation goals was the most challenging aspect of exploring unknown environments (after localization and filtering tasks, which were outside the scope of this project). Movements through tight areas and corners often require special handling; GMapping is quick to dismiss challenging points as unreachable for various reasons. Our program handles this by anticipating easy goals in fault conditions, allowing the robot to “unstick” itself without attacking particularly challenging frontiers directly.

In the recovery condition, we noted that a “crow flies” distance did not always produce the best approximate goal. For example, in an environment with inward concavity, or in a highly uncertain costmap, this distance heuristic could select points that were physically close to the goal but not along the theoretical path thereto. A better solution may be to incorporate a low-resolution A\* path planning search before entering the fault condition, to bound the region of the map in which fault-resolution goals could be assigned.

We frequently saw our robot colliding with floating obstacles due to unaccounted protrusions from the control computer (USB ports, LCD screen, etc). We did not attempt to redefine robot configuration information to solve this problem, but it would likely be a simple fix.

**Conclusion**

Throughout the process of completing this final project we have learned how to creating a mapping function for the TurtleBot using GMapping in the ROS environment. The robot was able to produce an accurate two-dimensional map of a small indoor environment by exploring detected frontiers in-turn and making intelligent goal decisions based on frontier volume. We also successfully implemented a form of simple fault detection and recovery, which dramatically improved performance and reduced the likelihood of a user-detectable failure.We successfully integrated all of our code with existing ROS libraries, and ultimately used a combination of our own code and the GMapping navigation stack to complete the challenge. The ease of integration within ROS highlights the value of this meta-operating system for research applications; had ROS and its constituent libraries been unavailable, we would have had to perform low-level integration and data-type conversion for a variety of sensors and search libraries.

**References**

[1] B. Gerkey, "Gmapping," in *ROS Wiki*. [Online]. Available: http://wiki.ros.org/gmapping.   
 Accessed: May 2, 2016.

[2] M. Keidar, E. Sadeh-Or, G. Kaminka, and MAVERICK Group, "Fast Frontier Detection for   
 Robot Exploration," in *MyWPI*. [Online]. Available: https://my.wpi.edu/bbcswebdav/pid-   
 628611-dt-content-rid-3338687\_1/courses/RBE3002-D16-MASTER/Fast%20Frontier%20  
 Detection%20for%20Robot%20Exploration.pdf. Accessed: May 2, 2016.

[3] D. Wang, Z. Wang, Q. Chen, and P.G.C.N. Senarathne, "Efficient frontier detection and   
 management for robot exploration," IEEE, pp. 114–119. [Online]. Available:   
 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6705430. Accessed: May 2,   
 2016.