Assignment 6

Due: 28. January 2022, 10:15 CET

General information: This assignment should be completed in groups of **three to four people**. Submit your assignment via the corresponding LernraumPlus activity. You can only upload one file, therefore make sure to place all your files required to run/evaluate your solution into **one archive** and just upload that archive.

Exercise information: In this assignment you will be implementing functions required for decision making, primarily focused on Markov Decision Processes (MDPs). For this assignment, I have provided quite a lot of boilerplate code for you. For the 2nd exercise you find a simple decision tree implementation as well as simple node classes for the different types of nodes which you should complete. For exercise 3, the *assignment6.zip* contains a package called *mdp* which contains files required for the MDP exercise instead of the graph reference implementation alongside the skeleton and example-test files. The package contains an implementation of a transition model for the agent we intend to train, a simple 2d gridworld implementation that serves as the environment the agent is acting in, as well as, a module that handles simple visualization of the environment. The skeleton file further includes the MDP class itself where you need to complete only 2 functions. The other functions are there to provide useful utilities, e.g. visualization to you.

Pay attention to the provided docstrings as they further explain what is expected from the different functions. **You need to ensure** that your solution can directly be **imported as a Python3 module** for automatic tests and at least contains the name of the skeleton file. The easiest way to achieve this, is to directly develop your solution in the assignment6.py file and submit that. You are free to use **numpy** and **matplotlib** (in case you want to do your own visualizations) as third party libraries.

Exercise 1: (5 Points)

Consider the following lotteries:

$$L_1 = [0.3, A; 0.7, B], U(A) = 200, U(B) = 30$$
$$L_2 = [0.8, A; 0.2, B], U(A) = -75, U(B) = 200$$
$$L_3 = [0.3, A; 0.2, B; 0.5, C], U(A) = 100, U(B) = -150, U(C) = 150$$
$$L_4 = [p, A; (1-p), B], U(A) = 115, U(B) = 65$$

Task 1: (3 Points) Rank the lotteries L_1, L_2 and L_3 according to their expected utility.

Task 2: (2 Points) Find the lowest value for p so that L_4 should be at least indifferent, if not preferred to all other lotteries.

You are free to do this manually or by implementing suitable representations for lotteries. If you do it manually, make sure to report the steps you took to reach your results and report the EUs for each of the lotteries.

Exercise 2: (5 Points)

Decision Trees are a prominent and powerful way of modeling and solving (simple) decision problems. These are tree-like graphs with three different types of nodes. In the *assignment6.py* you can find code for the three different kinds of tree nodes: ActionNodes, ChanceCodes and UtilityNodes. Furthermore, you can find an implementation of a simple DecisionTree class that already implements the *backward induction* algorithm in order to determine the best possible actions for all ActionNodes. However, it requires that the expected utilities (EUs) of all its nodes can be computed.

Therefore, you should implement:

Task 1: (0.5 Points) The *get_eu* function for the UtilityNode class.

Task 2: (1.5 Points) The *get_eu* function for the ChanceNode class.

Task 3: (3 Points) The *get_eu* function for the ActionNode class. This function should also store the action with the highest EU in the *best_action* attribute of the ActionNode class as the backward induction algorithm will make use of that.

You can assume that utility nodes are always leaves in the tree and make up the sum of all intermediate rewards. You can also find example decision trees in the skeleton file.

Exercise 3: (10 Points)

The final exercise deals with Markov Decision Processes, arguably the most common method to solve complex, multi-action decision problems as many real world scenarios can be expressed as (partially observable) Markov Decision Processes.

All the provided code for this exercise can be found in the skeleton file and the *mdp* module.

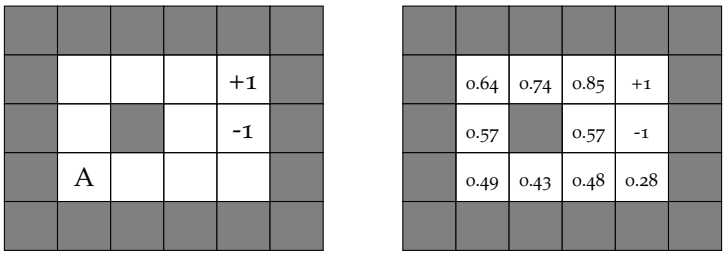

Figure 1: Example grid world (left) and corresponding optimal utility values (right). Your values may vary slightly depending on the machine you run it on.

Figure 1 (left) shows the simple grid world from the lecture. The world consists of walls and obstacles (dark gray), and several final states (+1, -1). Final states yield positive or negative rewards according to the values given. In the world, there is a tiny agent (A) that would like to move to one of the final states, of course preferring the highest rewards. At any given position, the agent can choose among four actions: moving north, east, south, or west.

Unfortunately, sometimes the agent gets confused and moves into a direction that differs from the direction it decided to move in. For instance, in the given example in Fig. 1, when the agent decides to move north, there is only a chance of 0.8 that it actually moves north and a chance of 0.1 that it moves east and a chance of 0.1 that it moves west. When the agent bumps into a wall or obstacle, it stays at the current state. The question is how the agent should decide to move under such uncertain

conditions in order to maximize the expected outcome.

As presented in the lecture, such a decision problem can be described as *Markov Decision Process* (S, A, T, R, γ), where S is a set of states, A a set of possible actions, T a transition probability for reaching state s' from state s by choosing action a, R a reward function for transitions from a state s to a state s', and γ a discount factor that is used to weight future rewards. By solving the *Markov Decision Process*, a policy can be obtained that provides the optimal action sequence for any given state of the world the agent might encounter.

One approach to solve a *Markov Decision Process* is to use *Value Iteration*. The basic idea is to compute the value (expected reward) of step $t+1$ by using an estimate of the value of step t:

$$U_{t+1}(s) := R(s) + \gamma max_a \sum_{s'} T(s, a, s') U_t(s')$$

After iteratively solving this so called *Bellman equation*, the expected values can be used to compute an optimal policy for the decision problem represented by the *Markov Decision Process*.

Task 1: (6 Points) Implement the *value_iteration* method in the *Markov Decision Process* class in assignment6.py to solve the MDP using *Value Iteration*. For this exercise, you can ignore the ϵ parameter of the signature.

Task 2: (1 Point) Extend your algorithm from Task 1 so that it automatically stops after convergence, i.e., when the expected value of a state changes less than a predefined threshold ϵ: $max_s(|U_{t+1}(s) - U_t(s)|) < \epsilon$.

Task 3: (3 Points) Implement the *get_policy* method of the provided MDP class, in order to compute an optimal policy based on provided expected utility values (ideally computed by Tasks 1/2).