MULTIVARIABLE CONTROL AND COORDINATION SYSTEMS (EE-477)

Case study:

Path tracking for an automated vehicle.

`http://react.epfl.ch`

Denis Gillet, Ezequiel Debada
Lausanne, EPFL, 2018

Exercise 3 Dynamic programming

In this exercise, we tackle the trajectory planning problem through a dynamic-programming-based approach. Trajectory planning refers to the task of planning, in a certain temporary horizon. So, the obtained sequence of control inputs are applied to reach a certain control objective in an optimal way. This is an important problem for autonomous vehicles because they must make control decisions considering specific navigation requirements such as obstacle avoidance and lane driving.

In particular, the problem is often solved in four steps as follows:

1. a certain cost function is defined related to the control objectives,
2. the current state of the vehicle and the range of valid control actions are used to predict the feasible trajectories which are supposed to be performed by the vehicle,
3. each trajectory is evaluated according to the cost function,
4. the best trajectory is selected out of the obtained pool of trajectories to be taken into account.

In this exercise, we address a simplified version of the trajectory planning problem, and we obtain its solution, say, an optimal trajectory, using dynamic programming method. According to this simplified version, we assume that the first control input u_1 is fixed, while the second control input $u_2 \in \Omega = \{u_{2,1}, u_{2,2}, \cdots, u_{2,n}\}$ can take a finite set of values and change every h seconds.

Regarding these assumptions, we build a graph representing the possible trajectories which can be followed by the vehicle. For example for a certain node i representing the state x_i, we create n new nodes $\{x_k, x_{k+1}, \cdots, x_{k+n-1}\}$ where the new states is computed as $x_{k+l} = x_i + \int_0^h f(x, u_1, u_{2,l+1}, t)\mathrm{d}t$ (with f representing the non-linear model of the system). Once the graph is completed, we assign a specific cost to each node according to the cost function. Then, we traverse the graph in a backward manner to find the optimal trajectory.

The provided script and functions create the system graph and compute the state of the system at every node. For simplicity, a randomized heuristic is used to shape the graph in such a way that from each node, only some control inputs in Ω are taken into account to create the new set of nodes. As already noted, you don't need to explore this feature of the script, however note that this heuristic parameter underlies obtaining different results in the course of consecutive executions of the script.

Graph representation We formally define the graph, then the cost assignment to nodes and the backtracking procedure to obtain the optimal trajectory are explained.

The graph $G = (N, E)$ represents the possible trajectories which can be followed by the vehicle. In particular, the set of nodes N indicates different states which can reached from the vehicle's current position. The set of edges E describes how the nodes are connected. In the code, nodes and edges are represented by arrays `nodes`$\in \mathbb{R}^{n\times 11}$ (here n is the number of the

nodes in the graph) and $\texttt{edges} \in \mathbb{R}^{m\times 4}$. Each row $\texttt{node}_i$ of the `nodes` array is built as

$$\texttt{node}_i = (i, t_i, \gamma_i, p_{xi}, p_{yi}, \theta_i, x_1, x_2, x_3, x_4, x_5), \tag{3.1}$$

that is, it contains the node id i, the time t_i corresponding to the node, a flag γ_i, the position of the vehicle $(p_{xi}, p_{yi}, \theta_i)$, and its state $(x_1, x_2, x_3, x_4, x_5)$. The flag γ_i is used to indicate whether the node has not been visited yet ($\gamma_i = 0$), has been visited and is a valid node ($\gamma_i = 1$), or has been visited and is a dead end ($\gamma_i = 2$); and it is used during the graph creation process (see Fig. 3.1). A node is said to be visited when it has some successor node. Otherwise, the node could be a dead-end (indicating that it violates the imposed constraints regarding lateral deviation form the path) or a non-visited node, which corresponds to the nodes at the tip of the branches of the graph.

On the other hand, every edge $\texttt{edge}_i$ of the `edges` array in the form of

$$\texttt{edge}_i = (j, k, u_1^{j\to k}, u_2^{j\to k}) \tag{3.2}$$

contains the ids of the nodes it connects (j and k) in addition to the control input used during such a transition. It is worth noting that you will be given both the `nodes` and `edges` arrays

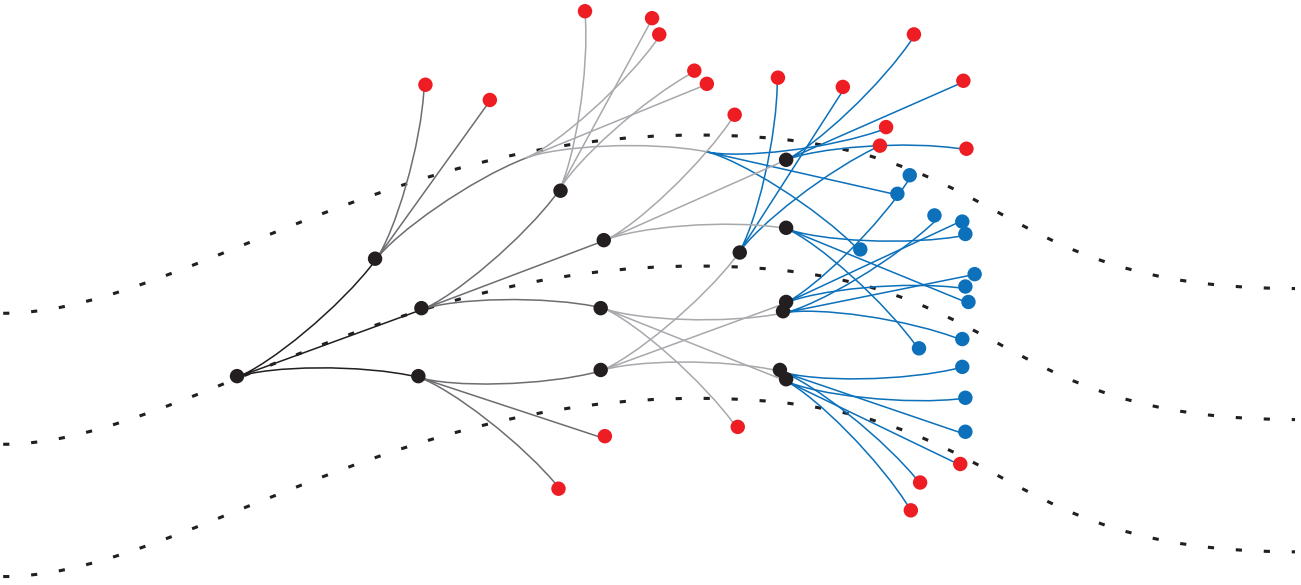

Figure 3.1: Example graph. Visited nodes in black, dead end nodes in red, and non-visited nodes in blue.

and you will not be involved in calculating their elements.

Obstacles To make the problem slightly more interesting, we consider not only the path tracking objective but also the avoidance of collisions with a certain set of static obstacles $\mathcal{O} = \{o_1, o_2, \cdots, o_{n_{obs}}\}$, where each $o_i \in \mathbb{R}^2$ contains the position information of the obstacle i.

Node costs In this exercise, we calculate the cost c_k of a node k as

$$c_k = c_j + c_{k,\text{path tracking}} + c_{k,\text{collision avoidance}} + c_{k,\text{dead end}} \tag{3.3}$$

where

- c_j is the cost of the predecessor node,
- $c_{k,\text{path-tracking}} = w_{\text{track-d}} x_2 + w_{\text{track-heading}} x_3$ is a cost term associated with the path tracking error,
- $c_{k,\text{collision-avoidance}} = \sum_i^{n_{obs}} \frac{w_{collision-avoidance}}{dist(x_k, o_i)^3}$ is a cost term associated with the proximity of a node to some specific obstacles,
- $c_{k,\text{dead-end}} = \infty$ if $\gamma_k = 2$ and $c_{k,\text{dead-end}} = 0$ otherwise. This cost term prunes dead-end nodes of the solution.

Backtracking Once the graph is built and the cost of each node is assigned, the optimal trajectory is backtracked. In particular, we only need to identify the set of terminal nodes $\mathcal{N}_T = \{i : \gamma_k = 0\}$. Then, we select the one with the smallest cost $i^* = \arg\min\{c_i : i \in \mathcal{N}_T\}$. We later use the information in `edges` to explore the graph from i^* in a backward manner by keeping track of the visited nodes, which will define the optimal trajectory.

3.1 Provided files

- `ex3.m`. The class whose methods have to be completed. A brief description of its functions is provided within this document, whereas more detailed information regarding inputs and outputs can be found in the matlab file itself.
- `exercise3_DynamicProgramming.m`. Matlab script that sets up and runs the required simulations. Even though, you are not asked to modify this script, understanding what the file does can be helpful to solve the problem.
- `utilities.m`. The class which includes auxiliary functions. This file is updated w.r.t the one provided in previous exercises, and some bugs are fixed.
- `circle, path_1, path_2, path_3`. mat files contain different paths to be tracked which can be used in the experiments.

3.2 Exercises

- Copy your solutions of the methods `getInitialState`, `getSystemParameters`, `getAllowedControlValues`, and `select_reference_path` from previous exercises.
- Complete the method `getAllowedControlValues` which returns an array including all possible combinations of control inputs that can be applied to the system. We assume that $u_1 = 3$ and $u_2 \in \{-\pi/4, 0, \pi/4\}$.
- Complete the method `getObstacle` which specifies the positions of the obstacles to be considered. You can start by positioning the obstacle(s) in an arbitrary location, and once you visualize the path to be tracked, you can refine the position of the obstacle so that it is on the vehicle's way.
- Complete the method `assignCostsToNodes` which explores the graph in a forward fashion from the initial node. This method assigns costs to the nodes. You can freely choose the values of the weights corresponding to the cost function, but the values you report must

result in collision-free trajectories. If you cannot find such a set of parameters, explain why.

- Complete the method `backtrackNodeValues` which explores the graph in a backward manner to collect the sequence of the nodes which define the optimal trajectory.
- Repeat the exercise for different sets of obstacles (changing the number of obstacles and their position). Is the algorithm always able to find a collision-free path? Why?
- Why is the graph creation process randomized from a certain point on?
- If you obtained unexpected results, discuss them.