Path Planning and Following in Mobile Robot

Juan Alvarez, Joel Santiago-Baretti, Erica Chen, Mohammed Ehab, Ningshan Ma
RSS: Robotics, Science, and Systems
Team 2

April 26th 2024

Path planning is a fundamental aspect of robotics which involves a robot determining a sequence of actions for an autonomous agent to navigate from an initial state to a goal state while avoiding obstacles and adhering to specific constraints. In this report, we explore a solution to this problem through exiting path planning methodologies. At a high level, our approach uses search based algorithm as A* to compute optimal paths from start pose to end pose and uses pure pursuit to command the robot to drive. Additionally, we investigate how various factors, including graph representation influence the robot's path planning process. **Keywords:** Path-planning, A* Search, Motion-planning

1 Introduction

Path planning is an essential element of autonomous navigation which entails an autonomous agent navigating from an initial state to a designated goal state while avoiding obstacles and following a developed path. This task is crucial for the implementation of a more precise controller and a successful self-driving robot.

Some problems that may arise from the implementation of a path planning algorithm are the development of non-optimized paths, improper integration with the driving controller, and unintended collisions. We discuss solutions to these problems in order to achieve our overarching goal of giving the autonomous robot an initial pose and a final destination, and then properly arriving at the desired destination. In order to construct an optimal path from the start pose to the end pose, an A* search algorithm was developed. The A* algorithm takes input from our graph-based representation and then creates an optimized path which avoids obstacles. The implementation of the A* algorithm with a Pure Pursuit controller and the necessary adjustments made to create a smooth and complete path are discussed. Elaborating further, the creation of a feasible graph representing our environment, the efficiency of the Pure Pursuit control, and the use of the A* search-based algorithm are modeled and explained.

2 Technical Approach

2.1 Path Planning Selection

2.2 Map Representation

To construct a feasible graph representing our environment for A* search, we begin by mapping the environment onto a grid-based representation. The information about the grid is provided in an occupancy grid, where obstacles are marked within the pixel coordinates. We iterate over neighboring pixels around a given node using nested loops to establish connections in the graph. For each neighboring pixel, if it falls within the map boundaries and is not obstructed by an obstacle, we add a path between them and add the neighboring pixel to the list of neighbors for the current pixel. To reduce the risk of collision and leave a safety margin around obstacles, we expand the boundaries of obstacles through dilation, effectively creating a buffer zone that helps avoid close corners. For our dilation choices, we experimented with different radius and chose the largest radius that doesn't close off any narrow path which is 10 pixels. Fig. 1 shows an example of a long path with dilation enabled to allow the car to navigate around walls and other obstacles without crashing, while Fig. 2 shows a path that is not viable because dilation was not performed.

We also experimented with changing our map space's granularity when creating the neighboring pixels, thus subdividing the pixel space from the resolution. As shown in Fig. 1, subdividing the pixels did give the

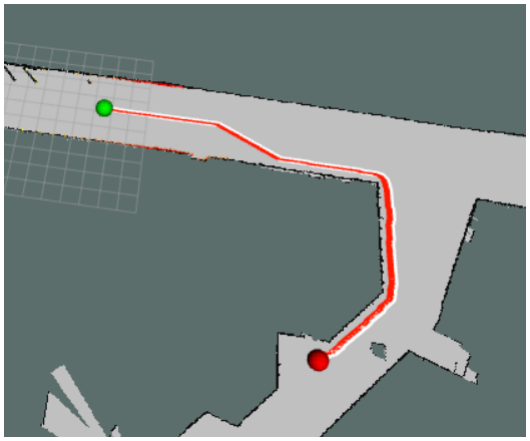

Figure 1: Path with dilation performed on map

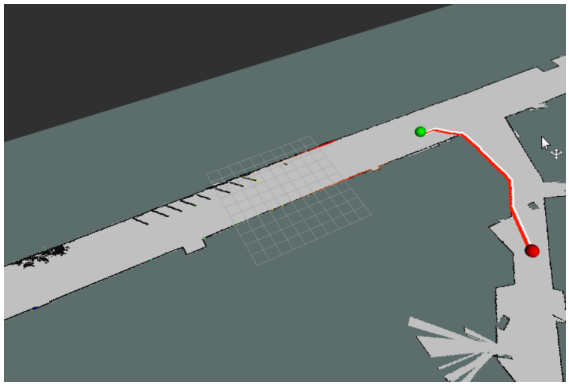

Figure 2: Path with no dilation performed on map

ability for the path to create shallower angles between neighboring map units. However, in testing we decided against implementing this due to steep costs for a shallow return.

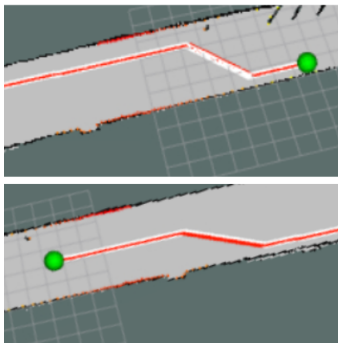

Figure 3: Comparison of ideal straight line path with pixel subdivision vs without pixel subdivision

2.3 Pure Pursuit

Once a path is planned, we need to implement an algorithm for the robot to follow it, and for that we chose pure pursuit control.
The idea behind pure pursuit is the following: draw a circle centered at the robot of radius L_1, and compute the last point P where the circle intersects the path; then, steer the robot as if it's heading directly towards that point. Since we are looking ahead in the path by a distance L_1, we will refer to L_1 as the lookahead distance.
The steering angle δ depends on the angle between the robot and the lookahead point, η, as well as the length of the robot L. We derived the following formula for η in Lecture 6:-

$$\eta = \tan^{-1}(\frac{2L\sin\eta}{L_1})$$

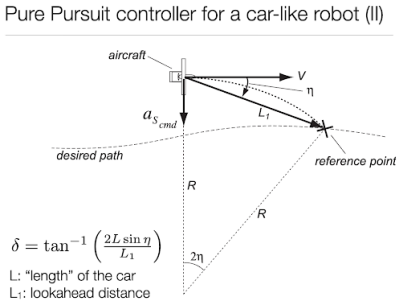

Figure 4: Illustration of pure pursuit control, from Lecture 6 slides.

In order to implement pure pursuit control for this lab, we need a way to compute the reference point P. The path is given as a piecewise-linear path with way-points $P_0, P_1, \ldots, P_n$. The path then consists of the line segments $(P_0, P_1), \ldots, (P_{n-1}, P_n)$. In order to compute P, we iterate over the segments one by one until we find a segment that intersects the circle of radius L_1 around the robot. We then compute the intersection point by solving a pair of simultaneous equations. Both of these operations can be done as follows: suppose the current segment we're checking is (P_i, P_{i+1}). Let $d = P_{i+1} - P_i$. We can parameterize the line through these points as $P_i + t \cdot d$, and in particular, the line segment is the set of points where $0 \leq t \leq 1$. If the robot location is P_r, we can then solve the equation $||P_i + t \cdot d - P_r||^2 = L_1^2$ to get the intersection between the circle of radius L_1 around the robot and the line through P_i and P_{i+1}. This is a quadratic equation in t, so we can solve it using the quadratic formula. If there's no solution, or if $0 \leq t \leq 1$ doesn't hold, the segment doesn't intersect the circle. Otherwise, if t^* is the solution, then $P = P_i + t^* \cdot d$. It could be the case that the quadratic equation has two solutions. To ensure that we obtain the latest point of intersection on the path, we pick t^* to be the larger between the two solutions.
Since this module was developed in parallel with the path planning module, the order of magnitude of how many line segments the path would contain was unclear, and we needed to ensure the most efficient implementation possible. Iterating over the segments one by one every iteration is inefficient and unnecessary

for the following reason: suppose that last iteration, the circle intersected the path at the fifth segment. Then, since the robot only moves forward, we do not need to check intersection with the first four segments again, since the robot already passed them. We can thus store a variable telling us which segment the robot is currently tracking and only increase that variable in time. This leads to a more efficient algorithm with amortized constant time complexity.

2.4 Path Planning

To implement path planning on our robot, we had to choose an appropriate path planning algorithm to use. We were given a choice of Search-based or Sample-based planning algorithms. Both options have various advantages and disadvantages. For example, sample-based algorithms generally run faster than search-based algorithms. In addition, some, such as RRT, are probabilistically complete, meaning that given enough time they will find a viable path if one exists. However, these algorithms are not optimal. They may always find a path, but there is no guarantee that it is the ideal path, and because of the probabilistic nature of every step of the algorithm, the path may be very jagged and include unnecessary random movements. There are ways around this, such as smoothing the paths, but we instead decided to use a search-based algorithm, A*. Search-based algorithms run slower on average than sample-based algorithms, but A* is both optimal and complete. This means that if a path exists, A* will always find it, and when it does, it will be the optimal path for whatever heuristic we use. We decided to use A* with a Euclidean distance heuristic. We also considered using Dubin's curves as a heuristic but after testing Euclidean distance we determined that the path generated by it was sufficient to be followed by the car without having to take into account more information about the car's dynamics.
To implement A*, we fed our map in as a graph where each pixel represented a node with some adjacent neighbors on which we could perform A*. As discussed in the Map Representation section, we also experimented with subdividing each pixel into multiple nodes to achieve higher granularity on the map and paths that were closer to optimal for certain situations. However, due to runtime issues and marginal gains in path optimality we decided against subdividing the pixels. We also defined a few helper functions to aid in our A* implementation, such as a heuristic function that calculated the Euclidean distance between two points, and functions that converted from pixel coordinates to map coordinates and vice versa. These were important because the A* algorithm took in start and end points as pixel coordinates, but had to output the path in xy map coordinates for it to be visualized correctly in the map.

3 Evaluation and Experimental Results

To evaluate our path following algorithm, we compared the given teacher assistant's path with our generated path. While their solution may not be the most ideal path, we determined that it was optimal enough to provide an adequate baseline. We utilized our A* implementation without pixel subdivision in our map space to generate Fig. 5.

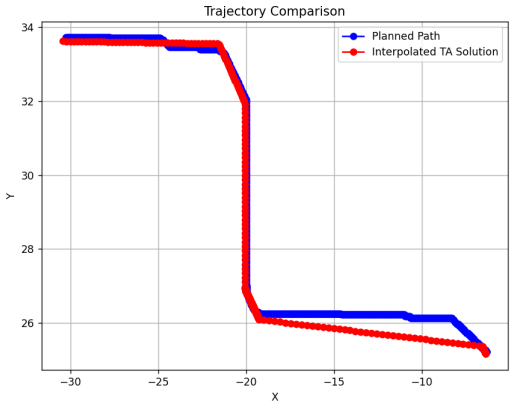

Figure 5: Graph of planned path without subdivided nodes. The average error was 0.42m while the average runtime was 2.3 seconds.

As shown, the path has a tendency to follow the grid in the map space. This is most evident from where $x \approx 6$ to $x \approx 20$. Our attempt at remedying this was to subdivide our pixels. In Fig. 6, we used the same path but subdivided each pixel into two.

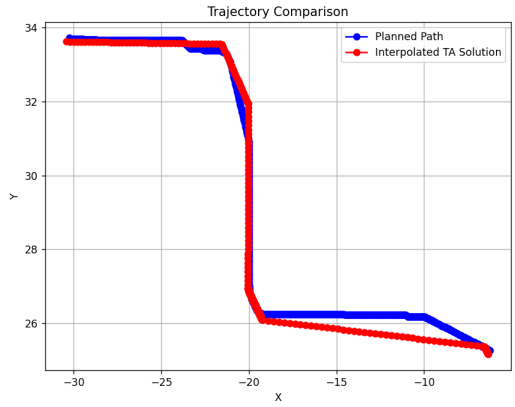

Figure 6: Graph of planned path subdividing each node into two. The average error was 0.35m while the average runtime was 24.4 seconds.

While the path still has a tendency to follow the grid space, the angle at which it takes to transition between the space is significantly shallower. However, when we compared the average errors between them, we

realized that this improvement was marginal, but the runtime increased by about a factor of ten. Further dividing our map space would likely exponentially increase our computation time while barely optimizing the path. Ultimately, we decided to revert back to our map representation without the subdivided pixels, as the average error was adequate and our algorithm did a sufficient job of avoiding walls in our real-life implementation. Given more time, we would likely be able to post-process our data to be able to fit a line during those long, angled straight paths.

Despite our path being sufficiently efficient, in our real-life implementation, there is a bit of oscillation on the real car that we do not observe in simulation. In an attempt to debug this, we hypothesized that the path following module might be running too slowly, since there were more than 300 line segments in the path. One reason we have that many line segments is an artifact of how the graph was built: only adjacent pixels are connected. Hence, a long straight line segment can end up being represented as a concatenation of hundreds of short ones.

To mitigate this issue, we decided to remove redundancies in the path by removing the waypoint P_i if it is collinear with P_{i-1} and P_{i+1}. After that modification, the number of line segments in the same path dropped to only 24. We then used the ros2 topic hz command to ensure the driving commands are being published in real-time, and they were indeed being published at 60 Hz. The lookahead point P was also being published at that rate and accurately tracked the path.

The oscillations marginally improved due to this modification, but they were still prevalent. Based on the evidence that every component of the system works perfectly in real-time, and that the oscillations do not arise as an issue in simulation, we concluded that the oscillations are more likely to be an artifact of our imperfect localization system rather than path following system.

4 Conclusions

Based on the smooth path following we obtained in simulation and the results we observed in practice, we conclude that our robot indeed accomplishes path planning and following, and that the oscillations observed can be smoothed out by revising and improving our localization. Another next step would be further processing our data to be able to create more efficient, angled lines.

Lessons learned by person:

- Ningshan: I learned to collaborate with another teammate on a specific module and sharing the responsibility for it. Juan and I collaborated on the path finding module and it turned out really well because we were able to debug together and discuss the steps.
- Ehab: I learned how to trust my teammates better and not worry about every detail. I tried to interfere as little as possible with the path planning module, and they just did a great job.
- Juan:
- Erica: From last lab, we put off collecting all the data until after we completed everything. But I learned that doing this incrementally is a much better alternative since it's hard to "revert" back to old code to collect data for comparison.
- Joel: I learned how collaborate with my teammates in order to gain a better understanding of the technical problems we faced. I developed my skills with RViz in order to simulate Ehab's pure pursuit on a sample trajectory.