A(E)3M33UI - Semestral project 3

Markov Decision Processes

1 Introduction

The goal of this task is to make a practical experience with Markov Decision Processes (MDP) by implementing the *Value Iteration* algorithm.

2 Technical notes

For the labyrinth environment, implement the *Value Iteration* (VI) algorithm. The implementation consists of a single python file which must contain a class called `Solver` which derives from the class `SolverBase` that is defined in the `mdp_testbed` package (i.e. the file `mdp_testbed/__init__.py`).

The class must implement these methods:

- `solve_mdp(self, environment: Environment)`

  This method is called by the testbed platform. This method should be responsible for all the actual computation needed for solving the MDP.

  The `Environment` class is located at the `mdp_testbed` package (i.e. in the file `mdp_testbed/__init__.py`).

- `get_action_for_state(self, state: State) -> Action`

  This method is called by the testbed platform. This method is responsible for returning the action that should be performed in the given `state`.

  The `State` class and the `Action` enum are located in the `mdp_testbed.internal` module (i.e. in the file `mdp_testbed/internal.py`).

- `get_value_for_state(self, state: State) -> float`

  This method is called by the testbed platform. This method is responsible for returning the expected value of the given `state`.

  The `State` class is located in the `mdp_testbed.internal` module (i.e. in the file `mpd_testbed/internal.py`).

The `SolverBase` class has two members: `gamma` and `p_correct` which are set automatically when running the solver. You just need to use them properly (`self.gamma`, `self.p_correct`).

A working dummy solution with all the necessary structure is in the file `dummy_solution.py`. This solution does no computation at all, it always performs the `NORTH` action and it returns the rewards as the values. However it can serve you as a starting point (from the programming point of view).

Description of the classes needed for solving this task is in the Section Brief Description of the Testbed Framework.

2.1 Restrictions and Constraints

Given Python and its dynamic nature, it is impossible to prevent you from using some methods and members that are physically available. However, there is a common convention: methods and members starting with an underscore, e.g. `_get_coords()`, are considered to be **private**. Therefore you are **not allowed** to use any such method or member field of the objects that are given to you through the three methods introduced in the previous section, i.e. of the objects of classes `Environment` and `State`.

Everything related to your solver needs to be contained in a single python file. You can import whatever you need to import (libraries, other parts of the testbed...) but your code must be self-contained in that one file. Don't be afraid of that - a working, correct solution can fit 50 lines including imports (though it is not very computationally effective).

3 Experimental Goals

Make experiments with the following values of the parameters:

- Discount factor $\gamma : 0.5, 0.9, 0.99, 1$
- Reward r_0 for each non-terminal state[1]: $-10, -3, 0, 5$
- Maximum error $\varepsilon : 0.1, 0.01, 0.001$

Report and discuss the results:

- The resulting policies $\pi(s)$ and value functions $V(s)$ for three selected interesting combinations of the parameters γ, r_0 and ε.
- The dependence of the number of iterations and the total runtime on the
  - Discount factor γ (for $\varepsilon = 0.01$ and $r_0 = -10$).
  - Maximum error ε (for $\gamma = 0.99$ and $r_0 = -10$).
- Compare the number of iterations needed for value function convergence and for policy convergence. Discuss the difference.

4 Evaluation and Submission

Maximum score is 10 points. If you do not submit a working implementation you get zero points.

Implementation covers 0-4 points and the evaluation criteria are:

- Satisfaction of the above specification.
- Successfull pass in a validation trial consisting of one maze selected by the teacher.

[1] All the rewards are fixed in the maze. You must use the editor in order to set them. See Running the program.

Report covers 0-6 points and the evaluation criteria are:

- A comprehensive and detailed description of the implementation (the `Solver` class and the rest of your solution file).
- Quantitative results of the experiments (tables, plots).
- A reasonable discussion of the experimental results.
- Grammatical and formal aspects.

Bonus covers 0-2 points:

- 1 point if the report is written in LaTeX

4.1 Submission

A ZIP archive must be submitted that contains the following files:

- A Python file (`*.py`) with the required source code.
- File `surname.pdf` with the report which includes these sections: Implementation, Experimental results, Discussion.
- LaTeX sources if LaTeX was used.

5 Brief Description of the Testbed Framework

5.1 Requirements

The framework is written in Python 3 so it has to be installed (tested on python 3.4). In order for the framework to work, these libraries are needed:

- numpy
- tkinter

5.2 Running the program

The entry point is in the file `mdp_testbed/__main__.py` which is a file that is run when the package `mdp_testbed` is run as a module. Hence you can run the program like this:

```
$ python3 -m mdp_testbed
```

Use the `-h` option (i.e. `$ python3 -m mdp_testbed -h`) to get help on how to run the editor/solution viewer.

5.3 Important classes

5.3.1 The `Action` enum

This enum is defined in the module `mdp_testbed.internal`. It is an **enum**, pretty similar to Java enums, meaning it is a "list" of possible values. These values are

- `WEST` or `W` (the enum has both these aliases that can be used interchangeably) - represents the direction to the west, i.e. to the left side.
- `EAST` or `E` - represents the direction to the east, i.e. to the right side.
- `NORTH` or `N` - represents the direction to the north, i.e. in the upper direction.
- `SOUTH` or `S` - represents the direction to the south, i.e. in the lower direction.

This enum represents the possible actions in the labyrinth. There are no actions for turning, you just go in the desired direction.

You can access the enum values by using the class name, dot and the desired enum value, e.g. `Acion.WEST`. You can also iterate through all the values like this:

```
for action in Action:
    print(action)
```

The piece of code above would produce:

```
Action.WEST
Action.EAST
Action.NORTH
Action.SOUTH
```

You can get the numerical value of an enum option by using the option's `value` member:

```
>>> Action.EAST.value
2
```

And you can get the enum option from the numerical value by calling the enum like a constructor with the numerical value:

```
>>>print(Action(2))
Action.EAST
```

5.3.2 The `State` class

This class represents a single state. There are no public methods or members of this class, hence you are not allowed to use any of them. However, the objects of this class can be tested for equality using the == operator and can be used as keys in dictionaries.

5.3.3 The `Environment` class

This class represents the API which you can use to interact with the MDP. The class defines these **public** methods:

- `set_probability_of_correct_transition(self, p: float)`

  This method sets the probability that a transition from one state to the other will be the one desired by an action.

  Example: the agent chooses the action `SOUTH`. With probability p she will end up in the cell "below" the the one she is now and with probability $1 - p$ whe will not.

- `get_reward(self, state: State) -> float`

  This method returns the reward for reaching the given `state`.

- `get_all_states(self) -> list`

  This method returns **all** states that exist in the MDP.

- `get_transision_probability(self, from_state: State, action: Action, to_state: State) -> float`

  This method returns the probability of a transition from state `from_state` to state `to_state` given an action `action` was performed.