Final Project Assignment
Probabilistic Modeling for the Pac-Man Game and other Scenarios

Artificial Intelligence
Grado en Ingeniería Informática
Spring 2017

Overview

In this project, you will implement reasoning techniques under uncertainty to be applied to the Pac-Mac game and other scenarios. The project is within the context of Markov Decision Processes (MDPs). For purposes of this work, we make use of a modified version of a software provided by the University of Berkeley[1], which you can download from Aula Global. The programming language is Python 2.7, which is friendly and easy to learn. You can find a short tutorial that introduces you to the Python programming language and the UNIX environment here: `http://ai.berkeley.edu/tutorial.html`.

The project consists of three main parts:

- Part I:
  - Software installation.
  - Execution of given examples.
  - Implementation of the Value Iteration algorithm.
  - Test evaluation using different parameter settings.
- Part II:
  - Implementation of a learning module for an MDP.
  - Test evaluation of different given scenarios.
- Part III:
  - Implementation of a new data structure to represent states.
  - Test evaluation of more complex scenarios.

Part I: Implementation of the Value Iteration algorithm

In this part, we make use of the *GridWorld* scenario, which consists of a robot moving on a finite grid of cells. It can move in four different directions: North, South, East, West. Each time the robot performs an action, there is some probability that it does not reach the expected state. Such probability can be set through the *noise* parameter, and remains the same during the whole simulation and for all actions. The default value of the *noise* parameter is 0.2. This means that, if the agent performs the *North* action, it will move to the above cell with a probability of 0.8, and will end up in any other reachable cell with a probability of 0.2.

[1] `http://ai.berkeley.edu/reinforcement.html`

To implement intelligent agents capable of automatically moving from an initial position to a goal position, we can use a Markov Decision Process (MDP) with the following features:

- States: represent the position of the agent. It takes the form (x, y), where x represents the grid's row, and y represents the grid's column. (For instance, the position $(0, 0)$ represents the bottom left grid cell.)
- Actions: represent the four different movements *North*, *South*, *East*, and *West*.
- Transition probabilities: the agent will move to the expected cell with a probability of (1-<*noise*>), and will end up in any other reachable cell with a probability of <*noise*>. If it comes to a unreachable cell, the Pac-Man agent will remain in the current cell.
- Reward function $R(s, a, s')$: represents the reward of reaching the s' state from the s state when action a is applied.
  1. When a state does not have any reward, it is called *living-reward* state. This is an input parameter and its default value is 0.
  2. When a state has a reward, it is called *exit* state. In this state there is only one applicable action that takes the system to a goal state.
  3. The reward in a terminal state is 0.
- Expected value function $V^t(s')$:

$$\begin{aligned} V^t(s) &= \arg\max_a Q(s,a) \\ Q(s,a) &= \sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma V^{t-1}(s')] \end{aligned} \tag{1}$$

  where $\gamma \in [0, 1]$ is the **discount factor**, which describes the preference of the agent for the current rewards over future rewards. Higher γ values give preference to future rewards.

Next, we detail the tasks to perform in this scenario:

1. **Execute** the *Gridworld* scenario typing in the terminal the following command:

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

   Check that when you press an arrow key only the 80% of the time the robot (the blue dot) moves to the right direction. There are two goal squares, one correct and another one incorrect with values +1 and -1 respectively. On top of that, you can control other aspects of the simulation. To list them all, type in terminal the following command:

```
python gridworld.py -h
```

   To execute the default configuration, type in the terminal the following command:

```
python gridworld.py -g MazeGrid
```

   Observe and understand the execution trace.

2. **Implement** part of the methods in *valueIterationAgents.py* between the following marks:

```
*** YOUR CODE STARTS HERE***

*** YOUR CODE  FINISHES HERE***
```

Right before those blocks, you might find the following line:

```
util.raiseNotDefined()
```

which needs to be commented since it stops the execution. However, it could be helpful for debugging purposes.

The methods you need to implement are:

- `computeQValueFromValues(state, action)`: it returns the Q-value for the pair (state, action), according to the expected $V(s)$ values in self.values.
- `computeActionFromValues(state)`: it computes the best action according to the expected $V(s)$ values in self.values.
- `doValueIteration()`: it performs a generic *Value Iteration* algorithm.

For testing your code, use the default settings of the *GridWorld* scenario (*BookGrid*), and execute it by typing the following command in the terminal:

```
python gridworld.py -a value -i <iterations> -k <executions>
```

where i is the number of iterations to compute a policy, and k is the number of executions performed by the agent applying that policy.

During execution, you can check the Q-values by pressing any key. After 5 iterations, the execution should show a similar figure to the one shown in Figure 1. For each cell, it shows the expected V-value computed. The policy is shown with an arrow, which points out the direction where the agent should move.

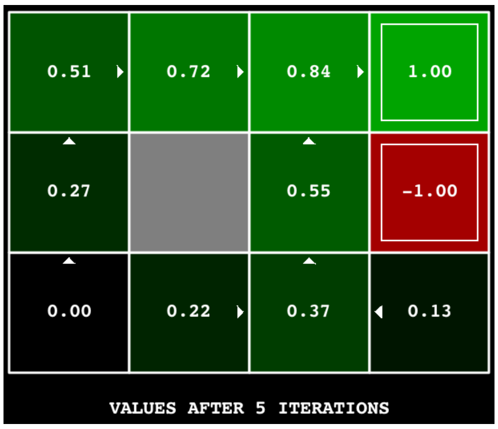

Figure 1: Output of the ValueIterationAgent agent after 5 iterations.

- Check if your implementation is correct by typing the following line in the terminal:

```
python autograder.py -q q1
```

Solve this problem on paper. For each iteration, compare if the output of your implementation is the same as the solution you got.

3. Check your implementation using the *BridgeGrid* scenario (Figure 2). It simulates a bridge with an agent initially located near to the left-hand side of the bridge. The goal is to move the agent to the right-hand side without falling into the water (cells with a reward of -100). To execute the simulation, type the following command in your terminal:

   ```
   python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
   ```

   When you run this scenario using the default values for the *discount* and *noise* parameters, you can see that the agent does not cross the bridge. Instead, it leaves the bridge through the cell with a reward of +1. Choose, therefore, different values for the *discount* and *noise* parameters so that the agent crosses the bridge. That is, it reaches the cell with a reward of +10. Remember that the *noise* parameter represents the probability that the agent does not reach the expected state (cell) when executing an action.

   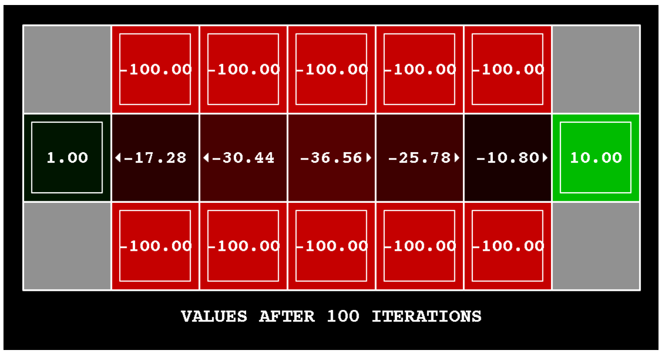

   Figure 2: An example of the BridgeGrid scenario.

   To check if those values are correct, answer `question2` in the `analysis.py` file, which is related to the "answerDiscount" and "anserNoise" parameters. Then, type the following command in the terminal:

   ```
   python autograder.py -q q2
   ```

4. Figure 3 shows the *DiscountGrid* scenario, which is a grid with two exits: one with a reward of +1, and another one with a reward of +10. The agent is initially located at the yellow cell; it has, therefore, two choices: (1) choose a safer and longer path (green arrow) that avoids falling off the cliff (red cells each of them with a reward of -10); and (2) choose a less safe, but shorter path (red arrow) next to the cliff.

   Choose values for the *discount* and *noise* parameters so that the ValueIterationAgent agent has the following behaviors:

   (a) Reach the exit (+1) choosing the less safe but shorter path (-10)
   (b) Reach the exit (+1) choosing the safer and longer path (-10)
   (c) Reach the exit (+10) choosing the less safe but shorter path (-10)
   (d) Reach the exit (+10) choosing the safer and longer path (-10)
   (e) Avoiding both exits and the cliff (this option should never end)

The answers to these questions should be included in the `analysis.py` file (`question3a` to `question3e`). For each question, there must be a function that returns a 3-tuple <*discount* , *noise*, *living_reward*>. Then, test your implementation by executing:

```
python autograder.py -q q3
```

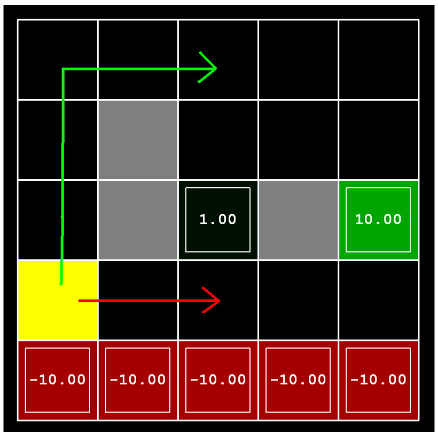

Figure 3: An example of the DiscountGrid scenario.

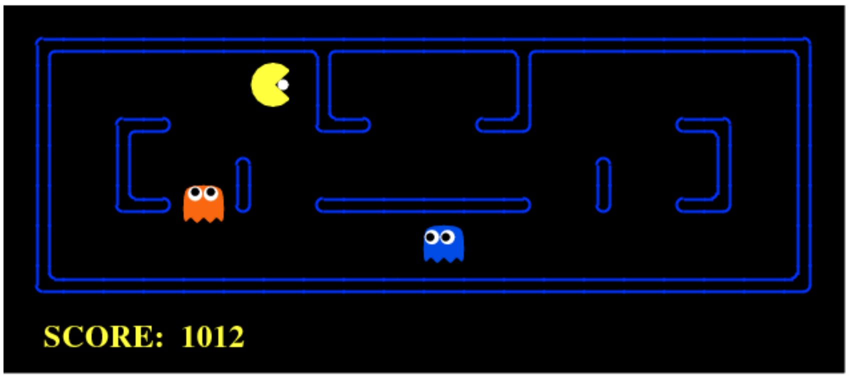

Figure 4: A simplified version of the PAC-MAN game.

Part II: Learning of the transition function in the PAC-MAN game

In this project, we are going to work with a simplified version of the PAC-MAN game. The scenario consists of a maze with walls where there is:

- A Pac-Man agent, which is represented by a yellow ball.
- A set of ghosts.
- Pac-Man's food, which is represented by white dots.

To win the game, the Pac-Man agent must eat all the food while avoiding to be eaten by a ghost. If the latter case happens, the game is over. Figure 4 shows an example of the game.

The number of possible states in this game is huge. For this reason, in this project we are going to work with small scenarios in order to compute reasonable policies.

In this part of the project, we assume that the transition function of the Pac-Man agent is unknown. Therefore, we need to learn it. In order to do that, we need to randomly execute the Pac-Man agent a high number of times. In this way, we could get information to learn the transition function. You need to implement the code needed to learn the transition function that represents the Pac-Man agent. This MDP uses a simplified state representation, namely *high-level* (the real state representation is called *low-level*), which needs to be understood before starting working on it. In order to do that, you need to edit the `featureExtractors.py` file, and understand the implementation of the `FullStateExtractor` class. The tuple representing the High-level state has the form $< posX, posY, IncFoodX, IncFoosY, IncGhostX, IncGhostY >$, where:

- $posX$ and $posY$ represent the position of the Pac-Man. The coordinates (0,0) (i.e. $posX = 0$ and $posY = 0$) refer to the lower right corner.
- $IncFoodX$ and $IncFoosY$ represent the distance of the Pac-Man to the closest food (ignoring walls).
- $IncGhostX$ and $IncGhostY$ represent the distance of the Pac-Man to the closest ghost (ignoring walls).

Table 1 shows an example of this representation.

Low level (map)	High-level (tuple)
`%%%%%%%`	
`% <   %`	
`% %%% %`	
`% %.  %`	(2, 5, 1, -2, -1, -4)
`% %%% %`	
`%G G  %`	
`%%%%%%%`	

Table 1: States codificaction: "`%`" represents the walls, "<" represents the Pac-Man, "`.`" represents the Pac-Man's food, and "`G`" represents the location of the ghosts.

The implemented code assumes that initially we only know the MDP actions, which are *North*, *South*, *East*, *West*, and *Stop*. States, transition functions, and rewards are generated through random execution rounds where the Pac-Man agent performs random actions. During execution, it keeps track of the states where the Pac-Mac is in to then compute the probability of the transitions. The transition function expresses the relation between each *state-action* pair and the next state. (In an MDP model the transition function may not be deterministic, therefore, we need to define the condition probability tables.) This implies that, if there is not enough training, the learned transition function will be partial. In other words, we will not get values for each *state-action* pair.

In this part of the project, you need to implement the following methods, contained in the `pacmanMdp.py` file:

- `updateTransitionFunction(self, initialMap, action, nextMap)`: this method updates the transition function using the 3-tuple `initialMap, action, nextMap`, when the Value Iteration algorithm is executed. The transition function is stored for the simplified (high-level) state representation. Therefore, this method first changes the state representation to a simplified (high-level) state representation.
- `getTransitionStatesAndProbabilities(self, state, action)`: this method returns $P(nextState \mid state, action)$ for each possible next simplified (high-level) state.
- `__init__` : class constructor; it is used to initialize variables. You need to initialize the `transitionTable` variable (without changing the name) according to the implementation you develop.

Once the transition function is learned, we are ready to execute the Value Iteration algorithm in the PAC-MAN game to compute a policy. Given that both transition functions and known states can be partial, the Value Iteration could generate a partial policy. When it comes to a partial policy and an unknown state appears, the agent performs a random action.

Check if your implementation is correct using simple scenarios. To debug, once the training is over, show the learned transition function. The following command trains the Pac-Man agent during `<train>` games;

executes `<total>` games, where `<total>` − `<train>` is the number of games where the behavior of the agent is observed; includes `<ghosts>` ghosts; uses the `<scenario>` scenario; provides the parameters `<iter>` (iteraction number) and `<discount-factor>` (discount factor) to the Value Iteration algorithm:

```
python pacman.py -p EstimatePacmanMdpAgent -x <train> -n <total> -k <ghosts>
-l <scenario>  -a "iterations=<iter>,discount=<discount-factor>"
```

Initially, we will use a couple of easy scenarios where ghosts are not included. In such scenarios, the transition function is deterministic. Therefore, the probabilities printed out on your terminal should be 1.0 for all transitions.

```
python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 0 -l superSmallGrid  -a
"iterations=50,discount=0.8"

python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 0 -l smallGrid2  -a
"iterations=50,discount=0.8"
```

Let us assume a case where there are some ghosts. In such case, you need to check that there is uncertainty in the transition function since the behavior of the ghosts is random.

```
python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 1 -l smallGrid2  -a
"iterations=50,discount=0.8"
```

Table 2 shows an example of the learned transition functions during training. It corresponds to a state $state_i$ where an action is executed (the example uses the initial state). For each action, there is a list of possible next states, $nextState_i$, each of them with an associated probability. This list takes the form $[((nextState_i),$ $P(nextState_i \mid state_i, action)), \ldots]$[2].

	State	Action	Learned transitions
k=0	(2, 5, 1, -2)	West	[((1, 5, 2, -2), 1.0)]
	(2, 5, 1, -2)	Stop	[((2, 5, 1, -2), 1.0)]
	(2, 5, 1, -2)	East	[((3, 5, 0, -2), 1.0)]
k=1	(2, 5, 1, -2, -1, -4)	West	[((1, 5, 2, -2, 1, -4), 0.466), ((1, 5, 2, -2, 0, -3), 0.533)]
	(2, 5, 1, -2, -1, -4)	Stop	[((2, 5, 1, -2, -1, -3), 0.555), ((2, 5, 1, -2, 0, -4), 0.444)]
	(2, 5, 1, -2, -1, -4)	East	[((3, 5, 0, -2, -2, -3), 0.411), ((3, 5, 0, -2, -1, -4), 0.588)]

Table 2: Transition function for the initial state in the `smallGrid2` scenario. When k=0 (no ghosts) there is not uncertainty: when k=1 there is uncertainty since the ghost can move to different locations. The last two elements of the tuple represent the distance of the Pac-Man to the closest ghost.

Part III: Additional experimental evaluation

Advanced training

This project allows specifying an external file as parameter. This file is used to save the solution of a set of training rounds. It is very useful to set different experiments and, therefore, learn a more general behavior. For instance:

- Computing a policy in a particular scenario, specifying: (1) different initial positions of the Pac-Man agent, (2) different positions of the food, and/or (3) different positions of the ghost(s).
- Computing a policy in a particular scenario, but executing the policy in a different one.

[2]This output corresponds to the output of the `getTransitionStatesAndProbabilities` function.

- Computing a policy with a particular number of ghosts and several food location, but executing the policy in another one with a different number of ghosts, food location, etc.

To use this file, you need to include the `table=<file-name>.p` parameter in the command-line arguments as follows:

```
python pacman.py -p EstimatePacmanMdpAgent -x 50 -n 55 -k 1 -l smallGrid2  -a
"iterations=50,discount=0.8,table=smallGrid2.p"
```

If the file exists, the training starts from the state defined in the file and overwrites the file when the training finishes.

Note: To execute a test using the information defined in a file, you need to run at least one training round (`-x 1`). Otherwise, the Pac-Man agent will use random actions. For instance, to run a 100 experiments using the `smallGrid2.p` file, type in the terminal the following command:

```
python pacman.py -p EstimatePacmanMdpAgent -x 1 -n 101 -k 1 -l smallGrid2  -a
"iterations=50,discount=0.8,table=smallGrid2.p"
```

The `layouts` directory contains a number scenarios that you can use for this game. However, you can also create your own scenarios, varying the number of ghosts, the food position, the grid shape, etc. Remember that the higher the number of states, the higher the training rounds you need to set (`-x`).

Modifying the simplified (high-level) state

If you check out the policy generated, you will observe that it is more difficult to find an identical case as the state gets more complex. However, if you do not provide information about the location of the food and the location of the ghosts, the Pac-Man agent will not be able to compute a policy that allows winning a relatively high number of cases. For this reason, we propose that you change the implementation of the `stateToHigh` function in the `pacmanMdp.py` file in order to generate the most appropriate attributes for the learning process.
This function receives a grid (low level state) and makes a call to `self.featExtractor.getFeatures` to get the attributes defined in the header. In other words, this function simply selects some of those attributes through the list defined in the constructor (`__init__`) of `pacmanMdp.py`. You could choose another set of attributes among the defined ones, or compute other attributes from the given ones. For instance, you can create a boolean attribute, set to $True$, to indicate if the ghost is in a given distance to the Pac-Man agent, and $False$ if it is further. This discretization could decrease the number of unknown states in the learned transition function.

Important: the high-level states should be variables defined as tuples, numbers, strings, booleans, and combinations (lists and dictionaries are not allowed).

Submission

This project must be done in pairs or individually. You must submit a zip file through a link that will be posted on Aula Global. Note that if you work in pairs, only one submission is needed. The zip file name must be *pr-ai-2017-students-code.zip*, where *students-code* is the last six digits of each students' NIA (e.g. pr-ai-2017-123456-654321.zip). It will contain:

- An essay containing the following sections:
  1. Introduction.
  2. Development: description of the tasks performed within the project. Include an explanation of each of the implemented methods.

3. Test evaluation: description and explanation of the performed test and the results obtained from such testing.
4. Conclusions: technical comments related to the development of this project.
5. Personal comments: difficulties, challenges, benefits, etc.

- A directory containing the source code, including the parts you have implemented.

Project submission deadline is May 12th!