Universitat de Barcelona

Facultat de Matemàtiques i Informàtica

Inteligència Artificial

Pràctica 3: Reinforcement Learning

Authors:
Martí Vila Rebellón
Martí Lázaro Bagué

Date:
December 13, 2025

Introduction

In this project, we were introduced to reinforcement learning, a method of "teaching" an agent to select the correct and optimal actions in order to reach a desired state. This learning process is implemented through a table known as the Q-table, which represents the relationship between states and actions. Throughout this document, it will be explained and demonstrated how this learning method is implemented across various scenarios, using different actions as well as different penalties and rewards.

Objectives

The objectives of this final practicum are summarized into two main exercises:

- **4x3 Grid Scenario:**
  - Implement Q-learning with a constant reward of -1 in each cell of the grid.
  - Implement Q-learning using a heuristic-based reward system in the grid.
  - Implement the "drunken sailor" behavior in the previously solved scenario.
- **Chess Scenario:**
  - Solve the same problems presented in the previous objective, but in an 8×8 scenario, where there is one black king that cannot move, and two moving pieces: a white king and a white rook.
- Analyze all the results and compare them with previous concepts learned in the course.

Questions

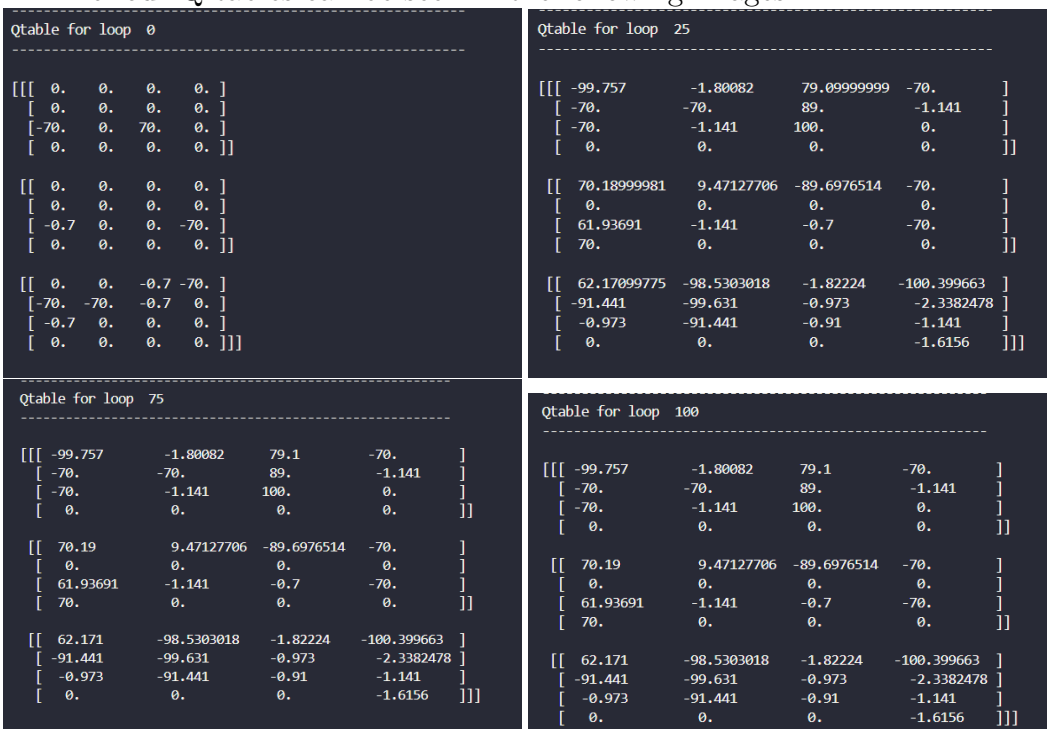
0.1 Print the first, two intermediate and the final Q-table. What sequence of actions do you obtain?

First of all, before starting with the requested explanation, we will describe the parameters shown in the Q-table prints.
To begin with, the title and the loop number are displayed in each print.

It is important to note that these prints are generated after the agent has performed its actions. Therefore, non-zero values can already be observed in loop 0. The most important aspect when analyzing the table is that each cell contains four values. Each of these values represents the penalty associated with one of the possible movements from that cell. In this way, we can understand what the agent has learned about each cell and the four possible actions it can take from that position. These values are updated after each loop, allowing us to observe the evolution of the Q-table over time.

The printing of the four required Q-tables is implemented in the main_Qtables.py file. As can be observed, the first and second tables, corresponding to the 1st and 25th loops, show an improvement in the values across the table. In contrast, the tables from loops 75 and 100 remain unchanged. This behavior indicates that the model converges at approximately the 25th loop, or even before reaching that point. This convergence is further demonstrated later in the document.

The four Q-tables can be seen in the following images:



As previously explained, there is no change in the table values between loops 75 and 100, nor between loops 25 and later iterations.

As shown in the four images, the Q-tables consist of a 3×4 grid, meaning

there are 12 cells, each containing four values. These values correspond to the Q-values assigned to the possible actions from each cell. The values are structured in the following order:

- 1st value [0]: Up
- 2nd value [1]: Down
- 3rd value [2]: Right
- 4th value [3]: Left

If a value is displayed as 0.0, it means that the model has not yet performed any calculations for that action from the corresponding cell. In other words, 0.0 is the initialization value of the Q-table. These values are typically observed during the first episodes of training and in invalid movements. For example, cells located at the borders of the grid show zero values for actions that would result in moving outside the grid, as such movements are not valid.

0.2 After trying for a bit, what is your parameter choice for alpha, gamma and epsilon? Why?

After conducting several tests and analyzing the effect of each parameter, we settled on the following configuration:

alpha: Based on the observation that most of the learning occurs during the initial stages of the algorithm, choosing a relatively high value for this parameter is crucial. Therefore, an $\alpha = 0.7$ was selected, as it allows the agent to learn efficiently while still stabilizing over time.

gamma: This parameter determines how much the agent prioritizes future rewards. In this case, the objective is to find the shortest path, so future rewards should be strongly prioritized. For this reason, a high discount factor was chosen, specifically $\gamma = 0.9$.

epsilon: This parameter controls how often the agent explores the environment (i.e., performs exploratory or random actions) in order to discover potentially better solutions. In this implementation, a decay of 10% per round was applied. Initially, the exploration rate is set to a high value, $\epsilon = 0.9$, and it gradually decays until reaching a minimum value of $\epsilon = 0.1$.

0.3 How do you judge convergence of the algorithm? How long does it take to converge?

We've implemented a method to calculate the convergence after 300 iterations for each combination of these values:

alphas = [0.1, 0.3, 0.5, 0.7]

gammas = [0.5, 0.7, 0.9]

epsilons = [0.1, 0.3, 0.5, 0.9]

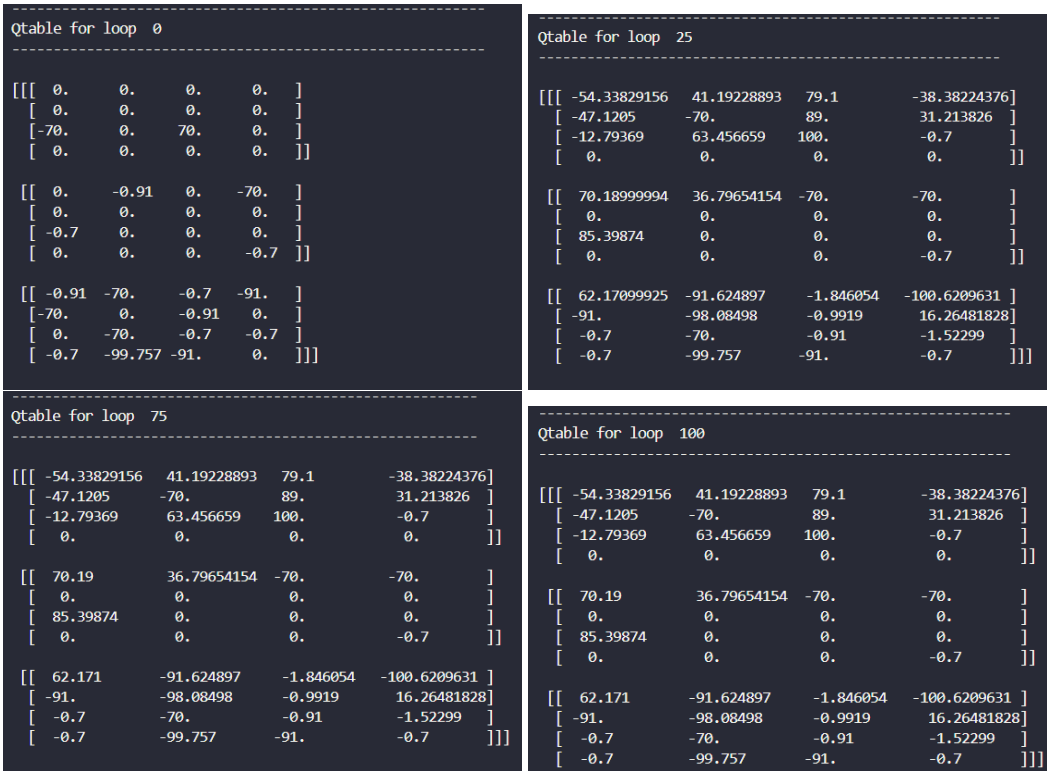
After analyzing the convergence table, we selected our initial assumption. By examining the parameter triplets that achieved maximum convergence and reward with the minimum number of iterations, this choice was confirmed. Convergence occurs, sometimes, in fewer than 5 iterations. However, due to the inherent randomness of the algorithm, we chose to ensure the stability by requiring three consecutive episodes with identical rewards. This convergence check is applied only after 5 episodes have been completed. While this method does not yield strictly stable results, such variability is inherent to the stochastic nature of the algorithms, representing a limited control measure.

```
alpha=0.1, gamma=0.5, epsilon=0.1 -> convergence=85.700,
   iterations=19
alpha=0.1, gamma=0.5, epsilon=0.3 -> convergence=80.700,
   iterations=19
alpha=0.1, gamma=0.5, epsilon=0.5 -> convergence=96.000,
   iterations=14
alpha=0.1, gamma=0.5, epsilon=0.9 -> convergence=96.000,
   iterations=13
alpha=0.1, gamma=0.7, epsilon=0.1 -> convergence=85.200,
   iterations=17
alpha=0.1, gamma=0.7, epsilon=0.3 -> convergence=71.000,
   iterations=25
alpha=0.1, gamma=0.7, epsilon=0.5 -> convergence=96.000,
   iterations=12
alpha=0.1, gamma=0.7, epsilon=0.9 -> convergence=96.000,
   iterations=12
alpha=0.1, gamma=0.9, epsilon=0.1 -> convergence=85.900,
   iterations=15
alpha=0.1, gamma=0.9, epsilon=0.3 -> convergence=65.700,
   iterations=16
alpha=0.1, gamma=0.9, epsilon=0.5 -> convergence=96.000,
   iterations=11
alpha=0.1, gamma=0.9, epsilon=0.9 -> convergence=96.000,
   iterations=12
alpha=0.3, gamma=0.5, epsilon=0.1 -> convergence=76.000,
   iterations=6
alpha=0.3, gamma=0.5, epsilon=0.3 -> convergence=65.500,
```

```
    iterations=13
alpha=0.3, gamma=0.5, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.3, gamma=0.5, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.3, gamma=0.7, epsilon=0.1 -> convergence=65.900,
    iterations=10
alpha=0.3, gamma=0.7, epsilon=0.3 -> convergence=70.400,
    iterations=7
alpha=0.3, gamma=0.7, epsilon=0.5 -> convergence=96.000,
    iterations=6
alpha=0.3, gamma=0.7, epsilon=0.9 -> convergence=96.000,
    iterations=9
alpha=0.3, gamma=0.9, epsilon=0.1 -> convergence=60.700,
    iterations=8
alpha=0.3, gamma=0.9, epsilon=0.3 -> convergence=70.800,
    iterations=10
alpha=0.3, gamma=0.9, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.3, gamma=0.9, epsilon=0.9 -> convergence=96.000,
    iterations=9
alpha=0.5, gamma=0.5, epsilon=0.1 -> convergence=60.800,
    iterations=7
alpha=0.5, gamma=0.5, epsilon=0.3 -> convergence=75.800,
    iterations=7
alpha=0.5, gamma=0.5, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.5, gamma=0.5, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.5, gamma=0.7, epsilon=0.1 -> convergence=75.800,
    iterations=10
alpha=0.5, gamma=0.7, epsilon=0.3 -> convergence=75.800,
    iterations=6
alpha=0.5, gamma=0.7, epsilon=0.5 -> convergence=96.000,
    iterations=9
alpha=0.5, gamma=0.7, epsilon=0.9 -> convergence=96.000,
    iterations=7
alpha=0.5, gamma=0.9, epsilon=0.1 -> convergence=60.700,
    iterations=22
alpha=0.5, gamma=0.9, epsilon=0.3 -> convergence=75.700,
    iterations=10
alpha=0.5, gamma=0.9, epsilon=0.5 -> convergence=96.000,
    iterations=7
alpha=0.5, gamma=0.9, epsilon=0.9 -> convergence=96.000,
    iterations=7
alpha=0.7, gamma=0.5, epsilon=0.1 -> convergence=85.700,
    iterations=7
alpha=0.7, gamma=0.5, epsilon=0.3 -> convergence=66.000,
    iterations=12
```

```
alpha=0.7, gamma=0.5, epsilon=0.5 -> convergence=96.000,
    iterations=5
alpha=0.7, gamma=0.5, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.7, gamma=0.7, epsilon=0.1 -> convergence=75.500,
    iterations=8
alpha=0.7, gamma=0.7, epsilon=0.3 -> convergence=85.700,
    iterations=8
alpha=0.7, gamma=0.7, epsilon=0.5 -> convergence=96.000,
    iterations=5
alpha=0.7, gamma=0.7, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.7, gamma=0.9, epsilon=0.1 -> convergence=65.700,
    iterations=18
alpha=0.7, gamma=0.9, epsilon=0.3 -> convergence=65.800,
    iterations=12
alpha=0.7, gamma=0.9, epsilon=0.5 -> convergence=96.000,
    iterations=7
alpha=0.7, gamma=0.9, epsilon=0.9 -> convergence=96.000,
    iterations=9
```

0.4 Q1b. i Answer the questions of the previous section for this case



In these images, we can observe how the values of the Q-tables differ significantly between the first exercise and this one. To further explain these changes, we established a set of predefined penalties in the grid, which act as a heuristic for the agent. This numerical distribution is known as the Manhattan Distribution, and it is intended to help the agent learn faster, as it discourages certain movements without requiring the agent to explore them explicitly.

Regarding the second question, our choice for the alpha parameter remains unchanged, as it is already set to a relatively high value. However, we have modified the values of gamma and epsilon.

Furthermore, when analyzing the convergence of the algorithm using the new reward table,

```
    alpha=0.1, gamma=0.5, epsilon=0.1 -> convergence=75.700,
       iterations=23
alpha=0.1, gamma=0.5, epsilon=0.3 -> convergence=80.900,
   iterations=19
```

```
alpha=0.1, gamma=0.5, epsilon=0.5 -> convergence=96.000,
    iterations=13
alpha=0.1, gamma=0.5, epsilon=0.9 -> convergence=96.000,
    iterations=17
alpha=0.1, gamma=0.7, epsilon=0.1 -> convergence=65.600,
    iterations=11
alpha=0.1, gamma=0.7, epsilon=0.3 -> convergence=65.900,
    iterations=11
alpha=0.1, gamma=0.7, epsilon=0.5 -> convergence=96.000,
    iterations=11
alpha=0.1, gamma=0.7, epsilon=0.9 -> convergence=96.000,
    iterations=12
alpha=0.1, gamma=0.9, epsilon=0.1 -> convergence=80.600,
    iterations=10
alpha=0.1, gamma=0.9, epsilon=0.3 -> convergence=70.900,
    iterations=11
alpha=0.1, gamma=0.9, epsilon=0.5 -> convergence=96.000,
    iterations=13
alpha=0.1, gamma=0.9, epsilon=0.9 -> convergence=96.000,
    iterations=13
alpha=0.3, gamma=0.5, epsilon=0.1 -> convergence=55.900,
    iterations=11
alpha=0.3, gamma=0.5, epsilon=0.3 -> convergence=90.900,
    iterations=10
alpha=0.3, gamma=0.5, epsilon=0.5 -> convergence=96.000,
    iterations=9
alpha=0.3, gamma=0.5, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.3, gamma=0.7, epsilon=0.1 -> convergence=75.600,
    iterations=6
alpha=0.3, gamma=0.7, epsilon=0.3 -> convergence=80.800,
    iterations=18
alpha=0.3, gamma=0.7, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.3, gamma=0.7, epsilon=0.9 -> convergence=96.000,
    iterations=9
alpha=0.3, gamma=0.9, epsilon=0.1 -> convergence=85.800,
    iterations=10
alpha=0.3, gamma=0.9, epsilon=0.3 -> convergence=85.900,
    iterations=6
alpha=0.3, gamma=0.9, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.3, gamma=0.9, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.5, gamma=0.5, epsilon=0.1 -> convergence=90.900,
    iterations=15
alpha=0.5, gamma=0.5, epsilon=0.3 -> convergence=90.900,
    iterations=8
alpha=0.5, gamma=0.5, epsilon=0.5 -> convergence=96.000,
```

```
    iterations=8
alpha=0.5, gamma=0.5, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.5, gamma=0.7, epsilon=0.1 -> convergence=85.500,
    iterations=9
alpha=0.5, gamma=0.7, epsilon=0.3 -> convergence=85.700,
    iterations=6
alpha=0.5, gamma=0.7, epsilon=0.5 -> convergence=96.000,
    iterations=7
alpha=0.5, gamma=0.7, epsilon=0.9 -> convergence=96.000,
    iterations=7
alpha=0.5, gamma=0.9, epsilon=0.1 -> convergence=50.800,
    iterations=13
alpha=0.5, gamma=0.9, epsilon=0.3 -> convergence=55.800,
    iterations=8
alpha=0.5, gamma=0.9, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.5, gamma=0.9, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.7, gamma=0.5, epsilon=0.1 -> convergence=80.700,
    iterations=11
alpha=0.7, gamma=0.5, epsilon=0.3 -> convergence=80.400,
    iterations=10
alpha=0.7, gamma=0.5, epsilon=0.5 -> convergence=96.000,
    iterations=7
alpha=0.7, gamma=0.5, epsilon=0.9 -> convergence=96.000,
    iterations=6
alpha=0.7, gamma=0.7, epsilon=0.1 -> convergence=70.800,
    iterations=6
alpha=0.7, gamma=0.7, epsilon=0.3 -> convergence=90.800,
    iterations=8
alpha=0.7, gamma=0.7, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.7, gamma=0.7, epsilon=0.9 -> convergence=96.000,
    iterations=8
alpha=0.7, gamma=0.9, epsilon=0.1 -> convergence=95.600,
    iterations=6
alpha=0.7, gamma=0.9, epsilon=0.3 -> convergence=80.500,
    iterations=10
alpha=0.7, gamma=0.9, epsilon=0.5 -> convergence=96.000,
    iterations=8
alpha=0.7, gamma=0.9, epsilon=0.9 -> convergence=96.000,
    iterations=6
```

As shown here, the heuristic-based approach achieves faster convergence. In many simulations, convergence is reached in fewer iterations. The best convergence occurs at the 5th iteration, with a reward of 96. Due to the randomness introduced by the exploration factor, some variation may occur,

however, in most cases, the algorithm converges at or near this point.

0.5 What is the effect of the new reward function on performance?

With these preset values assigned to each cell, we can conclude that this approach acts as a heuristic for the Q-learning algorithm. Incorporating this heuristic, the agent's learning process becomes significantly faster, as it is able to follow a pre-established path from the beginning.

In conclusion, convergence in this case is faster because the penalty values are non-uniform. This allows the model to initially follow a guided path, making it easier to identify and refine the optimal path toward the reward.

0.6 How does this relate to the search algorithms studied in P1? Could you apply one of those in this case?

In P1, we learned about the A* algorithm, which searches the shortest path in a grid-based environment through the use of a heuristic. The current case is very similar to the A* scenario, as the predefined penalties act as a heuristic, and the Q-learning algorithm guides the agent toward the goal by following the path with the lowest accumulated penalty, learning its own shortest path.
Indeed, A* could also be applied in this case, since the environment is deterministic, every action leads to a known and intended next state. Under these conditions, implementing A* would likewise result in the discovery of the shortest path.

1 Drunken sailor

1.1 Explanation of the changes implemented for the drunken sailor approach

There is some randomness in the original agent, but it only affects exploration, not the actual outcome of the agent's decisions, which is key to this approach.
The main difference lies in whether the agent considers the optimal path to the goal or just the immediate next step. This change is implemented in the environment itself, even if the agent selects a theoretically perfect action,

the environment may modify the outcome, influencing the agent's learning process.

1.2 Using the a or b choice

1.2.1 What is your parameter choice, why?

We have chosen option 1.b, the one with pre-established penalties in each cell. We consider this a better approach, as it allows for faster convergence and can also lead to improved results.

1.2.2 Assuming the sailor is in a state that allows learning: how many drunken nights are necessary for them to master the perilous path to bed? Compare to the previous, deterministic scenario.

Due to the random aspect of this scenario, it is nearly impossible to determine exactly how many episodes the "drunken sailor" agent will need to converge. Unlike the deterministic cases presented earlier, the next movement is uncertain because of the agent's stochastic behavior. Consequently, the agent will require more episodes to converge, although the exact number cannot be precisely predicted. Even though, we could approximate the episodes being similar or completely different to the deterministic approach.

1.2.3 What is the optimal path found? If we watched the sailor try to follow it, would they always follow the same path?

The sailor will not follow the same path every time. As explained earlier, the key difference between this scenario and the deterministic one lies in the environment. In this case, the environment class determines the agent's movement, introducing the random component. Learning proceeds as usual, but the agent cannot fully control its actions or predict the next state with 100% certainty. As a result, even if the agent calculates the optimal path, it may need to recalculate it periodically whenever the "drunken sailor" fails to execute the intended move and a new path must be determined.

1.2.4 Could we apply one of the algorithms in P1 here? Why? Hint: think of the notions of deterministic vs random and of path vs policy.

The algorithms implemented in P1 were deterministic in nature, and we do not believe that implementing them here would work properly. Those

algorithms require complete knowledge of the next possible states in order to precalculate the optimal movement. In this scenario, however, there is a 1% probability that the agent will not know the next move, which would disrupt the proper functioning of deterministic algorithms.

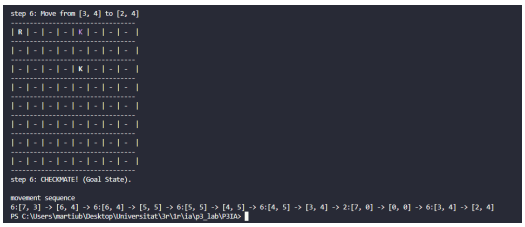
2 chess Scenario

In this second section of the memory, we're meant to implement the Q-learning we achieved in the previous part, in a chess scenario, similar to the P1.

2.1 a.Q-Learning with "simple" reward

This section is centered in a chess scenario which every cell applies a reward of -1 to the pieces, like the first grid scenario.

2.1.1 i. Sequence of Actions and Q-Tables



This sequence corresponds to 2000 episodes using the simple mode of penalties throughout the board. This represents the best performance our agent has achieved so far, and, as observed, it follows the optimal path. However, due to the randomness in the agent's exploration function, executing AiChess with this configuration may yield different results each time. There is a significant possibility that the agent may not complete the sequence successfully and therefore may not achieve checkmate. In fact, the majority of the time we've executed the agent in this conditions of 2000 episodes the agent doesn't get to the checkmate, with 3000 the possibilities increment.

Regarding the sequence of Q-tables, the number of states and actions in this scenario is much larger than in the first section. As a result, printing the Q-tables to the terminal was not feasible. For this reason, all Q-tables for this simple environment are provided in a JSON file attached to this document. This JSON file is generated when executing aichess.py with the user's

preferred configuration, and it is saved in the folder where the document is created.
[1]

2.1.2 ii.After trying for a bit, what is your parameter choice for alpha, gamma and epsilon? Why?

- **Learning rate: 0.7** → This relatively high value allows the agent to achieve faster convergence. Initially, we tried an even higher value, but this caused the agent to enter cycles, repeatedly returning to previous states because they had higher immediate rewards, even though they were not the goal. With a learning rate of 0.7, we achieve both correct and fast learning behavior.

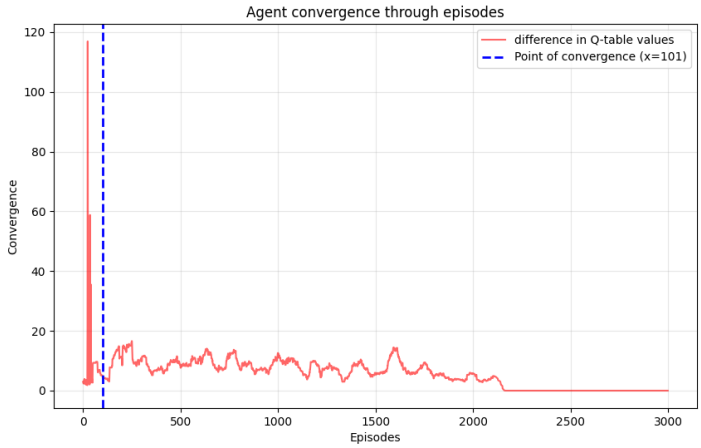
- **Discount factor: 0.9** → A high discount factor encourages the agent to prioritize long-term rewards over immediate ones. With gamma = 0.9, the agent aims to reach the maximum possible reward, avoiding states that are only beneficial in the short term but ultimately penalizing progress toward the goal.

- **Exploration rate: 0.9** → Since the agent starts in an empty environment, a high exploration rate encourages random exploration over exploiting known actions. This is essential in this scenario to ensure that the agent sufficiently explores the environment before converging on the optimal path.

2.1.3 iii.How do you judge convergence of the algorithm? How long does it take to converge?

Due to the large number of states and actions, we decided to create a new convergence function. Now instead of using reward as an indicator for convergence, we used a Qtable variation based method. This function generates a plot to visualize the agent's convergence making it easier to analyze the information.

[1] Read trained_agent_qtable_simple.json



As explained previously, convergence in 2000 episodes is possible but challenging with the simple reward system. Therefore, we extended the training to a maximum of 3000 episodes, as shown in the plot. The plot can be divided into three main parts:

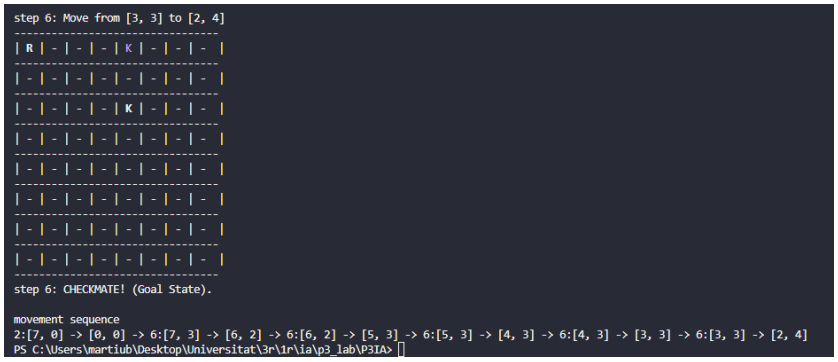
The first part shows significant variation. The red line represents the difference between consecutive Q-tables in each episode. In this initial phase, high variation is expected, since the agent has no prior information about the environment.

The second part, occurring after the blue line, shows the Q-tables becoming more stabilized, although full convergence has not yet been reached. The values are settling, but some fluctuations remain.

In the final part, a plateau is observed, where the difference reaches absolute zero. This indicates that the Q-tables have become invariant, and the agent has achieved convergence.

2.2 b.Q-Learning with Heuristic-Based Reward

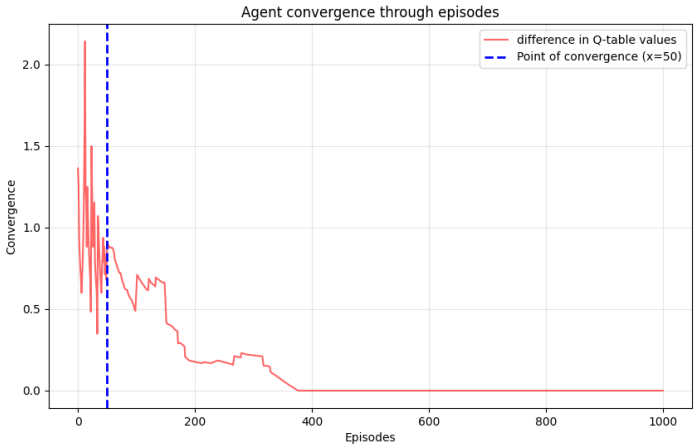
2.2.1 i. Answer the questions of the previous section for this case.



As in the simple environment, this scenario provides a similar optimal path, with minor variations across consecutive executions. However, in this case, the agent consistently reaches checkmate every time. The main difference is that with only 1000 episodes, the agent reliably discovers the optimal path, making this approach more efficient and dependable.

Regarding the parameters, we have not changed any of them. Since they primarily influence the environment, and the environment has not changed, there is no need to modify them. The parameters perform effectively in this scenario, just as they did in the previous one.

Focusing on the convergence of this new scenario, it appears to be faster than in the simple reward system. This is clearly illustrated in the following plot:



The plot also shows three distinct sections, similar to the simple reward system. However, the speed of convergence is significantly higher, as is evident from the plot.

2.2.2 ii. What is the effect of the new reward function on performance

This new reward function has made the agent more reliable and faster. Now, the agent can achieve checkmate almost every time, and in fewer episodes. With the simple reward function, convergence typically required between 2000 and 3000 episodes. Using the new reward system, however, the agent achieves checkmate within 1000 episodes with approximately 99% probability. This makes the learning process more efficient in both speed and reliability.

2.3 c. Drunken Sailor Scenario (Stochastic Chess)

2.3.1 i.Stochasticity Implementation

To implement stochasticity, as we previously applied in the simple scenario of Part 1, the goal is to allow the AI to occasionally execute an action different from the one it computed as optimal. In this implementation, activating the "drunken sailor" random factor only requires setting a non-zero value for the stochasticity parameter in the learning configuration. When this parameter is set to 0, no randomness is applied, and the agent always executes the action it computes.

2.3.2 What is your parameter choice? Why?

We have not significantly changed our parameters. A discount factor of 0.99 is clearly necessary. A high value allows the agent to prioritize long-term rewards, and in the context of the "drunken sailor" scenario, it reduces the impact of short-term penalties caused by random actions.

The learning rate remains unchanged, although it could potentially be increased. With this relatively high value, the agent maintains consistency along high-reward paths, mitigating the effect of stochasticity.

Finally, the exploration rate does not need to be adjusted. The environment still starts empty, and the introduction of the "drunken sailor" does not require changes to this parameter.

2.3.3 Assuming our obsessive sailor is in a state that allows learning: how many games do they have to play before they are satisfied that they have found the best strategy and can go to bed? Compare to the previous, deterministic scenario.

In this "drunken sailor" scenario, the AI may require different amounts of time in each execution to determine an appropriate strategy. The randomness introduced in the agent's movements makes the time needed to converge to a stable path largely unpredictable.

As a result, the time required for the drunken sailor to reach the optimal path strategy can range from being comparable to that of the deterministic agent to an almost unbounded duration. In extreme cases, persistent randomness may cause the agent to continuously recalculate its path after each step, significantly delaying convergence.

2.3.4 What is the optimal path found? If we watched the sailor try to follow it, would they always follow the same path?

In this stochastic scenario, the most optimal path remains the same as in the previous example. Since there is only a 1% probability that the "drunken sailor" fails to execute a move, the agent often follows the optimal path, similar to the standard agent. While the path to the goal is identical, the drunken sailor does not always follow it perfectly, meaning that the actual trajectory may vary from episode to episode.

2.4 d.Compare the application of Q-learning in this chess scenario with that of the grid of exercise 1. How do the two scenarios differ? How does that translate into your results?

Considering the two scenarios, there are several key differences:

- **Dimensions:** The Chess scenario is almost twice the size of the previous grid, increasing from a 4×3 grid to an 8×8 grid. This increase in grid size makes achieving the objective more difficult and significantly increases computational complexity.
- **Actions:** In the first grid scenario, each state has a fixed set of four possible actions (up, down, left, right). In the Chess scenario, not only are there more possible actions per state, but different pieces have different available actions. For example, the king may have fewer possible actions than the rook.

These two factors contribute to the greater complexity of the Chess scenario compared to the grid scenario. The agent's learning process is more challenging due to the increase in dimensions.
Additionally, differences are observed in the results. The Chess scenario produces a larger set of outcomes compared to the grid scenario, as the number of movements, possible actions, and associated rewards is significantly greater.

2.5 Q-Learning vs. Search Algorithms (P1)

The core difference between Q-learning and the search algorithm lies in their ability to handle uncertainty and their output. In a deterministic environment, Search Algorithms are generally more efficient for finding the single optimal path from a starting state to a goal state, as they require a complete model of the environment to work with and explore it systematically. Q-learning, however, is a model that learns an optimal policy through trial-and-error, which ables it to proceed in a bigger range of scenarios. The critical distinction is in a stochastic environment, Q-learning is explicitly designed to handle randomness because it learns with many episodes, which is perfect in these cases, whereas standard Search Algorithms are inapplicable as they rely on predictable transitions to function correctly.

Conclusions

This project provided a comprehensive insight into the mechanics of reinforcement learning, evolving from fundamental problem-solving to complex state-space navigation. A central focus of our analysis was the correlation between environmental design and agent behavior, specifically how hyperparameter tuning dictates action selection. We determined that intentionality in environment-agent architecture is a prerequisite for efficient convergence. Empirically, heuristic-based reward shaping (as the one shown, Manhattan distance) proved significantly more effective than standard sparse penalties (specifically, the constant -1 step cost). Furthermore, the "Drunken Sailor" scenario served to validate the agent's adaptability and learning process within a stochastic framework. However, a critical limitation remains scalability; as the scenario expands, the computational cost grows exponentially, highlighting the significant challenge of applying these models to larger state spaces.