Miguel A. Ruiz

Zeyad Shureih

ROB-456

12/08/20

Final Project Report

1. Discussion of the exploration problem and your solution - how did you pick the next point(s) to visit?

The exploration problem was solved by starting the robot without a goal position. On every map update, the nearest unknown value is computed via the flood fill algorithm. When the top node in the priority queue is an unmapped space, we set this as our goal position, and begin our A*star procedure for finding the shortest path.

Our flood fill algorithm is modified to create smoother paths around obstacles by artificially inflating the distance of a node from the starting point by drawing a circle around the point *r* points wide. If any point within the circle is a wall, we increase the distance in the costmap by a factor of 100. This allows paths to be generated for start and end points that are near obstacles, while avoiding most obstacles in the path between the two points. However, this modification can become extremely expensive for larger values of *r.* We found that in path planning, the optimal value for *r* was between 8 and 10 in order to create paths with wide enough turns around obstacles. In the closed house world, our flood fill and get path algorithms took around 5 minutes to chart a path to the other side of the map.

Once we have navigated to a point, and that point has been mapped, we repeat the procedure. Using our updated map, we find the path to the nearest unknown point, and navigate to it with the help of our subgoal algorithm.

2. Discussion of your map regeneration/new planning points policy - when did you decide to get a new map and re-plan?

We detected our robot getting stuck by starting a timer every time we navigated towards a new subgoal. On a map update (or an odometry update when testing in part 2), we check to see if the robot is at its subgoal. If it isn't, and the difference in time between when we set the subgoal and the most recent update is greater than one minute, we rechart a path for the goal.

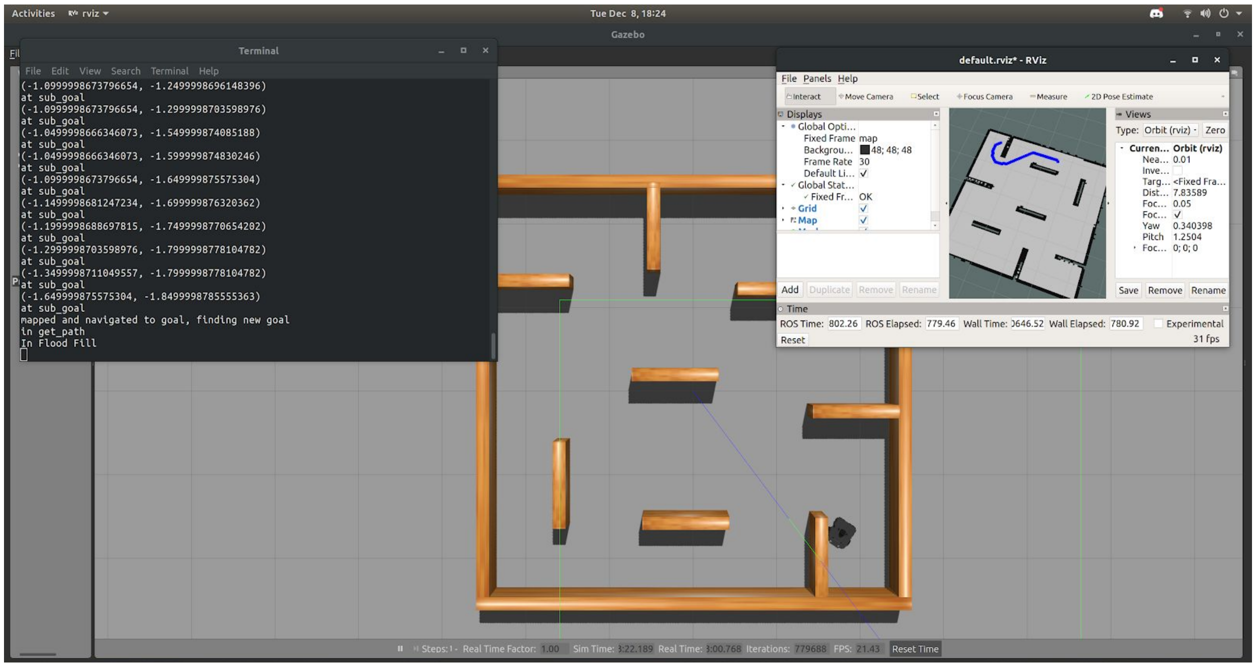
In part 2, this would result in a similar path to before, and was not as helpful when the robot was just having trouble getting around a corner. However, in part 3, this proved useful during our tests because the usual case was that the robot had mapped the goal or subgoal, but had not successfully navigated to it. Prompting the robot to chart a new path at this point led to it leaving the room it was stuck in successfully.

3. Discussion of your next way point algorithm (how did you get the robot to follow your plan?)

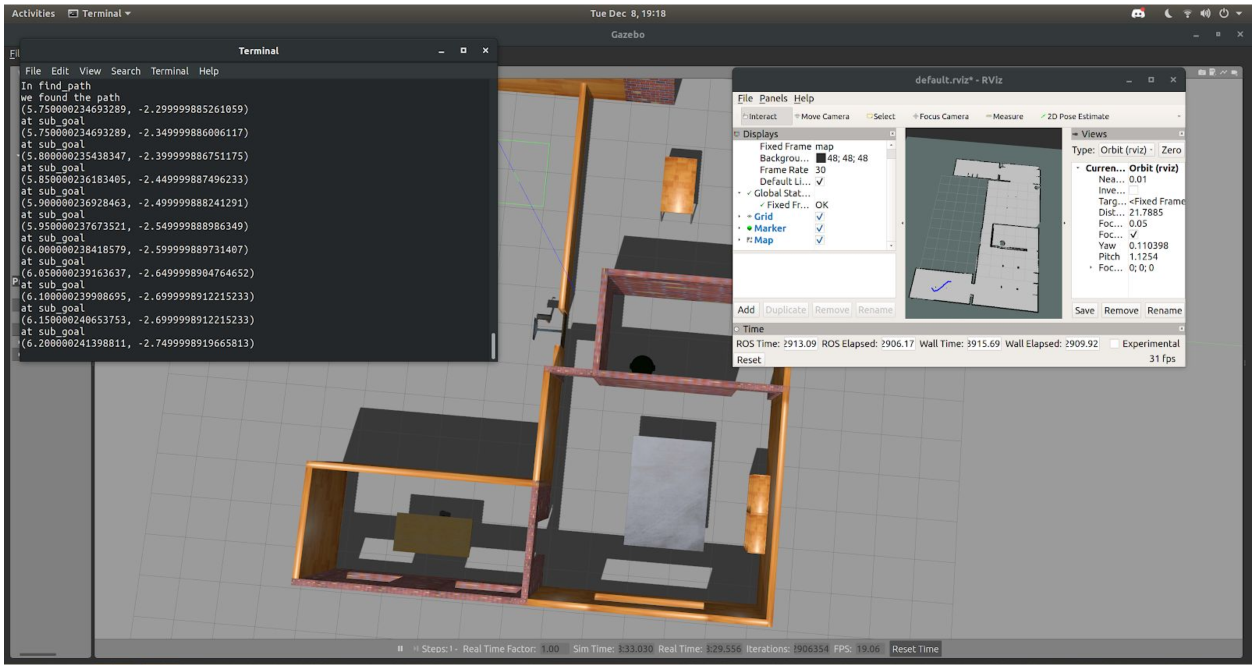
Our robot is given a list of subgoals in which it navigates before the end goal. At first, we found subgoals by splitting the path into chunks of four, given that the overall path was less than a specified length, and then recalculating the path once we reached the first subgoal. This was time expensive, especially for longer paths, given that we were recalculating the costmap at each subgoal. In retrospect, we could have used a single costmap given that, in part 2, we were using a static map of the entire world. We also encountered an issue with this approach in that subgoals would be on the other side of walls or corners, and the robot had extreme difficulty navigating around these obstacles.

Our second approach was to find the corners of the path, and save these as a list of subgoals. When we reach a subgoal, we remove it from the list and set the next point in the list as the sub goal. Originally, the intention was to compute the direction between points in the path, and at a change in direction greater than a few degrees, mark the midpoint of the vector as a subgoal. However, what we ended up implementing was a simple check to see if, not the previous point, but the point before that, was at a diagonal from the most recent point on the path. While this does create many arbitrary subgoals in path's with long stretches of diagonal movement, it creates many useful subgoals around obstacles and corners. Because our robot must be within 0.25 rviz/gazebo units of a subgoal in order to remove it from the list, the robot tends to handle corners by taking slower, wider turns.

4. Analysis of your algorithm's exploration performance. Did it result in full coverage of the environment (provide a screenshot from rviz)?



```
(-1.0999998673796654, -1.2499998696148396)
at sub_goal
(-1.0999998673796654, -1.2999998703598976)
at sub_goal
(-1.0499998666346073, -1.549999874085188)
at sub_goal
(-1.0499998666346073, -1.599999874830246)
at sub_goal
(-1.0999998673796654, -1.649999875575304)
at sub_goal
(-1.1499998681247234, -1.699999876320362)
at sub_goal
(-1.1999998688697815, -1.7499998770654202)
at sub_goal
(-1.2999998703598976, -1.7999998778104782)
at sub_goal
(-1.3499998711049557, -1.7999998778104782)
at sub_goal
(-1.649999875575304, -1.8499998785555363)
at sub_goal
mapped and navigated to goal, finding new goal
in get_path
In Flood Fill
```



```
In find_path
we found the path
(5.750000234693289, -2.299999885261059)
at sub_goal
(5.750000234693289, -2.349999886006117)
at sub_goal
(5.800000235438347, -2.399999886751175)
at sub_goal
(5.850000236183405, -2.449999887496233)
at sub_goal
(5.900000236928463, -2.499999888241291)
at sub_goal
(5.950000237673521, -2.549999888986349)
at sub_goal
(6.000000238418579, -2.599999889731407)
at sub_goal
(6.050000239163637, -2.6499998904764652)
at sub_goal
(6.100000239908695, -2.6999998912215233)
at sub_goal
(6.150000240653753, -2.6999998912215233)
at sub_goal
(6.200000241398811, -2.7499998919665813)
```

Our algorithm appears to be successful, though with some serious flaws.

1. **Provide suggestions on how your waypoint allocation algorithm could be improved.**

Because our flood fill algorithm is $O(n^4)$, as our robot explores more of the map, in the worst case it takes exponentially longer to find the path from our current point to the nearest unobserved point. Our algorithm is worst case $O(n^2 \pi r^2)$ because for each point on the map, if it is not a wall, we check the πr^2 nodes around it.[1]

The first improvement that comes to mind is to remove our "near wall" function, which checks all the points near a given point and inflates the cost of that point on the map if it is within r points of a wall. We would replace it with a path post-processing technique to separate a point on a path from walls it is near. This algorithm would likely be, in the worst case, $O(n^2 + p^2)$ where p is the number of points in the path. Because there will never be as many points in the path as there are total points in the map, in other words $n > p$, our overall time complexity would remain $O(n^2)$ and we would see an overall exponential reduction in time to compute a path.

Another improvement would be to run flood fill concurrently from both our starting point and our goal. When the two floods meet, the path would be computed from the point they meet at to the starting and ending points. This approach would have the same overall time complexity as standard flood fill. In the best case scenario, it would explore half as many nodes as standard flood fill, and in the worst case scenario it would explore the exact same number of nodes as standard flood fill.

2. Did your robot move fast enough? Get stuck?

Our algorithm took around 13 minutes to map the entirety of the closed maze world, and approximately the same amount of time to map a quarter of the closed house world. It did bump into a wall once or twice in the process, but only ever got stuck and had to re-plan once. What

[1] I am shaky on the math, but I believe this would be equivalent to $O(n^4)$ for the overall time complexity

mostly happened was when the robot tried to turn away from the wall, the back of it would scrape against the wall. When it did get stuck, our robot was trying to drive under a table, and because the lidar scanner can not see contours in the floor of the map, it did not see that under the table was a raised ledge, and would not try to go around it. Thankfully, our re-plan feature was able to get the robot to explore a different part of the world, as it had technically mapped the area it got stuck in.

In order for the robot to handle turns correctly, given our multiple subgoals around corners, our robot only accelerates when it is pointed in the direction of the subgoal, or in the absence of a subgoal, the goal itself. This, in combination with our expensive implementation of A*, leads to a painfully slow SLAM algorithm.

5. Provide suggestions for how your global planning (which points to explore next) could be improved

The best way to explore is to move in a forward direction without having to turn around for the best solution. that being said the robot once it chooses a path it will explore it regardless. What we would have liked to do is that while implementing A*, in the exploration phase, to have a more forward expanse of BFS, so that the robot doesn't have to turn as often unless it encounters a wall. Yes, collision avoidance will take over as the robot explores the best case scenario. However, having the robot drive in a linear pattern up until it approaches a wall is far easier to think of and perhaps implement.

5. **For example, if the entire environment is not mapped, explain what changes would fix this. If the entire environment is mapped, explain how a different explore point algorithm may have performed exploration more quickly.**

Because of how A* works in that it's a more refined BFS and Uniform cost search. an exploration that benefits keeping a forward momentum over a lateral exploration would be something that would fix unwanted crashes. Keeping a distance from the wall is something we discussed during our presentation. However, I have seen that segmenting a section and centering the pathing through each section ground zero allowed for easier traversal. This involves having to have a map already. Therefore a forward traversal compared until a wall is reached is far more cost effective.