3 Computer Lab: Reinforcement Learning

This week you will be implementing a reinforcement learning agent and applying it to the standard testbed "Mountain Car". The goal is to draw a heat-map of the two dimensional value function. The exact RL learning algorithm you use to compute it doesn't matter that much, but the result does.

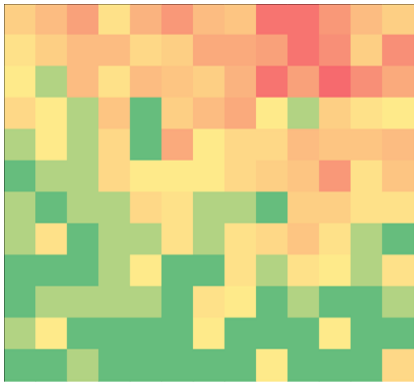
Mountain Car is a simplification of simulated car on a hill that is too steep to drive onto without any initial speed. The task was an early favourite benchmark in reinforcement learning because the optimal solution requires that the car first drives away from the goal, misleading algorithms such as A*, at least initially. The state of the car is two dimensional, consisting of its position and speed. The position ranges from -1.2 to 0.6, although episodes end as soon as the car reaches a position greater than 0.5. The speed of the car ranges from -0.07 to 0.07. The actions that the car can execute are full throttle ahead, full throttle reversed or no throttle at all. Full throttle increases or decreases the speed by 0.001. The hill is sine-shaped and gravity pulls with a 0.0025 force. There is no friction caused by tires or air. The exact computation can be found in the example code supplied. The rather depressing reward is defined as -1 for each time step until the episode ends. Hence, the goal is to leave as soon as possible.

There are two options readily available for you to interface with this environment. The first and recommended choice is through OpenAI Gym (https://gym.openai.com). This is a Python interface for reinforcement learning environments, including the old benchmarks such as Mountain Car, but also more interesting ones such as the Arcade Learning Environment. You can find how to install gym on your Mac OS or Linux system at https://gym.openai.com/docs. On EleUM, I provide some example Python code that generates interaction episodes using random action selection.

The second option is to interface with the Java version of the environment implemented by yours truly. The source of this code is also available on EleUM. It should not be treated as a good example of Java coding, but it seemed to pass most stress tests last year. It does seem to run faster than the Gym version if you turn off the graphical display on both.

The third option is to re-implement the environment yourself in your language of choice: Matlab, R, prolog, ... Without graphics, this is really easy.

Then you need to decide on a reinforcement learning algorithm. Although the actions are discrete, the state space is in principle continuous. This means you can either decide to work with this as is and use some kind of generalisation technique, or discretise the state space. Your free to make your own choice for an algorithm, but keep in mind that you need to visualise the state-value function at the end.



Handing In

Your submission should include your code and a README to allow me to test it. Besides that report on your choice of reinforcement learning algorithm and how easy it was to (i) implement and (ii) get it to learn something useful. Explain how you dealt with the continuous state space. Explain how you constructed the visualisation of the state-value function. Explain which parameters you tuned to get the best visualisation.[1] As last week, the deadline falls before the last set of lectures.

[1] Now there is a hidden hint if there ever was one!