PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

Task 3 - Tabular Methods

IIC3675 - Reinforcement Learning

Group 40

Group members
William Aarland, Pascal Lopez Wilkendorf

Delivery date: 12th of May 2025

Table of Contents

1 Task a) On- vs. Off-Policy

Both Q-learning and Sarsa are Temporal-Difference (TD) learning algorithms; however, they differ in that Q-learning is *Off-Policy* and Sarsa is *On-Policy*. The main difference lies in how the different algorithms update their Action-Value function Q. For Sarsa, the agent gathers experience by following the same policy as it is trying to improve. As we can see in Equation (1) the action-value function is updated on the actual action taken. In contrary, the Off-Policy learns target policy, while not necessarily acting on the target policy, but rather a behavior policy. In Q-Learning the agent updates the action-value function as if it were acting greedily, even if the agent did not. As seen in Equation (2), where the agent is maxing the action-value function in the current state over all actions, and updating the action-value function thereafter.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{1}$$

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right] \tag{2}$$

2 Task b) Sarsa and Q-Learning acting fully greedy

Assuming both the Sarsa (Algorithm 1) and Q-learning (Algorithm 2) algorithms act greedily, their behavior becomes equivalent in terms of both action selection and value updates. Although the two algorithms differ fundamentally in their learning approach (as discussed in Section 1) this difference disappears under a fully greedy policy.

In this case, the next action a' selected by Sarsa is given by:

$$a' = \arg\max_{a} Q(s', a) \tag{3}$$

As a result, both algorithms perform the same update to the action-value function. The Sarsa update rule (1) becomes numerically identical to the Q-learning update rule (2), since both use the maximal future action value. Therefore, in a deterministic environment with fully greedy action selection, Sarsa and Q-learning are behaviorally and computationally equivalent.

Furthermore, it is evident that the order of operations in the algorithms differ. However, under the strict assumptions of acting fully greedy, fully deterministic and observable, the order does not matter. Since both algorithms choose the next action A using the current policy (fully greedy), take action A, and observe reward R and state S', they experience identical transitions. In Sarsa, the agent then explicitly selects the next action A' from state S' using the same greedy policy and uses $Q(S', A')$ to update the previous value $Q(S, A)$ (Line 8 in Algorithm 1. In contrast, Q-learning does not select an actual next action but instead uses the maximal estimated value $\max_a Q(S', a)$ directly in its update.

Finally, it might look like Sarsa takes two actions per step — first A, then A' — but that is not the case. Only A is actually taken in the environment. A' is chosen only to help calculate the Q-value update. So in both algorithms, only one action is actually executed per step. Therefore, when both are acting greedily, they take the same actions and use the same Q-values to update, which makes their behavior effectively the same in that setting.

So to summarize, under the assumptions of acting fully greedily, deterministic, and the environment is fully observable. Sarsa and Q-learning will act, learn, and update in the same manner.

Algorithm 1 Sarsa (on-policy TD control) for estimating $Q \approx q_*$ [1].

```
1: Parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
2: Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, except that $Q(\text{terminal}, \cdot) = 0$
3: for each episode do
4:     Initialize $S$
5:     Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
6:     while $S$ is not terminal do
7:         Take action $A$, observe $R$, $S'$
8:         Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
9:         $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$
10:        $S \leftarrow S'$; $A \leftarrow A'$
11:    end while
12: end for
```

Algorithm 2 Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$ [1].

```
 1: Parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
 2: Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, except that $Q(\text{terminal}, \cdot) = 0$
 3: for each episode do
 4:     Initialize $S$
 5:     while $S$ is not terminal do
 6:         Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
 7:         Take action $A$, observe $R$, $S'$
 8:         $Q(S, A) \leftarrow Q(S, A) + \alpha\, [R + \gamma \max_a Q(S', a) - Q(S, A)]$
 9:         $S \leftarrow S'$
10:     end while
11: end for
```

3 Task c) Sarsa, Q-learning, and 4-step Sarsa

The task was to compare the performance of Q-learning, Sarsa, and 4-step SARSA in the Cliff Environment. For this experiment, the parameters $\epsilon = 0.1$, $\alpha = 0.1$, and $\gamma = 1$ were used.

From the plot in Figure 1, it is evident that the 4-step SARSA algorithm converges more quickly and stabilizes at a less negative average return compared to the other two algorithms. This is likely to the fact that 4-step SARSA incorporates information from multiple future steps during learning, in contrast to the one-step updates used by Q-learning and standard SARSA. This multi-step approach helps to smooth out high-variance rewards and accelerates convergence.

Initially, during the first 100 episodes, Q-learning and SARSA show similar learning progress. However, as training continues, SARSA stabilizes at a less negative average return than Q-learning. Our initial hypothesis was that Q-learning would learn a riskier but shorter path along the cliff edge, leading to higher average returns. Contrary to this expectation, the results show that Q-learning experiences greater instability and converges to a lower average return. A plausible explanation is that Q-learning's off-policy updates encourage the optimal (but risky) policy, while the agent continues to follow an ϵ-greedy behavior during training. This leads to more frequent falls into the cliff, resulting in lower rewards. In contrast, SARSA's on-policy nature encourages a safer route during training, reducing the frequency of high-penalty outcomes.

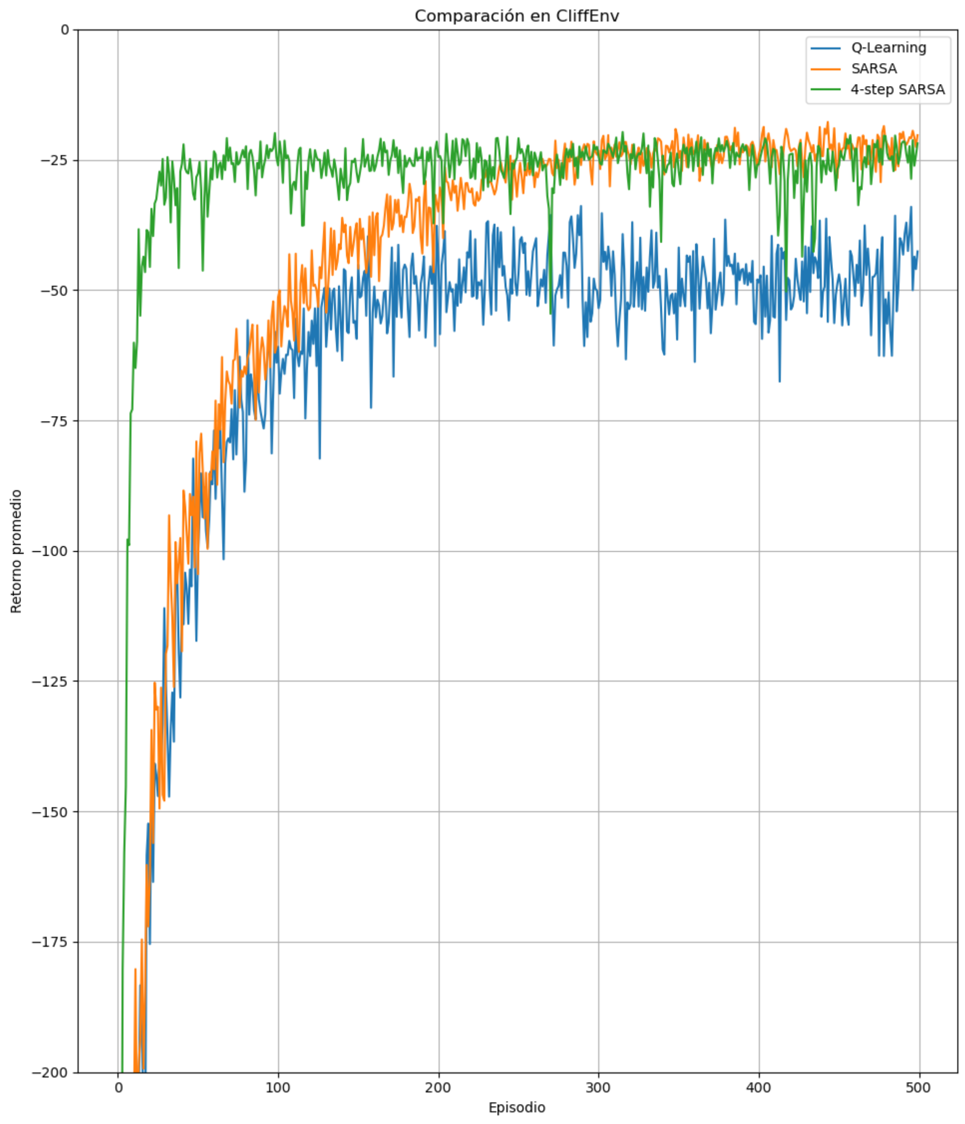

Figure 1: Comparison of average return per episode in the Cliff Environment for Q-learning, SARSA, and 4-step SARSA

4 Task d) Dyna-Q and RMax

This task incorporates comparing the average reward from Dyna-Q algorithm and the RMax algorithm in the `EscapeRoomEnv`. For Dyna-Q with different planning step sizes, and using $\gamma = 1$ for both, and for Dyna-Q using $\alpha = 0.5$, $\epsilon = 0.1$, and initializing all Q-values to 0.

Table 1 shows the results of the experiment. With the number of planning steps in the Dyna-Q algorithm, the average return per episode decreases, showing that by incorporating a simulated environment helps the agent learn a better policy more efficiently. Notably, the RMax algorithm outperforms even the Dyna-Q algorithms with the highest number of planning steps. Even with a high number of planning steps, making the Dyna-Q algorithm resemble more a model-based approach, the explicit explorative nature of RMax gives stronger guidance to the agent in such an environment as `EscapeRoomEnv`.

Method	**Planning Steps**	**Average return**
Dyna-Q	0	-1749.39
Dyna-Q	1	-1394.15
Dyna-Q	10	-1258.04
Dyna-Q	100	-900.79
Dyna-Q	1000	-651.86
Dyna-Q	10000	-606.91
RMax	–	-547.70

Table 1: Average return per episode in the `EscapeRoomEnv` environment when using Dyna-Q with different amounts of planning steps, compared to RMax.

Regarding whether Dyna-Q approaches the RMax algorithm when the number of planning steps becomes sufficiently large; the answer is no. Although both algorithms combine planning, and learning, the main difference lies in their approach to exploration. While Dyna-Q uses ϵ-greedy exploration, it does not include an explicit exploration mechanism such as RMax. Whereas RMax sets the reward of every state that has been visited fewer than k times to the maximum reward (giving an optimistic exploring strategy). Therefore, although Dyna-Q with many planning steps may behave similarly to RMax in some aspects, they are not equivalent in terms of exploratory behavior or learning efficiency.

5 Task e) Multi-goal environments

In this task, we evaluate and compare the performance of various tabular reinforcement learning algorithms in a multi-goal environment. The domain used is `RoomEnv`, where each episode begins with a randomly sampled goal state. The environment is fully observable, and the state is represented as a pair $\langle s, g \rangle$, where s is the agent's current state and g is the designated goal. The algorithms evaluated include Q-learning, Sarsa, 8-step Sarsa, and multi-goal version of both Q-learning and Sarsa.

For this task the hyperparameters $\alpha = 0.1$, $\epsilon = 0.1$, $\gamma = 0.99$, and all Q-values initialized to 0. Each method is trained with 100 repetitions, with 500 episodes per repetition.

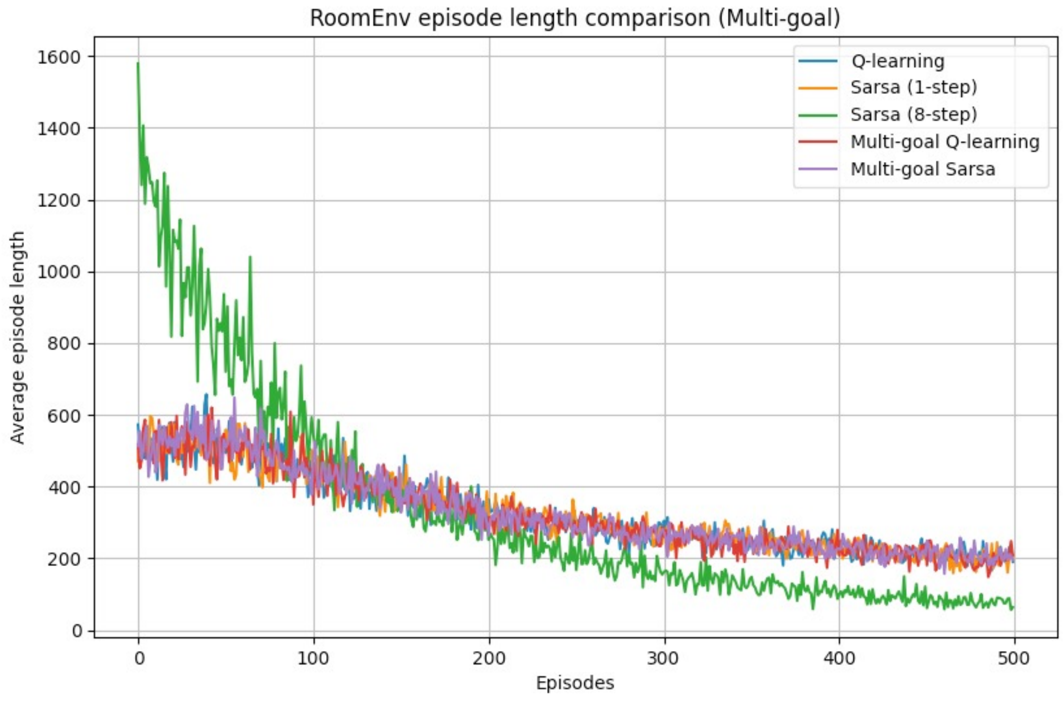

Figure 2: Comparison of average episode length over 500 episodes for Q-learning, SARSA (1-step), SARSA (8-step), and their respective multi-goal variants in `RoomEnv`.

Training the different agents in the `RoomEnv` environment provides several insights. Notably, the 8-step SARSA agent begins with a higher average episode length, but eventually outperforms all other methods after approximately 200 episodes. This aligns with theory, as multi-step methods can propagate reward signals more efficiently across time, allowing the agent to learn optimal paths more quickly in sparse-reward environments. In contrast, the multi-goal agents do not demonstrate a significant improvement over standard Q-learning or 1-step SARSA. While multi-goal learning is theoretically advantageous—since it reuses each experience to update the value function for all possible goals, thereby aiding the agent generalize—this advantage was not clearly reflected in the results. A possible explanation is the sparsity of rewards in the environment, where each transition yields a zero reward except when the state is the goal yielding a reward of +1. Making many updates less informative. As a result, the benefit of multi-goal generalization may have been diminished in this particular setting. Based on the results of the experiment, there does not seem to be a clear advantageous method when comparing the multi-goal Sarsa, and Q-learning. Since both methods correlate and follow each other closely.

6 Task f) Multi-Agent Reinforcement Learning

In this task, we explore different multi-agent reinforcement learning (MARL) settings using three variants of a predator-prey environment. The goal is to understand how the coordination and competition between agents affect learning efficiency.

The different environments are `CentralizedHunterEnv`, `HunterEnv`, and `HunterAndPreyEnv`. For `CentralizedHunterEnv` the prey has a fixed policy of running away, and the hunters are controlled conjointly. In `HunterEnv` the difference is that the hunters are working independently, a decentralized approach. The last environment, `HunterAndPreyEnv`, is similar to `HunterEnv` with a decentralized learning for the hunters, but contrary to the other environments, the prey also learns. All the experiments were run with $\gamma = 0.95$, $\epsilon = 0.1$, $\alpha = 0.1$, and with Q-values initialized to 0. The environments were run 30 times for 50 000 episodes.

6.1 Results

As shown in Figure 3 the Competitive Q-learning setup (`HunterAndPreyEnv`) converges the slowest. In this environment, the prey also learns, giving a more dynamic behavior between the agents, simulating a more realistic behavior. Since the prey adapts its policy over time, the hunters must constantly re-learn how to catch it. Initially, the average episode length is relatively low, likely because all agents are acting randomly with high exploration. Neither the hunters nor the prey have yet learned effective strategies. As training progresses, the average episode length increases, indicating that the prey is learning to avoid the hunters, while the hunters struggle to coordinate. The decentralized nature of the hunters would also explain the increased episode length, as each hunter must act independently, and without coordination could lead to increased difficulty in catching the prey.

In the decentralized environment `HunterEnv`, the initial episode lengths are significantly higher than in the centralized version (`CentralizedHunterEnv`). This is expected, as the hunters in the decentralized case must learn to coordinate purely through shared rewards, without access to each other's internal states or policies. However, over time both approaches converge to similar episode lengths, suggesting that coordination can occur even without centralized control. Interestingly, the decentralized version appears to slightly outperform the centralized agent in the long run. This could be attributed to greater flexibility and scalability, as decentralized architectures are typically more adaptable in larger or more complex multi-agent systems.

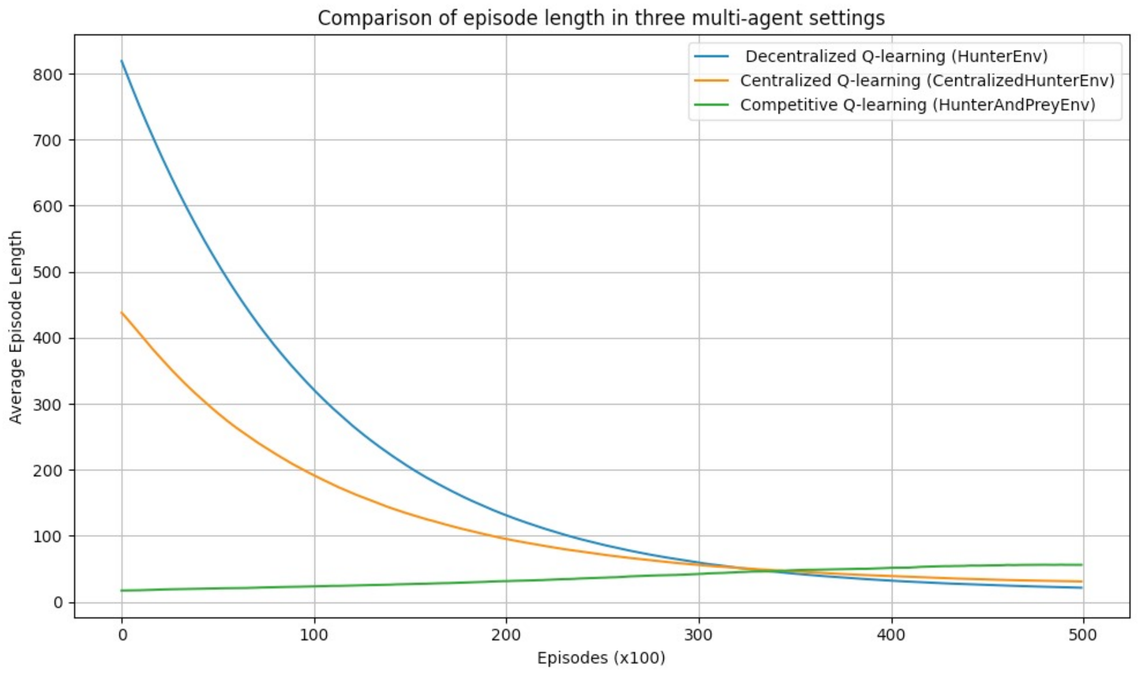

Figure 3: Smoothed average episode length over training for three multi-agent Q-learning setups: centralized cooperative, decentralized cooperative, and decentralized competitive.

7 Task g)

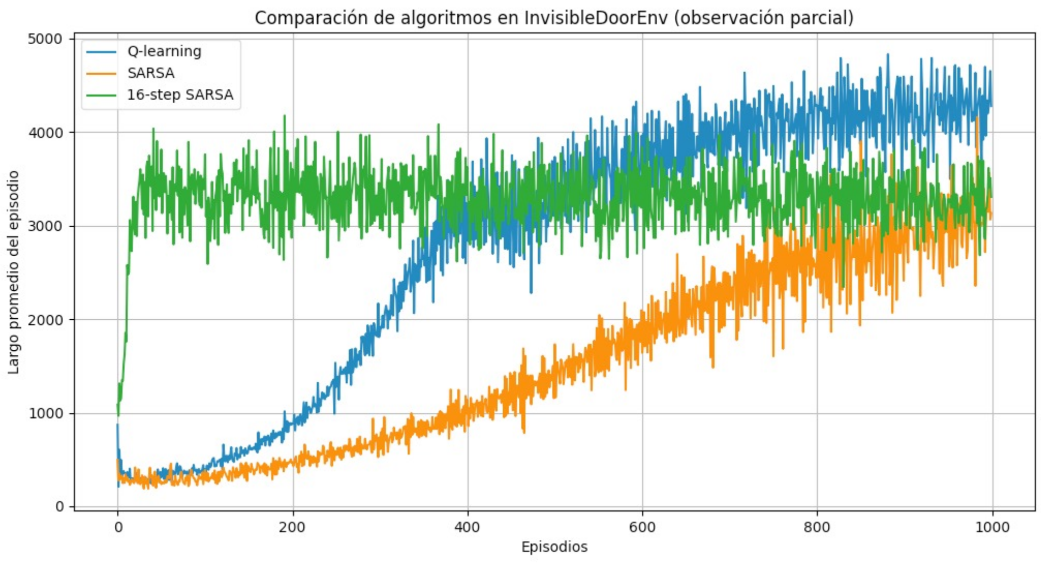

Figure 4: Average length per episode on InvisibleDoorEnv

In this experiment, we evaluate the performance of different reinforcement learning algorithms in a partially observed environment, specifically InvisibleDoorEnv. In this type of environment, the agent cannot fully observe the state of the environment, which breaks the Markov property and makes memoryless policies (based only on current observation) suboptimal. We notice that as the agent learns the lengths of episodes become longer. This is because at first the problem could be solved by chance, but as the episodes progress, the agent prefers suboptimal but safe policies.

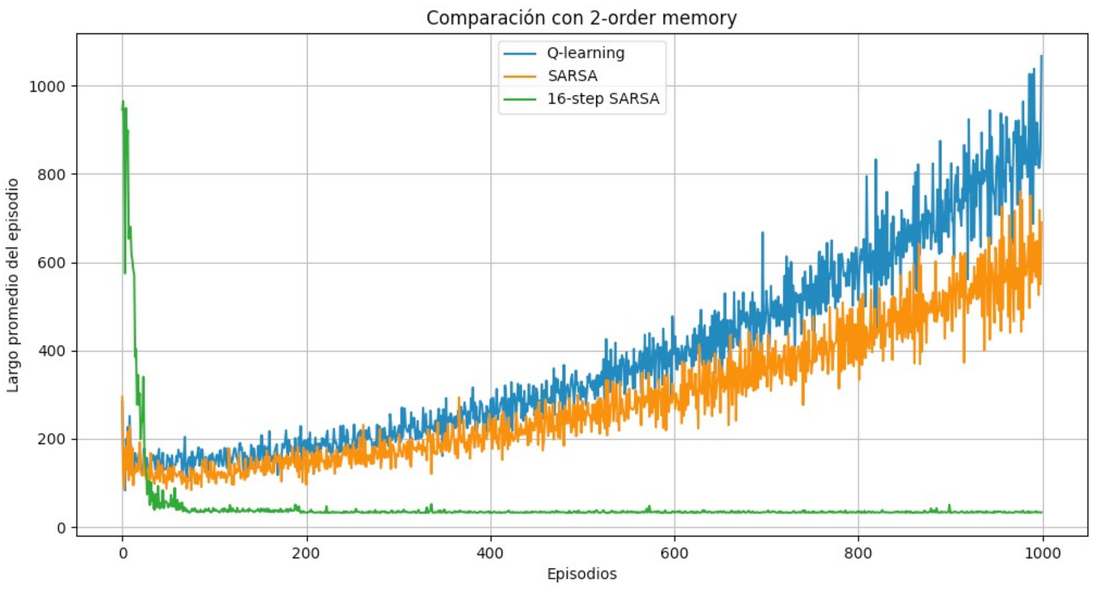

Figure 5: Average episode length in InvisibleDoorEnv using 2-order memory

In this experiment, by adding a second-order memory, the agent can now make decisions considering not only the current observation, but also the immediately preceding observation. This allows it to infer something about the hidden state of the environment.In the case of nstep sarsa, this algorithm manages to converge because by using many steps and having a memory of order 2, it can better exploit the reward signal in long trajectories, and learns the structure of the environment faster.

In the cases of Q-learning and SARSA, performance improves but it is still not able to converge to the optimum.

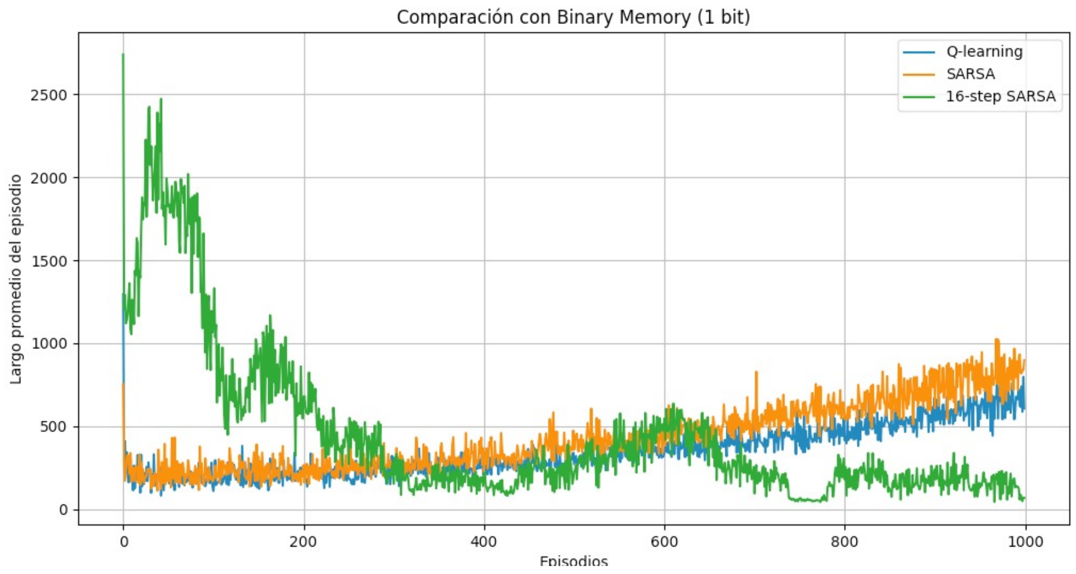

Figure 6: Average episode length in InvisibleDoorEnv using binary memory.

We observe that SARSA-16 converges, but more slowly than with 2-order memory. This is expected because, although binary memory gives the agent control over memory, learning to use it correctly requires exploring combinations between actions and memory decisions, which adds complexity to the action space. We also observe that Q-learning and SARSA move closer to the optimal approach; both algorithms now learn more about the structure of the environment and achieve policies that are closer to optimal.

References

[1] Richard S. Sutton and Andrew Barto. *Reinforcement learning: An Introduction.* eng. Second edition. Adaptive computation and machine learning. Cambridge, Massachusetts London, England: The MIT Press, 2020. ISBN: 978-0-262-03924-6.