Autonomous Agents

Assignment 3: Multi Agent Planning and Learning 19 October, 2012

By:
Paris Mavromoustakos
Georgios Methenitis
Patrick de Kok
Marios Tzakris

Introduction

In this last assignment, we added multiple predators to the environment, while making the prey intelligent, thus harder to catch. Our previous implementations already considered the prey to be an agent, for that reason we only added minor changes to our prey's functions, allowing it to use learning methods. The prey will now move equiprobably towards all directions, with a chance to trip (stay in the previous position), otherwise one predator would never be able to catch it.

What we also changed is that the prey and predator(s) move simultaneously, in a single timestep. That means, the agents can be next to eachother and swap positions, without considering their next move to be "safe" or not.

Lastly, we now consider this implementation to be a zero-sum markov game, because the predators will receive a -10 reward if they crash into eachother, while the prey will be receiving a +10 reward. Beside that, the predators will receive +10 for catching the prey which gets -10 for getting caught.

Exercise 1

In the first excercise, we use the 11x11 grid where we add one prey and multiple predators. The program 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). Those are the two possible and unique absorbing states. We should note that, if two predators collide and the prey is caught in the same timestep, the predators' collision is more "important" and defines the reward distribution.

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

As we can see in the table above, adding more predators to this particular implementation does reduce the number of steps needed to catch the prey but dramatically increases the number of collisions between predators.

Exercise 2

Independent Q learning

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

The predators will now learn to move towards the prey but the prey will also learn to avoid getting close to where the predators are. 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 chance to catch it. If this probability was not taken into consideration, the prey would always run away from the predators.

Minimax-Q Algorithm

In this section, we have implemented the minimax-Q algorithm, as described in the paper "Markov games as a framework for multi-agent reinforcement learning" by M.L. Littman, 1994.

Description of the algorithm

First of all, 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 one agent tries to maximize and its "opponent" tries to minimize. So, for example, when the game is in state s, the predator will consider the prey picking the next possible action o that is optimal for itself, thus worst for its opponent. As a consequence, the predator will choose the next possible action s that maximizes its reward, always considering the worst case scenario. But on the other hand, the prey will consider the predator picking the next possible action that is optimal. Generally, it is a fact that in the minimax-Q algorithm each agent considers its opponent to be optimal (picking the optimal actions each time), and defines its next action according to that.

Implementation of the algorithm

The algorithm implementation consists of three parts: Initialisation, choosing the next action and learning.

Minimax-Q learning initializes a $Q[s, a, o]$ table for each agent, with initial value = 1. For each possible state s, it also initialises a $V[s]$ table with initial value = 1 aswell. The probability between all the possible actions $a \in A$ deriving from each state s is distributed equally, with $P(a|s) = 1/|A|$, and is also saved in a probabilities table, $pi[s, a]$. Also, learning rate α is also initialised = 1 and will descend as time goes by. That means, the agents will learn based on their immediate past in the beginning of the algorithm, but will consider their past to become more important in the next stages.

After the initialization of the data, each agent will have to choose its next action. That depends on a variable ϵ, so as, the agent will choose the action with maximum probability in the $pi[s, a]$ table with probability $1 - \epsilon$, but will 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 and not its expected reward.

The learning (and last) part of the algorithm will be described in pseudocode, as written in the paper:

- After receiving reward rew for moving from state s to s' , via action a and opponent's action o,
- Let $Q[s, a, o] = (1 - \alpha) \times Q[s, a, o] + \alpha \times (rew + \gamma \times V[s'])$

- Use linear programming to find $pi[s, .]$ such that $pi[s, .] = argmax(pi'[s, .], min(o', \sum a', pi[s, a'] \times Q[s, a', o']))$
- Let $V[s] = min(o', \sum a', pi[s, a'] \times Q[s, a', o'])$
- Let $\alpha = \alpha \times decay$

  Note that $decay \in (0, 1)$ is the variable we use to reduce the value of α.

Linear Programming

WOLF

1 Conclusion