Autonomous Agents

Assignment 3: Multi Agent Planning and Learning 21 October, 2012

By:
Paris Mavromoustakos
Georgios Methenitis
Patrick de Kok
Marios Tzakris

1 Introduction

In this last assignment, we added multiple predators to the environment, while making the prey learn. By doing so, it should be harder to catch. Our previous implementations already considered the prey to be an agent, but it only used a random policy. Because of our earlier design decisions we only made minor changes to the prey's functions, allowing it to use learning methods just as predator agents do. The prey starts off with a probabilistic policy, where each action has the same probability to be chosen. However, there is a 0.2 chance that the prey trips, and no action is performed. This tripping probability will overrule every policy; without this, it might always outrun the predator.

Another update is that the prey and predators now move simultaneously. First, we coded their moves asynchronously. Because the actions are handled in a single time-step, the agents can be next to each other and swap positions, without having them bump into eachother.

Last, we now consider this implementation to be a zero-sum Markov game, because when the predators (collectively) receive a reward, the prey receives the same reward, multiplied by -1. Predators receive a negative -10 reward if they crash into each other, while the prey receives a positive $+10$ reward in that case. Beside that, the predators will receive reward $+10$ for catching the prey. When this happens, the prey gets -10 for getting caught.

2 Exercise 1

In the first exercise, we use the 11×11 grid where we placed one prey and multiple predators. Our implementation requests the number of predators as input from the user, initializes the prey at position $< 5, 5 >$ and puts the predators in random positions.

The agents then move randomly on the grid until two predators move into the same position (Collision) or a predator catches the prey (Catch). These two are the only absorbing states. We should note that, if two predators collide and the prey is caught in the same time-step, the predators' collision is considered to have precedence and defines the reward distribution.

2.1 Results

Table 1 presents the results we had for the first part of the exercise. In the last column of the table we present the average number of needed time-steps for the predators to catch the prey. In this experiment, the predators and prey always act randomly. Adding more predators with a random policy to the environment reduces the number of steps needed to catch the prey but dramatically increases the number of collisions between predators. This is only logical, as there are more occupied squares, which an agent may enter.

Table 1: Multiple Agents' Random Implementation

Number of Predators	Total No. of Collisions	Total No. of Catches	Average Step No. until Catch
1	0	500	185
2	175	325	59
3	255	245	27
4	292	208	15

3 Exercise 2

For the second exercise, we have decided to implement both independent Q-learning, as well as minimax Q-learning.

3.1 Independent Q-Learning

In the first part of the second exercise, we made both prey and predators learning by implementing the *Q-Learning* algorithm on every agent. That means that all agents save and use their own $Q(s, a)$ table, from which they choose the next action using ϵ-greedy action selection.

Q-learning assumes the environment is stationary. Because other agents are modeled as part of the environment by Q-learning, we break this assumption. We are aware that it is not theoretically correct to use Q-learning in such a setting.

All predators now have their own learning ability which will lead them to choose actions aiming to their individual success. Considering the reward function discussed above, we expect a cooperative behavior among the predators, as all predators will get a positive reward in case a predator manages to catch the prey. Considering the prey, we only expect it to be smart enough so as to avoid getting caught without tripping. In this implementation, P_{trip} (the chance of the prey tripping) becomes important, as it allows the predators to move faster than the prey, thus getting a bigger chance to catch it. If this probability was not taken into consideration, the prey would always run away, especially when it competes against only one predator.

We have seen that independent *Q-Learning* has satisfying results in this multi-agent framework. However, the space complexity of this algorithm makes it computationally intractable in environments that there is a big number of agents. For example, in our 11×11 grid, adding four agents, means each agent is going to need to save up to $11^{2\times4} \times 5 = 1.071.794.405$ state-action pairs in its Q-table, given that each agent has $11^2 = 121$ possible positions, and thus states, and the selected agent has 5 actions. Table 2 presents how the number grows exponentially as the number of agents increases in our grid world. We were able to implement independent *Q-Learning* even for four agents, three predators and one prey, considering prey's position always at $< 5, 5 >$, and transform its action to predators' movement. Running the same algorithm for five agents was unfeasible, due to memory limitations of our laptops and the university computers..

Table 2: Multiple Agents' Random Implementation

Number of Agents	# State-Action
1	605
2	73.205
3	8.857.805
4	1.071.794.405
5	129.687.123.005
6	$1.569214188 \times 10^{13}$

3.1.1 Results

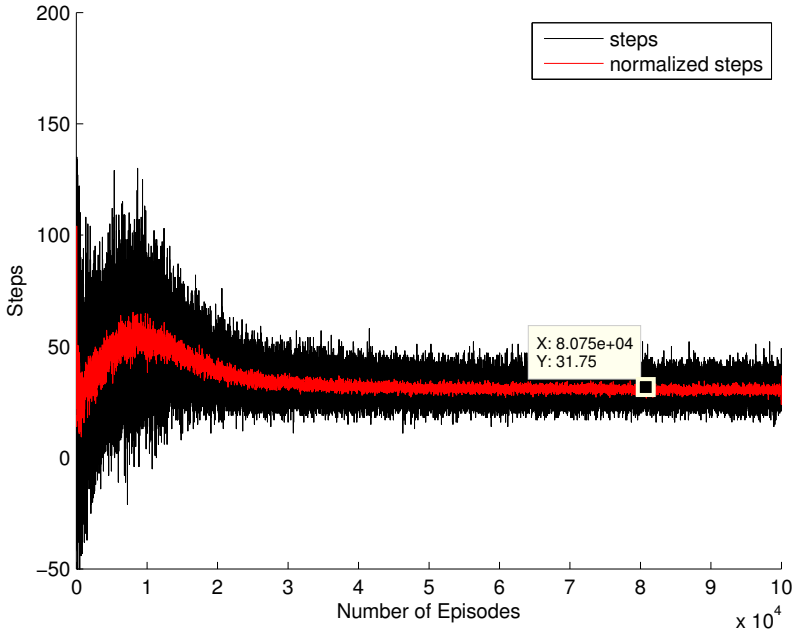

Figure 1: Performance of independent *Q-Learning* algorithm, multi-agent environment (2 predators, 1 prey), $\alpha = 0.7$. Episodes to converge ~ 3000.

In this section, we are going to illustrate through the following figures the performance[1]that independent learning had in this multi-agent system. Figure 1, 2, 3, and 4 illustrate our results in *independent Q-Learning* with a learning rate of $\alpha = 0.7$, $\alpha = 0.5$, $\alpha = 0.2$, respectively. In addition, the discount factor γ is always fixed at 0.7, as we have seen it to perform really well for single agent Q-Learning. We can figure out that, as α is decreased, independent *Q-Learning* needs more episodes to converge. All of the three versions converge almost at the same number of steps, indicating that an optimal policy has been found in all the three cases.

One more interesting thing that these figures indicate, is the negative number of steps. Every time that there is a collision between the predators, we are saving the number of steps multiplied by -1

[1]Normalized number of steps is a an average over the 20 neighbors for each episode. Given a sequence of steps for each episode $S(e)$, $s_{normalized}(e) = \frac{1}{20+1}\sum_{i=-10}^{10} s(e+i)$.

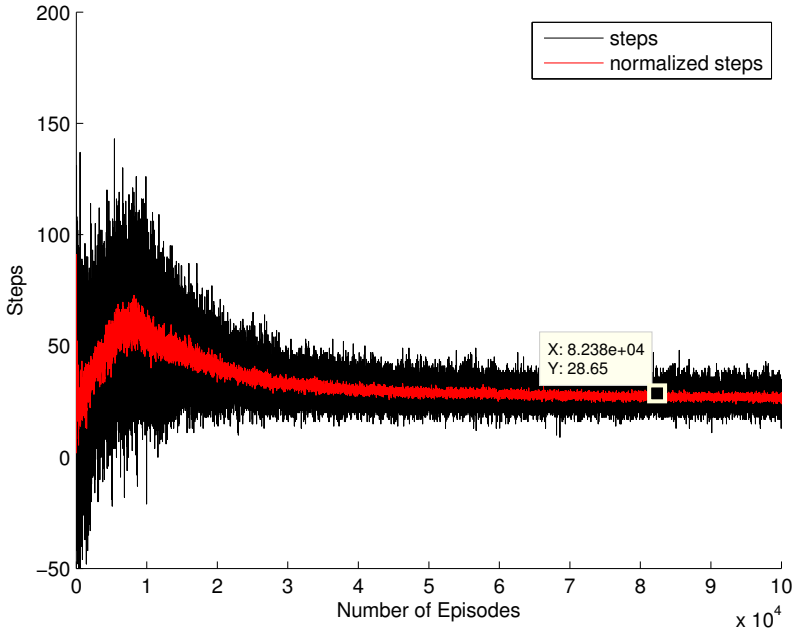

Figure 2: Performance of independent *Q-Learning* algorithm, multi-agent environment (2 predators, 1 prey), $\alpha = 0.5$. Episodes to converge ~ 5000.

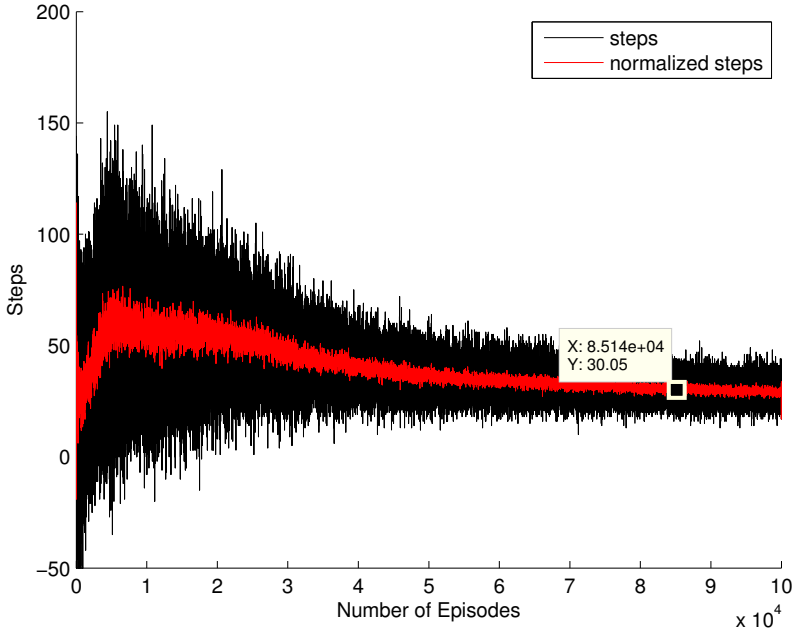

Figure 3: Performance of independent *Q-Learning* algorithm, multi-agent environment (2 predators, 1 prey), $\alpha = 0.2$. Episodes to converge ~ 8500.

to be able to clearly illustrate their learning process. At the beginning of predators' learning we can see that there is a random movement resulting in a big number of collisions between them. After one thousand episodes, the predators have learnt to avoid collisions and are focusing only on catching the

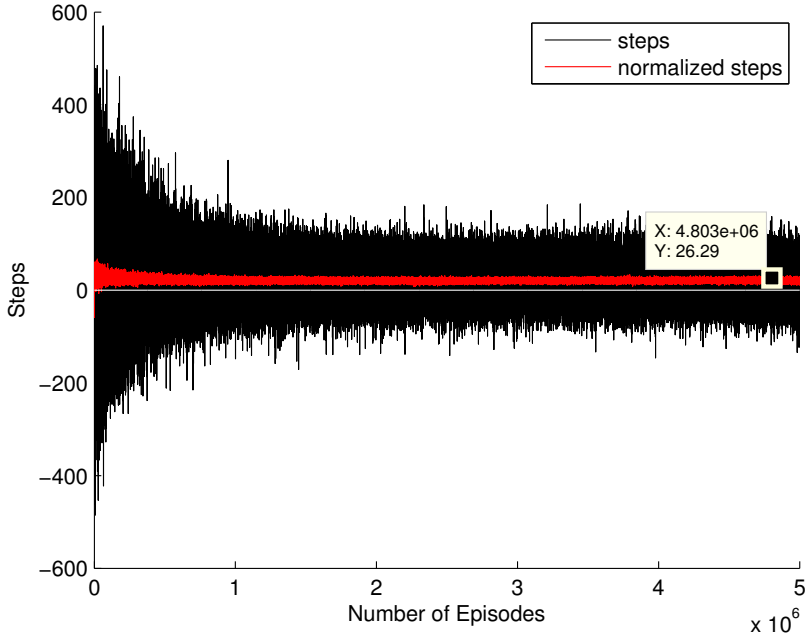

Figure 4: Performance of independent *Q-Learning* algorithm, multi-agent environment (3 predators, 1 prey), $\alpha = 0.2$. Episodes to converge $\gg 5 \times 10^6$.

prey as quick as possible. Lastly, the peak that appears in all figures indicates the transition from the exploring phase (predators are still testing "bad" moves and paths) to the exploiting phase, where predators have enough knowledge on how to avoid collisions and start choosing optimal paths towards the prey.

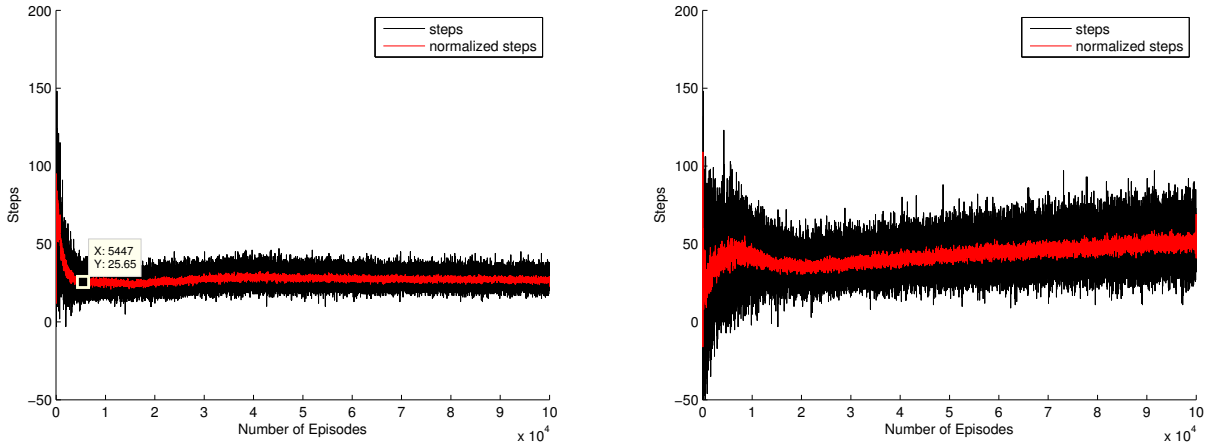

Figure 5: **Left:** Further testing of independent *Q-Learning* algorithm, multi-agent environment (2 predators, 1 prey), $\alpha = 0.7$, Q-initialization to zero value. **Right:** Further testing of independent *Q-Learning* algorithm, multi-agent environment (2 predators ($\alpha = 0.9$), 1 prey ($\alpha = 0.1$)).

We also tried some changes in the parameters of the independent *Q-Learning* algorithm in order to watch the agents' behavior under certain circumstances. Figure 5 presents these two tests. In the first case, each agent initialized its Q-table with zero values (pessimistic). As we expected, the algorithm

converged really fast, to an average of 25 steps, as the predators found an optimal policy fast enough without any need of further exploration. In the second case, the learning rate of the predators was significantly higher from the prey's. Predators learnt an optimal path really fast while the prey was still trying to improve its choices. We can see that after 1000 episodes, the average number of steps increases continuously, indicating that the prey has at that moment started to find an optimal path as well.

3.2 Minimax Q-Learning

In the second part of the first exercise we implemented the *minimax Q-Learning* algorithm, as described in the paper *"Markov games as a framework for multi-agent reinforcement learning" by M.L. Littman, 1994.*

3.2.1 Description

First, the algorithm discussed in that paper is only described in a zero-sum two player Markov game. The obvious goal of both agents is to maximize their expected sum of discounted rewards, just like in an MDP. However, this algorithm discriminates the two agents by introducing a single reward function $R(s, a, o)$, which each agent tries to maximize and the other one (opponent) tries to minimize. For example, when the game is in state s, the predator will consider the prey choosing the optimal action o which will minimize the expected reward of its opponent. As a consequence, the predator will choose the next possible action s that maximizes its reward, always considering the worst case scenario. On the other hand, the prey will consider the predator picking the next possible action that is optimal. It is assumed that in the minimax Q-learning algorithm each agent considers its opponent to be optimal (picking the optimal actions each time), and the agent defines its next action according to that assumption.

The *minimax Q-Learning* algorithm consists of three parts: Initialization, choosing the next action and learning. Each agent initializes its own $Q[s, a, o]$ table, with initial value, 1. For each possible state s, it also initializes the state value $V[s]$ table with initial value of 1, as well. The probability between all the possible actions $a \in A$ deriving from each state s is distributed equally, with $\forall a_i \in A : P(s, a_i) = 1/|A|$, and is also saved in the policy table, $pi[s, a]$. Learning rate α is also initialized as 1 and will descend as the steps in each episode goes on. That means, the agents learn based on their immediate past in the beginning of the algorithm, but they consider their past to become more important in the next stages as they do not learn anymore if learning rate becomes very low.

After the initialization of the data, each agent will have to choose its next action. That depends on a variable ϵ. Agents choose the action with maximum probability in the $pi[s, a]$ table with probability $1 - \epsilon$, but they choose a random action among the possible ones with probability ϵ. This is very similar to ϵ-greedy action selection, but this time, the action is chosen based on its probability of being chosen by the policy, and not on its expected reward. The learning part of the algorithm is described below:

- After receiving reward R_t for moving from state s to s' , via action a and opponent's action o,

$$Q_{t+1}[s, a, o] \leftarrow (1 - \alpha) \times Q_t[s, a, o] + \alpha \times (r + \gamma \times V[s'])$$

- We used linear programming to find $pi[s, .]$ such that:

$$pi[s, .] \leftarrow \arg\max_{pi'[s,.]} (\min_{o'} \sum_{a'} (pi[s, a'] \times Q[s, a', o']))$$

  In fact we need to maximize what our opponent agent wants to minimize. The maximization term goes to the probabilities of choosing our set of actions, given that our opponent is going to minimize it.

For the necessary linear programming part we used 6 variables. One is used to model the minimization of the opponent's expected return m, and the remaining 5 for our actions, p_1,p_2,...,p_5. Linear programming needs constraints in order to maximize the given linear combination of variables. Our main function to maximize was $f(m)$, and the constraints were:

  - $\sum_i p_i = 1.0$
  - $p_i \geq 0.0$
  - $\sum_{a'}(pi[s, a'] \times Q[s, a', o']) \geq m$, $\forall o' \in O$

- The value of the state s will be equal with the maximized value of variable from the linear program m: $V[s] = m$
- Learning rate will be always decayed in each update: $\alpha = \alpha \times decay$. $decay \in (0, 1)$ is the variable we use to reduce the value of α.

3.3 Results

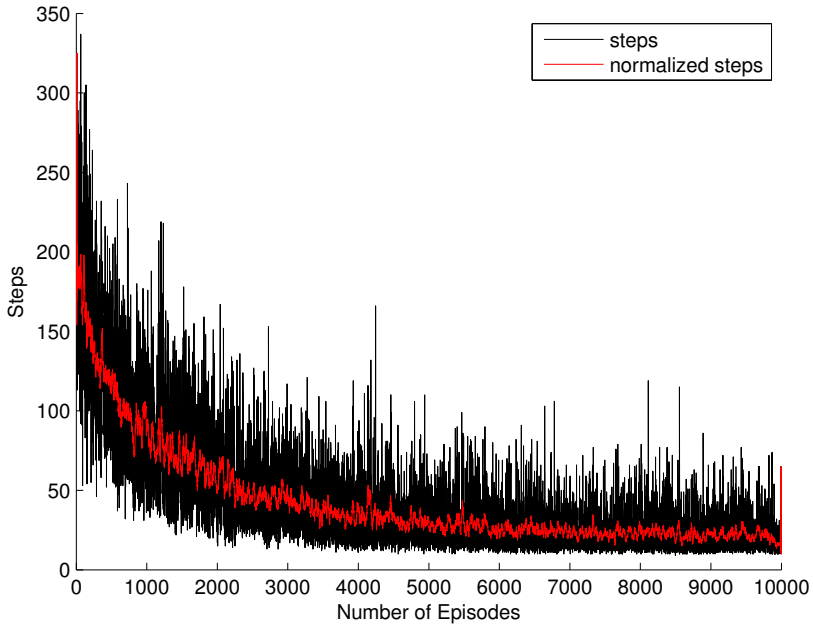

Figure 6: Performance of *Minimax Q-Learning* algorithm, multi-agent environment (1 predator(*Minimax Q-Learning*), 1 prey(random)), $\alpha = 0.5$, $\gamma = 0.5$. Episodes to converge ~ 8000.

First, we wanted to test our implementation of the *minimax Q-learning* algorithm. We have done so by using predators which learn according to this method, and a prey with a random policy. Figure 6 and Figure 7 shows the results we have in the approach with the random prey. We can see that the algorithm converges to an optimal policy for the predator, although, we see some peaks in the graph. This occurs because the main characteristic of the *minimax Q-learning* is that we always consider our opponent to be optimal. This means that, even though the predator was competing against a prey which moved randomly, it was choosing actions according to what the predator assumed to be the prey's optimal next possible action. This might lead the predator to non-optimal choices.

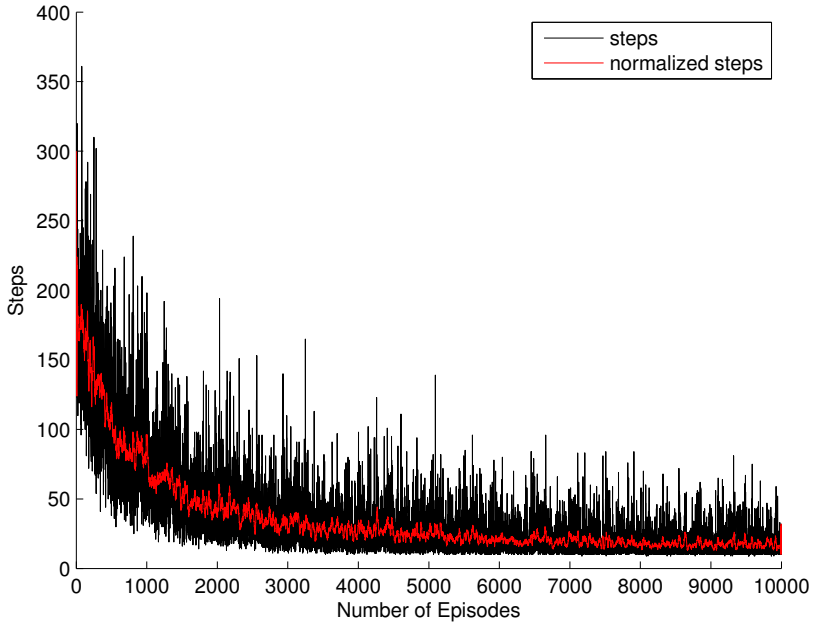

Figure 7: Performance of *Minimax Q-Learning* algorithm, multi-agent environment (1 predator(*Minimax Q-Learning*), 1 prey(random)), $\alpha = 0.7$, $\gamma = 0.7$. Episodes to converge ~ 6500.

Next, we did *minimax Q-learning* but now all agents utilize the learning method. Figure 8 presents our results of this experiment. The results are clearly unsatisfying even though we tried to make sure there are no bugs in the algorithm implementation that cause this behavior. It is obvious that the predator does not seem to converge to a faster (optimal) path. Both agents use the policy that it is generated by the linear program in each update, and both agents use random actions with probability $p_{random} = 0.1$. The red line indicates that more than 300 steps are needed for the predator in average to catch the prey. We expected to see better results, taking into consideration the fact that the prey's trip chance would make it easier for the predator to catch.

4 Exercise 3: WoLF-PHC

For the final part of the exercise we chose to implement *WoLF-Policy Hill Climbing*. This algorithm is described in detail in the slides of the Autonomous Agents course. In addition, the paper *"Efficient Learning in Games", by Raghav Aras, Alain Dutech and Francois Charpillet*, was really useful for us in order to implement this algorithm.

The general idea behind this algorithm is that each agent chooses its actions using the π-table with respect to the probability of each action. Next, each agent updates its Q-table:

$$Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

Next, each agent has to update its average policy $\bar{\pi}$. A counter is associated with every state, tracking the number of visits by the agent in the past. Using this counter, agents are able to update the value of their current state's policy for all possible actions:

$$\forall b : \bar{\pi}(s,b) \leftarrow \bar{\pi}(s,b) + \frac{1}{V}(\pi(s,b) - \bar{\pi}(s,b))$$

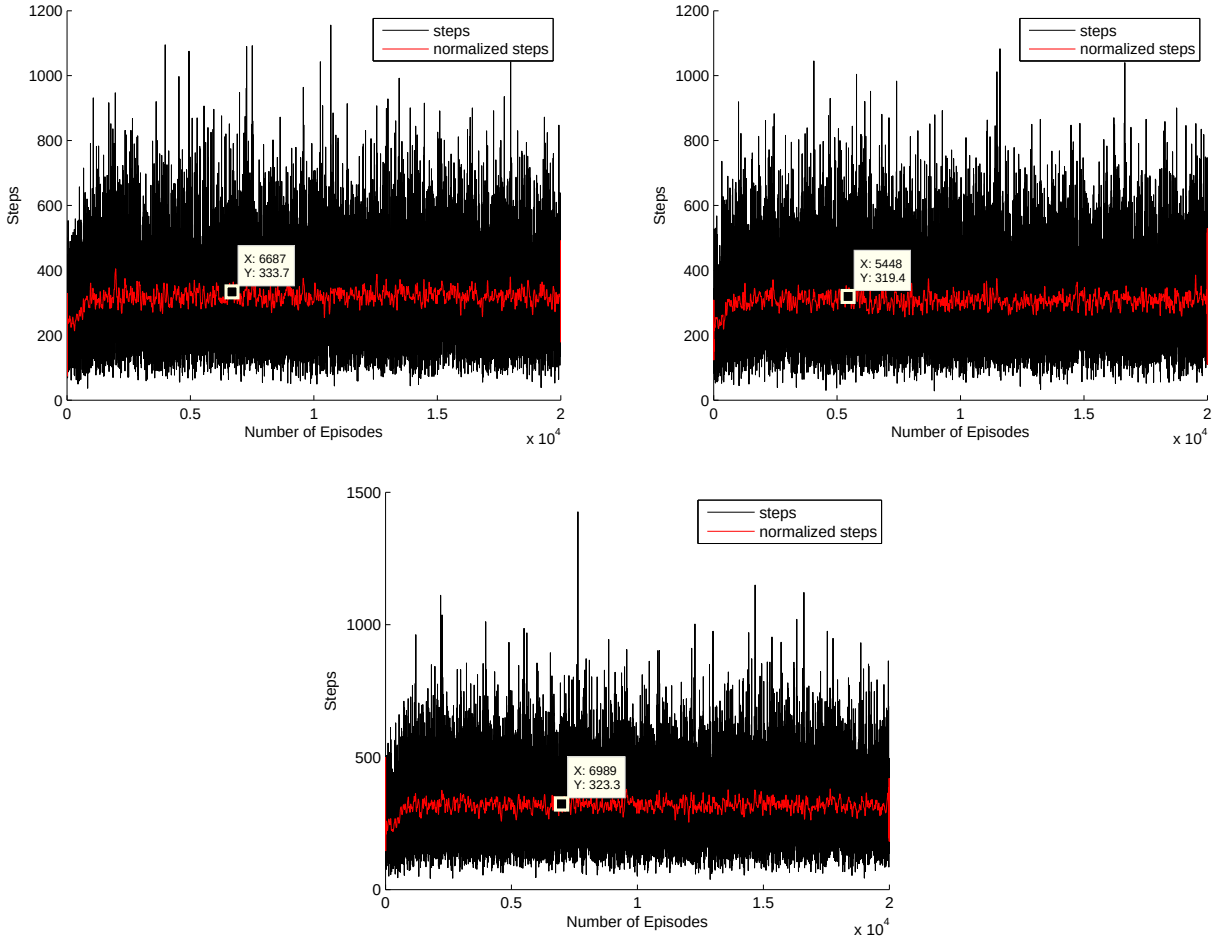

Figure 8: Performance of *Minimax Q-Learning* algorithm, multi-agent environment (1 predator(*Minimax Q-Learning*), 1 prey(*Minimax Q-Learning*)), **Left**: $\alpha = 0.5$, $\gamma = 0.7$, **Right**: $\alpha = 0.7$, $\gamma = 0.7$, **Middle**: $\alpha = 0.9$, $\gamma = 0.7$.

where V is the number of visits in this state. An important difference from the other learning algorithms is observed at the policy update. This algorithm uses δ_{win} and δ_{lose} to change the probability of choosing the next action. δ_{win} is used whenever the agent choose an action which has a higher value than the average policy, and δ_{lose} is used when $\pi(s,a) < \bar{\pi}(s,a)$. This unique way to update the policy being followed by the agent is the main characteristic of this algorithm.

4.1 Results

Figures 9, 10, 11 and 12 present the results we have found with *WoLF-PHC* learning algorithm. We can see four different implementations with different values for δ. Through testing we realize that the performance of this algorithm depends heavily on the ratio $\frac{\delta_{lose}}{\delta_{win}}$.

Then we took the best parameters from the above experiments, $\delta_{lose} = 0.3$, $\delta_{win} = 0.15$ and perform the same experiment for two predators and one prey. Figure 13 presents the results. *WoLF-PHC* converges to an average of 50 steps after two million episodes. It is clear that predators learn how not to collide with each other. Unfortunately, we were not able to perform the same test for more episodes due to long run-time of the algorithm, although, we believe that the learning process is depicted clear

enough in this figure.

4.1.1 Observations

- In Figure 9, we used a ratio $\frac{\delta_{lose}}{\delta_{win}} = 2.5$. Fast converge of the algorithm in an average of 76 steps.
- In Figure 10, we used a ratio $\frac{\delta_{lose}}{\delta_{win}} = 2$. Slower converge than the first implementation , in an average of 54 steps. Few random placed peaks up to 300 steps.
- In Figure 11, we used a ratio $\frac{\delta_{lose}}{\delta_{win}} = 3$. Slower converge than the first two implementations, in an average of 35 steps. More random placed peaks up to 900 steps.
- In Figure 12, we used a ratio $\frac{\delta_{lose}}{\delta_{win}} = 5$. Same converge with the previous implementation, in an average of 39 steps. Lots of random placed peaks up to 3000 steps (Are not shown in the figure due to scale equality of the four figures).

The first thing we want to mention is that the convergence speed of this learning algorithm mainly depends on the values we used for δ_{win}. As δ_{win} is decreased, the algorithm converged in less average steps. This happened because for every visited state each agent just adds a small value in the best policy causing the same effect the learning rate α has in pure *Q-Learning*.

Second, we have noticed an increasing number of peaks as the ratio $\frac{\delta_{lose}}{\delta_{win}}$ is going up. A big ratio means that agents are able to perform huge changes in their policies as δ_{lose} is now a lot bigger than the δ_{win}. One can imagine this to mean that the prey is always disappointed from its policy and tries to change it causing these "abnormal" peaks.

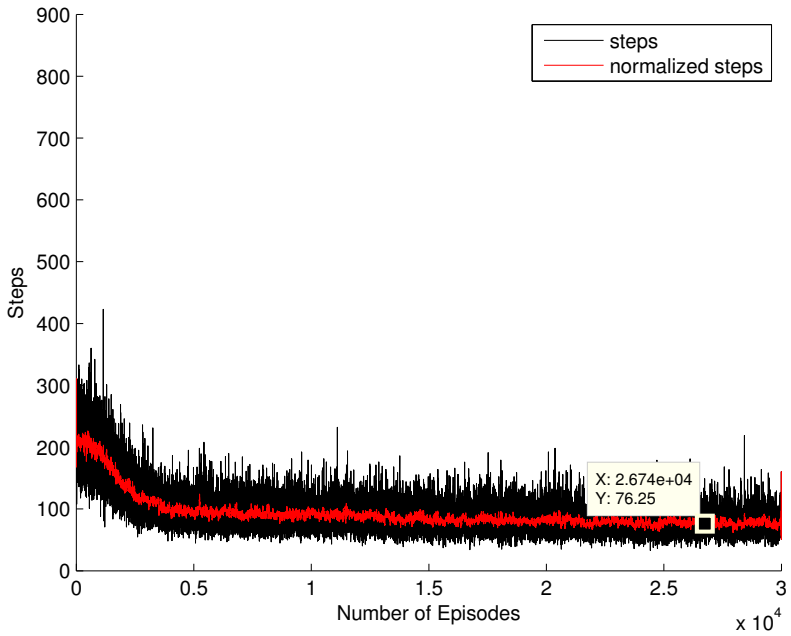

Figure 9: Performance of *WoLF-PHC* algorithm, multi-agent environment (1 predator(*WoLF-PHC*), 1 prey(*WoLF-PHC*)), $\alpha = 0.7$, $\gamma = 0.7$, $\delta_{lose} = 0.5$, $\delta_{win} = 0.2$.

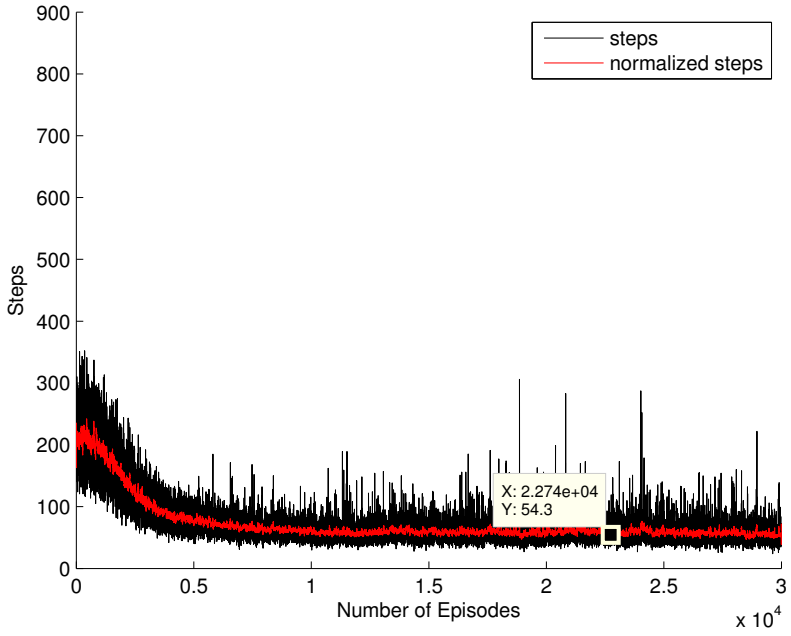

Figure 10: Performance of *WoLF-PHC* algorithm, multi-agent environment (1 predator(*WoLF-PHC*), 1 prey(*WoLF-PHC*)), $\alpha = 0.7$, $\gamma = 0.7$, $\delta_{lose} = 0.3$, $\delta_{win} = 0.15$.

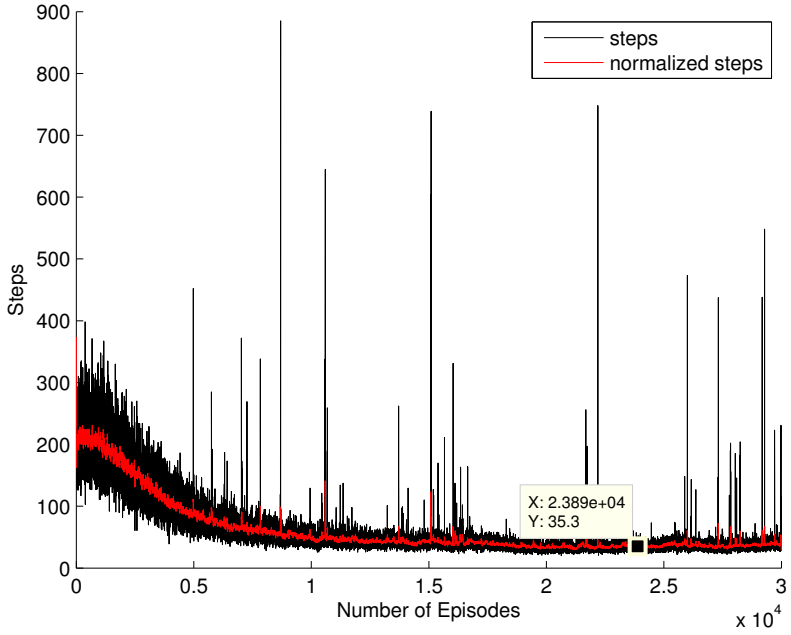

Figure 11: Performance of *WoLF-PHC* algorithm, multi-agent environment (1 predator(*WoLF-PHC*), 1 prey(*WoLF-PHC*)), $\alpha = 0.7$, $\gamma = 0.7$, $\delta_{lose} = 0.15$, $\delta_{win} = 0.05$.

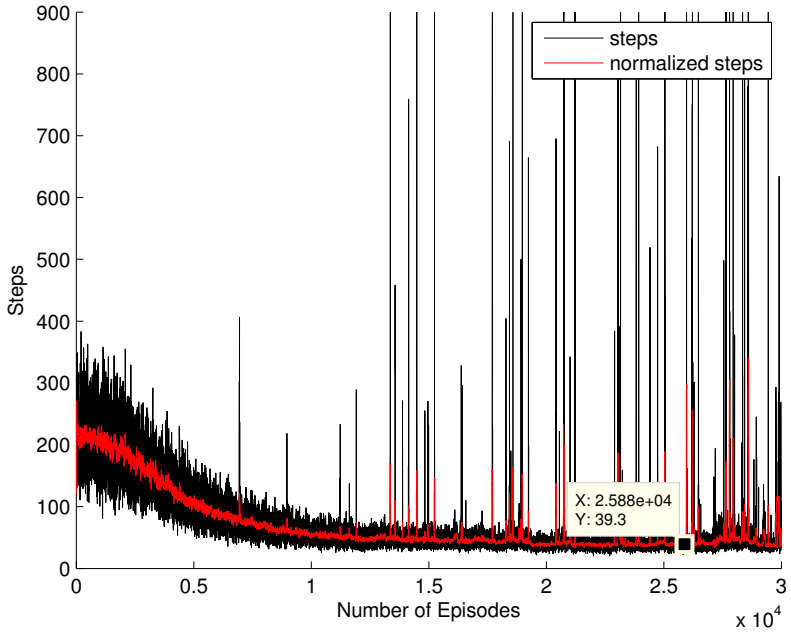

Figure 12: Performance of *WoLF-PHC* algorithm, multi-agent environment (1 predator(*WoLF-PHC*), 1 prey(*WoLF-PHC*)), $\alpha = 0.7$, $\gamma = 0.7$, $\delta_{lose} = 0.1$, $\delta_{win} = 0.02$.

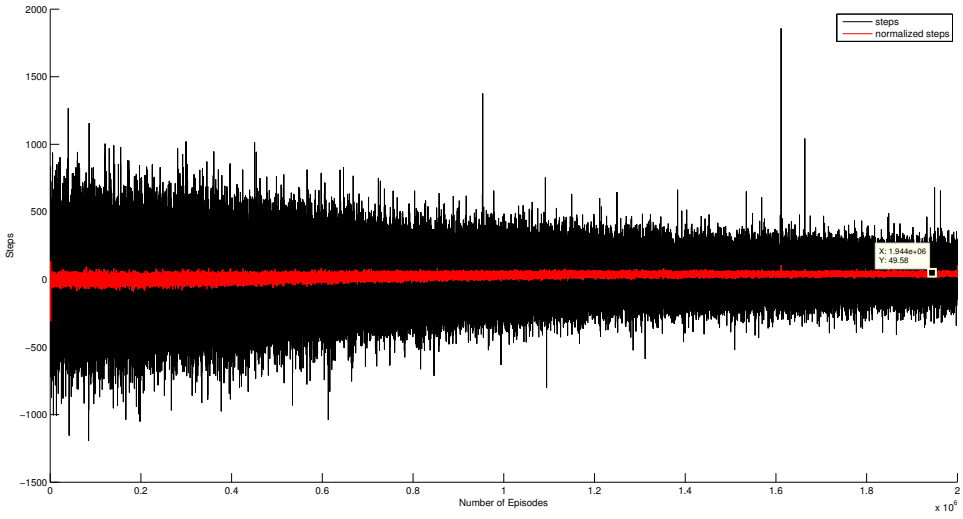

Figure 13: Performance of *WoLF-PHC* algorithm, multi-agent environment (2 predator(*WoLF-PHC*), 1 prey(*WoLF-PHC*)), $\alpha = 0.7$, $\gamma = 0.7$, $\delta_{lose} = 0.3$, $\delta_{win} = 0.15$.

5 Conclusion

In this last assignment of the Autonomous Agents course, we implemented and experimented on different learning algorithms that can be used for multi-agent frameworks. We understand how more than one agents can dramatically increase the space complexity of these specific algorithms, and, considering that fact, we understood the reason why planning and learning in multi-agent systems have triggered

a great research interest over the last decade.