Lab 6 Report: Path Planning

Wells Crosby, Paarth Desai, Ian Gatlin
Samay Godika, John Posada

6.4200/16.405 Robotics: Science and Systems

April 27, 2023

1 Introduction

Author: Samay Godika

Over the course of the semester, our team has implemented various different systems, from path following algorithms to localization. In this lab, we worked on bringing everything together to have one cohesive setup. Now that our robot is able to determine its location in a given environment, it is necessary for it to know what its next steps to take are. In this lab, we worked on planning a path given a start point and destination. Usually, the start point is taken to be the robot's location. Planning a trajectory is very important; it is vital for an autonomous car to not only have a path which is the shortest, but also one that is valid(i.e., only traversing legal roads, not trying to go through walls, etc.). The next step is to follow the created trajectory. The robot needs to make sure that it is adhering to the path that was created and going along that accurately. At the same time, it must be able to handle obstacles in its path, using the safety controller to avoid collisions. We created a pure pursuit controller which our robot used to follow the path planned. On the whole, this entire system is quite important on a large scale. In a real world autonomous vehicle, the user would simply give it the destination. The car would not only have to plan the path, but also follow it safely, legally, and accurately, all while accounting for real world obstacles and imbalances. Thus, we can see that this lab is one that is very applicable to the real world. During our process of working on it, we made sure to not

take shortcuts but instead attempt to implement the most efficient and correct code - we understood that this lab was one in which our robot could not malfunction in. The report describes our approaches to the different modules of the lab, and our results. After getting our implementations working, we evaluated it and tuned it based on different criteria, described in the sections below. Overall, we learnt a lot in this lab, and each part taught us a new important detail in the world of robotics!

2 Map Construction

Author: Wells Crosby

When starting the path planning program, we are given a known occupancy grid, which is a 2D array 'image' holding the probability from 0% to 100% of that 'pixel' on the map being occupied with -1 being unknown. Creating an efficient map representation to do shortest path search through is a key element of creating performant code. The given occupancy grid for Stata basement has size of over 1000 X 1000, which means running a shortest path search algorithm on this would effectively be on a graph with over a million vertices. Reducing the size of the graph we search on will yield huge performance gains. In order to do this we are going to take a relatively small random sample of points on the image. First we are going to threshold our occupancy grid, making any position with an unknown or greater than 5% occupancy probability to be definitely occupied. We will then dilate this image, so that any position within a 10 pixel radius of an occupied pixel will also be considered definitely occupied. We do this so that positions which are close to any objects are also off limits, so that we can prevent collisions. With this new image we are going to take a random sample of some number of unoccupied positions (we found that 1000 was a good number for Stata basement). These positions will be the vertices of our graph, we now need to connect them with edges. We are going to simply find some number (we found 20 to be a good number for Stata basement) of the closest vertices for each vertex. With these nearest vertices we will check if the straight line path to each of these vertices goes through any occupied position, if not, we will form an edge to that vertex. We then want to do a transformation of each of the positions of our vertices from the occupancy grid frame (pixel coordinates in the occupancy grid) to the world frame using the given translation and rotation values provided when we first get the occupancy grid.

We now have a graph representation of the map which is much smaller than the occupancy grid to run our shortest path algorithm on.

3 Path Planning

Author: Paarth Desai

After deciding how we wanted to represent our search space, we had to determine the fastest and most accurate way to plan our path. There are two primary methodologies within which we could choose to plan our path: search-based planning and sample-based planning. I'll give a brief introduction to each space of algorithms before elaborating more.

The search-based path planning space is a family of algorithms which take in a graph (a structure of nodes which are connected by edges, with each edge having a cost associated with it) which represents a given environment and then search this graph to find a path from a given start and end location. To break it up into further steps, these algorithms will take in their graph/tree, and then search this tree from the given start node looking for the end node in an ordering dependent on the specific algorithm itself, and then, if the algorithm finds a path, it will construct that path and return it. Any of these algorithms will only not find a path if it searches every node connected to the start node and does not find the end node, as they are all structured such that they will search the entire space completely. There are further families of algorithms within this space, such as greedy and optimal algorithms, but we will discuss these later.

Now moving onto sample-based algorithms, these algorithms typically require the environment to be represented as an occupancy grid, which was explained previously. They then sample points randomly in the open space in the grid, and depending on the algorithm, use these sampled points to construct a graph. These algorithms are moderately efficient, but not guaranteed to find the optimal solution. With this said, we will now discuss how we went about choosing our own algorithm.

3.1 Search-based Path Planning

Author: Paarth Desai

We primarily used a search-based path planning algorithm, as we believed that for speed, our algorithm should guarantee the optimal path within our

constructed graph. This also ruled out the greedy algorithms within search based path planning, as this type of algorithm optimizes for runtime rather than for the optimal path. Because we wanted to make sure we found the shortest path, this led us to having two main choices: Djikstra's algorithm and A* search. Djikstra's algorithm, also known as uniform cost search, works by beginning at the starting node, and then calculating the cost from each neighbor from the starting point, and building paths until the end node is reached. The algorithm will also update visited nodes' costs when it finds less costly paths to those nodes, allowing it to ensure that it will return the shortest path in the end. The algorithm sounds great, however, it can be improved for our purposes, and this is where A* comes into play. A* search is very similar to Djikstra's algorithm, however it introduces a heuristic function which allows for more efficiency. A* uses a user-designed heuristic to estimate the distance remaining from all nodes to the end node, and uses this estimate to choose the priority of which paths it would like to explore first. This function allows the algorithm to essentially search in the direction of the goal, so long as the heuristic function is appropriate. The image below, taken from redblobgames.com, displays visually the difference between these two algorithms when A* is implemented correctly.

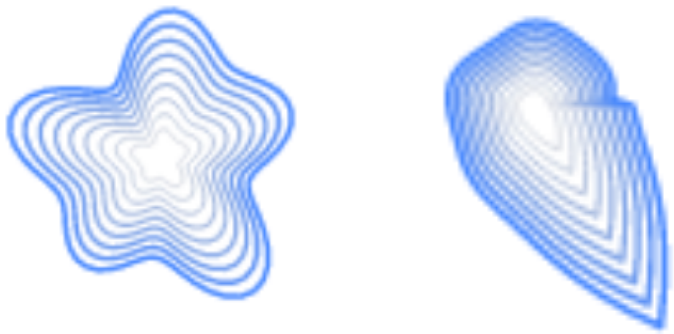

Figure 1: On the left: Dijkstra's algorithm. On the right: A*.

As mentioned earlier, both of these algorithms always find the shortest path when implemented correctly, however A* is typically faster due to the

nature of the heuristic function, whereas Djikstra's takes less space as it only stores values corresponding to the cost of edges on the graph, whereas A* stores these values in conjunction with the heuristic cost values. With this said, for our purposes, A* is certainly the better choice, as we are looking to generate a path as quickly as possible, and we can also generate an appropriate heuristic for this task.

The heuristic we chose is the Euclidean Distance between two points. For A* to get the optimal path, the heuristic must never overestimate the actual distance between two nodes, and because euclidean distance is the absolute shortest possible path between two locations, without accounting for any obstacles, it can never be an overestimate, making it a heuristic which is always admissible, and ensuring that A* will find the optimal path. Doing this allowed us to have good performance both on the robot and in simulation, but as we will discuss in the rest of this report, we wanted to try other algorithms as well.

3.2 Sampling-based Path Planning

Author: Ian Gatlin

To supplement our A* algorithm, our team also developed two sampling path algorithms to test relative effectiveness of all the different path finding algorithms in different situations. To begin, we used a modification of the probabilistic roadmap (PRM) algorithm. Wells described this implementation earlier in the report when he talked about sampling 1000 unoccupied locations in the Stata basement to create a graph. Once we have this graph, a user would need to a search based algorithm like A* on it to find a path. The key feature of PRM is that unlike a true occupancy grid, which is what search based algorithms are normally run on, PRM selects a random subset of points to run the search on. When a search algorithm is run on an occupancy grid, every point in the 2D grid is treated like a node in the graph. This almost guarantees an optimal solution, but often the granularity is unnecessary and can be too expensive for the efficient deployment of software in some situations.

This is what PRM solves by taking a sample of points from the occupancy grid. This is not a purely sampling based method, but rather a way to reduce complexity of search algorithms, which can be very effective in certain situations. If the car knew that it was in a more open area with less obstacles,

it would be smart to sample less points since having granularity in an open region is unneeded. The search algorithm would have to search through more nodes in a more granular graph only to come up with similar paths.

However, if the car was in a more obstacle-crowded region, it would be more necessary to sample a greater amount of points. If there are so many obstacles that the start node and the end region aren't connected in the graph after edges were drawn, the search algorithm that would run on the PRM graph would not return a solution. To solve this, the algorithm would have to sample more points to add to the graph, but this trial and error method can become very time consuming if many consecutively growing graphs fail to return solutions. Understanding the problem you are trying to solve when choosing variations in even simple path planning algorithms can greatly affect the magnitude of algorithmic performance.

The second sampling based path planning method we attempted to implement is the rapidly-exploring random tree algorithm. This works by building a graph systematically towards a goal, rather than just building a framework for a search algorithm to perform on. The algorithm starts with a start node and an end area. Then a point in the occupancy grid is sampled, and will be potentially added to the graph by connecting it to the existing node that it is closest to.

> This is an opportunity to incorporate dynamics into the algorithm since it is relatively more state dependent than PRM. At the beginning of the algorithm, the car has a known location and orientation, so the algorithm can intelligently sample a point that is in front of the robot to build this path, rather than sampling a point directly behind it that it would not be easily able to turn to. However, this was not present in our RRT implementation, but it could be useful for path planning in more constrained areas which may be present in the final challenge.

Before this addition becomes official, the prospective node and path are checked if they are in collision zones, but if they are not, they are added. The algorithm will continue sampling points and adding edges to the graph in this manner. Once an added point is added that is determined to be in the end region, the search terminates, and a path is returned immediately. This algorithm works well in situations where the search graph may need to be constantly reconstructed, which is the case if obstacles are moving around.

3.3 Comparsion of Algorithms

Author: Ian Gatlin

Currently, we have working implementations for A* on both a discretized grid graph and a random-point mesh graph. We are debugging our RRT implementation as it could be very helpful in the final challenge, but we do not have metrics for it. We used two paths to benchmark the performance of our two implemented algorithms.

	# of Nodes	Time until path found (ms)
A* w/ Grid Graph	2000	6.65
A* w/ Mesh Graph	1000	4.18

Table 1: Comparison of Graph Size to Search Time (short)

	# of Nodes	Time until path found (ms)
A* w/ Grid Graph	2000	16.82
A* w/ Mesh Graph	1000	11.21

Table 2: Comparison of Graph Size to Search Time (long)

The proportional speedup between the tests is relatively close, being at 0.63 and 0.67 respectively. This approximately 33% decrease in time is not significant on the scale that we are working at since these times are in milliseconds, but by order of magnitude this speedup can make a big difference. For this reason, we would most likely want to use the more optimal solution for our application, which would be the A* on an discretized grid. We are looking forward to seeing how these different algorithms continue to comparatively perform, and how optimized versions of them and RRT compare for the final challenge.

4 Pure Pursuit Controller

Author: Samay Godika

The next step after planning the trajectory is to follow it. For this, we implemented a pure pursuit controller. This involved two key steps. The

first step is to find the point on the trajectory closest to the car. This point updates in real time depending on the robot's position; for instance, if the car moves forward on the trajectory, the closest point will be the new location of the car. If the car moves off the trajectory, this point informs the car of where it actually should be and helps reorient it.

After finding the closest point on the trajectory, the next step is to calculate a lookahead point. We cannot simply tell the robot that its lookahead point is the final destination, since that may make it not follow the trajectory smoothly or not even complete the path at all. Instead, we calculate a lookahead point for the robot - a goal destination which is just slightly ahead of the robot on the trajectory. To do this, we assume that the robot is the center of a circle, and we use a function which calculates the intersection point of the circle and the trajectory. The intersection point ahead of the robot is the one we use as the lookahead goal. The robot then drives itself to this lookahead point. With the lookahead updating in real time, the robot is able to drive itself to the destination. Here, we were able to reuse code from the parking controller lab, with just a few modifications to ensure it doesn't stop upon reaching the lookahead point as it did with the cone. After our first implementation, we could see that the robot's movement was shaky. In the evaluations section below, the plots of the errors at different speeds are shown. To improve this, we worked on tuning the parameters in the PD controller. In our next steps, we plan to tune it further to make the robot run smoothly at higher speeds.

4.1 Implementation Evaluation

Author: John Posada

In order to ensure that our implementation of path planning and control was robust to different conditions and parameters, we established several qualitative and numerical metrics to evaluate each aspect of our algorithm.

First, we evaluated our pure pursuit controller to both compare the effects of its different parameters and to properly tune them. We were able to use the same trajectory and initial conditions during testing to make sure each trial depended on our parameter choices, as opposed to initial condition noise, although it is important to note that our localization algorithm uses Monte Carlo methods, and so introduces noise that way. We used instantaneous distance and instantaneous heading error between the car and the desired

trajectory, and an accumulated average error over the entire trajectory.

$$e^{i+1}_{\text{accumulated}} = \frac{1}{\Delta t^{i+1}} \left(\Delta t^{i} e^{i}_{\text{accumulated}} + e_{\text{distance}} \right) \tag{1}$$

$$e_{\text{average}} = \frac{e_{\text{accumulated}}}{T} \tag{2}$$

To tune the controller, we first tried using the Ziegler-Nichols method to choose proper PID gains. Testing at 4 m/s, we found consistent oscillations about a straight line trajectory at a proportional gain of approximately 2.0. The oscillations had an average period of 0.37 seconds, which produced gains Kp = 1.2, Ki = 8.0, Kd = 0.045. These gains produced inadequate results, and the resulting controller could not follow the testing trajectory. In order to avoid integrator windup, we instead tried a PD controller using Ziegler-Nichols derived gains, but this controller was also inadequate compared to regular P control, producing large oscillations and significant distance errors.

In the end, experimentally derived PD gains were used, with Kp = 0.4 and Kd = 0.01. These gains were evaluated at 2 m/s, 3 m/s, and 4 m/s, along the entire loop testing trajectory. The graphs below show the performance of this controller along straight lines.

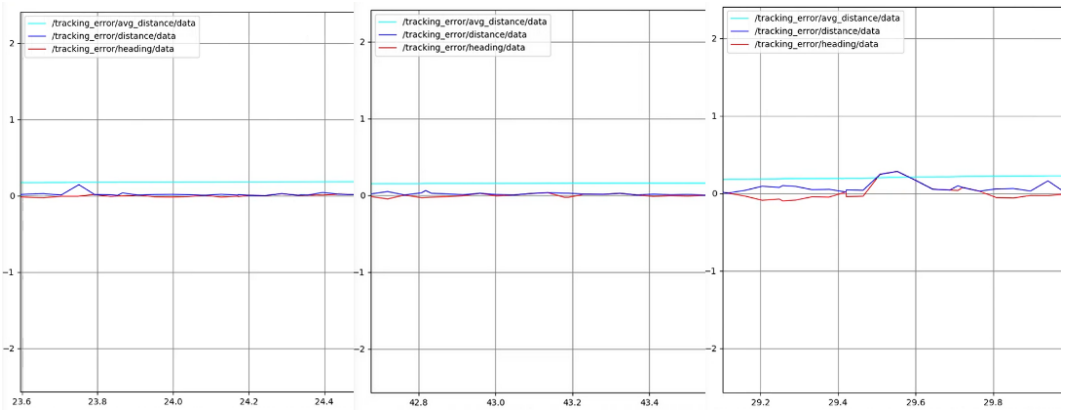

Figure 2: Error metrics reported while traversing a straight line. From left to right: 2 m/s, 3 m/s, 4 m/s.

The controller errors increased along sharp turns, which is to be expected as the controller must respond to discontinuous jumps in desired heading

along these turns. In every case, the error metrics found were within acceptable bounds. The graphs below show the performance of this controller along turns.

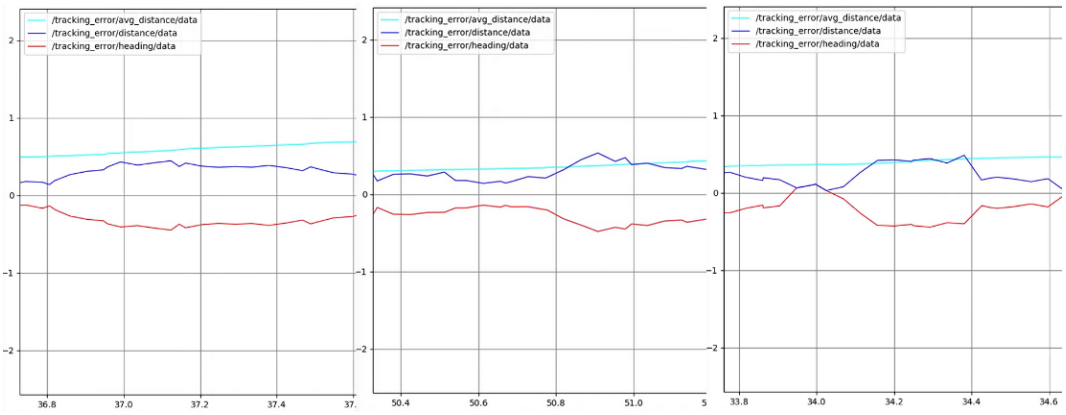

Figure 3: Error metrics reported while executing a right turn. From left to right: 2 m/s, 3 m/s, 4 m/s.

Overall, the distance and heading errors almost always stayed below 0.2 m and 0.2 radians. In addition, we found that the calculated average from Eq. 2 were low, and so used these gains going forward. The average errors were 0.0212m, 0.0239m, and 0.0343m at 2 m/s, 3 m/s, and 4 m/s respectively.

Finally, we also measured the time it took to generate our A* paths. We tried using a grid-like approach as well as a random-mesh approach to graph generation. We found that while the mesh approach allows us to easily modify the total number of vertices we want in our graph, it could often produce non-straight trajectories to paths that would best be performed using a completely straight trajectory. On the other hand, using the grid-like approach, the paths generated were much cleaner, but would take slightly longer to generate. As discussed above, testing on a very long path through the Stata basement map, we found that while it took 0.16 seconds on average to generate on a gridded graph, it took about 0.11 seconds to generate on the mesh graph.

5 Conclusion

Author: John Posada

Our team was able to effectively implement a quick path planning algorithm and pure pursuit controller in order to allow our robot to receive a desired goal pose, and reach it quickly and safely. This implementation continues from our previous implementation of Monte Carlo Localization (MCL), which allowed our robot to produce a high confidence estimate of its own pose using both its odometry measurements and its LIDAR measurements. Path planning is a problem that has many different solutions, and so our team was able to try out several different methods of slightly altering our implementation in order to ensure that our final solution was very efficient and robust to different speeds and perturbations on the robot. Our pure pursuit controller is properly tuned to ensure that it can follow straight lines and turns at varying speeds, which is also supported by our MCL algorithm. A* search was implemented and is able to quickly find paths throughout the given map, achieving times of less than 0.2 seconds in even very long paths. Our graph generation tested both a grid-like approach and a mesh-like approach, which had tradeoffs in speed of path generation and path jaggedness. Finally, we tried PRM and RRT as alternative path planning methods, which can avoid the time of generating a graph at initialization.

Overall, the algorithm implemented was effective on the robot throughout Stata basement to find its way from any location to any other location and navigate as quickly and as safely as it can. With much room for improvement and many other alternative paths to investigate in each aspect of the implementation, our team is looking forward to continuing on testing, debugging, and developing new features as we approach the final challenge, in which our robot will have to generate and navigate paths with high confidence. We've seen through this lab that our robot has achieved a very satisfying form of autonomy, being able to take a desired end result and doing the rest of the work itself!

6 Lessons Learned

6.1 Wells Crosby

In this lab there were a lot of lessons learned. On the technical side I learned a lot about efficient image processing (in relation to the occupancy graph), learning to work with the necessary python libraries needed to do the processing and then validate the results was fun. Additionally, learning graph

construction was a fun challenge, there were many intermediate steps such as effective sampling, 2D ray casting, and nearest neighbors search which provided a manageable challenge. Another huge area of learning in this lab was in how we integrated each component we worked on. I struggled with knowing exactly what output format was expected of my part, and how my result would connect to the other parts. For our future work, defining the structure of the program early on will make this more manageable, in addition to defining what the expected inputs and outputs of each of our parts are.

6.2 Paarth Desai

This lab made me think a lot more about the way we must apply path planning algorithms in the real world. I have a decent amount of experience using path planning algorithms from other courses, however this was my first time using them on an actual robot, and there were certain aspects which I'd never considered that were integral. For example, the fact that we had to create our own representation of a graph/occupancy grid from only an image of the environment which we were traversing was almost jarring for me, as I'd always considered the graph representation to be a given in these types of problems.

6.3 Ian Gatlin

In this lab I realized how applicable algorithms that many students have already learned about all throughout course 6 classes can be used in really cool real world ways. The search algorithms for cars that we developed all have tradeoffs, and therefore are optimal in different situations. This is a common theme in engineering, where no one solution can check all of the boxes. This is very clear in this lab heading into the final challenge where we will have to program the car to do very different tasks. I think we did a good job this lab splitting up work where everybody was doing something. I felt like everyone was very involved and its very nice to have a complete team of five heading into the end of the semester. I agree with Wells that sometimes it was hard to fit our code together. Talking even more about this beforehand can save us all a lot of time, as when I was coding the RRT algorithm, I wasn't sure if I needed to create my own discretization or if one was already built. Our code is relatively messy and we should write more

documentation and specification for it, but this requires a system in place, so it would be challenging to implement at this point.

6.4 Samay Godika

This lab was a big learning experience. Actually seeing how localization, path planning, and pure pursuit worked in conjunction with each other taught me a lot from a technical point of view. I not only learned how these systems worked independently, but also how they integrate with each other, which is not an easy task! We had to make sure that each block of code was running efficiently so that none of them were bottlenecks for the remaining parts. From a communication standpoint, I learnt how it is important to distribute tasks equally and communicate openly about the status of the work done. For instance, if someone has taken on an extra load, we worked together to ensure that no person felt overwhelmed. Overall, I enjoyed working with my teammates on this lab a lot and look forward to the final challenge!

6.5 John Posada

I felt this lab was one of our most difficult, but really was super satisfying to see done! Our team did really well in splitting up the work, integrating smoothly, and making sure we were making progress all the time, even if we kept running into bugs. Being able to integrate parts from our last couple of labs (localization, control, safety) was really cool and we were able to discuss each how each aspect affected each other - for example, we found that because we had not optimized some of our computations during the localization lab, our pose estimate would sometimes not update quick enough for very fast turns during pure pursuit. This really helped us think more about forward planning and making sure our implementations now will really help propel our future implementations. For the final challenge, we're dedicating some time to thinking about the best way to have the ROS node architecture, to make sure our computations aren't repeated, are optimized, and happen only when they have to. Overall, I think our team did really well and I'm glad we all work together super well! We're super excited to keep pushing on for the final challenge and think we've got some good ideas for each part's implementation.