Grundlagen der Künstlichen Intelligenz

Programming Exercise: Graph Search-Based Motion Planner with Motion Primitives

Edmond Irani Liu, Prof. Matthias Althoff Last updated: November 8, 2019

Finding safe motions in complex traffic situation is one of the major challenges in autonomous driving. Typically, one has to develop a large software framework before one can start developing new motion planning techniques. Such a framework typically includes software for representing road networks, reasoning on the networks, collision checking, simulating vehicle dynamics, and visualizing results, to name only a few aspects. Under CommonRoad software framework, all of these are taken care of. A standard CommonRoad scenario taken from the Stachus intersection in Munich is shown in Fig.1.

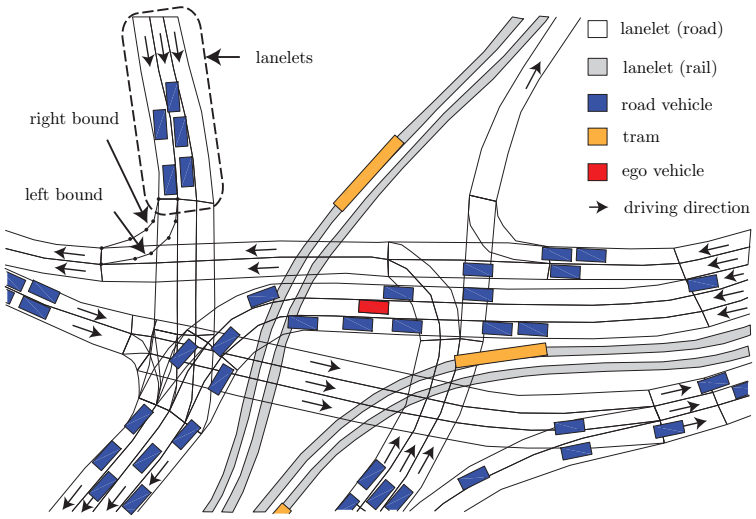

Figure 1: Visualization of Stachus in the city center of Munich under CommonRoad

To further facilitate the use of CommonRoad, we have written tutorials which demonstrate basic functionality of CommonRoad in a step-by-step manner. CommonRoad also contains a benchmark to which you can upload your solutions to the motion planning problems and see how well you perform in comparison to others. While initially most of the uploads will come from the students of this course, we will further promote the benchmark worldwide and also organize a competition with a prize.

1 Installation

1.1 Regular Installation

Please follow the instructions given in commonroad-search/README.md if you wish to install the software on your own machine.

1.2 Virtual Machine

We provide you with a Virtual Box image of Lubuntu 18.04 (ca. 6.5 GB after unzipping), in which all necessary modules are installed. Please download it via this link. The downloading password and default login password are both commonroad. In the terminal, please navigate to /home/commonroad/repos/commonroad-search/ and use the command jupyter notebook to launch the Jupyter notebook. If a warning prompts up showing that no kernel is found, please proceed by selecting python3.

2 CommonRoad Scenarios

Before implementing the search algorithm, it is essential to correctly understand the structure of motion planning problems in CommonRoad, which consists of:

- **Road network:** The road network consists of a series of road segments which are called lanelets. A lanelet is defined by its left and right bound, where each bound is represented by an array of points (a polyline), as shown in Fig.2.
- **Obstacles:** A scenario typically has both static and dynamic obstacles. Static obstacles do not move over time, such as construction zones and parked vehicles. Dynamic obstacle are moving objects and are mostly traffic participants, such as cars, trucks, motorbikes, bicycle riders, pedestrians and so on. For both of these obstacles, a geometry shape is provided; for dynamic obstacles, the movement over time is given by a trajectory, which is essentially a list of states over time.
- **Initial state:** The initial state of the planning problem has the following values:
  - **Position:** A two-dimensional vector $[x, y]^T$ in [m].
  - **Steering Angle:** A scalar δ in [rad].
  - **Velocity:** A scalar v in [m/s].
  - **Orientation:** A scalar $\theta \in [-\pi, \pi]$ in [rad].
- **Goal region:** The goal region of the planning problem is constructed from the Cartesian product of:
  - **Position Region:** A two-dimensional polygon within $\mathbb{R}^2$; each axis has the unit [m].
  - **Orientation Interval:** An interval of allowed orientation in [rad].
  - **Velocity Interval:** An interval of allowed velocities in [m/s].
  - **Time Interval:** An interval of allowed time of arriving at the goal region in [s].

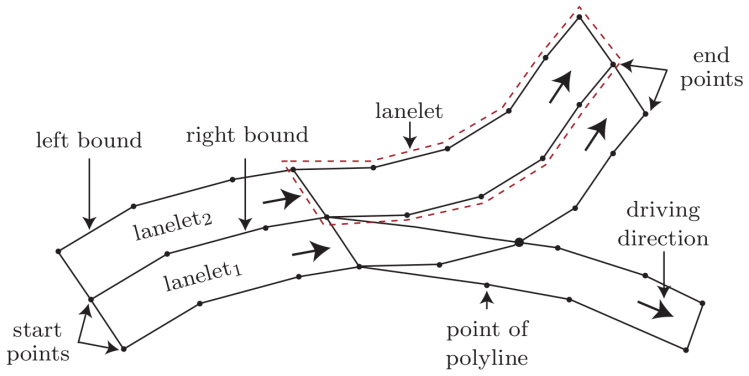

Figure 2: An example of lanelet.

Please note that all time stamps in CommonRoad are represented by an integer k. The time in seconds is implicitly given by the fixed, global time step size r (usually set to 0.1 second) of the scenario, so that $t = kr$. For example, time stamp $k = 61$ corresponds to time $t = 6.1$s, given that r is set to 0.1s.

A valid solution trajectory to a planning problem satisfies the following conditions:

- It starts at the initial state of the planning problem.
- It reaches the goal region of the planning problem within the given time interval.
- A vehicle does not collide with any obstacle and stays on the road while following this trajectory.

- The trajectory comply with a given vehicle model. In this tutorial, we assume that the Kinematic Single-Track model is used. See here for more details.

Please refer to notebooks/tutorials/1.Brief_Introduction_to_CommonRoad_io.pdf and notebooks/tutorials/tutorial_commonroad-io.ipynb for a brief introduction and a practical tutorial on CommonRoad-io, respectively.

3 Software Documentation

Your task is to implement a heuristic function and/or a search algorithm of your choice for the motion planning problems. Specifically, this algorithm should produce a trajectory by concatenating given motion primitives. A motion primitive is a short trajectory whose drivability under certain vehicle models is guaranteed. Fig.3 shows an exemplary search tree that is spanned via motion primitives, where states are indicated by dots. Every short trajectory between two successive dots is a motion primitive.

For this exercise we provide approximately 3000 motion primitives. These primitives all start at position $[x = 0, y = 0]^T$, have a duration of 0.5 seconds and differ in velocity, steering angle, orientation, acceleration, and length. If the velocity and the steering angle of the first state of a following primitive are equal those of the last state of a preceding primitive, we say these two primitives are **connectable**. notebooks/tutorials/motion_primitives_generator.ipynb demonstrates how are the motion primitives generated.

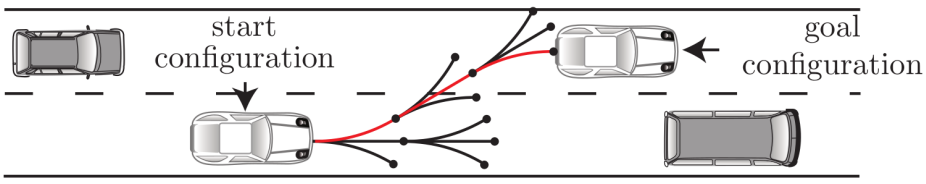

Figure 3: A search tree with motion primitives.

Within motion_automata/automata/MotionPlanner.py, we have already implemented a MotionPlanner class for your convenience. Some of its attributes are:

- **startLanelet_ids:** Ids of the lanelets on which the start position is.
- **goalLanelet_ids:** Ids of the lanelets on which the goal region is.
- **desired_time:** Time interval in which the ego vehicle has to arrive at the goal region.
- **desired_orientation:** Desired orientation interval at the goal region.
- **desired_velocity:** Desired velocity interval at the goal region.
- **priority_queue:** A priority queue used within the algorithm.
- **initial_distance:** Distance between initial state and the center of the goal region.
- **lanelet_cost:** The dictionary lanelet_cost returns a cost for each lanelet, which is equal to the number of number of lanelets to be traversed from each lanelet to reach the goal region. The function returns -1 if it is impossible to reach the goal region from the current lanelet. If the goal region is contained within current lanelet, the cost will be 0.

Besides these attributes, we have also provided some helper functions to solve for the trajectory. A detailed description of these helper functions can be found in the documentation of the class. For a quick familiarization with the available functions, we summarize them below:

- **distance:** takes two points in 2D plane as inputs, and calculates their distance according to a distance metric. We provide the following metrics: Euclidean, Manhattan and Chebyshev distances.

- **dist_to_closest_obstacle:** Returns the Euclidean distance to the closest obstacle in a lanelet. If there is no obstacle in the segment the function returns ∞.
- **check_collision:** To guarantee safety, it is important to ensure that the vehicle does not collide with other obstacles while following the trajectory. Therefore, a collision checker is provided which checks if a collision occurs for all states of the planned trajectory. Furthermore, the collision checker also investigates if the vehicle stays on the road network in the scenario.
- **reached_goal:** Returns true if all conditions of the goal region are satisfied.
- **remove_states_behind_goal:** Removes all states of a motion primitive that the vehicle would follow after reaching the goal region.
- **translate_primitive_to_current_pos:** Translates a motion primitive to current position. Note: All motion primitives start at position $[0, 0]^T$.
- **append_path:** Concatenates a motion primitive to current trajectory.
- **get_successor_primitives:** Returns all possible successive motion primitives of the given motion primitive.
- **is_goal_in_lane:** A Boolean function which returns true if the goal is reachable by recursively following the successive lanelets of the given lanelet.
- **calc_time_cost:** Returns the time needed to follow the motion primitive.
- **calc_path_efficiency:** Returns the ratio of traveled distance of the motion primitives to the time needed to follow such motion primitives.
- **calc_angle_to goal:** The orientation to the goal is calculated based on the current position and the center of the goal region.
- **curvature_of_polyline:** Returns the absolute sum of all local curvatures of a polyline. This function can be used to penalize non-smooth trajectories in order to derive a trajectory with higher comfort for the passengers.
- **orientation_diff:** Returns the difference of two given orientations. The result falls in the range of $[-\pi, \pi]$.
- **num_obstacles_in_lanelet_at_time_step:** Returns the number of obstacles in the given lanelet at given time step.
- **calc_heuristic_distance:** Summed absolute distance from every state along the motion primitive to the center line of the lanelet. Furthermore, the lanelet of the first and the last state are returned. This value can be used to penalize movements that have a large deviation from the center line of a lane.
- **calc_lanelet_orientation:** Returns the orientation of the lanelet at the specified position.
- **length_of_polyline:** Returns the length of a polyline.

Note: You do not have to use all of these attributes and functions. You are also free to come up with your own ideas and to combine them together to search for trajectories within a reasonable time.

For understanding the provided source code under motion_automata/automata, we suggest the following reading order:

1. States.py: a start state class and a final state class are defined for motion primitives.
2. MotionPrimitive.py: the MotionPrimitive class and some of its operations are defined.
3. MotionPrimitiveParser.py: a helper class to read in and parse the pre-generated motion primitive data is defined.
4. MotionAutomata.py: the MotionAutomata class which reads in and manipulates motion primitives is defined.

5. MotionPlanner(_*).py: definition of MotionPlanner class. Note that we have provided two example planners for your reference.

Finally, please refer to notebooks/tutorials/tutorial_commonroad-search.ipynb for a practical tutorial on how to execute the search.

4 Task & Evaluation

You should implement your own heuristic function and/or search algorithm. You are free to refer to, add and change the two already implemented motion planners. After implementing your own code, you should evaluate its performance on a series of scenarios. To pass the programming exercise, you should solve at least X planning problems of your choice. The exact number of X will be announced soon.

To select the scenario of your choice, please refer to CommonRoad Scenarios and its repository. The ranking of your solutions in the benchmark is irrelevant to your passing of the exercise. However, if you are ambitious enough, you might want to submit solutions with a higher ranking. A better search algorithm also helps you to get closer to the best student reward :)

5 Tips on Heuristic Function

Searching blindly in a high-dimensional graph can be very resource-consuming (e.g. time, memory, etc.), and having a well-thought-out heuristic function can dramatically decrease the amount of need for such resources. Generally, we want to reward states that lead us closer to the goal state while obeying some constraints. As the initial state of the planning problem and the goal region both could have state components including position, velocity, orientation, etc., we can try to formulate a heuristic function that penalizes large deviation to the goal state by considering:

- **positional distance:** the positional distance (in Euclidean, Manhattan or other distances) between the (x, y) position of a given state and goal region (e.g. center of goal region).
- **velocity difference:** the velocity difference between the velocity of a given state and goal state (e.g. center of the desired velocity interval).
- **orientation difference:** the orientation difference between the orientation of a given state and goal state (e.g. center of the desired orientation interval).

Besides these state components, one might also want to consider some other factors such as:

- **lanelet id:** we can retrieve the id of the lanelet on which the state is located. By this we can determine whether the examined state is located on the lanelet of the goal state, and reward such states.
- **obstacles on lanelet:** contrary to previous metric, if there are obstacles located on the lanelet of the goal state, we might want to make a lane change, thus penalizing such states.
- **trajectory efficiency:** we can calculate the ratio of the length of the trajectory travelled so far to the time required to travel such a trajectory.

More often than not, these calculated metrics are accompanied by different weights to collectively form the final heuristic function; however, we are mostly only able to make a guess of these weights and fine-tune them empirically. The ultimate heuristic function will have the following form:

$$h(state) = \sum_i weight_i \times metric_i$$

As a reference, you may find more explanation on planning algorithms in the book Planning Algorithms by LaValle.

6 Uploading Solutions

Your solutions should be uploaded to CommonRoad Benchmark. Prior to uploading the solutions, you should create an user account and log in with it. Instructions on how to upload the solutions are provided on the website. Only PUBLISHED solutions with SUCCESS status are considered submitted.

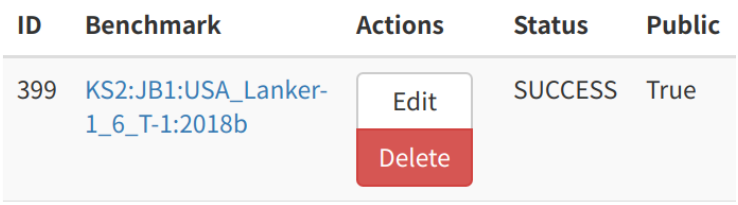

Figure 4: Feasible & published solution.

6.1 Benchmark ID

The full benchmark ID is composed of three parts. Let us introduce the IDs of the vehicle Model, the Cost function, the Scenario, and the Benchmark via letters M, C, S and B, respectively. The benchmark ID is constructed by separating partial IDs with colons and has the following form:

$$\text{B} = \text{M:C:S}$$

For instance, for M=KS2, C=JB1, S=OV001, the benchmark ID B will be KS2:JB1:OV001. The vehicle model is a differential equation representing the vehicle dynamics. For this exercise, the vehicle is modeled via a Kinematic Single-Track model.

6.2 Cost Function

The performance of your solutions are evaluated via cost functions. The general form of cost functions is depicted by

$$J_C(x(t), u(t), t_0, t_f) = \Phi_C(x(t_0), t_0, x(t_f), t_f) + \int_{t_0}^{t_f} L_C(x(t), u(t), t)\mathrm{d}t,$$

where Φ_C is the terminal cost and L_C is the running cost. Given a cost ID, the functions Φ_C and L_C can be uniquely determined. A list of possible cost functions is provided in Table.1 in Cost Function Documentation. As your solution will be evaluated based on the indicated cost function in the solution file, to achieve a higher rank, try using a similar cost function in your own planner.

6.3 Batch Processing

If you wish to find the solution to a batch of scenarios at once, you can make use of the batch processing script provided in notebooks/batch_processing/batch_processing.ipynb. While uploading the solutions, you can also select multiple solutions to submit them at once.

7 Deadline and Requirements

Final submission of the results will close on Sunday, 08.12.2019 at 23:59. Your submission should at least consist the following files:

1. Your own implementation of MotionPlanner.py (if you write your own motion planner, make sure they can be executed with least extra dependencies)
2. Screen-shot of your user profile on CommonRoad website indicating that you have successfully uploaded accepted solutions.

3. A folder containing all of your submitted solutions.

These items should be put directly under the submission folder, i.e., not be put under sub-folders or be compressed. If applicable, do not use absolute paths, but instead use relative paths to the working directory. If you have questions regarding this exercise, please raise them in the CommonRoad Forum, as other persons may have asked the same questions already.

Lastly, good luck with the exercise and hope you enjoy :)