ECE 484: Principles of Safe Autonomy	Spring 2021
Written by: Tianqi Liu, Minghao Jiang, Weihang Liang, Sayan Mitra	*MP 4: Path Planning*
	Due date: April 30, 11:59PM

1 Introduction

In this MP, you will implement a path planning module and add it on top of a controller similar to what you developed in MP2. The module will accept the map of Gazebo environment with some walls and obstacles, the GEM car's starting state, and the goal state as input, and it will produce as output a feasible path, that is, a series of way points, that takes the car from the start position to the goal.

You can get the ground truth car position from a ROS topic published by Gazebo simulator. Then the controller should drive the vehicle to the goal area without colliding with the obstacles in the Gazebo simulator. You will implement both A* and Hybrid A* path planners.

Same a previous MPs, this MP is divided into two parts. In the first part of this MP (section 2), you will be working on some theoretical problems about planning and value iteration. In the second part of this MP (section 3), you will work with the Gazebo simulator. You will need to use the knowledge from previous part of this MP and lecture, to develop path planning algorithms.

For part one, you will solve Problems 1-3. For part two, your will need to submit your solution to Problems 4-7. Name all the group members and cite any external resources you may have used in your solutions. More details for submission are given in Section 4. All the regulations for academic integrity and plagiarism spelled out in the student code apply.

Learning objectives

- Path planning
- A* and Hybrid A* Algorithm

System requirements

- Ubuntu 16
- ROS Kinetic
- ros-kinetic-ros-control
- ros-kinetic-effort-controllers
- ros-kinetic-joint-state-controller
- ros-kinetic-ackermann-msgs

2 Written Problems

Problem 1. A* search (10 points)

(a) Construct an example of a graph G with at least 6 nodes and 10 edges where A^* search fails to find the optimal path. Clearly mark the values of the heuristic function h at each vertex, the *start* and the *goal* vertices, the optimal path. Then show the step-by-step execution of the algorithm in a table to illustrate the sub-optimal path that the algorithm finds. *Do not use the graph from the lecture slides or search for an example online.*

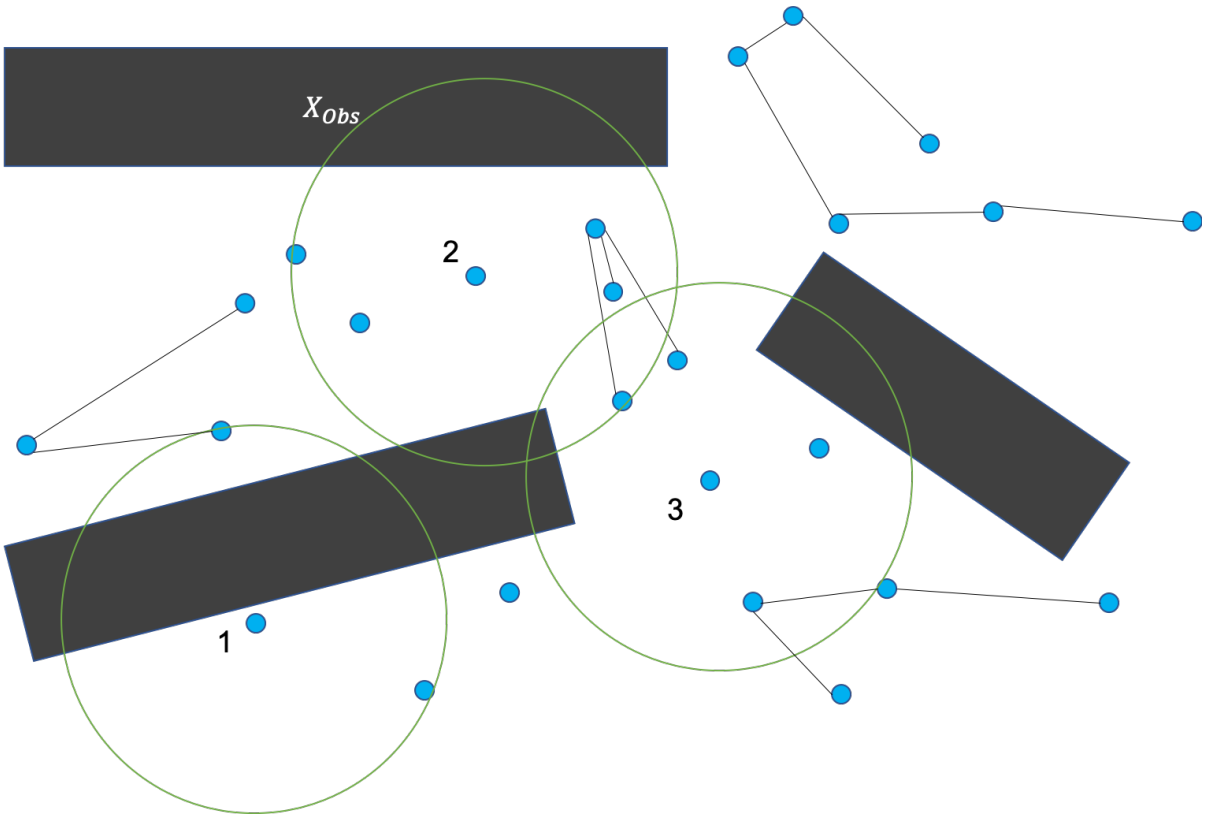
(b) Change the heuristic function h on the above graph G to an admissible heuristic. Illustrate the execution of A^* on this changed graph.

Problem 2. Probabilistic Road Maps (10 points)

(a) Recall the Probabilistic Road Maps algorithms from lecture. In particular, recall that the function $Near(G = (V, E), v, r)$ returns the vertices in V that are within r distance of v.

Consider the partially constructed PRM shown in the figure below. Suppose the next three vertices chosen in the outer for loop are marked by 1, 2, and 3, and the corresponding r-radius circles is shown. Suppose also that *CollisionFree* paths are found by drawing straight lines. Then, draw the edges that are added to he PRM graph after these vertices are processed.

(b) What are some of the advantages of PRMs?



Problem 3. Value Iteration (15 points) Let's extend the example of value iteration from class. Now, suppose we discretize the space into four states, where the final state is a violation of the stop sign.

As before, you can either stop or choose to move forward.

- When you perform a stop action, you stay in your current state with probability 1 and the MDP terminates (i.e., no more transitions occur).
- When you perform the move forward action, there is a 0.9 chance you move forward one state, but there's a 0.1 chance you move forward two states (unless you are in x_2, where you move to x_1 with probability 1).

Let's expand our reward function:

- If you enter x_1, you receive a collision penalty, $r(x_1, *) = r_c$.
- If you stop in x_2, you receive a goal reward: $r(x_2, stop) = r_g$.
- If you terminate early (in x_3 or x_4), you receive a small reward: $r(x_3, stop) = r(x_4, stop) = r_s$.

(a) Create a table that captures the state transition function.

(b) Implement value iteration with this set of values: $\{r_c, r_g, r_s, \gamma\}$.

(c) How does the policy change when you vary the payoff values?

3 Implementation

In this part of the MP, you will need to implement A* and Hybrid A* algorithms to help the GEM car navigate in either Highbay laboratory or ECE building. The first step is to implement the 2 path planning algorithms and test them with some test cases. Then, to drive the GEM car in Highbay or ECEB, you need to use the provided Dubin's car controller.

To start, you need to download the code from the gitlab repo . All the code blocks you need to write should be inside the file `a_star.py`, `hybrid_a_star.py` and `controller.py` from `/src/mp4/`. This document gives you the first steps to get started. You will have to take advantage of tutorials and documentations available online. Cite all resources in your report. **All the regulations for academic integrity and plagiarism spelled out in the student code apply.**

3.1 Module architecture

As usual, we will discuss the important functions needed to run the code. Note that you only need to implement some of the functions. The functions marked by + are not *required* for you to implement, but you can experiment with them.

Gazebo simulator$^+$ This whole MP will be tested in Gazebo simulator [1], which simulates the physical characteristics of objects like velocities, forces, friction, and collision. Each object is modeled in "models" folder. Some of the objects can be observed and controlled through ROS topics. Those need to be set up in .urdf files.

A* path planner A* algorithm is a graph search algorithm. For an admissible heuristic cost-to-go, it is optimal. That is, it is guaranteed to find the optimal path provided one exists. With the information of current GEM (discretized) position (x, y), the goal position (x', y'), and the positions of obstacles in the environment, the vanilla A* path planner needs to generate a path containing a series of adjacent way points that leads the vehicle from start position to goal position.

Hybrid A* path planner Hybrid A* algorithm is an extended version of A* that works over continuous spaces [2]. It can take into account the physics of the vehicle.

Controller Given the position of the goal point, the controller should navigate the vehicle to reach it. Note that this controller is different from the one you used in MP2.

3.2 A* path planner

The A* algorithm remains one of the most popular searching algorithm since its introduction in 1960s. It is has been successfully applied to many applications including robot planning. It uses a heuristic function h to estimate the most likely optimal cost-to-go. The main idea of A* algorithm is to minimize the cost function: $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost from the starting position to current position n and $h(n)$ is the estimated cost from n to goal.

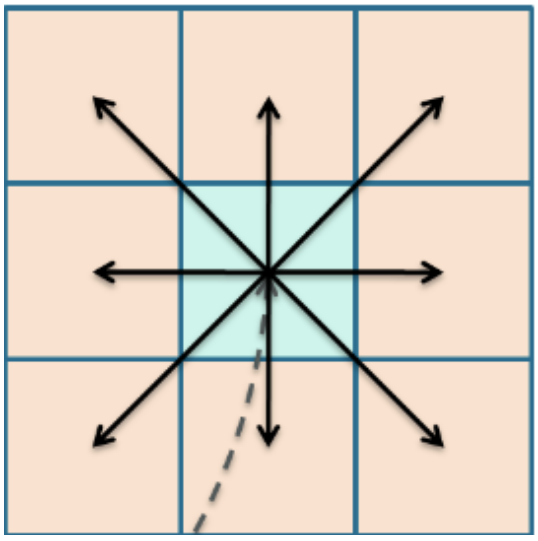


Figure 1: Simple A* [4]

The A* algorithm only works in discrete graphs. We need to divide our environment into a grid and assume the car is always at the center of that grid. To make our life easier, we can just set the size of the grid to be 1. Thus, each state will only contain the vehicle's x, y coordinates, which will always be integers. The vehicle can move to any of the 8 neighboring cells. To decide which cell to move to, we just need to minimize the heuristic function $h(n)$. You can use the Euclidean distance to the goal position as the heuristic function. You can also try other heuristic functions like Manhattan distance, L1-distance, and others.

In order to find the optimal path, we need to store not only the state itself, but also the path connecting to that state, and the current estimated cost f and actual cost g. We call such a bundle of information a *Node*. The nodes will be sorted based on their f values. For the starting node, the estimated cost h is the Euclidean distance between starting position (sx, sy) and goal position (gx, gy), and the actual cost g is 0.

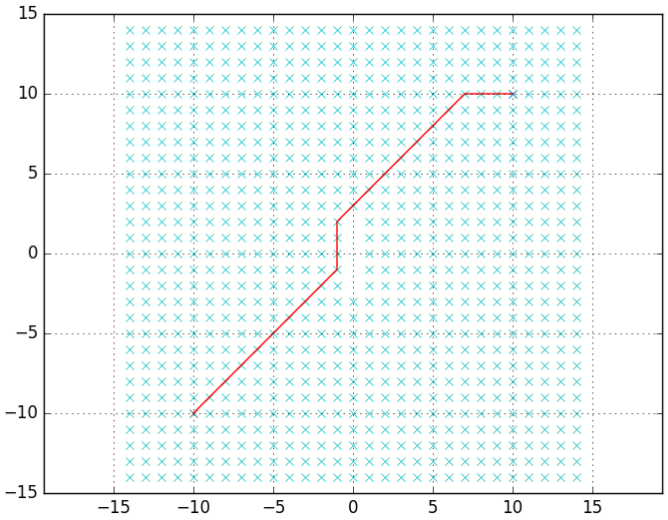


Figure 2: A path found by A* search. A test case.

Once you finished the simple A* code in a_star.py, you can just run it. A simple test case is offered. If everything works fine you are supposed to see a planned path connecting the starting point with the goal point.

```
python a_star.py
```

3.3 Hybrid A* path planner

The real world is better modeled in terms of a continuous space. Also, the results from A* algorithm may not be implementable by the actual vehicle since the physical constraints of the actual vehicle could may be properly represented. For example, a vehicle cannot stop instantaneously because of inertia; a wheeled vehicle has a minimum turning radius. and cannot change its heading too dramatically. These physical constraints make driving safety harder.

Hybrid A* works with continuous state space and physical constraints. Hybrid A* is a modified version of A*. It has been successfully applied on many autonomous driving platforms, such as Junior from the Stanford [2], which won the second place in the famous DARPA Urban Challenge in 2007. Read the above paper for a succinct overview of the algorithm.

The idea behind hybrid A* is that each node, represented by a coordinate (x', y') in discrete space, should be mapped to a coordinate (x, y) in continuous space that is reachable from the starting point based on the physical model of the vehicle. To represent the real world, we need to have the position of the vehicle in continuous space (x, y). Just as before, we also need to represent the vehicle state in discrete space (otherwise we will have uncountably many of nodes). In addition to x, y the nodes will also contain the heading angle θ. To simplify the calculation, we round all angles to integers (use degrees instead of radians). So the discrete state becomes (x', y', θ'). If the discrete states of two nodes are the same, we can treat them as if they are at the same position.

In order to ensure that the planned path is actually achievable for our vehicle, we need to introduce the Dubin's model that we have discussed previously in the modeling lecture. Instead of using the neighboring states of the current states as new states, now for each of the possible steering δ and speed v , we use this model to calculate the new states in continuous space. Note l is the length of the vehicle.

$$x_{new} = x + vcos(\theta)$$
$$y_{new} = y + vsin(\theta)$$
$$\theta_{new} = \theta + \frac{v}{l}tan(\delta)$$

The new cost g (at the new state) should be increased based on the control commands. For example, larger steering input should have higher cost.

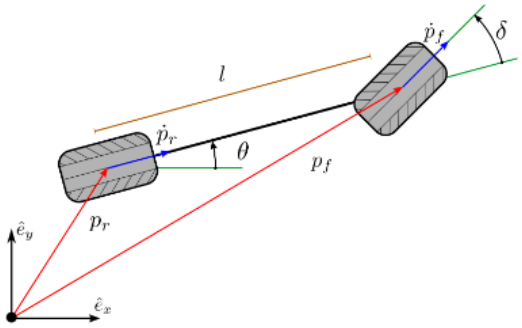


Figure 3: Rear wheel car model [3].

Once you finish the hybrid A* implementation in in hybrid_a_star.py, you can just run it. A simple test case is offered. If everything works fine, after a few seconds you should see a plot of a feasible planned path. Notice that this path, unlike the A* generated path, can be followed by our vehicle model very smoothly.

```
python hybrid_a_star.py
```

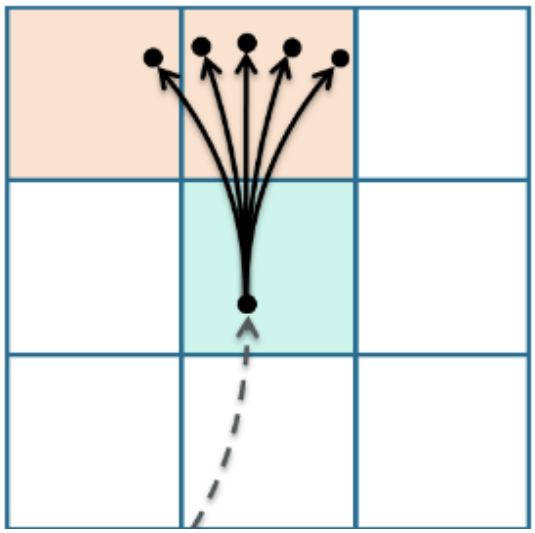


Figure 4: Construction of the graph for the hybrid A* algorithm. The edges are defined by discrete states that can be reached by the vehicle model with different inputs [4].

3.4 Design choices to consider

1. If your code runs too slowly, you may try to use existing python data structures like heapq to improve the performance. These implementations are highly optimized and usually bug free. Again, cite all libraries used in your report.
2. To further increase the performance, for each possible motion you can compute and store the changes to the current state before the propagation starts.
3. If you want to have a smoother path, you can add more steering angles to the possible steering control in the beginning of Hybrid_astar.py.
4. If your generated path contains too much reversing, you can adjust the cost g to "encourage" going forward.
5. When converting floats to integers in Python, use round() rather than int().

3.5 Start Gazebo simulator environment

Just like in previous MPs, MP4 also requires Gazebo simulator. First, go to the root folder of git repo, which is also the root for our catkin workspace, and build the code

```
catkin_make
```

Now, if you look in your current directory you should have a 'build' and 'devel' folder. Inside the 'devel' folder you can see that there are several setup.*sh files. Sourcing any of these files will overlay this work space on top of your environment. For every new terminal you open, you need to source again in that terminal before you run anything related to this ROS package.

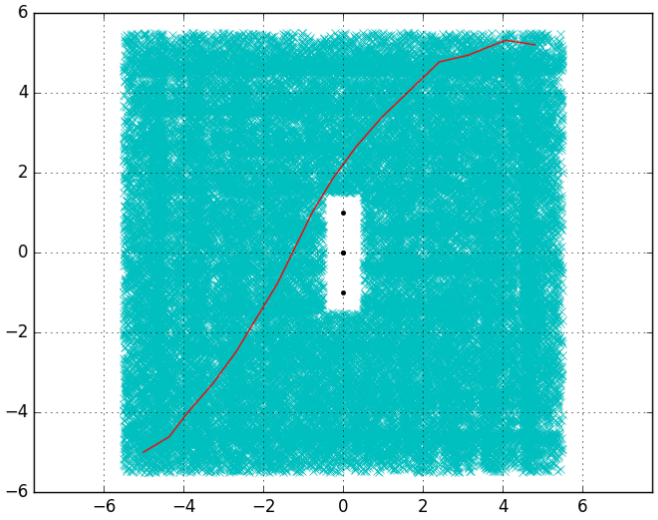


Figure 5: Hybrid A* Test Case

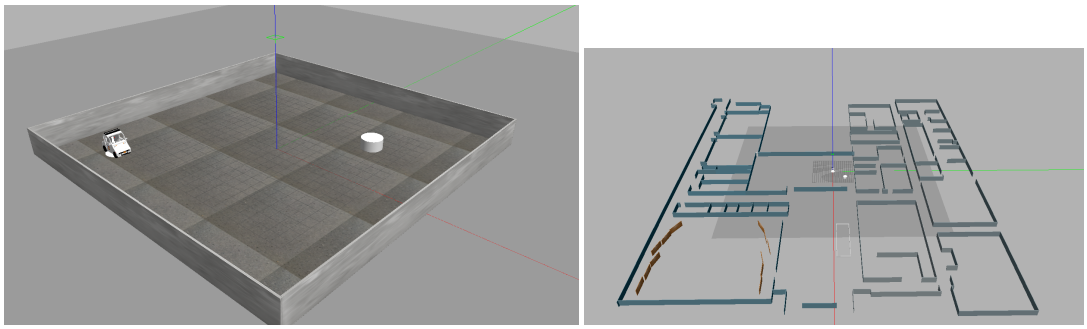


Figure 6: Two simulation environments for you to work in. Highbay (left) and ECEB1 (right).

```
source devel/setup.bash
```

Now we can launch the Gazebo. The GEM car, the surrounding walls and obstacles should show up in the simulator. We have created two environment for you. One is the "highbay" environment, the other is "eceb" environment. Fill in the environment name at <environment> and remember to use the same environment for mp4.py.

```
roslaunch mp4 mp4_<environment>.launch
```

3.6 Start the controller

To start the controller, We need to start the mp4.py. It will use the path planner to find the path from current location of the vehicle to the target position. Then it will call the Dubin's car controller to drive the car in Gazebo simulator.

To test your A* in Gazebo, run:

```
python mp4.py --algo a_star --env <env> --gx x1 x2 --gy y1 y2 --gtheta theta1 theta2
```

To test your Hybrid A* in Gazebo, run:

```
python mp4.py --algo hybrid_a_star --env <env> --gx x1 x2 --gy y1 y2 --gtheta theta1 theta2
```

During your development, if you want to move the GEM car to a different position (x, y), just run:

```
python reset.py --x <x> --y <y>
```

3.7 Test cases

We will grade you MP based on the following test cases. You can use reset.py to set the starting state,then go to mp4.py to set the goal state.

1. highbay, Start:(-10, 10), End:(10, 10, 0)
2. eceb, Start:(0, 0), End:(30, 0, 90)
3. eceb, Start:(-75, 60), Then:(-50, 45, -90), End:(-95, 50, 90)
4. eceb, Start:(50, 30), Then:(77, 15, -90), End:(90, -15, 0)
5. eceb, Start:(-100, -100), End:(50, 80, 0); (For this test case, you are only required to use A* path planner. The Hybrid A* might take too much time to finish.)

3.8 Report

Problem 4 (20 points) Observe the results from your code, what have you noticed? Attach the plots of paths for all test cases that your A* and Hybrid A* path planners found in your report. Please also record the time your planner takes to find the paths.

Problem 5 Design choices (25 points)

(a) What heuristic function are you using and why?

(b) Please use at least one different heuristic function other than the one in (a) and run all test cases. Plot the time it takes for path planners with different heuristic functions to find paths. (Your should plot the two runtimes with different heuristic functions against each of the test cases.) What have you noticed?

(c) How do you discretize the control inputs (speed and steering angle)?

(d) What are some other interesting design decisions you considered and executed, in creating the path planning module?

Problem 6 (10 points) Record a video for executions of all test cases with both A* and Hybrid A* path planner (except the last test case which you only need to show A* working). The video should include the Gazebo window. Provide a link to the video and include it in the report.

Problem 7 (10 points) Demo For this MP, you will need to demo your code to one of the TA. You can do the demo during any of the lab sessions or office hours. The TA will primarily check if your planner can plan an optimal path with A* and a reasonably good path with Hybrid A* and if your controller can follow the path generated. Note that the TA might pick any of the test cases.

4 Submission instructions

Each group should write a report that contains the solutions, plots, and discussions. This report should be submitted to Gradescope **per group** with filename `MP4_groupname.pdf`. Please include links to the video and your solution code in the report.

References

[1] C.E. Aguero, N. Koenig, I. Chen, H. Boyer, S. Peters, J. Hsu, B. Gerkey, S. Paepcke, J.L. Rivero, J. Manzo, E. Krotkov, and G. Pratt. Inside the virtual robotics challenge: Simulating real-time robotic disaster response. *Automation Science and Engineering, IEEE Transactions on*, 12(2):494–506, April 2015.

[2] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabriel Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, and Sebastian Thrun. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics*, 25:569 – 597, 09 2008.

[3] Brian Paden, Michal Cáp, Sze Zheng Yong, Dmitry S. Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Trans. Intelligent Vehicles*, 1(1):33–55, 2016.

[4] J. Petereit, T. Emter, C. W. Frey, T. Kopfstedt, and A. Beutel. Application of hybrid a* to an autonomous mobile robot for path planning in unstructured outdoor environments. In *ROBOTIK 2012; 7th German Conference on Robotics*, pages 1–6, May 2012.