**Implement a basic driving agent**

Implement the basic driving agent, which processes the following inputs at each time step:

* Next waypoint location, relative to its current location and heading,
* Intersection state (traffic light and presence of cars), and,
* Current deadline value (time steps remaining),

And produces some random move/action (None, 'forward', 'left', 'right'). Don’t try to implement the correct strategy! That’s exactly what your agent is supposed to learn.

Run this agent within the simulation environment with enforce\_deadline set to False (see run function inagent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

*In your report, mention what you see in the agent’s behavior. Does it eventually make it to the target location?*

The implemented agent chooses randomly to perform a move/action not considering waypoint location, intersection state or traffic. Although the move/action is 100% random, the agent eventually gets to destination but it’s very rare that it arrives on deadline. In the first 5 trial the agent arrived to destination in 200+ time steps but on the 6th it arrived on time.

Another thing that caught my eye was that this random agent tends to get stuck in intersections because it is not considering the environment to take an action. So for example it may choose to move right on North-South traffic or just choose the ‘None’ action when there could be a reasonable action to do. In this sense, I made the Agent choose a random action based the possible action it can take given the environment (considering both the traffic lights and the oncoming traffic). For example, it can not choose to go forward on red lights.

Although this new agent gets to destination at a much lower number of time steps, it still doesn’t get to destination on deadline.

**Identify and update state**

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

*Justify why you picked these set of states, and how they model the agent and its environment.*

**Implement Q-Learning**

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that.

Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

*What changes do you notice in the agent’s behavior?*

**Enhance the driving agent**

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

*Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?*

*Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?*