Multi-Agent Systems and Distributed AI
Lab course Assignment3

Reinforcement Learning

Roberto Valenti *0493198*
Felix Hageloh *0425257*

Introduction

In this assignment we implemented two different learning algorithms for software agents. Both are aimed to learn an optimal policy in a Markov Decision Process. The methods are Value Iteration and Q-learning. The learning methods were implemented for the pursuit environment with one predator and one prey for Value Iteration, and two predators and one prey for Q-learning.

Theory

Value Iteration

During a Markov Decision Process an agent is allowed to take consecutive actions a_t, a_{t+1}… and receives from the environment an immediate reward or reinforcement for the resulting states s_t, s_{t+1}… . Clearly an agent will want to maximize this reward for all future states also referred to as total future reward. This total future reward obviously depends on the policy of the agent, and an optimal policy requires and optimal utility function. This optimal utility function again depends on the policy of the agent, hence when defining the optimal utility function U*(s) we get a recursive definition, as follows:

$$U^*(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U^*(s'). \quad (1)$$

Where R(s) is the reward function that assigns a reward to any state s. S is the current state and a the current action, while s' is the next state.
The goal of value iteration is to learn this optimal utility function. This is done by replacing the last equation with an assignment, which solves the problem of this recursive definition:

$$U(s) := R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s'). \quad (2)$$

The states in our scenario are the relative positions of the predator to the prey and we initialize U(s) for all possible states to a random number. When applying (2) iteratively it has been proven that U(s) converges to its optimal version U*(s). To ensure convergence we introduce a discount rate [gamma] between 0 and 1.
We can apply this function because we know P(s'|s, a) which is the transition model, namely we know that the prey has 0.2 probability to move in either direction, and 0.2 probability to stand still. Also we know that if the predator is next to the prey, the prey will not move on top of the predator and thus the number of moves decreases and the probabilities do accordingly. Moreover, we know all possible actions for a predator and their effect on the world state.
For the reward function R(s) we can apply simple rues, namely the reward is 1 when the predator catches the prey (s is 0,0) and -0.1 for all other states where doesn't catch the prey. Obviously the reward for catching the prey "encourages" the predator to move towards the prey (and catch it) while the negative reward in all other states "discourages" the predator to take detours when moving to the prey.

Q-learning

Since in this scenario we don't know the state transition model (we don't make any assumptions about the movements of the other predator) we can not use the definition of the optimal utility function. Instead we use Q learning which estimates the optimal action value function Q*(s,a) with

$$Q(s,a) := (1-\lambda)Q(s,a) + \lambda\big[R + \gamma \max_{a'} Q(s',a')\big] \quad (3)$$

Implementation

Naturally our implementation is in Java and uses Policy class to the define an agents policy for both learning methods. The policy class incorporates either the optimal utility function U*(s) or the optimal state action pairs Q*(s,a).

Value Iteration

Since the state space in this scenario is quite small we can easily store the entire utility function in memory. Since a state consists of two numbers (relative x position of the predator and relative y position) we can represent U(s) as a 2d matrix. This implemented in java as a 2D array of doubles, and thus each entry represents the utility value, and the coordinates of the array represent the state. Obviously there needs to be some mapping between the world state and the actual coordinates, since in the world state 0,0 corresponds to the center (the prey) while in the array coordinates this is the lower left entry in the matrix.
We then implemented the learning function `valueIteration()` that for each state uses the definition in (2) to calculate the new utility value, an repeats this process until the maximum change in any value is less than 0.001. The discount rate, gamma, can be set as global variable in policy. Furthermore we had to implement a reward function which exactly follows the definition in the theory part, namely the reward is 1 for the world state 0,0 and -0.1 otherwise.
We call this function `valueIteration()` when the agent is initialized. Then after the optimal util function is learned we can start the simulation. During the simulation we choose the next action of the agent using the `maxAct()` function, which implements and optimal policy. For each possible action we sum over the utilities of the possible next states, weighted with their probabilities. Then we pick the action with the biggest sum and return it. Possible next states are determined using the `nextStates()` method that takes into account the effects of the current action plus the possible movements of the prey. It the returns an array of possible states together with their probabilities.

Q-Learning

Like for value iteration we store all Q values in a matrix, but since our state consists of four numbers now (predator x,y and prey x,y) we need a 4D one. Since we use Q-Learning only in cases were the prey is at least 3 squares away, our matrix is of size 7 in all dimensions. However, this doesn't cover one special case: the prey might be at most 3 squares away, but the other predator might be further away. Thus we increased the last dimension to 8 and used the entry [prey x][prey y][0][8] to store the Q value for this special case.
We then initialize all values to a random number. For learning we implemented the method qLearnStep that uses the definition in (3) to update the corresponding entries of the Q matrix. Since this function requires the current and the next sate, we save the current state inside the Predator class. We then make the next step and observe the world state and call the qLearnStep method with this new state and the previously saved state. The reward that an agent receives is now not a function of the state anymore, but simply an observation that the agent makes. In

Pursuit these observations are implemented as events and so the reward method takes as an input the current event and maps the following rewards to the following events:

- COLLISION. Reward = -1
- PENALTY: Reward = -0.2
- CYCLE END. Reward = 1
- NO EVENT. Reward = -0.1

These values were obtained experimentally, which is describe in the next section. However, we can say that in general we want to discourage collisions and penalties, and encourage captures. Like before we want to also discouraged predators from taking detours or doing nothing, thus we also "punish" if no event occurs. However, this resulted in some problems, because essentially agents get punished whenever moving within three squares of the prey. Outside this zone the old policy is intact and thus no rewards or punishments are given. This way the agents learn very quickly to move outside this 3 square zone when they are just on the border of it. This is because the Q value for this action is the highest since no punishment was ever given for it. Thus we have to introduce another event, which corresponds to an agent stepping out of the 3 square zone and also punish it. Since the resulting next state has no Q value attached to it we had no choice but skip this part of the formula so the Q value gets updated as follows: $Q(s,a) = (1-\lambda)\, Q(s,a) + \lambda(R + \gamma)$. While this seem questionable it helped to improve the results drastically

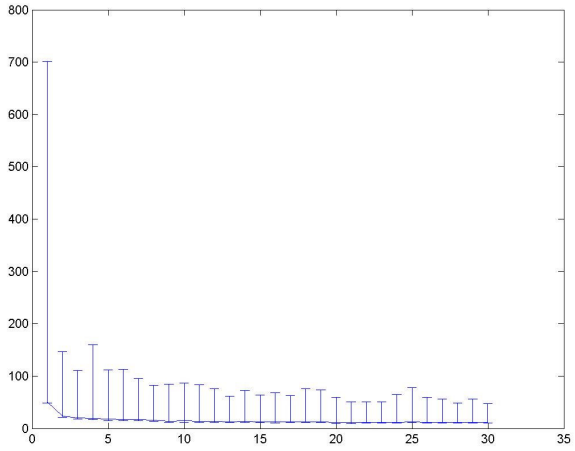
Results

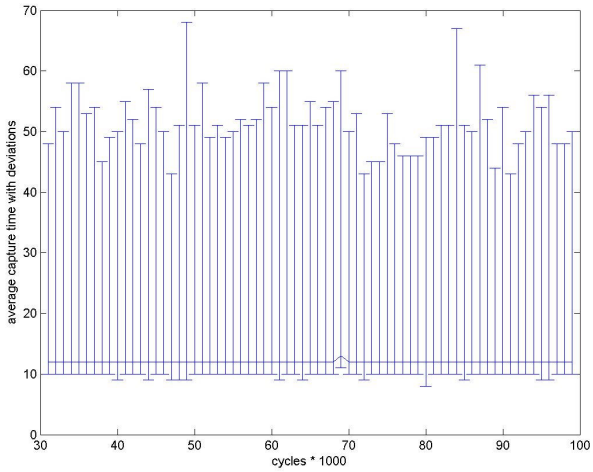
Value Iteration

The resulting average capture time after value iteration is 10.8 in 5000 cycles. Compared to the first assignment this is not a major improvement and analyzing the predators movements we can observe that it is not moving optimally all the times (it is not moving diagonal to the prey). We could not find a way to improve this and had to give up due to time constraints.

Q-Learning

As mentioned earlier, we had to experiment with different reward values to achieve optimal results. The ones that are listed in the last section are the ones that caused the best performance. The following graphs give an overview of the capture times at different stages in the learning process. The runs used $\lambda = 0.9$ and $\gamma = 0.9$:





The first graoh shows capture times for the first 30 000 cycles and the next for the last 70 000 cycles. Each data point represents the average of 1000 cycles. The bars above and beneath indicate the maximal and minimal capture time during those 1000 cycles. We can see that the algorithm converges after about 25 000 cycles.

Conclusion

Our implementation of iterative earning gave good but not optimal results and we had no time to investigate the causes of this.
Q-Learning on the other hand gave very good results, converging to an average capture time of 12 cycles. However, the algorithm took a long time to converge and we did not make use of an exploration strategy to ensure that we don't end up in a local minimum. Again this could not be fixed due to time constraints but we believe that our results are not from a local minimum.