Reinforcement Learning - Assignment 1

Gustavo Teodoro D. Beck
gtdb@kth.se
940218-0195

Fernando García Sanz
fegs@kth.se
970718-0312

1 The Maze and the Random Minotaur

(a) Formulate the problem as an MDP.

As a Markov Decision Problem, this problem can be defined as follows:

- **Time horizon**: This problem has a finite time horizon defined as T. Either the Minotaur catches the Player or the Player escapes before the Minotaur is able to catch him.
- **State space**: Both the Player and the Minotaur can be located in every tile of the grid that has not a wall in it. Therefore, the total number of possible states is calculated as $40 \times 40 = 1600$. There is a total of 1600 different possible states in the game.
- **Actions**: The actions that can be performed are constrained by the maze and the position (state) of the Player and the Minotaur. The action space for both agents is *A = {Up, Down, Left, Right, Still}*, and depending on the problem, the Minotaur may or may not remain still. However, for the Player, depending on its state (e.g. close to a wall), the possible actions are limited, being the selected action the one that returns the highest reward according to Bellman's equation (problem *b*) or the best policy discovered with Value Iteration algorithm (problem *c*). The Minotaur, on the other hand, moves randomly, and it's allowed to jump a maximum of one wall tile.
- **Rewards**: A reward of 1 is given when the Player reaches the exit position and the Minotaur is not there at the same time. Otherwise, he obtains a reward of 0 (i.e. still in the maze, caught by the Minotaur, or time runs out).
- **Transitions**: Due to the action constraints, if the transition probabilities were represented by a transition matrix, it would result into a very sparse matrix. Since the actions only allow a move of maximum one tile to the player, and maximum of two to the Minotaur (if it crosses the wall), most of the possible states will be unreachable from a given state.

(b) Solve the problem, and illustrate an optimal policy for T=20. Plot the maximal probability of exiting the maze as a function of T. Is there a difference if the Minotaur is allowed to stand still? If so, why?

In order to solve the problem, a dynamic programming approach has been used. Bellman's equation is the method employed for computing the expected rewards for the different states at a given time. The Player will use the outcomes in order to choose the most rewarding action regarding to the expected recompense.

$$V(s) = \max_{a \in A} \left[r(s,a) + \sum_{s' \in S} p(s' \mid a, s) V(s') \right] \tag{1}$$

Given a life expectancy $T = 20$, it is possible to compute backwards the expected rewards in each one of the states for each time step. Saving the previously computed results, it is possible to quickly obtain those of the previous time step, making the computations way faster when compared to other strategies.

According to the Minotaur allowed moves, the exiting probability for the player as a function of T changes. There are three cases that can be discussed:

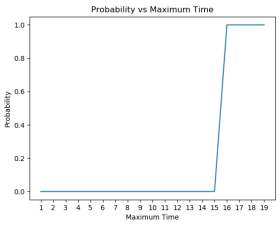

Figure 1: Probability with Minotaur unable to stay still and jump.

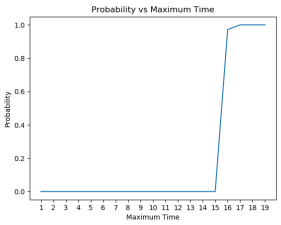

Figure 2: Probability with Minotaur unable to stay still.

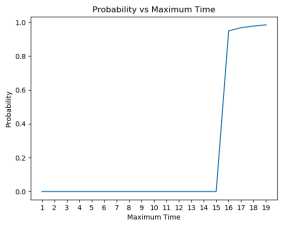

Figure 3: Probability with Minotaur able to stay still.

As it can be seen, as an special case, when the Minotaur is allowed to move without constrains inside the maze 1-tile actions, and forbidden to stay still, a step can be observed between $T = 15$ and $T = 16$. This is due to the fact that the Minotaur and the Player start in different module tiles, and, therefore, if they constantly move, they will never be in a tile of the same module at the same time. In the optimal policy, the Player just needs to move across the optimal path towards the exit in order to win, without caring about the Minotaur. Nevertheless, when the moves of the Minotaur are constrained and it can therefore reach tiles of the same module as the Player's tile at the same time, this plot changes. Now, Minotaur's motion randomness plays a more important role, and although the probability is still high when the value of T is enough to reach the exit from the Player's initial position, it is not a step function anymore. The Player, in some cases, will play safer if there exists a possibility of the two agents reaching the same tile at the same time.

(c) Assume now that your life is geometrically distributed with mean 30. Modify the problem so as to derive a policy minimizing the expected time to exit the maze. Motivate your new problem formulation. Estimate the probability of getting out alive using this policy by simulating 10,000 games.

For this case, our goal is to compute the optimal police independently of the final *T*, since we don't know beforehand which one is going to be the *T* drawn by the geometrical distribution. Therefore, we implemented a Value Iteration algorithm. The essence of this algorithm lies on computing the best value function (based on a discounted version of the Bellman's equation (2)) and assigning the according policy (action) to the best policy vector, i.e. the best action that the player can make in each possible state (all the 1600 possible states). Once all states have been computed we need to check the computed value function vector with the previous one until they converge according to a tolerance factor.

$$V(s) = \max_{a \in A} \left[r(s,a) + \lambda \sum_{s' \in S} p(s' \mid a, s) V(s') \right] \tag{2}$$

$$threshold = 0.0001 \qquad \lambda = (mean - 1)/mean \qquad tolerance = (1 - \lambda) \times threshold/\lambda \tag{3}$$

Once the value functions converge, the optimal policy can be applied on all the 10,000 games. After discovering the optimal police this is the histogram of winning, running out of time, or getting caught by the Minotaur:

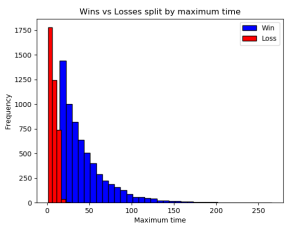

Figure 4: Histogram Winning x Losing

In our analysis, the Player was never caught by the Minotaur and had a success rate of 61.3%. It's is clear that once the time drawn by the distribution is bigger the player always wins and the distribution of losses are concentrated closer to T smaller than 20 seconds.

2 Robbing Banks

(a) Formulate the problem as an MDP.

As a Markov Decision Problem, this problem can be defined as follows:

- **Time horizon**: This problem has an infinite time horizon. There is no final state; if the Police catches the Player, they both go back to their initial positions in the next time step.
- **State space**: Both the Player and the Police can be located in every tile of the grid. Therefore, the total number of possible states is calculated as $18 \times 18 = 324$. There is a total of 324 different possible states in the game.
- **Actions**: The actions that can be performed are constrained by the grid and the position (state) of the Player and the Police. The action space for the Player is A = *{Up, Down, Left, Right, Still}*, while for the Police is the same but without the *Still* action. However, for the Player, depending on its state (e.g. next to the grid limits), the possible actions are limited. The Police moves randomly but always towards the Player, with the intention of catching him. This also limits the possible actions Police can perform in a given state.
- **Rewards**: A reward of 10 is given when the Player stays in a bank tile and the Police is not there at the same time. If the Police catches the Player, he receives a reward of -50. For all other cases, the retrieved reward is zero.
- **Transitions**: Due to the action constraints, if the transition probabilities were represented by a transition matrix, it would result into a very sparse matrix. Since the actions only allow a move of maximum one tile to the player and to the Police, most of the possible states will be unreachable from a given state.

Similarly to problem *1 (c)*, the solution for this problem lies on finding the best policy independently on T, especially because in this game the time horizon is infinite. Therefore, once again we computed the best policy based on the Value Iteration algorithm, however this time we created a grid search to identify the best discount factor for our problem. After discovering the best policy (once the value function converges) for all the 324 states, it is time to start robbing the banks, where each action of the robber will be made based on the state of the robber and police.

(b) Solve the problem, and display the value function (evaluated at the initial state) as a function of λ. Illustrate an optimal policy for different values of λ – comment on the behaviour.

For this problem we decided to demonstrated the effect of the discount factor (λ) displaying the value function evaluated at the initial state and the reward of the robber if we played a game with 1000 turns with each optimal policy (figures 5 and 6).

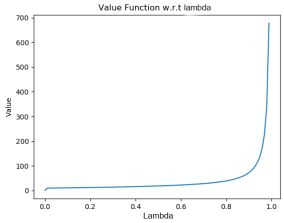

Figure 5: Value function evaluated at the initial state for different discount factors λ.

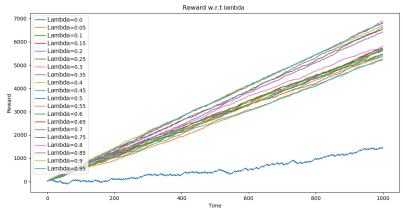

Figure 6: Reward over time for different discount factors λ.

It's clear to see that higher the discount factor, the bigger the value function for the initial state and the higher the reward over time, which indicates that the robber "plays" better, without getting caught or remaining much time outside a bank. There are two main reasons for this to happen: (1) with a

higher discount factor, smaller is the tolerance, which leads to a better approximation of the optimal policy, and (2) with a higher discount factor, higher is the influence of the second term of the equation 2, which leads to more iterations to achieve convergence, and probably to a better approximation.

As an illustration, figure 7 shows that the robber is smart enough to traverse through all the banks in order to earn more money (higher reward).

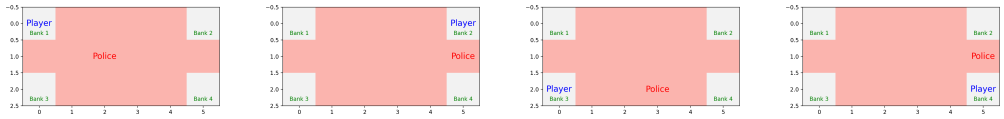

Figure 7: Player in the different banks during the same game.

3 Bank Robbing (Reloaded)

(a) Solve the problem by implementing the Q-learning algorithm exploring actions uniformly at random. Create a plot of the value function over time (in particular, for the initial state), showing the convergence of the algorithm. Note: Expect the value function to converge after roughly 10,000,000 iterations (for step size $1/n(s,a)^{2/3}$***, where*** $n(s,a)$ ***is the number of updates of*** $Q(s,a)$***).***

In this problem our goal is to compute a policy for all the possible states, however, since it is not a MDP, we do not have full knowledge about the environment. In other words, the robber has to randomly learn where he receives a positive reward and when he receives a negative reward. Therefore, in this problem, the matrix Q is learned by exploring all possible moves at random and by the algorithm Q-learning. The core of the algorithm it to initialize as zeros a Q-matrix of size "*all possible states* $\times$ *all possible actions*", for each step (one out of the 10,000,000), we will compute the value function of all possible next actions for that particular state, assign the best value to the Q-matrix according to the best action, and then perform a random move. After many iterations, each coordinate of the Q-matrix will have a utility determined by the value function and the processed described. We can see in figure 8 that the agent explores the space state quite fast and converges to an optimal policy close to 1,000,000 iterations. The value function in figure 8 corresponds to the best value at each step for the initial state.

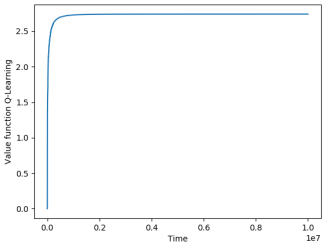

Figure 8: Q-learning value function for the initial state over time.

(b) Solve the problem by implementing the SARSA algorithm using ϵ***-greedy exploration (initially*** $\epsilon = 0.1$***). Show the convergence for different values of*** ϵ***.***

$$\text{action} \rightarrow a = \begin{cases} \text{uniform}(\mathcal{A}_s) & w.p. \quad \epsilon \\ \text{argmax}_{b \in A_s} Q^t(s,b) & w.p. \quad 1-\epsilon \end{cases} \tag{4}$$

$$Q^{(t+1)}(s,a) = Q^{(t)}(s,a) + 1_{(s_t,a_t)=(s,a)}\alpha_{n^{(t)}(s_t,a_t)}\left[r_t + \lambda Q^{(t)}(s_{t+1},a_{t+1}) - Q^{(t)}(s_t,a_t)\right]$$
$$\text{where } n^{(t)}(s,a) := \sum_{m=1}^{t} 1[(s,a) = (s_m,a_m)] \tag{5}$$

In SARSA algorithm, the Q-matrix is updated at each step (one out of the 10,000,000) based on: current state, current action, current reward, next state, and next action. The action and next action are computed in a ϵ-greedy exploration manner, as shown in equation 4. This strategy allows the agent to explore (randomly move) the state space, but also exploit (move according the highest value of Q in that state and next state) its current knowledge about the experience gained during the steps. Equation 5 shows how to update Q.

We solved this problem for different values of ϵ (figures 9, 10, 11, and 12); it is clear that assigning bigger values for ϵ constrains the agent for exploring the state space, which will lead the agent to exploit only a small number of states, and, therefore, returning a smaller value function.

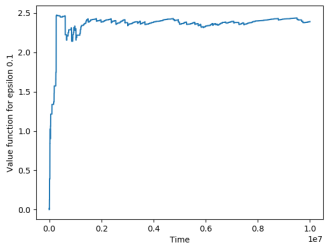

Figure 9: Value function for initial state $\epsilon = 0.1$.

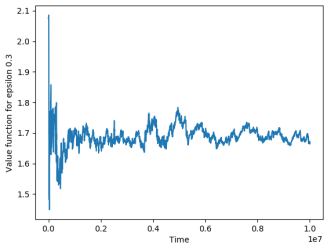

Figure 10: Value function for initial state $\epsilon = 0.3$.

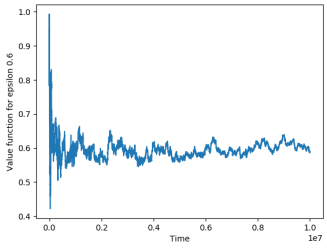

Figure 11: Value function for initial state $\epsilon = 0.6$.

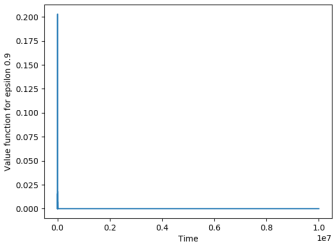

Figure 12: Value function for initial state $\epsilon = 0.9$.

4 RL with Linear Function Approximators

The algorithm employed for this case is, back again, SARSA. As this algorithm has been explained before we will not go too much into technical details of the algorithm itself here. The algorithm has trained for a total of 200 different episodes. Each episode corresponds to each one of the environment simulations until a final state is reached. It is important to remark that the knowledge acquired in a previous episode is transferred to the next one.

The main differences of this approach are the use of Fourier basis functions $\phi_i(s) = \cos\left(\pi \boldsymbol{\eta}_i^\top s\right)$ and the computation of a weight matrix W that maps the basis functions with accordance to the possible actions. These functions are approximations used to parameterize the state-value function of a given policy, the value function, or the Q-function using a low dimensional parameter. They are really useful for large state/action-spaces (like continuous spaces).

The parameters employed for this task are the following:

- Discount factor (γ): 1.
- Eligibility parameter (λ): 0.7.
- Problem order (p): 2.
- Momentum: 0.6.
- Dimensions (position and velocity): 2.
- Epsilon (ϵ): Range of 200 values.
- Number of basis functions (m): 9.
- Initial learning rate (α): 0.001.

It is necessary to highlight some improvements of the original algorithm. In order to make the learning process more stable, stochastic gradient descent with *Nesterov* acceleration has been employed. Besides, the eligibility trace values have been clipped between -5 and 5, avoiding the exploding gradient problem. Finally, the learning rate (α) has been scaled for each basis function (ϕ), obtaining a customized learning rate for each. This customization process has been performed as follows:

$$\alpha_i = \begin{cases} \alpha/\|\boldsymbol{\eta}_i\|_2 & \text{if } \|\boldsymbol{\eta}_i\|_2 \neq 0 \\ \alpha & \text{if } \|\boldsymbol{\eta}_i\|_2 = 0 \end{cases} \tag{6}$$

Besides this, the following matrices are generated:

- W: $m \times k$ matrix of randomly initialized $\boldsymbol{w}_{a_k}^T$ vectors.
- N: $dimensions \times m$ matrix of $\boldsymbol{\eta}_m^T$ vectors.
- Z: $m \times k$ matrix of eligibility traces initialized to zero.
- V: $m \times k$ matrix of velocity terms initialized to zero.

The term k represents the number of actions (3 in this case).

For this algorithm to succeed, it needs to first explore the biggest amount of states possible, and then, exploit the collected information in order to find a good policy. This is why we have not employed a fixed epsilon value, but a different epsilon for each episode. For the first episodes, the values of epsilon will be close to one, promoting choosing random actions, but as the number of elapsed episodes increases, the values of epsilon decrease, promoting more greedy actions over randomness. By that time, the algorithm would have collected information about different states due to the initial randomness and it could make a more appropriate use of these data now. The different epsilon values were calculated as a range, starting from zero, in the interval $[0, 1)$ with a step of $1/number\ of\ episodes$ between values, then inverted in order to employ the greatest values first.

With these settings, the obtained results were the following:

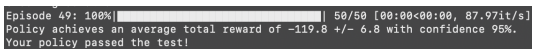

Figure 13: Solution in terminal.

The evolution of the reward over the episodes can be seen here:

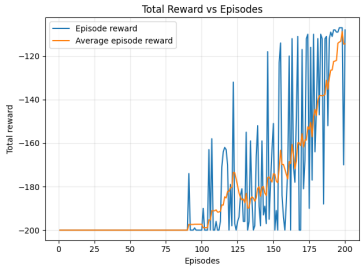

Figure 14: Total reward during the episodes.

In order to comprehend the effect of the eligibility parameter (λ) and the learning rate (α), we first fixed $\alpha = 0.6$ and increased λ to 1.0 and then, decreased λ to 0.3.

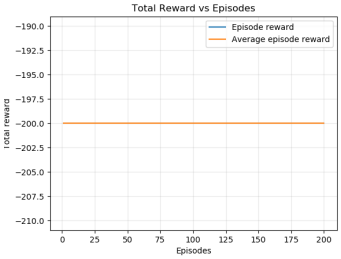

Figure 15: Total reward during the episodes for $\lambda = 1.0$.

Figure 16: Total reward during the episodes for $\lambda = 0.3$.

Afterwards, we fixed $\lambda = 0.7$ and increased α to 0.01 and then, decreased λ to 0.0001.

Figure 17: Total reward during the episodes for $\alpha = 0.01$.

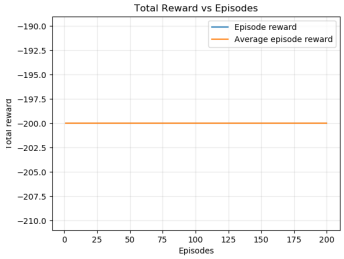

Figure 18: Total reward during the episodes for $\alpha = 0.0001$.

From figures 14, 15, and 16, we can infer that the eligibility factor affects how much we update the weights. If the value is to low the learning process will be slower, since we are updating less. However, if it is to big, the weights will update to much and my overshoot and pass by the correct solution and never reach the goal. In terms of the learning rate, if the learning rate is to big it might happen that the solution will be going back and forth with regards to reaching the goal, as can be seen in figure 17. On the other hand, if the learning rate is to small (figure 18 the agent might never reach the goal in 200 steps since the learning process is very slow.

References

[1] G. Konidaris, "Value function approximation in reinforcement learning using the fourier basis," *Computer Science Department Faculty Publication Series*, p. 101, 2008.