COMP4620/8620: Assignment 1

Alban Grastien

The purpose of this assignment is to verify your understanding of Markov Decision Processes and their basic search algorithms.

-1 Installation

This assignment is supposed to be completed in `python3` together with the `scikit-decide` library, and `gym` that it depends on. You will therefore need to install these on your machine.

On Windows for instance, you can install anaconda following the instructions on Anaconda documentation website[1]. Then use the following commands:

```
conda create -n comp4620 python=3.7
conda activate comp4620
pip3 install gym
pip3 install scikit-decide
```

Don't forget: if you use the procedure above, you will need to type `conda activate comp4620` every time you open a new terminal before you can run the code.

The documentation for `scikit-decide` is available here:

`https://airbus.github.io/scikit-decide/reference/`

0 Description of the Files Provided

0.1 interface.py

This file provides an interface with `scikit-decide` and does not need to be modified.

0.2 statemachine.py and example1.py

The file `statemachine.py` contains an implementation that allows to interface a finite state machine with the `scikit-decide` framework. Generating an MDP requires to provide the list of transitions as well as the initial state.

The file `example1.py` gives an example. For instance, the line

```
SMTransition("Fork2" , "DoNothing", 0, [("NoFork", 0.1), ("Fork2", 0.9)]) ,
```

indicates that performing action `DoNothing` from state `Fork2` will lead to state `NoFork` with probability 0.1 and to state `Fork2` with probability 0.9; whichever state is reached, the cost of the transition is 0. Make sure that the sum of probabilities adds up to 1. In the example of this file, the initial state is `NoFork`.

These files do not need to be modified.

[1] `https://docs.anaconda.com/anaconda/install/windows/`

0.3 policy.py and value.py

`policy.py` contains a class to manipulate policies; this file does not need to be modified.
`value.py` contains a class to manipulate value functions.

0.4 vi.py

`vi.py` implements the Value-Iteration algorithm on top of `value.py`.

0.5 pi.py

`pi.py` implements the Policy-Iteration algorithm on top of `value.py` and `policy.py`.

0.6 rtdp.py

`rtdp.py` implements the RTDP algorithm on top of `value.py`.

1 VI, PI, and RTDP (10 points)

In this exercise, you will show your understanding of the basic Markov Decision Process algorithms. You are supposed to modify all methods that contain the `TODO` keyword in its code (instead of the real code, there is usually a dummy value). We now describe these methods.

Each method is given a test file, that can be used to verify the correctness of the method. For instance, you can test your implementation of method `q_value` by running the command `python3 test_11.py`. That a test passes **does not** imply that the implementation is correct, only that it seems correct.

You can also test your implementations by running the main files of `vi.py`, `pi.py`, and `rtdp.py`.

1.1 Method `q_value` from `value.py`

This method computes the Q-value of the specified pair (state,action). The Q-value of an action is the expected (discounted) cost of applying the action under the assumption that the value function of the reached state is accurate. Test this method with file `test_11.py`.

1.2 Method `compute_single_policy_backup` from `value.py`

This method computes and returns a new ValueFunction (it shouldn't modify the current function) that associates each state s with the Q-value of (s, a) where a is the action for s according to the Policy given as input. If a state is terminal, the value of this state should be zero. It also returns the error that resulted from the backup (i.e., the maximum difference between the original value function and the new value function). Test this method with file `test_12.py`.

1.3 Method `compute_value_function_of_policy` from `value.py`

This method computes and returns an estimate of the ValueFunction associated with the specified policy. The implementation of this method should use `compute_single_policy_backup`. Test this method with file `test_13.py`.

1.4 Method `compute_bellmann_backup` from `value.py`

This method computes and returns the ValueFunction resulting of a Bellmann backup, as well as the Bellmann error. If a state is terminal, the value of this state should be zero. Test this method with file `test_14.py`.

1.5 Method `greedy_action` from `value.py`

This method computes the best action that should be performed in the specified state according to the current value function, together with the Q-value of this action. Test this method with file `test_15.py`.

1.6 Method `backup` from `vi.py`

This method performs the value iteration (VI) algorithm using the `single_backup` method provided in the code. Test this method with file `test_16.py`.

1.7 Method `refine_policy` from `pi.py`

This method updates `_policy` in the class Policy Iteration by performing one iteration of the policy iteration (PI) algorithm. Test this method with file `test_17.py`.

1.8 Method `one_step_simulate_and_update` from `rtdp.py`

This method performs one step in the simulation (picks the greedy action, simulates it, backs up the state). Test this method with file `test_18.py`.

2 Understanding the Weaknesses of VI (5 points)

In this exercise, you will show your understanding of the weaknesses of Value Iteration. Edit the file `student_example1.py` in order to provide a Markov Decision Process that Value Iteration will find hard to solve, despite the small number of states.

Finish the implementation of the method `hard_instance()` while following the following constraints:

1. The state machine contains six states, named s1 to s6. The initial state is s1. Only one action, a1, is available in s6; this action costs 0 and loops on state s6.
2. The cost of any action is always in the interval [0,1].
3. The probability of a transition from one state to another is either 0 or at least 0.1.

The object produced by the method is a State Machine defined in file `statemachine.py` and described in subsection 0.2.

The exercise is considered to be successful if VI requires more than $1,000$ iterations to finish with an ε value of .5 and a γ value of 1. In practice, it is possible to design examples that require over $50,000$ iterations.

Test this method with file `test_21.py`.

3 Modelling (5 points)

In this exercise, you will show your understanding of how to model a Markov Decision Process. Edit the file `student_example2.py` in order to output a Markov Decision Process that represents the system described below. **The test files `test_3*` give examples of the correct results of your methods, in case the text below is ambiguous or incorrect.**

Use one of the solvers developed in Section 1 in order to learn a policy for this system. (Using VI may be slow, so do not be surprised by the execution time.) Answer the question at the end of the file `student_example2.py` that asks you to explain the behaviour of solver.

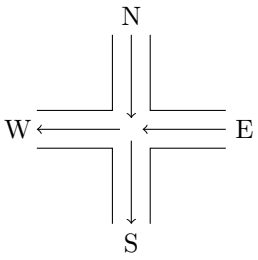

Figure 1: Simplified traffic light

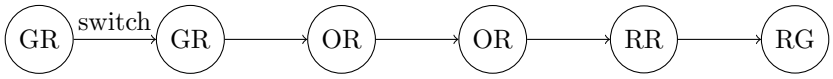

Figure 2: Sequence consecutive to traffic light switch: each state represents the colour of both lights, where G stands for green, O for orange, and R for red.

3.1 Description of the System

The goal is to optimise a (highly simplified) traffic light intersection (Figure 1). We assume that cars can move in only two directions:

direction 1 North to South,

direction 2 East to West.

In the first direction, cars arrive at a rate of τ_1: this means that at every time step, the chance that a new car is trying to cross the intersection is τ_1; the rate of arrival in the second direction is τ_2.

When the traffic lights do not allow passage in one direction, all cars that want to travel in this direction queue, and the queue increases as new cars arrive. The queue is limited to `max_queue` vehicles; when a new vehicle arrives, it takes a different path.

When the traffic lights allow passage in one direction, one of the queueing cars is allowed to cross (and disappears from the queue).

In general, the traffic lights allow passage in exactly one direction, sometimes zero. If the traffic light is currently green for direction $\vec{i}$ (and therefore red for direction $\vec{j}$), the agent may choose the action "Switch". This action does not change the lights immediately, but instead starts the following sequence:

- At time step t (when the switch is initiated), the lights remain green for $\vec{i}$ and red for $\vec{j}$.
- At time step $t+1$, the lights become orange for $\vec{i}$ (the next queueing car passes) and remain red for $\vec{j}$.
- At time step $t+2$, the lights remain orange for $\vec{i}$ (but the next queueing car will pass with only 50% chance) and remain red for $\vec{j}$.
- At time step $t+3$, the lights are red for both $\vec{i}$ and $\vec{j}$.
- At time step $t+4$, the lights are red for $\vec{i}$ and become green for $\vec{j}$. The action "Switch" becomes available again.

The cost of any transition is the number of cars that are queueing in the next state.
In order to bound the state space, we limit the number of cars queueing in any direction to `max_queue`.

Methods and Classes Available

The actions available to the agent are described in the class `TrafficLightAction`. They are `SWITCH` and `DO_NOT_SWITCH`.

The class `SingleLightState` represents the state of a single traffic light (in either direction NS or EW). The following methods are available:

- `next_state(self,other,act)` calculates what will be the next state of this traffic light, given the current state, the current state of the other traffic light, and the action chosen by the agent.
- `next_cars_queueing(self, current_cars: int, tau: float) -> List[Tuple[int, float]]`: returns a probabilistic distribution of how many cars will be queueing in the next state, given that `self` is the next state of the light, the current number of car, and the rate τ of cars coming. For instance, if `self` is green, the current number of cars is 4, and τ is 0.3, then the number of cars queueing in the next step will be: either 3 with 70% chance, or 4 is 30% chance. This is represented by a returned value of $[(3, .7), (4, .3)]$.

The class `TrafficLightState` represents a state of system, defined as the state of the light in each direction as well as the number of cars queueing.

3.2 Methods that Need to be Implemented

All the methods that need to be implemented are part of the `TrafficLightDomain` class.

3.2.1 Method `_get_transition_value`

This method indicates the cost associated with the transition from the specified state to the specified state through the specified action. Test this method with file `test_31.py`.

3.2.2 Method `_get_next_state_distribution`

This method indicates what the state distribution is after applying the specified action in the specified state. Use the methods provided to simplify the implementation. Test this method with file `test_32.py`.

3.2.3 Method `_get_applicable_actions_from`

This method indicates what actions are applicable in the specified state. Test this method with file `test_33.py`.

3.2.4 Method `_get_observation_space_`

This method indicates the list of states. While compute this list precisely (i.e., only those states that are reachable) is beneficial for computational reasons, you are allowed to return a superset of states. Test this method with file `test_34.py`.

3.2.5 Wrapping up

Run now the main method from `student_example2.py` and answer the question at the bottom of the file. This requires a working version of ValueIteration (from the first part of the assignment). If your implementation does not work, you may replace it with a working implementation of PolicyIteration or RTDP. The main file may take a few seconds to run successfully.