Single Agent Planning

Lab Assignment 1
Autonomous Agents
Master Artificial Intelligence
University of Amsterdam

Camiel Verschoor
StudentID: 10017321
UvAnetID: 6229298
`Verschoor@uva.nl`

Steven Laan
StudentID: 6036031
UvAnetID: 6036031
`S.Laan@uva.nl`

Auke Wiggers
StudentID: 6036163
UvAnetID: 6036163
`Auke.Wiggers@uva.nl`

September 21, 2012

1 Introduction

In this report we will discuss the predator versus prey world Markov Decision Process (MDP). The basic environment of the predator versus prey MDP is a 11 by 11 toroidal grid (see figure 1), which means that the north and south side and the east and west side are connected. The predator and prey can be anywhere on the grid, however they cannot have the same position. The starting position of the prey is $(5, 5)$ and of the predator is $(0, 0)$.

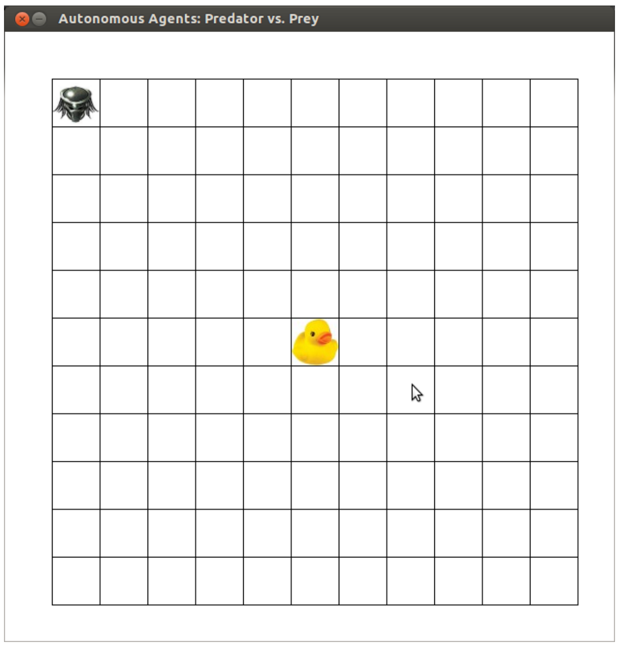

Figure 1: The 11 by 11 toroidal grid pf the predator versus prey world MDP.

In the MDP the prey behaves in a known way, and therefore can be part of the environment. The prey has five actions:

- $ACTION\_NORTH$
- $ACTION\_EAST$
- $ACTION\_WEST$
- $ACTION\_WEST$
- $ACTION\_STAY$

The prey performs action $ACTION\_STAY$ with probability $P(ACTION\_STAY) = 0.8$ and performs any of the other actions with equal probability (total probability of 0.2), given that a prey will never move into a predator.

The predator is the agent in the MDP and has the same actions as the prey. The goal of the predator is to catch the prey, which occurs when the predator stands next to the prey and performs an action towards the prey. The reward for catching the prey is 10 and the immediate reward for any other action is 0.

This report will focus on the planning scenario, which means that the entire MDP is known to the agent. Therefore, the agent can determine the optimal policy even before interacting with the environment. In this report the optimal policy is determined on the basis of methods presented in the book Reinforcement Learning: An Introduction [Sutton and Barto, 1998].

2 Method

This section describes the methods that are applied in this report. The following methods will be discussed: the simulator and visualization, the state representation, the transition and reward function, iterative policy evaluation, value iteration and policy iteration.

2.1 Simulator and Visualization

The simulator of the environment can simulate the predator and prey according to their policy. The simulation can handle both state representations (see section 2.2) as the algorithms for both representations are implemented independently.

For debugging purposes a visualization tool was constructed for the MDP based on the Pygame[1] library. The visualization represents the following elements of the MDP:

- Location of the predator.
- Location of the prey.
- 11×11 Grid.

In figure 1 a screenshot is shown of the visualization tool. The visualization tool shows the simulation of the MDP until the prey is catched and by pressing the r key the simulation is restarted.

[1]http://pygame.org/

2.2 State representation

The standard state representation suggested by the assignment is one where the state is represented by two pairs of coordinates: the location of the prey and the location of the predator. Because either one can be on either square of the grid, the size of the state space is polynomial in the size of the environment: $O(n^4)$ where n is the length of one edge of the environment. In the given exercise these dimensions are 11×11, thus obtaining a state space that is 11^4 if naively implemented using a table.

2.2.1 Reduced State Representation

Instead of keeping track of the specific coordinates of both predator and prey, it is also sufficient to only maintain the relative position from one to another. In this scenario the world is toroidal, which means that the playingfield wraps. For example: when you move down in the bottom row, you will end up in the top row. This means that all situations (or states) where the relative position is the same, is essentially the same state. For example the situation where the prey is at (2,3) and the predator at (4,5) is analoguous to the situation where the prey is at (9,10) and the predator at (1,2). This representation only needs $O(n^2)$ space.

2.2.2 Minimal State Representation

In the extreme case only a few states are needed. This is due to the symmetry of the environment. In order to make an optimal decision, the predator only needs to know where the prey is, relatively. The reduced state representation took advantage of this fact by only looking at the relative position. However, the exact distance to the prey is irrelevant. If the predator just knows in which direction to walk, this is sufficient to obtain optimal behavior. Therefore the minimal state representation consists just of the direction of the prey. In a rectangular grid, this breaks further down to four directions: north, east, south, and west. Then we need only one extra state to denote the terminal state, thus the state can be represented using only 5 different states.

However, note that this is not sufficient information for the prey, since the information of the distance of the predator is lost. Therefore, the prey cannot determine when he is moving into the predator, which is part of his policy. For this reason, this state representation was not implemented. The predator does not need this information, he only needs the transition function for planning, which can be easily computed.

2.3 Transition function and reward function

For all three algorithms to work, two functions need to be specified: the reward function and the transition function. The reward function returns the immediate reward for a state-action-pair. In this case, the immediate reward is 10 when the predator catches the prey, otherwise it is 0. The reward function is implemented as a method of the environment. The environment has access to both the location of the predator and the prey and is therefore in a good position to compute the reward. The predator does know the location of the prey, via the environment, so it could be implemented in the predator as well (or the prey for that matter). The reason we chose to implement it in the environment was to keep the chains as short as possible. In the environment the location of the prey can easily be retrieved by `self.prey.location`, whereas in the predator the same location is found by `self.environment.prey.location`, which yields a longer chain of references.

The transition function is also implemented in the environment, for the same reason. The transition function provides for each state, action and next state the probability of reaching that next state, given the action and the previous state. In our implementation the method `nextStates` does slightly more than that. Instead of only assigning probabilities to s, a, s'-triples, it also computes the possible next states for the given state. It returns a dictionary (or hashmap in other languages), containing the different next states and their probabilities. The transition function is online, instead of a lookup-table. This means that for each call the probabilities are computed only then, instead of computing all probabilities beforehand and just looking them up during when needed.

2.4 Iterative Policy Evaluation

The goal of policy evaluation is to obtain the value function given a policy. The value function assigns a value to every state, where a higher value is more desirable. The algorithm for policy evaluation is given in algorithm 1, as specified by Sutton and Barto [Sutton and Barto, 1998].

Algorithm 1 Policy Evaluation

```
Input $\pi$, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in S^+$
repeat
    $\Delta \leftarrow 0$
    for each $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
until $\Delta < \theta$ (a small positive number)
```

Since the Python programming language has built-in set operations, the implementation is quite similar to the pseudocode in algorithm 1. A difference with the pseudocode is that the implementation does not need an input policy. As there is only one predator in the environment (for now), the algorithm will take the policy of the only predator and use that for evaluation.

2.5 Value Iteration

Value Iteration is a method to obtain the optimal policy given a MDP. This is done by iteratively updating the best actions to take in each state. The pseudocode as written by Sutton and Barto for value iteration is given in algorithm 2.

2.6 Policy Iteration

The goal of policy iteration is to obtain a better policy using the evaluation of the current policy. Eventually, after sufficient iterations, the improved policy is the optimal policy. The pseudocode for policy iteration is given in algorithm 3.

Again, the implementation looks very similar to the pseudocode. However, whereas the pseudocode has all the code for policy improvement and policy iteration in its body, in our implementation those are seperate methods. Policy evaluation returns a value function, which is input for the policy improvement.

Algorithm 2 Value Iteration

```
Initialize $V$ arbitrarily, e.g. $V(s) = 0$ for all $s \in S^+$
repeat
    $\Delta \leftarrow 0$
    for each $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end for
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi$, such that
    $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
```

Algorithm 3 Policy Iteration

```
1. Initialization
    $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}$ arbitrarily for all $s \in S$

2. Policy evaluation
    repeat
        $\Delta \leftarrow 0$
        for each $s \in S$ do
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
        end for
    until $\Delta < \theta$ (a small positive number)

3. Policy improvement
    policy-stable $\leftarrow$ true
    for each $s \in S$ do
        $b \leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
        if $b \neq \pi(s)$ then
            policy-stable $\leftarrow$ false
        end if
    end for
    if policy-stable then
        Stop
    else
        Go to 2
    end if
```

3 Results

This section reports on the results that have been found by the accompanying methods.

3.1 Simulation

The environment and agent were simulated for 100 runs, with a random policy for the agent. The average and standard deviation were computed for the number of iterations:

Average	272.3 steps
Standard deviation	19.73 steps

3.2 Iterative Policy Evaluation

Iterative policy evaluation can be used to find the values fo a lstatlse., for a discount factor $\gamma = 0.8$. Stated below are values found by policy evaluation for sample states:

- `Predator(0,0), Prey(5,5)`: 0.005724141401102881
- `Predator(2,3), Prey(5,4)`: 0.1819507638515225
- `Predator(2,10), Prey(10,0)`: 0.1819507638515225
- `Predator(10,10), Prey(0,0)`: 1.1945854778368168

The algorithm took a total of **207** iterations to converge to the limit.

3.3 Reducing the statespace

The number of iterations for both the 2D state-representation and the 4D representation (as described in 2.2) is shown in table 1. Results indicate that the total number of iterations of the value iteration algorithm for the reduced statespace will be less than the number of iterations for the normal 4D state representation, for all discount factors $0 > \gamma > 1$. For a more Measurements were performed on a dual-core system, 2.3GHz, with 6GB RAM.

	Reduced		Normal	
Discount	Iterations	Time (seconds)	Iterations	Time (seconds)
0.1	0.314	20	21	33.96
0.3	0.360	25	26	43.97
0.5	0.394	28	30	50.94
0.7	0.430	30	36	60.01
0.9	0.466	33	37	60.59

Table 1: Number of iterations for value iteration, for both state representations and for different discount factors. Values are obtained after the algorithm has converged to the limit.

	Reduced		Normal	
Discount	Iterations	Time (seconds)	Iterations	Time (seconds)
0.1	3	1.40	6	251.25
0.3	4	1.91	5	278.87
0.5	4	2.46	5	364.06
0.7	4	3.22	6	586.23
0.9	5	7.23	6	996.03

Table 2: Number of iterations and runtime for policy iteration, for both state representations and for different discount factors. Values are obtained after the algorithm has converged to the limit.

0,4348	0,6065	0,8453	1,1765	1,6463	2,1169	1,6463	1,1765	0,8453	0,6065	0,4348
0,6065	0,8328	1,1750	1,6587	2,3345	3,0759	2,3345	1,6587	1,1750	0,8328	0,6065
0,8453	1,1750	1,6587	2,3415	3,3044	4,4805	3,3044	2,3415	1,6587	1,1750	0,8453
1,1765	1,6587	2,3415	3,3044	4,6737	6,5116	4,6737	3,3044	2,3415	1,6587	1,1765
1,6463	2,3345	3,3044	4,6737	6,5116	10,0000	6,5116	4,6737	3,3044	2,3345	1,6463
2,1169	3,0759	4,4805	6,5116	10,0000	0,0000	10,0000	6,5116	4,4805	3,0759	2,1169
1,6463	2,3345	3,3044	4,6737	6,5116	10,0000	6,5116	4,6737	3,3044	2,3345	1,6463
1,1765	1,6587	2,3415	3,3044	4,6737	6,5116	4,6737	3,3044	2,3415	1,6587	1,1765
0,8453	1,1750	1,6587	2,3415	3,3044	4,4805	3,3044	2,3415	1,6587	1,1750	0,8453
0,6065	0,8328	1,1750	1,6587	2,3345	3,0759	2,3345	1,6587	1,1750	0,8328	0,6065
0,4348	0,6065	0,8453	1,1765	1,6463	2,1169	1,6463	1,1765	0,8453	0,6065	0,4348

Table 3: All the values of the different positions after value iteration has converged, for discount factor $\gamma = 0.7$.

4 Implementation

4.1 Usage Guide

In order to run all our software sucessfully the following two programmes should be installed:

- Python 2.7 interpreter[2].
- Pygame library[3], to run our visualization.

Type the following command in the terminal to launch our main program:

```
>>> python main.py
```

Type the following commands in your python interpreter to run the implementations individually in the standard state representation:

```
# For the standard state representation
>>> from EnvironmentNormal import EnvironmentNormal
>>> Environment = EnvironmentNormal()

# Perform policy evaluation
>>> Vpe = Environment.policyEvaluation( 0.8 )

# Perform policy iteration
>>> Vpi = Environment.policyIteration( 0.8 )

# Perform value iteration
>>> Vvi = Environment.valueIteration( 0.8 )

# Read out the varibles.
>>> Vpe[ (0, 0, 5, 5) ]
>>> Vpi[ (2, 3, 5, 4) ]
>>> Vvi[ (2, 10, 10, 0) ]
```

Type the following commands in your python interpreter to run the implementations individually in the reduced state representation:

```
# For the standard state representation
>>> from EnvironmentReduced import EnvironmentReduced
>>> Environment = EnvironmentReduced()

# Perform policy evaluation
>>> Vpe = Environment.policyEvaluation( 0.8 )

# Perform policy iteration
>>> Vpi = Environment.policyIteration( 0.8 )

# Perform value iteration
>>> Vvi = Environment.valueIteration( 0.8 )
```

[2]http://www.python.org/, note that on UNIX systems python is already available
[3]http://pygame.org

```
# Read out the variables.
>>> Vpe[ ( 5, 5 ) ]
>>> Vpi[ ( 3, 1 ) ]
>>> Vvi[ ( 2, 1 ) ]
```

4.2 Class Hierarchy

The class hierarchy can be seen in figure 2. There is an abstract class for the environment, which contains the algorithms for policy evaluation, policy improvement, policy iteration and value iteration. These algorithms are implemented so that they do not depend on the state representation. However, some subroutines of the algorithms do depend on the state representation. These methods need to be implemented by the child-classes.

There are two children of the abstract environment class, namely EnvironmentNormal and EnvironmentReduced. Each has a different state representation: EnvironmentNormal has the normal 4 dimensional state representation (two pair of coordinates), whereas EnvironmentReduced has a 2D representation (relative coordinates).

The environment contains an instance of the prey class and an instance of the predator class, which are used for simulation. Both predator and prey have a policy of their own.

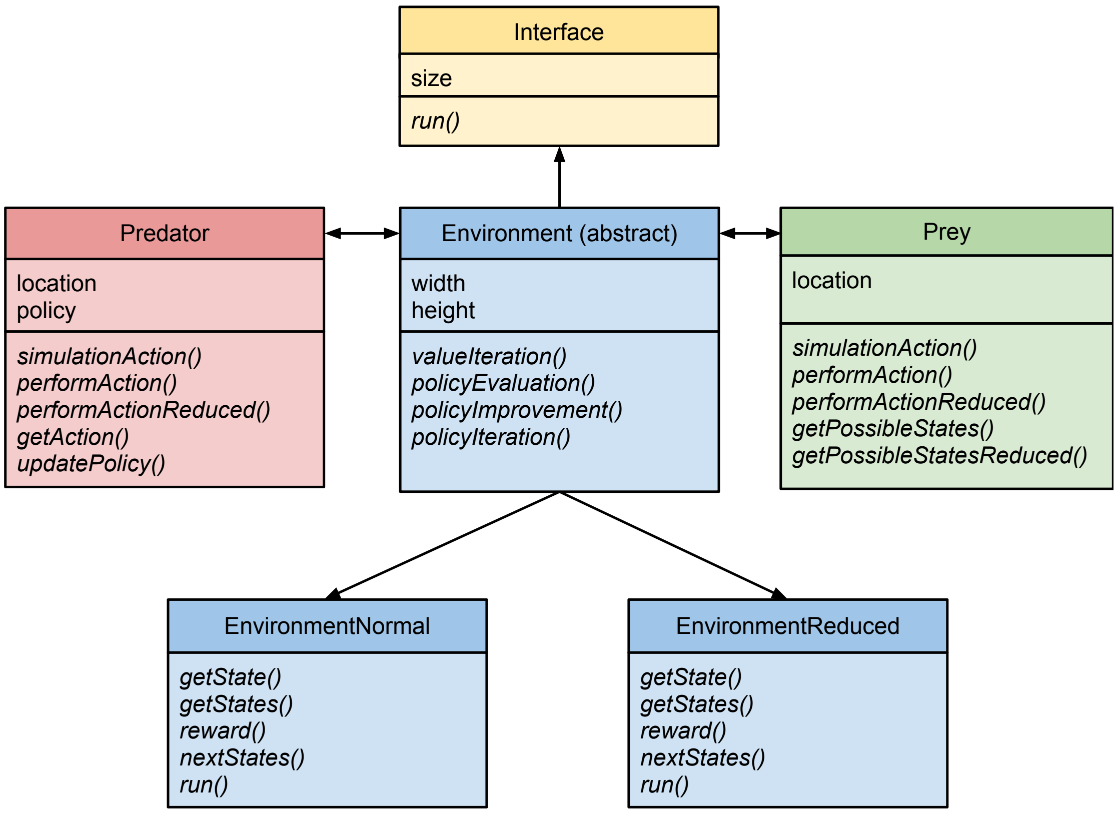

Figure 2: A graphical representation of the used class structure.

4.3 File Structure

This section describes the file structure of our implementation. The file structure consist of the following files:

Main.py This file is the main script that constructs a Interface object and runs the implementation.

Interface.py This file describes the Interface class, which is a visualization of the environment and is constructed for debugging purposes. This class runs the environment and represents it with a graphical visualization. The environment simulates the movements of the predator and the prey.

Environment.py This file describes the Environment class, an abstract class containing all the main variables and functions of the environment.

EnvironmentNormal.py This file describes the EnvironmentNormal class, a subclass of Environment describing the algorithms for the normal state representation.

EnvironmentReduced.py This file describes the EnvironmentReduced class, a subclass of Environment describing the algorithms for the reduced state representation.

Prey.py This file describes the Prey class that implements the prey of the MDP.

Predator.py This file describes the Predator class that implements the predator of the MDP.

5 Conclusion

In this assignment, multiple algorithms were used to find an optimal policy for an agent. The agent is in an environment in which it hunts for a prey that uses pseudo-random policy. Policy evaluation is used to determine the value of states for the agents policy. Value iteration and policy iteration are both employed to improve the policy. Results indicate that value iteration is the faster of the two algorithms for policy improvement. This is to be expected, as policy iteration has to perform policy evaluation (an iterative algorithm with relatively high computational cost) on every iteration. Policy improvement enables the predator to catch the prey in as few moves as possible.

Reduction of the statespace greatly decreases computational costs for both algorithms. Further reduction of the statespace is possible, although we did not implement it.

In further assignments, where multiple predators will catch a smarter prey, the current class structure can be upgraded to implement other learning and planning algorithms.

References

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). Reinforcement learning: An introduction.