Artificial Intelligence – Programming Assignment 3

UNIZG FER, academic year 2015/16

Handed out: 24.5.2016. Due: 30.5.2016. at 23.59.

Introduction

In this lab assignment you will solve problems from the domain of reinforcement learning. The code you will be using can be downloaded as a zip archive at the web page of the university here or at the github repository of the class here.

In the scope of this lab assignment, you can use the autograder, which will give you adequate feedback regarding the correctness of your implementation. You run the autograder by typing `python autograder.py`. If you want to run the autograder for a specific question, do it as follows: `python autograder.py -q q1`.

The code for the project consists of Python files, a part of which you just need to read and understand, a part of which you need to modify yourselves and a part that you can ignore. After you download and unpack the zip archive, you will see a list of files and folders which we briefly describe below:

Files you will edit:	
valueIterationAgents.py	The logic of the value iteration algorithm
qlearningAgents.py	The logic of Q-learning based algorithms
Files you should study:	
mdp.py	Description of the general markov decision process
learningAgents.py	Logic describing the agents
util.py	Helper classes and methods
gridworld.py	GridWorld description
featureExtractors.py	File that extracts features for approximate Q-learning
Helper files you can ignore:	
environment.py	Abstract class for reinforcement learning problems
graphicsUtils.py	Helper functions for graphic display
graphicsGridworldDisplay.py	Graphic display of the GridWorld
crawler.py	Crawler logic
graphicsCrawlerDisplay.py	Graphic display of the crawler
autograder.py	Helper file for running tests

The code for the lab assignments was adapted from the course "Intro to AI" at Berkeley, which allowed the usage of their Pacman environment for other universities for educational purposes. The code is written in Python 2.7, which you require an interpreter for in order to run the lab assignment.

After you've downloaded and unpacked the code and positioned your console in the subdirectory where you have unpacked the archive, you can test the GridWorld by manually controlling the movement with arrows by typing:

```
python gridworld.py -m
```

GridWorld is a world where your movement is made difficult by the fact that you only have an 80% chance to actually do the action that you wanted to do, and a 20% chance to make a random action. The noise can be modified with the parameter '-n', such as:

```
python gridworld.py -m -n 0.5
```

In which case we set the noise rather high, to 50%. This parameter as well as the rest of them can be seen by calling the parameter '-h (help)'. The output is as follows:

```
Options:
  -h, --help            show this help message and exit
  -d DISCOUNT, --discount=DISCOUNT
                        Discount on future (default 0.9)
  -r R, --livingReward=R
                        Reward for living for a time step (default 0.0)
  -n P, --noise=P       How often action results in unintended direction
                        (default 0.2)
  -e E, --epsilon=E     Chance of taking a random action in q-learning
                        (default 0.3)
  -l P, --learningRate=P
                        TD learning rate (default 0.5)
  -i K, --iterations=K  Number of rounds of value iteration (default 10)
  -k K, --episodes=K    Number of epsiodes of the MDP to run (default 1)
  -g G, --grid=G        Grid to use (case sensitive; options are BookGrid,
                        BridgeGrid, CliffGrid, MazeGrid, default BookGrid)
  -w X, --windowSize=X  Request a window width of X pixels *per grid cell*
                        (default 150)
  -a A, --agent=A       Agent type (options are 'random', 'value' and 'q',
                        default random)
  -t, --text            Use text-only ASCII display
  -p, --pause           Pause GUI after each time step when running the MDP
  -q, --quiet           Skip display of any learning episodes
  -s S, --speed=S       Speed of animation, S > 1.0 is faster, 0.0 < S < 1.0
                        is slower (default 1.0)
  -m, --manual          Manually control agent
  -v, --valueSteps      Display each step of value iteration
```

As we have covered in the lectures, the problem we are solving in GridWorld is the one of optimizing the total utility while moving through the map. The difficulty in this problem is that you have noise while moving, and the actions that you choose don't need to necessarily take you to the same place. In the GridWorld, this manifests in the way that you have a fixed chance of $1 - n$ to take the action that you chose, and a chance of n to move randomly.

As usual, the states are defined by coordinates (x,y), while the transitions are defined by the sides of the world towards which the agent is moving (south, east, north, west).

The lab assignment consists of four subtasks, and the correct implementation of each one of them carries the same weight. To ease the grading, each of the subtasks carries five points, with a total sum of 20. Your final points will be scaled and the actual maximum for the third lab assignment is 6.25.

Problem 1: Value iteration (5 points)

In the file valueIterationAgents.py fill in the code missing to implement value iteration. A reminder from the lectures - value iteration is an algorithm that tries to calculate the utility of every state of the map by using the recursive Bellman equation:

$$V_0(s) = 0$$
$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

As a parameter to the constructor of your *ValueIterationAgent* you receive an argument *mdp*, which is the definition of the underlying Markov Decision Process. In the case of GridWorld, this is the implementation of the abstract class *mdp.MarkovDecisionProcess* which is located in *gridworld.py*. The methods that describe the Markov Decision Process are:

`mdp.getStates():`

Returns the list of all possible states as a list of tuples (int, int)

`mdp.getPossibleActions(state):`

Returns the list of strings representing all possible actions for a given state

`mdp.getTransitionStatesAndProbs(state, action):`

Returns a list of pairs(tuples) consisting of (nextState: tuple(int, int), transitionProbability: float)

`mdp.getReward(state, action, nextState):`

Returns the float precision reward for a given state, action, nextState triple

`mdp.isTerminal(state):`

Checks if a state is final (returns True or False). Final states don't have any transitions (so you should handle the case where the list of transitions is empty)!

The methods you need to complete are: *__init__*, *computeQValueFromValues*, *computeActionFromValues*. SInce you have pre-defined values of the transition and reward functions, most of your computation should be done in the initialization. While solving use the already initialized class *util.Counter*, which is an extension of the Python's default dictionary with default values of all keys set to zero - so you don't need to initialize the value for each state.

Note 1.1 Make sure that while updating the values you use a helper structure where you will store the intermediate result - in the stap $k + 1$ of value iteration, the values V_{k+1} depend only on the values from the previous step V_k - make sure you don't use the values that you have calculated in step $k + 1$.

In order to test your implementation, the result of running the following command:

```
python gridworld.py -a value -i 5
```

should look exactly like this:

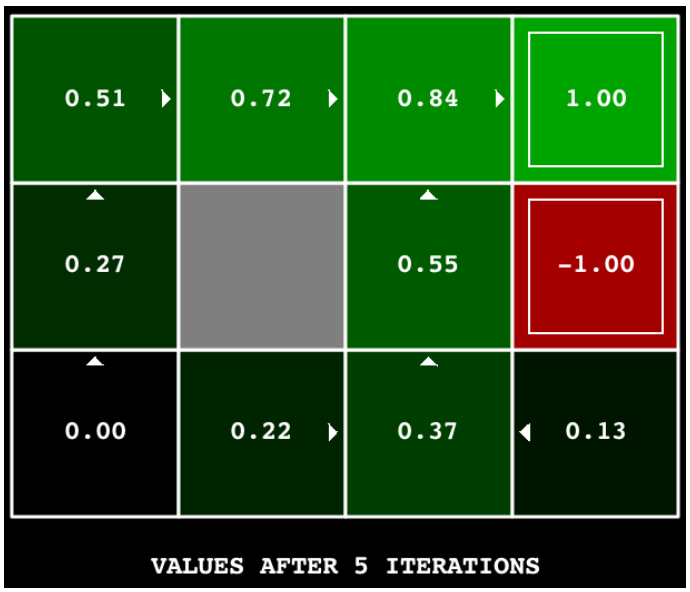

Figure 1: Value iteration example

Problem 2: Q-learning (5 points)

In the file qlearningAgents.py, fill in the code for the Q-learning algorithm. A reminder from the classes, the Q-learning algorithm works on the running average principle, and the update formula looks as follows:

$$Q_0(s,a) = 0$$
$$Q_{k+1}(s,a) \leftarrow Q_k(s,a) + (\alpha)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a)]$$

Where α is the learning rate, while γ is the decaying factor.

The methods you need to fill in in the QLearningAgent class are: *__init__* , *getQValue*, *computeValueFromQValues*, *computeActionFromQValues*, *getAction* and *update*. The QlearningAgent class has variables that it inherits from the ReinforcementAgent class, which even though you can't see explicitly, exist. The variables that were inherited, and you will need in the scope of the assignment are: *self.epsilon* - exploration probability for the ϵ-greedy approach (Only in assignment 3.), *self.alpha* - learning rate, and *self.discount* - the decaying factor. All the falues are in float precision

Another useful function which is not explicitely seen is `self.getLegalActions(state)`, which returns the list of possible actions for a given state.

Note 2.1 Take care of the fact that if you visited just one Q-state of a state, and it has a negative value, the optimal move can be one of the moves that you have not explored yet (due to the implicit default value of zero). Take care to explicitly initialize the moves so you can handle these cases. In case of ties in the Q-values of multiple Q-states, you should solve the ties by taking a random action from the ones with the maximum values. For this, the method `random.choice()` is useful, as it takes a list of elements as an argument and returns a random one with uniform probability.

Note 2.2 Take care of the fact that when accessing Q-values, you use the method`getQValue`, since it is used later on in approximate Q-learning.

Note 2.3 You can use tuples as keys to your dictionaries and Counters!

Problem 3: ϵ-greedy (5 bodova)

Modify your Q-learning algorithm by implementing the ϵ-greedy approach. As a reminder, the ϵ is a small probability of taking a random action while learning the Q-values. The probability ϵ is available as a class variable *self.epsilon*, while the method util.flipCoin(p), might also prove useful, as it returns True with probability p, and False with probability $1 - p$.

Without any additional changes to your code after adding the ϵ-greedy approach, you can test your implementation by running the crawler and playing around with the parameters:

```
python crawler.py
```

Problem 4: Approximate Q-learning (5 bodova)

Implement approximate Q-learning in the file qlearningAgents.py in the class *ApproximateAgent*.

As a reminder, an approximate Q-value is the form:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

Where W is the weight vector, while $f(s,a)$ is the feature vector for a Q-state (s,a). Every weight w_i from the vector is mapped to one feature f_i. The functions that will calculate the features are already defined and implemented in the file *featureExtractors.py*, and the class variable *self.featExtractor* which is available in your ApproximateAgent has a method *getFeatures(state, action)* which returns a Counter with feature values for a given Q-state.

At the beginning of the process, you should initialize the weight to zero (hint: use the Counter), and then update them by using the formula from the classes:

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$
$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$

If your implementation works, the learned pacman controller should without any problems solve the following map:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumGrid
```

While with a bit longer training time, he should have no more problems with a larger one:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumClassic
```