ECE 498SMA: Principles of safe autonomy	Spring 2020
Written by: Tianqi Liu, Sayan Mitra	*MP 3: Path Planning*
	Due date: March 6, 11:59PM

1 Introduction

In this MP, you will implement a path planning module and add it on top of the controller 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.

In the real world, we will need hardware and software to localize the vehicle. Localization will be covered in next MP, but right now you have perfect indoor localization. That is, 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.

To start, you need to download the code from a 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`. 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.**

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 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. You should have already finished this part in MP2. If you do not have the controller code, ask us.

3 Development instructions

3.1 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. You can use the Euclidean distance to the goal position as the heuristic function.

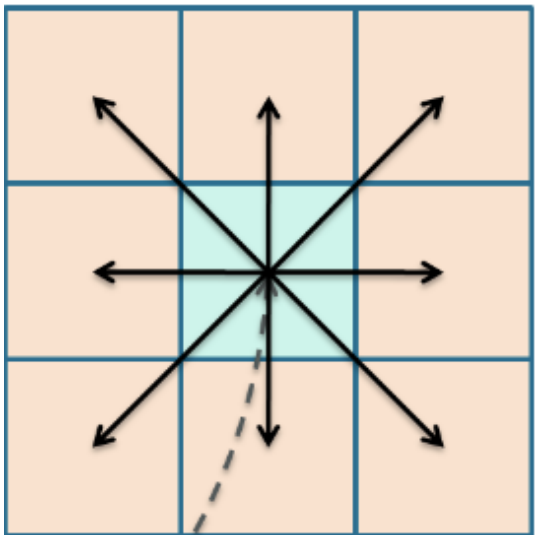

Figure 1: Simple A* [4]

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.

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. Then at each step, we can set the actual cost g be to the actual traveling distance from starting point to current position.

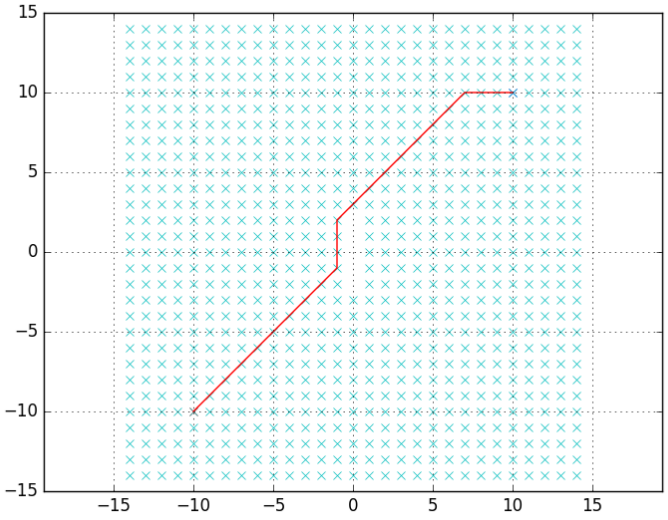

Figure 2: A path found by A* search. A test case.

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. Feel free to change the start and goal state to test the robustness of your code. (The green crosses represents explored positions, it is not required but we recommend you to plot those for debugging purposes.)

```
python3 a_star.py
```

3.2 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 can round all angles to integers, making $\theta = \theta'$. 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 rear wheel 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 increase in cost g.

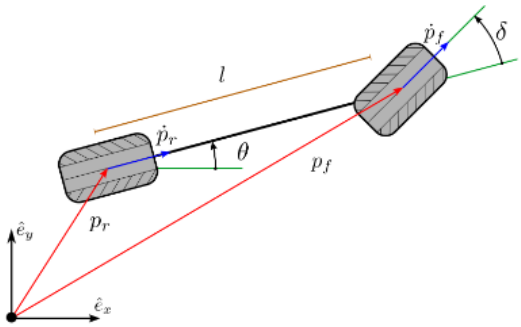

Figure 3: Rear wheel car model [3].

Once you finish the hybrid A* implementation in in 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. Feel free to change the start and goal state to test the robustness of your code.

```
python3 hybrid_a_star.py
```

3.3 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.

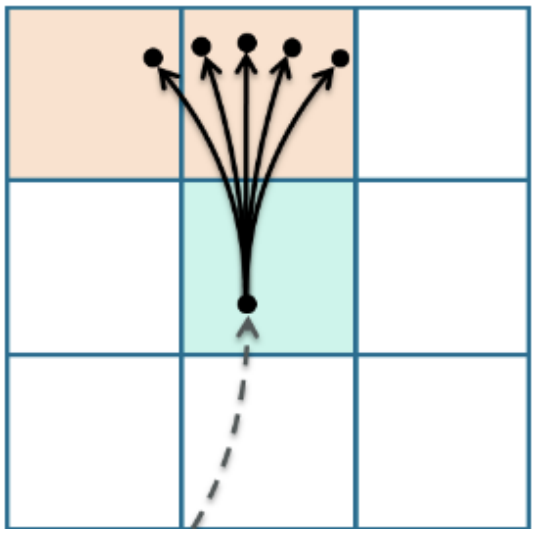

Figure 4: Construction of the graph for the hybrid A* algorithm. The edges are defined by discrete states that can be reached by the vehicle model with different inputs [4].

3. 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. The controller is also an important factor in overall performance. You may adjust your k1, k2 and k3 in controller.py to make the GEM car follow the path better in Gazebo.
5. If your generated path contains too much reversing, you can adjust the cost g to "encourage" going forward.
6. When converting floats to integers in Python, use round() rather than int().

3.4 Start Gazebo simulator environment

Just like in MP2, MP3 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
```

Hopefully you did not get any errors. 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.

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

Then we need to import the model we created into Gazebo.

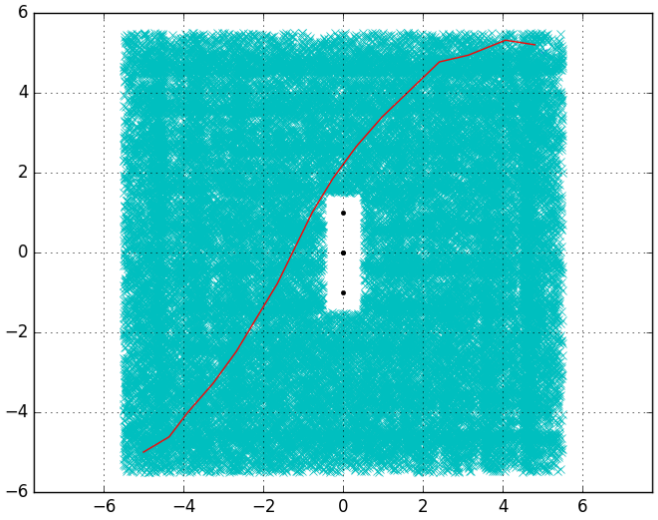

Figure 5: Hybrid A* Test Case

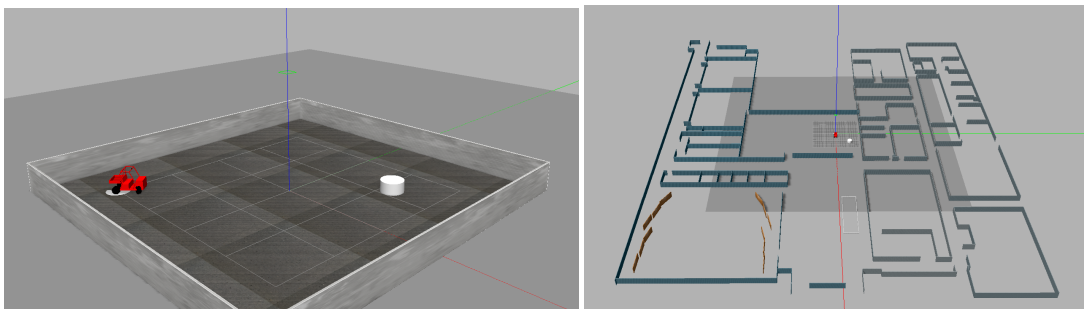

Figure 6: Two simulation environments for you to work in. Highbay (left) and ECEB1 (right).

```
export GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:<path to work space>/src/mp3/models
```

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 mp3.py.

```
roslaunch mp3 mp3_<environment>.launch
```

3.5 Start the controller

Just like MP2, we need to start the mp3.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 controller you developed in MP2 to drive the car in Gazebo simulator. Hence you need to copy some of the code blocks from the controller.py from your src folder in MP2 into the controller.py in src folder in MP3.

To test your A* in Gazebo, run:

```
python mp3.py a_star <environment>
```

To test your Hybrid A* in Gazebo, run:

```
python mp3.py hybrid_a_star <environment>
```

When the path is found, you will see a plot of the path, like what you see in a_star.py and hybrid_a_star.py. Save those figures and attach them to your report. Close the figure and the GEM car will start to move in gazebo simulator.

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

```
python respawn.py x y
```

3.6 Test cases

We will grade you MP based on the following test cases. You can use respawn.py to set the starting state, then go to mp3.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:(50, 30), End:(80, 0, -90)
4. eceb, Start:(-75, 60), End:(-95, 50, 90)
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. There will be a 5-point extra credit if your Hybrid A* planner can find the optimal path for this test case within 3 minutes when demo to TA.)

4 Report

Each group should upload a short report (2-5 pages). First, all group members' names and netIds should be listed on the first page. Then, discuss the the following questions possibly with screenshots:

1. Observe the results from your code, what have you noticed? For all test cases, attach the plots of paths that you A* and Hybrid A* path planners found in your report.
2. What are the interesting design decisions you considered and executed, in creating the path planning module?

3. What is each member's contribution and the number of hours spent (this does not have to be equal)? Give a rough breakdown of the time spent in the table format we have given earlier.

4. Any feedback you have for SafeAutonomy498-team on the Lab and the MP?

5 Submission instructions

Before the deadline, students need to submit the code(.py files) that you have changed, and the report to Compass 2G by one of the members in your group (homework needs to be submitted individually). Then in the following week, each group should demo the code to the TAs. You can choose to demo right after your lab or come to any of the office hours. All members should show up and be prepared to answer some questions. The TA might pick a random person and ask that person to answer the question. So all students should be familiar with the content.

6 Grading rubric

- 25%: Code is submitted on Compass 2G with quality results on the test cases(5% for each test cases)
- 30%: The TA will pick one of the hard test cases for A* and Hybrid A* planner. Your code should generate good results on the test cases during the demo. The running time for any test case should be < 5 minutes.
- 20%: TAs questions are answered properly during the demo.
- 25%: Report clearly answers all the questions

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.