Single Agent Learning

Lab Assignment 2
Autonomous Agents
Master Artificial Intelligence
University of Amsterdam

Camiel Verschoor	Steven Laan	Auke Wiggers
StudentID: 10017321	StudentID: 6036031	StudentID: 6036163
UvAnetID: 6229298	UvAnetID: 6036031	UvAnetID: 6036163
Verschoor@uva.nl	S.Laan@uva.nl	Auke.Wiggers@uva.nl

October 5, 2012

1 Introduction

In this report, we will describe an implementation of several learning algorithms in the predator versus prey environment, which is a 11 by 11 toroidal grid (see figure 1). This 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 goal of the agent is, as was the case in the last assignment, catching the prey in as few moves as possible.

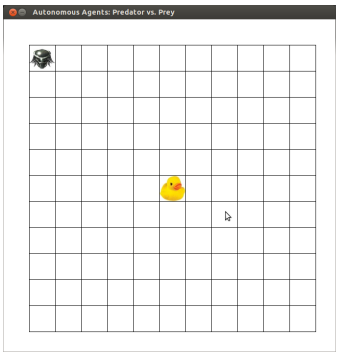
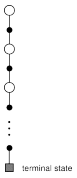


Figure 1: The 11 by 11 toroidal grid of the predator versus prey world.

In contrast to the previous assignment, transition and reward functions are not known to the agent. Only experience and observations can be used to plan optimally (that is, catching the prey in as few moves as possible). This report will focus on various learning algorithms that can learn from these experiences and observations.

(a) Backup diagram of the Q-learning algorithm.

(b) Backup diagram of SARSA.



(c) Backup diagram of on-policy Monte Carlo.

Figure 2: Backup diagrams for the implemented algorithms.

2 Method

This section describes the methods that are applied in this report. The following methods will be discussed:

- Q-Learning
- SARSA
- On-policy Monte Carlo Control

2.1 Q-learning

The goal of Q-Learning is to obtain the value of combinations of a state-action pair, based on observations. The algorithm for Q-Learning is given in algorithm 1, as specified by Sutton and Barto [Sutton and Barto, 1998]. Its backup digram, shown in figure 2a, shows that an action, the state following that action and the maximization over next possible actions are involved.

Algorithm 1 Q-Learning

Initialize $Q(s, a)$ arbitrarily
repeat forever
  Initialize s
  repeat
    Choose a from s using policy derived from Q (e.g., ϵ-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  until s is terminal

2.2 SARSA

SARSA is an algorithm that is comparable to Q-learning, as it also aims to obtain the value of state-action pairs. The main difference with Q-learning is that instead of choosing the action a' that maximizes $Q(s', a')$, a policy is derived from Q and an action is selected based on that policy. This is shown in the backup diagram in figure 2b.

Algorithm 2 SARSA

Initialize $Q(s, a)$ arbitrarily
repeat forever
  Initialize s
  Choose a from s using policy derived from Q (e.g., ϵ-greedy)
  repeat
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $a \leftarrow a'$
    $s \leftarrow s'$
  until s is terminal

2.3 On-policy Monte Carlo Control

The On-Policy Monte Carlo Control (OPMCC) is different from both SARSA and Q-Learning in the sense that it uses one whole episode per iteration. This is illustrated in figure 2c. In algorithm 3, the pseudocode for On-Policy Monte Carlo Control can be seen. The algorithm is on-policy, meaning the policy of the agent is changed between episodes. To ensure that exploration never stops, an ϵ-greedy policy is used. While the algorithm does not need exploring starts, their influence on the performance was tested.

Algorithm 3 On policy Monte Carlo Control

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s, a) \leftarrow$ arbitrary
  $Returns(s, a) \leftarrow$ empty list
  $\pi \leftarrow$ an arbitrary ϵ-soft policy
repeat forever
  (a) Generate an episode using π
  (b) For each pair s, a appearing in the episode:
    $R \leftarrow$ return following the first occurrence of s, a
    Append R to $Returns(s, a)$
    $Q(s, a) \leftarrow$ average($Returns(s, a)$)
  (c) For each s in the episode:
    $a^* \leftarrow \arg\max_a Q(s, a)$
    For all $a \in \mathcal{A}(s)$:

$$\pi(s, a) \leftarrow \begin{cases} 1 - \epsilon + /|\mathcal{A}(s)| & \text{if } a = a^* \\ \epsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

3 Results

In this section, the influence of different settings of the parameters are shown and evaluated.

3.1 Q-learning

The Q-learning algorithm was tested using different initialization of variables, such as the discount factor, learning rate, and the optimistic initialization of Q.

3.1.1 Initialization

The effect of the optimistic initialization on the learning performance, that is the starting value of Q: $Q_0(s) = c$. We tested three different settings for c while keeping the rest of the parameters fixed. For a learning rate $\alpha = 0.1$, discount $\gamma = 0.7$, the performance of the agent is shown in figure 3. The results indicate that starting with a lower optimistic value will result in better performance early on, but will, due to the maximization term in the algorithm, lead to fewer exploratory actions. However, since the plots are smoothed, and only contain averaged values, the peaks that occur due to insufficient exploration are not seen.

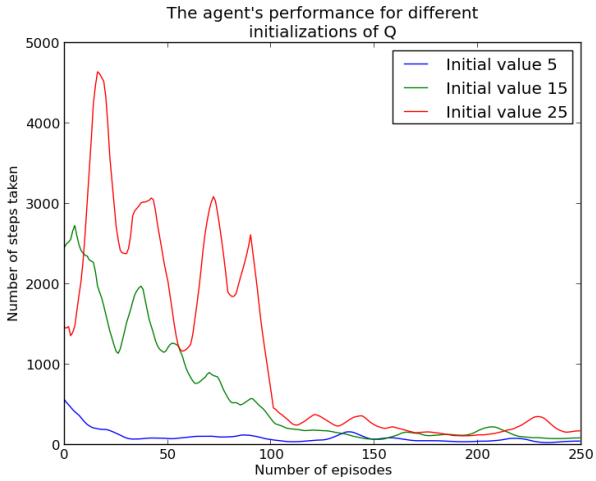
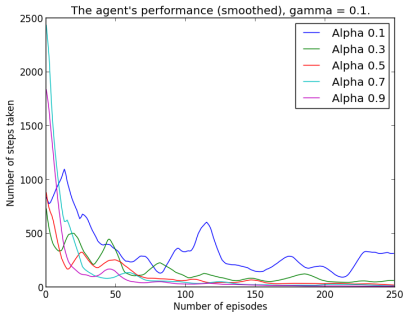


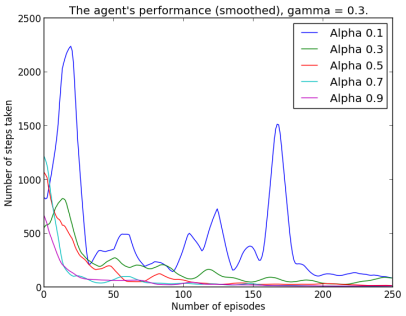
Figure 3: Performance of the agent plotted against the number of episodes, for different optimistic initializations of Q.

3.1.2 Learning rate and discount

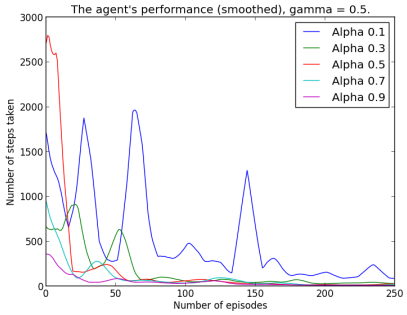
Altering the learning rate α and discount factor γ has also shown to influence performance (see figure 4). As is to be expected, a higher learning rate rapidly lowers the total number of steps the agent takes in a single episode. A low discount also causes the agent to reach the prey as fast as possible, as long episodes will become worthless faster than with a high discount factor.



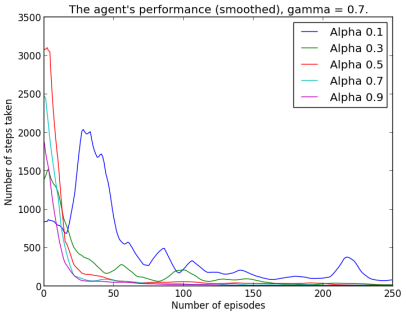
(a) Performance of the agent plotted against the number of episodes, for a discount of 0.1.



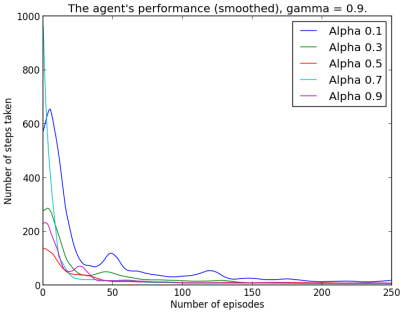
(b) Performance of the agent plotted against the number of episodes, for a discount of 0.3.



(c) Performance of the agent plotted against the number of episodes, for a discount of 0.5.



(d) Performance of the agent plotted against the number of episodes, for a discount of 0.7.

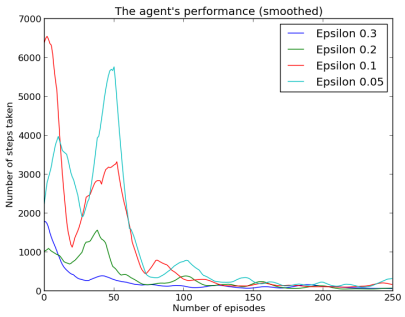


(e) Performance of the agent plotted against the number of episodes, for a discount of 0.9.

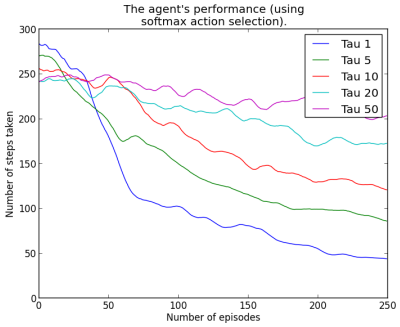
Figure 4: Plots of performance for different discount factors and learning rates. Results were found by simulating the agent and environment 100 times after each episode (when a new estimate for Q was found).

3.1.3 Greediness

The agents performance was also tested for different ϵ. Setting ϵ to a large number would cause the agent to take exploratory actions with a fairly high probability, with bad performance as a result. Setting it too low, however, will result in insufficient exploring, The setting of the ϵ for the ϵ-greedy policy. See figure 5a for the results. Tests were performed for a learning rate $\alpha = 0.1$, discount $\gamma = 0.7$.



(a) Performance of the agent plotted against the number of episodes, using Q-learning and ϵ-greedy action selection.



(b) Performance of the agent plotted against the number of episodes, using softmax action selection.

Figure 5: Performance plots for Q-learning, showing the effect of altering different parameters.

3.1.4 Softmax

Softmax action selection was implemented according to the explanation in chapter 2.3 from [Sutton and Barto, 1998]. The performance of the agent was plotted for different temperature τ, shown in figure 5b. The probability of an action, determined by softmax, is calculated using equation 1. Results indicate that, as temperature τ approaches zero, performance improves faster than for a high τ (figure 5b). This is to be expected: In the limit, where $\tau \to 0$, the softmax action selection will be equal to the greedy policy and thus the agent will always prefer actions that maximize return over exploratory actions. However, setting τ at such a low number may cause bad performance, as there may not have been sufficient exploration.

$$p(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}} \tag{1}$$

3.2 SARSA

The same experiments were conducted for the implementation of the SARSA algorithm.

3.2.1 Initialization

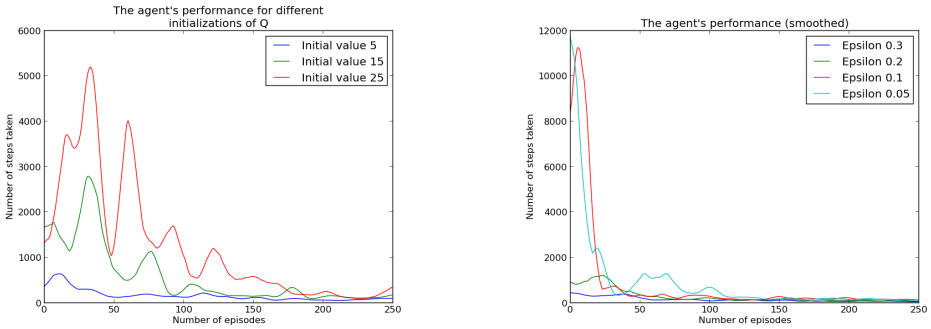
The optimistic initialization of Q influenced the results for SARSA, similar to Q-learning, as can be seen in figure 6a.

3.2.2 Learning rate and discount factor

The same performance tests were conducted for SARSA as for Q-learning. Results can be found in figure 7.

3.2.3 Greediness

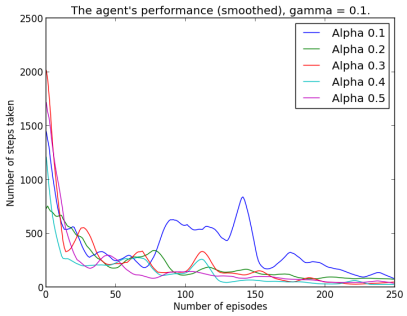
Results for different ϵ are displayed in figure 6b.



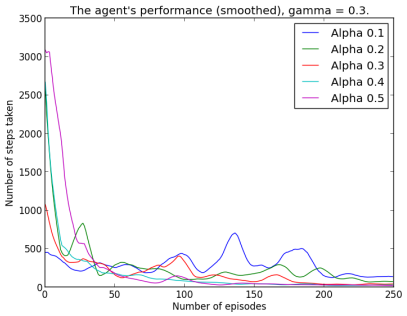
(a) Performance of the agent plotted against the number of episodes, using SARSA.

(b) Performance of the agent plotted against the number of episodes, using SARSA.

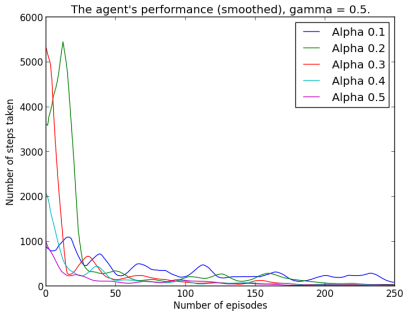
Figure 6: Performance plots for SARSA, showing the effect of altering different parameters.



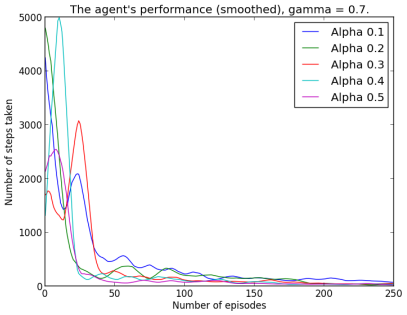
(a) Performance of the agent plotted against the number of episodes, for a discount of 0.1.



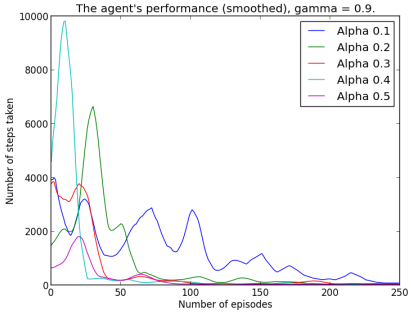
(b) Performance of the agent plotted against the number of episodes, for a discount of 0.3.



(c) Performance of the agent plotted against the number of episodes, for a discount of 0.5.



(d) Performance of the agent plotted against the number of episodes, for a discount of 0.7.



(e) Performance of the agent plotted against the number of episodes, for a discount of 0.9.

Figure 7: Plots of performance for different discount factors and learning rates. Results were found by simulating the agent and environment 100 times after each episode (when a new estimate of Q was found).

3.3 On-policy Monte-Carlo Control

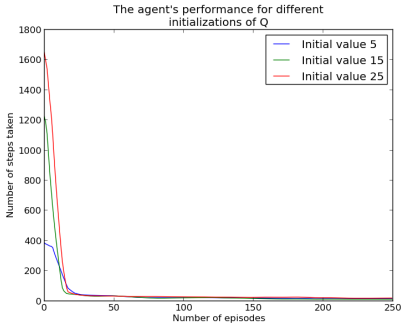
Similar experiments were conducted for the implementation of on-policy Monte Carlo. The main difference is that Monte Carlo does not use a learning rate α.

3.3.1 Initialization

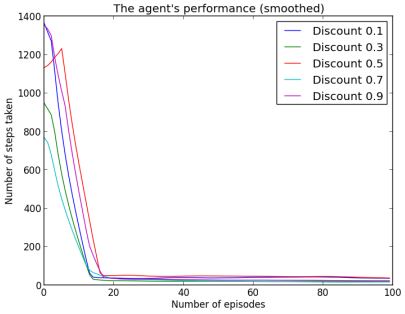
The results for on-policy Monte Carlo, for different optimistic initializations of Q, are given in figure 8a. Tests were performed for a learning rate $\alpha = 0.1$, discount $\gamma = 0.7$.

3.3.2 Discount factor

The performance of the agent, plotted against the number of episodes, can be found in figure 8b. This plot shows that the discount factor γ is almost insignificant in the predator-prey task, as each of the discount factors causes the state-action values to converge before the twentieth episode.



(a) Performance of the agent plotted against the number of episodes, using on-policy Monte Carlo.



(b) Performance of the agent plotted against the number of episodes, using on-policy Monte Carlo.

Figure 8: Performance of the agent, using on-policy Monte Carlo control. Influence of different parameters is shown in these plots.

4 Implementation

4.1 Usage Guide

In order to run all our software successfully the following two programs should be installed:

- Python 2.7 interpreter[1].
- Pygame library[2], to run our visualization.
- Numerical Python[3], to enable the use of matrices and corresponding functions.
- Matplotlib[4], to plot the performance of the agent.

Type the following command in the terminal to launch our main program, which shows the implementations of the algorithms in action:

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

Type the following commands in your python interpreter to run the implementations of the algorithms individually:

```
# Initialization of variables and classes.
>>> from EnvironmentReduced import EnvironmentReduced
>>> episodes = 1000
>>> alpha = 0.5
>>> discount = 0.1
>>> epsilon = 0.1

# Q-learning with epsilon-greedy action selection
>>> Environment = EnvironmentReduced()
>>> Q, Return_list = Environment.Predator.qLearning(episodes, alpha, gamma, epsilon)
# Q[(0,1)] contains the value of action (0,0) (standing still) in state (0,1).
>>> Q[(0,1)][(0,0)]

# Q-learning with softmax.
>>> Environment = EnvironmentReduced()
>>> Q, Return_list = Environment.Predator.qLearning(episodes, alpha, gamma, epsilon, False)

# SARSA
>>> Environment = EnvironmentReduced()
>>> Q, Return_list = Environment.Predator.SARSA(episodes, alpha, gamma, epsilon)

# On-policy Monte Carlo.
>>> Environment = EnvironmentReduced()
>>> Q, Return_list = Environment.Predator.onPolicyMonteCarlo(episodes, epsilon, gamma)
```

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

4.2 Class Hierarchy

The class hierarchy can be seen in figure 9.

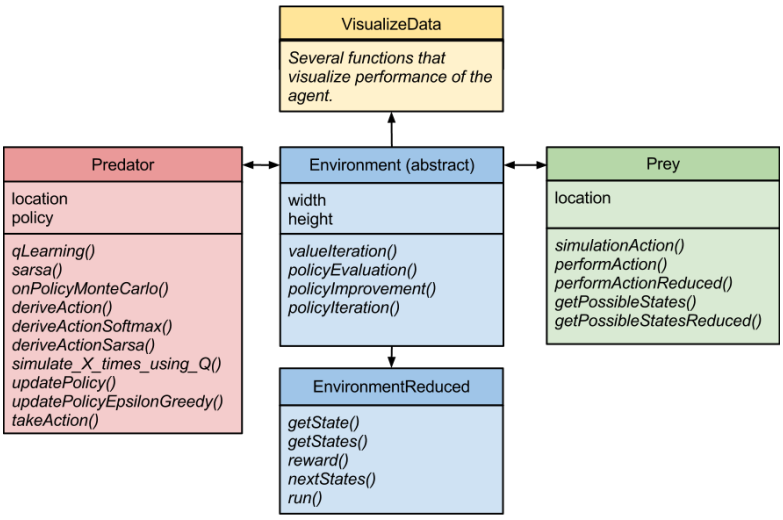


Figure 9: The used class hierarchy.

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 not used in the current assignment. There are two children of the abstract environment class, namely, EnvironmentNormal and EnvironmentReduced, but only EnvironmentReduced, which uses a more concise state representation, is used. For this reason, EnvironmentNormal was not included.

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. Added in this assignment are several functions in Predator, which are implementations of the methods described in this report. Also added is a class VisualizeData that creates an instance of both the environment (reduced state-space) and a predator in that environment, and uses these to plot performance of the agent in that environment.

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 runs several methods from the Predator class, and displays the results.

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

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.

Predator.py This file describes the Predator class that implements the predator, and contains implementations of the learning algorithms.

VisualizeData.py This file contains functions that enable plotting of performance.

4.4 Discussion

The influence of altering the different parameters will be discussed in this subsection.

Altering the optimistic initialization of Q greatly influences performance of the agent: For Q-learning, high initial values will result in slow convergence (see figure 3), due to the high value of unexplored actions and the maximization term in the algorithm. However, a very low initial value will result in insufficient exploratory actions. This is not shown in the figure, due to smoothing and averaging of the data. Changing the initialization of Q will hardly affect on-policy Monte Carlo, as it overwrites the values rather than updating them.

Q-learning and SARSA both performed optimally (for this task) when learning rate is high, and discount factor is low (as seen in figures 4 and 7). However, if the discount factor is set too low, the amount of exploration will not be sufficient on the long term. Again, smoothing of the data hides the random peaks in the number of steps needed to catch the prey. These peaks will occur when a situation arises that has not been encountered sufficiently often. Altering the discount of on-policy Monte Carlo control, again, hardly affects the convergence speed of the algorithm, and subsequently, the performance of the agent (figure 8b).

Increasing ϵ for the ϵ-greedy policy will result in fast convergence when compared to off-policy method Q-learning, as a low ϵ will cause low probability of taking exploratory actions (see figure 5a). Q-learning determines the optimal policy but does not take the exploratory actions into account. It assumes that the predator will always take the optimal action, while there is no guarantee that it takes that action. SARSA, on the other hand, creates a policy that takes these exploratory actions into account and will assume that the predator will sometimes take suboptimal actions.

In conclusion, we can say that for this task, a high learning rate, a low discount factor, a reasonably high ϵ and low initial values for Q are optimal and will cause a quick boost in the predator's performance. However, for a more difficult problem, these parameters may cause insufficient exploration and will thus not always suffice.

5 Conclusion

In this assignment, several Reinforcement Learning techniques were used to learn a predator to catch the prey as fast as possible. Different initial values were used, parameters were adjusted, and the performance of the agent tested for these configurations. Results indicate that it is possible to find good parameters for such a simple task, however, an equilibrium between fast convergence and sufficient exploring is hard to attain.

Due to the simplicity of the task (the prey stands still 80% of the time), fast convergence is possible with a high learning rate and low discount factor. This may prove to be a lot more difficult in the next assignment, where the prey actually tries to evade being caught.

References

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