Foundations of Artificial Intelligence

Programming exercise: *Search-based Motion Planning with Motion Primitives*

Edmond Irani Liu (M.Sc.), Prof. Matthias Althoff Last updated: November 14, 2020

1 Introduction

Finding safe motions in complex traffic situations 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 usually includes software for representing road networks, reasoning on the road networks, collision checking, simulating vehicle dynamics, and visualizing results, to name only a few aspects. The CommonRoad benchmark suite handles all of these. A standard CommonRoad scenario taken from the city center of Munich (Stachus) is shown in Fig.1. CommonRoad also provides a ranking system

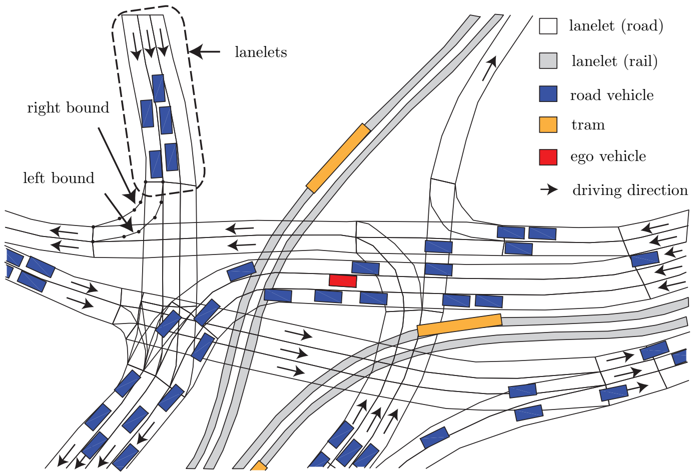


Figure 1: Visualization of Stachus in CommonRoad

to which one can upload his/her solutions to the motion planning problems, and see how well one's planner performs in comparison to others.

Your task in this exercise is to implement a heuristic function and/or a search algorithm with motion primitives to solve CommonRoad scenarios. The notion of motion primitives are explained in the following sections.

2 Installation

2.1 Regular Installation

If you are using Ubuntu 18.04, you can directly follow the instructions given in `README.md` if you wish to install the software on your own machine.

2.2 Virtual Machine

A Virtual Box image of Lubuntu 18.04 is provided, in which all necessary packages are installed (requires ca. 11.5 GB space after unzipping). It can be downloaded via this link. The downloading and the default login password are both commonroad. In the terminal, use the command `jupyter notebook` to launch the Jupyter notebook, and navigate to `/Desktop/commonroad-search/`.

If necessary, you can adjust the resolution by setting **Preferences >Monitor settings >Resolution**.

2.3 Docker

A docker file and a prebuilt docker image for this exercise is available. Refer to docker/README.md for more information.

3 CommonRoad Scenarios

3.1 Scenario

Before implementing the search algorithms, it is essential to correctly understand the structure of CommonRoad scenario files. The most important elements in a Scenario are:

- **Lanelet 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 contains both static and dynamic obstacles. Static obstacles do not move over time, such as construction zones and parked vehicles; dynamic obstacles are mostly traffic participants such as passenger cars, trucks, motorbikes, cyclists, pedestrians, etc. Both of these obstacles are modeled via geometric shapes. In addition, the movement of a dynamic obstacle over time is captured by a trajectory, which is essentially a list of states over time.

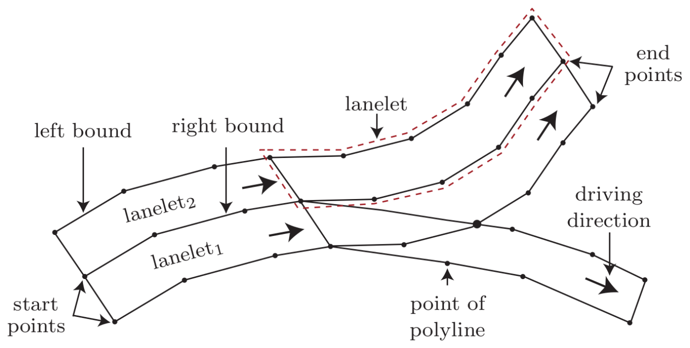


Figure 2: Example of a lanelet.

3.2 Planning Problem

The motion planning problems are described with PlanningProblem objects, which have the following attributes:

- **Initial state:** The initial state of the ego vehicle in the planning problem. Some of its attributes are:
  - **Position:** A two-dimensional vector $[x, y]^{\mathrm{T}}$ 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:** One or multiple goal states which should be reached by the ego vehicle. Some of its attributes are:
  - **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 arrival at the goal region in [time step].

Note that all time stamps in CommonRoad are represented by time steps $k \in \mathbb{N}^+$. 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 step $k = 61$ corresponds to time $t = 6.1$s, given that r is set to 0.1s.

Based on the types of the goal states, the planning problems (and the scenarios) can be categorized into:

- **Regular**: goal states contain explicit requirement on positions, velocities, orientations, time steps
- **Survival**: goal states only contain requirement on time steps

In survival scenarios, you only need to find a solution trajectory that does not collide with other traffic participants up to the indicated time intervals.

3.3 Solution

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

- It starts at the initial state of the planning problem.
- It reaches one of the goal states of the planning problem.
- The trajectory complies with a given vehicle model. In this tutorial, we assume that the Kinematic Single-Track model is used. See Vehicle Models for more details.
- A vehicle of the specified vehicle type does not collide with any obstacle and stays on the road while perfectly following this trajectory.

4 Software Description

4.1 Code Structure

The provided source code contains the following folders:

- SMP/:
  - maneuver_automaton/: contains code related to motion primitives and maneuver automata.
  - motion_planner/: contains code related to motion planners.
  - batch_processing/: contains code related to batch processing for parallel execution.
  - route_planner/: contains code related to route planner.
- tutorials/: contains essential tutorials for this programming exercise.
- scenarios/: contains scenarios to be solved by the motion planners.
- outputs/: folder to hold the outputs (solutions of motion planners, log files, GIF animations, etc.).

4.2 Motion Primitives

Motion primitives are short trajectory segments that can be driven by a given vehicle model. By concatenating the primitives, a drivable trajectory can be constructed that leads the vehicle from the initial state to the goal state. Fig.3 shows an exemplary search tree that is spanned via motion primitives, where the initial and final states of primitives are indicated by dots. The path between two successive dots are the intermediate states of a primitive.

In this exercise, all of the generated primitives start at position $[x = 0, y = 0]^\mathrm{T}$, have a duration of 0.5 seconds and differ in velocity, steering angle, orientation, acceleration, traveled distance, etc. If the velocity and the steering angle of the *initial* state of a *successor* primitive are equal to those of the *final* state of a *predecessor* primitive, we say that these two primitives are *connectable*.

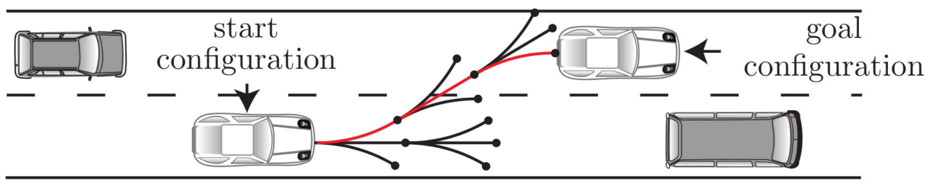


Figure 3: A search tree with motion primitives.

4.3 Motion Planners

For the better understanding of the basic search algorithms, 6 (un)informed algorithms that search for a trajectory using motion primitives have been implemented, namely:

1. Breadth First Search
2. Depth First Search
3. Depth-limited Search
4. Uniform Cost Search (aka Dijkstra's algorithm)
5. Greedy Best First Search
6. A* Search

Their implementations can be found in SMP/motion_planner/search_algorithms/. A base class SearchBaseClass is created to capture the common characteristics of these planners. Then, based on the way they are implemented (sequential search, best first search, etc.), different children classes are derived.

4.4 Helper Functions

For your reference, we provide a set of helper functions for heuristic cost computation within the base class SearchBaseClass. You do not have to use all of the provided functions, and of course you are free to come up with your own ideas.

4.5 Tutorials

The provided tutorials for this exercise go as follows:

- 0_commonroad_input-output: this is the first tutorial on CommonRoad Input-Output, and shows how CommonRoad XML-files can be read, modified, visualized, and saved.
- 1_search_algorithms: this exemplifies how one can combine motion primitives with (un)informed search algorithms to find solutions to simple motion planning problems.
- 2_commonroad_search: this demonstrates how one can use the (un)informed search algorithms, combined with motion primitives, to solve real motion planning problems introduced in CommonRoad.
- 3_motion_primitive_generator: this helps you understand how are the motion primitives generated, in case you want to generate your own set of motion primitives.
- 4_batch_processing: this shows how the search for solutions can be boosted via parallel computation with multi-threading using the given batch processing script.
- 5_route_planner: this explains how the route planner package can be used to plan high-level routes for CommonRoad scenarios to better guide the search tree.

4.6 Getting Started

To get started, we suggest going through all of the tutorials mentioned above. After that, you may look into the source code for implementation details in the following order:

1. maneuver_automaton/motion_primitive.py
2. maneuver_automaton/maneuver_automaton.py
3. motion_planner/motion_planner.py
4. motion_planner/search_algorithms/

5 Task & Evaluation

As mentioned above, your task is to implement a heuristic function and/or a search algorithm with motion primitives to solve CommonRoad scenarios. You are free to refer to and alter the provided motion planners. You should evaluate the performance of your planner against the 500 scenarios under the folder scenarios/exercise/.

To pass this exercise, you should solve at least 340 out of the 500 given scenarios. Don't be scared, roughly 320 scenarios are already solvable with the provided out-of-the-box planners :). Note that even if you produce more solutions to a scenario, it will still be considered as solving one scenario only. The ranking of your solutions in the leader board is irrelevant to the passing of the exercise.

6 Tips on Heuristic Function

Searching blindly in a high-dimensional state space can be very resource-consuming (time, memory, computing power, etc.); on the other hand, having a well-thought-out heuristic function can dramatically reduce the need for such resources. In general, we want to reward states that lead us closer to the goal state while obeying some constraints. The goal region of the planning problem mostly consists of state components including position, velocity, orientation, etc., we can therefore 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 the goal region (e.g. center of the goal region).
- **velocity difference:** the velocity difference between the velocity of a given state and the goal state (e.g. center of the desired velocity interval).
- **orientation difference:** the orientation difference between the orientation of a given state and the goal state (e.g. center of the desired orientation interval).
- **time difference:** the time difference between the time step of a given state and the goal state (e.g. center of the desired time step 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 the 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 traveled so far to the time required to travel such a trajectory as trajectory efficiency, and reward those trajectories with higher efficiencies.

Typically, these metrics are accompanied by different weights to collectively form the final heuristic function. The optimal weights are hard to determine, thus we mostly have to make an initial guess and fine-tune them empirically. The ultimate heuristic function typically have the following form:

$$h(\text{state}) = \sum_{\text{i}} \text{weight}_{\text{i}} \times \text{metric}_{\text{i}}(\text{state})$$

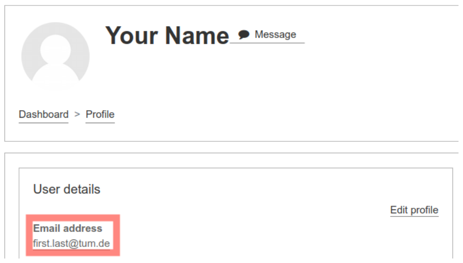
As a reference, you may find more explanation on planning algorithms in the book Planning Algorithms by Steven M. LaValle.

Additionally, a route planner package has been provided (see tutorial 5), which is able to generate feasible routes (and associated reference paths) that lead the initial lanelet(s) to the goal lanelet(s). One may incorporate additional information extracted from these routes and reference paths into the search algorithms to guide the search tree towards the goal.

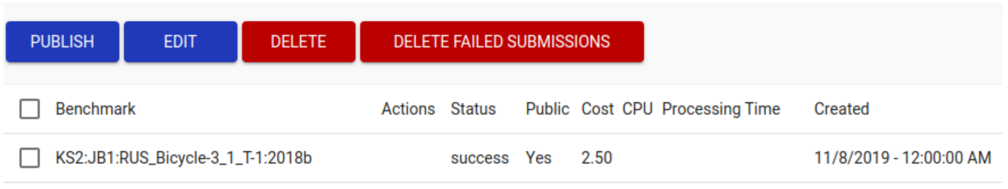
7 Submitting Solutions

Your final solutions should be uploaded to the CommonRoad Benchmark. Here are the steps of dong so:

1. Register here with the **same email address as shown on your Moodle** profile (otherwise we cannot assign the account to your student ID and you cannot enroll in the step 2.)


Your Name
Message
Dashboard > Profile
User details
Edit profile
Email address
first.last@tum.de

2. Enroll in the programming exercise here.
3. Upload your solution xml files under Username/New Submission. Wait for all submissions to be evaluated and verify their evaluation status on Username/My Submissions.
4. Select all submissions and publish them. Only submissions with status Success and Public will be counted!



☐	Benchmark	Actions	Status	Public	Cost	CPU Processing Time	Created
☐	KS2:JB1:RUS_Bicycle-3_1_T-1:2018b		success	Yes	2.50		11/8/2019 - 12:00:00 AM

5. Check your rank in the course leader board under commonroad.in.tum.de/challenges.

Note: You still have to submit your planner and other required files as explained in Sec. 8

7.1 Benchmark ID

The full benchmark ID is composed of three parts. Let us denote the IDs of the vehicle model, the cost function, the scenario, the version of the scenario, and the benchmark as letters M, C, S, V, and B, respectively. The benchmark ID is constructed by concatenating partial IDs with colons, which raises the following form:

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

For instance, for M = KS2, C = SA1, S = USA_Lanker-1_2_T-1, V = 2020a, the benchmark ID will be KS2:SA1:USA_Lanker-1_2_T-1:2020a.

7.2 Cost Function

After submitting the solutions to the CommonRoad website, they are evaluated via cost functions specified in the benchmark ID. The lower the cost, the higher the rank. The general form of cost functions is

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

where x is the state, u the control input, t the time, Φ_C the terminal cost and L_C the running cost. Given a cost function ID, the functions Φ_C and L_C are uniquely determined. A list of possible cost functions is given in Table.1 in Cost Function Documentation. Please DO NOT use cost function JB1 in this exercise as it only considers the time duration of the trajectory, which in the end will result in having the same costs for very different trajectories.

8 Deadline and Requirements

You should submit your work on Moodle before 18.01.2021. Your submission should contain the following files:

1. Your own implementation of search_algorithms/student.py. It should be callable with the Student-MotionPlanner method in motion_planner.py and should generate your solution files. If you make use of other packages, make sure to also provide a requirements.txt file. We should be able to run your code on the virtual machine image (see Sec. 2.2) by installing the requirements and copy-pasting your student.py and other files (if any).
2. A compressed file containing all of the valid solutions which you have submitted to the CommonRoad website. (make sure to provide the compressed file in .zip format)
3. Your configuration file for batch processing batch_processing_config.yaml.
4. Your motion primitive XML files, if you have created your own set of motion primitives.

These items should be put directly under the submission folder on Moodle, i.e., not be put under sub-folders or be zipped further. If applicable, do not use absolute paths in your code, 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 students may share the same question.

9 Beyond the Limits

If you are ambitious enough, we have a special challenge for you: you may solve as many scenarios as you want from all of the currently available CommonRoad scenarios, which can be accessed via the public repository. A scenario selection page is available on the CommonRoad website. We have the following rewards for students with the most solved scenarios:

1. **Top 10**: invitation to a Pizza Night gathering
2. **Top 3**: special certificate stating your outstanding performance
3. **Top 1**: €200 cash

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