Assignment 2 Planning for Self-Driving Cars

Due: 20.12.2023, 12:00AM

Introduction

Welcome to Assignment 2 of 'Techniques for Self-Driving Cars'! Make sure you have watched our lectures on planning. This file will lead you through the provided framework and present the four exercises.

Framework

For the assignments in this course, we provide you with a framework which manages the integration with the CARLA simulator and lets you focus on the implementation of a particular technique. The framework can be downloaded from eCampus as a `zip` file and has the following structure:

```
.
|-- ex2_planning.pdf
|-- params.yaml
|-- pyproject.toml
|-- scripts
|   |-- main_planning.py
|-- setup.py
|-- src
|   |-- pyilqr
|   |-- sdc_course
|       |-- control
|       |   |-- controller.py
|       |   |-- decomposed.py
|       |   |-- __init__.py
|       |   |-- mpc.py
|       |   |-- pid.py
|       |   |-- purepursuit.py
|       |   |-- stanley.py
|       |-- __init__.py
|       |-- perception
|       |   |-- __init__.py
|       |   |-- perception.py
|       |   |-- utils.py
|       |-- planning
|       |   |-- a_star.py
|       |   |-- behavior_planner.py
|       |   |-- global_path.p
|       |   |-- global_planner.py
```

```
|       |   |-- global_route_planner_dao.py
|       |   |-- graph.py
|       |   |-- __init__.py
|       |   |-- local_planner.py
|       |   |-- simple_global_planner.py
|       |   |-- simple_local_planner.py
|       |-- utils
|           |-- car.py
|           |-- __init__.py
|           |-- scenario1.yaml
|           |-- scenario2.yaml
|           |-- scenario3.yaml
|           |-- town03_map.png
|           |-- town03_signs.txt
|           |-- utility.py
|           |-- window.py
|           |-- world.py
|-- tests
    |-- test_planning.py
```

You can already see that it is similar to Assignment 1. We just added some files which are related to planning. We also provide the solutions for the PID and Stanley controllers in case you had problems solving Assignment 1. You can of course also use your own `controller.py` file, just make sure to use our new `params.yaml` with some updated parameters for planning.

File you need to edit:

Files	Description
`graph.py`	A graph class for storing and handling nodes and edges.
`a_star.py`	Search method which uses A Star to compute the shortest path from a start node to an end node.
`local_planner.py`	Class which computes a local reference trajectory and velocity based on the current maneuver and perception.

File you might have a look at:

Files	Description
`car.py`	Describes the car class.
`world.py`	Sets up the CARLA world.
`utility.py`	Consists of several helper functions which are used in the framework.
`global_planner.py`	Class which builds a graph from the carla map and computes the global reference path using your A Star implementation.
`behavior_planner.py`	A finite state machine keeping track of the current maneuver.
`params.yaml`	All the parameters for control and planning.

We will now explain the framework we implemented for you:

Global Planner

In Assignment 1, we used the `SimpleGlobalPlanner` to get a fixed global path from a `.txt` file. In this assignment, we want to navigate to an arbitrary goal on the CARLA road network by using the `GlobalPlanner`.

Our CARLA car does not need any fuel, however, we still want to drive on the shortest path! Therefore, we need a graph of the road network and a global planner that outputs the list of reference waypoints with the fewest costs. As you know from the lecture, a graph consists of nodes which are connected by edges. In our case, a node represents the car's location and an edge encodes additional information like cost or speed limits.

How do we obtain a graph from CARLA? Remember, we should only add nodes and edges which are valid. This means we only add an edge between two nodes if the CARLA car is allowed to drive there. In this assignment, we already implemented a `_build_graph()` method in the `GlobalPlanner` that does that. You can have a look at the `global_planner.py` to see how that works. In Short: We can access a list of road segments from the CARLA map and represent start and end of a segment as nodes in the graph. Then, we connect adjacent segments and check if a lane change is possible on a segment. Here you can see how the graph looks like in the CARLA world:

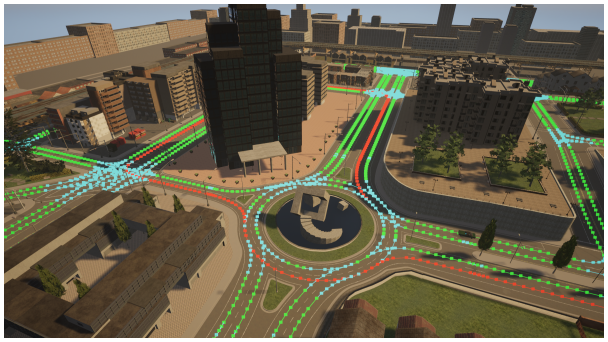
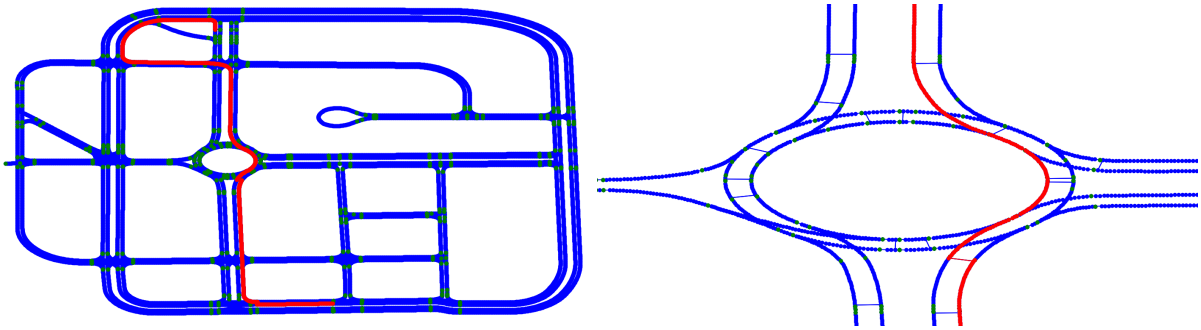


Figure 1: Road Network Graph

The green color indicates that the segment belongs to the semantic class 'street', whereas blue is a junction with more than two edges connected to the node. The red path is an example for a global path. You can already see that the shortest path is sometimes not intuitive if you need to follow the driving rules.

You might ask yourself why there are so many plotted waypoints. In fact, we have more waypoints plotted than nodes exist. As mentioned above, each segment represents an edge with an entry node and an exit node. Since the segment can also be a curved street, we need more waypoints in between to retrieve the final path. Therefore, our edges also store a list of 'in-between' waypoints. When extracting the global path from the list of optimal nodes, we just put these additional waypoints in between the entry and exit waypoints.

Have a look at these top down images for a better understanding. We indicate nodes as green dots, additional waypoints as blue dots and the shortest path between two nodes as red dots. Would you have taken the same path?



One last thing about the graph: The possibility of changing the lane within a segment is represented by one additional edge in the segment. This is not accurate since we could also change the lanes in the middle of a segment, but this simplification is sufficient for the global planner. You want a more specific behavior of the CARLA car for different situations? We provided you with a very simple `BehaviorPlanner`!

Behavior Planner

Our Behavior planner is a finite state machine with three states: `junction` if we are driving on a segment which is a junction, `overtaking` if there is a parked car nearby or `default` otherwise. Why? We just want to drive as fast as the speed limit and reduce the speed for making turns at intersections and for overtaking. One could also handle more manuevers in this class like yielding, emergency braking, parking.

Local Planner

The local planner outputs a local reference path and a target velocity which are then passed to the controller. Given the global path and the current position, the local planner

1. generates trajectory candidates with a fixed lookahead horizon,
2. scores them according to some cost function,
3. sorts the trajectories by score and returns them.

We decided to generate more candidates by also considering adjacent lanes. This makes it possible to overtake vehicles on a two lane road even if the global path wants us to stay in the lane. The scoring of trajectories allows for avoiding those that end up in a collision and favoring those that stay close to the global path.

In this assignment, we will consider two scenarios. For the first three tasks you should set the scenario in the `params.yaml` to `'scenario1'`. In this scenario, you can select a goal location on the small map on the upper left. The second scenario `'scenario2'` fixes the start and end location and we will spawn some parked vehicles to check if your local planner is working properly.

Evaluation:

You can check your implementation for the first three tasks by running

```
$ pytest tests/test_planning.py --no-summary -s
```

Also, after solving task 2 and 3 you can run

```
$ python3 scripts/main_planning.py
```

to see the car following a trajectory. Click on the small map to change the goal location If your A Star search is correctly implemented, the car will recompute its global plan by finding the shortest possible path to the goal.

The last task is carried out with `'scenario2'` (change it in the `params.yaml`) and you can check your implementation by watching the car drive. We will evluate this part by looking at the results in the simulator. You pass task 4 if you are able to follow the global path on a smooth local trajectory without crashing into obstacles.

Note: The evaluation script is supposed to support you for debugging. We can not guarantee that it catches all possible errors and sometimes your implementation can be correct but our script fails to see that.

Submission:

Once you have completed your assignment by editing the relevant scripts in the framework, upload your solution as a `zip` file via eCampus with the filename `LastName1_LastName2.zip`. We will check the files that you need to edit. Make sure that your solution is working, when we download a fresh framework and replace it with the abovementioned files.

Please make only one submission per team.

Academic Dishonesty:

We will check your code against other submissions in the class. We will easily know if you copy someone else's code and submit it with some minor changes. We trust that you will submit your own work only. Please don't let us down. Note that in the case of plagiarism, you will be expelled from the course. We are strict on that.

Task 1: Complete Missing Methods in the Graph Class [25 Points]

There are some missing methods in our graph class to access specific elements from it.

Your task: Finish the implementation of the methods `get_path`, `get_children` and `get_cost`. Each method should consist of one line. Have a look at the class members and how nodes and edges are added to the graph.

Hint: We use unique node IDs to specify a node!

Task 2: A Star Search [25 Points]

The global planner is used to compute the shortest path on the graph. You have seen different search algorithms like depth-first, breadth-first, uniform-cost or A Star search in the lecture. Here, the cost between nodes is the distance the car needs to travel on that segment which is not always the same. Also, we know where the goal is and can use an admissible heuristic.

Your task: Fill in the rest of the `a_star_search` function in `a_star.py`. Use the `open_list` list to keep track of unexplored nodes and their estimated costs, the `closed_list` to track already explored nodes, the `came_from` dictionary for mapping a node to its predecessor, the `accumulated_cost` dictionary for mapping a node to its accumulated cost and finally the `estimated_cost` for mapping a node to its estimated cost including the heuristic.

If you don't solve this task, you can still do task 3 and 4. In that case, the global planner loads a fixed global path which only holds for the inital start and end locations.

Hint: You might use some of the methods implemented in Task 1.

Task 3: Cubic Spline Fitting for Local Planner [25 Points]

The `LocalPlanner` in `local_planner.py` generates multiple candidate trajectories by fitting a parametric curve between the vehicle's current pose $p_0 = (x_0, y_0, \theta_0)$ and a desired end pose $p_1 = (x_1, y_1, \theta_1)$.

Our parametetric curve is defined by two cubic polynomials for the x- and y-position of the car with respect to a running variable $u \in [0, 1]$ which can be treated as time normalized to 1. The curve is defined by

$$x(u) = a_3u^3 + a_2u^2 + a_1u + a_0$$
$$y(u) = b_3u^3 + b_2u^2 + b_1u + b_0.$$

The paramaters of the polynomials need to be determined to follow the constraints induced by start and end configuration. To fix the start and end heading angles, we introduce boundary conditions on the first derivatives in x and y since the orientation is given by

$$\tan\theta_u = \frac{c_u \sin\theta_u}{c_u \cos\theta_u} = \frac{y'(u)}{x'(u)}$$

with a scaling parameter c_u as an additional degree of freedom that does not influence the orientation. **Your task:** Fill in the method `_fit_cubic` that takes the start and end configurations as (x, y, θ) as well as the scaling parameters c_0 and c_1 and returns a matrix $P \in \mathbf{R}^{2\times4}$ reading

$$P = \begin{bmatrix} a_3 & a_2 & a_1 & a_0 \\ b_3 & b_2 & b_1 & b_0 \end{bmatrix}.$$

The parameters c_0 and c_1 influence the curvature and can be set in `_compute_parameters` to get different shapes of trajectories. Try it! Test your A Star algorithm and the local planner by launching `python3 scripts/main_planning.py` and replanning by clicking on a new goal location on the small map. The different local trajectories are rendered in your screen.

Hint: You can solve the system of linear equations with `np.linalg.solve`.

Task 4: Collision Avoidance [25 Points]

This is the final task. You will score the different trajectory candidates to find the best one. First, set the scenario parameter in `params.yaml` to `'scenario2'`. This will spawn some parked cars we do not want to collide with. The parked cars will be detected by our perception module which is assumed to be given for now.

In `main_planning.py`, we call the method `get_local_path` that takes the vehicle and the list of parked cars as input. We then determine the local plan's end position on the global path and generate a `parameter_list_unscored`. This list contains parameters computed in Task 3 for fitting curves to the start and multiple end locations depending on the road geometry and c parameters.

Your task: Fill in the method `_score_and_sort` in the `LocalPlanner`. This method takes the `parameter_list_unscored` and returns a list of parameter matrices which is reordered according to the score. Make sure the trajectory the car should take is the first element in the returned list, therefore `parameter_list_scored[0]`. The `main_planning.py` will plot all candidates in your window and select the first element as the local path.

How to compute the score? That is part of your task! Think about possible metrics for evaluating how good a trajectory is in terms of reference tracking and collision avoidance.

We will check this task by running your code and evaluating the path the car takes. You passed this task if your car is able to stay close to the global path while not crashing into any obstacles.

Hint: Use the `_generate_trajectory` method for generating a list of `n_points` locations for a parameter matrix. You can try to use a small `n_points` for reducing the computation time and still achieving a good performance.