Learning and Optimization Algorithms

Implementation and Analysis
November 2025

Authors:

Alejandro Martínez Hermosa
Martín Serra Rubio

Contents

1 Introduction

For this analysis, we had to implement the reinforcement learning algorithms Q-Learning, Sarsa, Dynamic Programming, and Monte Carlo, as well as a genetic algorithm in the *Frozen Lake* game, created by OpenAI.

2 Game Explanation

The game is very simple. Given a board of dimensions $n \times m$, the goal is for the agent to find a path from its tile (marked as a chair) to the target tile (marked with a gift). If it falls into one of the frozen lakes or exceeds/reaches 100 steps on a 4×4 map or 200 steps on an 8×8 map, the agent loses.

To reach the objective, the agent can move in any of the 4 possible directions, represented directly with an integer for each direction from 0 to 3. However, because the parameter `is_slippery` is enabled, the agent will have a certain probability of performing the action in any of the other two parallel directions. These two actions are equiprobable.

To know which tile it is on, in each movement the position of the agent is indicated with an integer represented by the following formula:

Position Formula

$$\text{current\_position} = \text{current\_row} \cdot \text{number\_columns} + \text{current\_column}$$

This formula simply represents that each tile has an integer of the form indicated in the image below:

Regarding the reward obtained, the agent will receive a reward of +1 if it reaches the objective. Otherwise, the reward obtained will be 0.

3 Design Decisions

3.1 Reinforcement Learning Algorithms

First, we generated the state probability matrix by initializing all probabilities to 0. In the rows, we added the possible states, that is, all the tiles on the board, and in the columns, we added the possible movements, i.e., right, left, up, or down.

Its implementation has been quite simple since, as mentioned in the game explanation section, the possible states are numbered in order starting from 0, just like the actions. Therefore, it was very easy to implement this, as the indices of the arrays we use with the `numpy` library start at 0, meaning the first element of the array has index 0.

3.1.1 Q-Learning

Regarding the Q-Learning algorithm:
The modification of this matrix was done using the following formula, thus implementing the Q-Learning algorithm:

Q-Learning Update Function

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

Regarding the code structure, we follow the classic Q-Learning scheme, maintaining a table Q that updates after each interaction with the environment. We implement an episode loop where the agent selects actions via an ε-greedy policy and adjusts Q values using the standard update equation. Furthermore, we organize the program into separate functions for initialization, update, and action selection, which facilitates code readability and reusability.

3.1.2 Sarsa

Regarding the Sarsa algorithm:
The modification of this matrix was done using the following formula, thus implementing the Sarsa algorithm:

Sarsa Update Function

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

Regarding the code structure, the SARSA implementation follows the on-policy approach, updating Q values based on actions selected by the agent's own policy. The episode loop integrates action choice via an ε-greedy strategy and the continuous update of the Q table, maintaining a modular organization to facilitate algorithm clarity and extensibility.

3.1.3 Monte Carlo

Regarding the Monte Carlo algorithm:
Regarding the code structure, the Monte Carlo method is based on generating complete episodes and updating $Q(s, a)$ values from the average return of all visits. We implement the ε-greedy policy incrementally and use trajectory lists to record transitions and calculate returns, maintaining the code organized in independent functions.

Monte Carlo Update Function

$$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$$

A two-phase structure was implemented: first, the complete episode is generated and recorded in a temporary list under an ϵ-greedy policy to ensure exploration, and then a backward sweep is performed to calculate the accumulated return (G). Following the pseudocode, the First-Visit logic is applied (checking the history to ignore repeated states in the same episode) and the Q matrix is updated via an incremental mean (1/N) instead of a fixed rate, thus calculating the exact mean of the returns obtained.

3.1.4 Dynamic Programming

Regarding the dynamic programming algorithm:
When the environment has the attribute `is_slippery=True`, the agent's actions become unpredictable: moving in a specific direction does not guarantee reaching the desired state; rather, the agent has a probability of ending up in a different state.

Dynamic programming works by assuming that the environment model is completely known and deterministic, meaning we know for sure which state will be reached after each action. When transitions are not deterministic, as occurs with `is_slippery=True`, this no longer works and makes it very difficult to calculate the exact optimal policy, as there is no guaranteed path to the goal and planning based on a fixed model becomes very complicated.

3.2 Genetic Algorithm

Regarding the genetic algorithm:
The pseudocode was taken from slide 108 of the document **Topic 4 Search (complete)**, with this, the only difficulty was creating the individual model and the evaluation function for each one.

Regarding the individual: each individual is an array of length 16 representing a policy the agent will follow; each slot of the array contains a number $i \in \{0, 1, 2, 3\}$ which is the move it will make in that state.

Regarding the function fitness$(p : \text{policy}, \text{episodes} : \text{int})$, all it does is that each policy "plays" a number of *episodes* times and evaluates how good it is by dividing total_rewards/episodes.

As an extra addition, we have created **elite** individuals; these individuals are the best individuals of the current generation and we keep them so as not to lose them. With this individual/fitness model, the same process is repeated *generations* (function parameter) times; each generation selects the elites, evaluates all individuals, and creates a list of children selecting parents according to the *fitness* function. Once with the children, mutations are applied to them and they are joined with the elites obtaining the new generation which will be subjected to this cycle again.

4 Performance Comparison

4.1 Q-Learning

To analyze the performance of Q-Learning, we have generated graphs showing the probability (normalized to 1) of success in each episode for the last 100 measured episodes.

How do the parameters affect the graph?:

Base Experiment Parameters

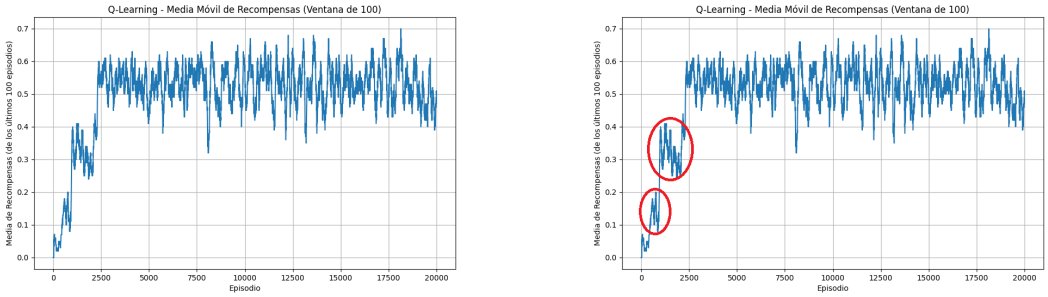
How do the parameters affect it?:

- α: is the value that determines how smooth the curve is; this gives more or less importance to the new data arriving at the matrix.
- γ: is the value that determines how high the curve can reach; thanks to this, future actions are taken into account just like current ones.
- ϵ: is the value that determines how *greedy* the policy is; a very high ϵ can make what is learned useless as it always chooses random moves, while a low ϵ can cause it to find bad paths as good solutions and stagnate. It might be convenient to have an ϵ_{decay} to modify ϵ as the agent learns, since initially data is not very reliable and it should be less *greedy*.
- *episodes*: is the simplest parameter to see; if there are few episodes the policy will not learn enough and may stay in a local maximum or a bad zone, while if there are too many executions we only unnecessarily prolong it as it may have already reached the maximum.

Depending on the different values of the parameters α, γ, ϵ and *episodes* of the function, different graphs are obtained:

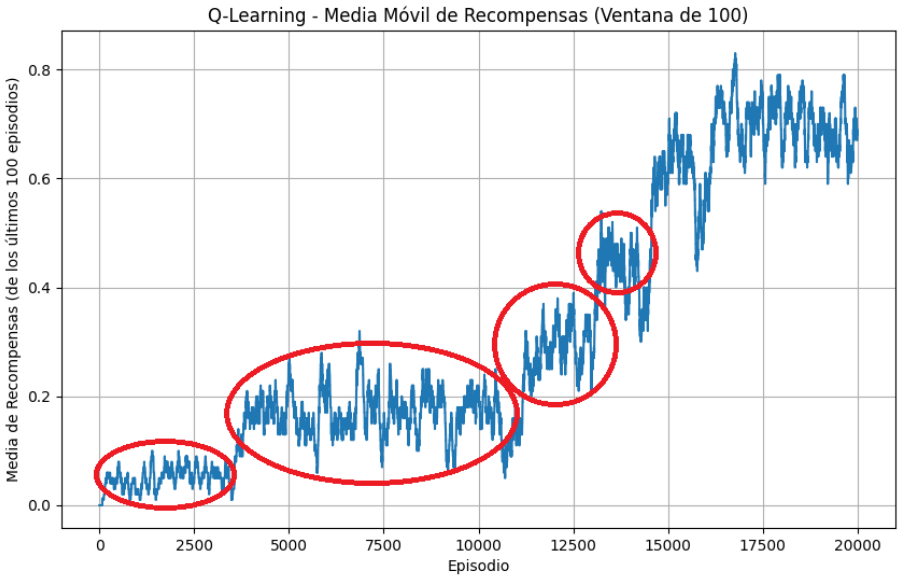
This first graph shows a standard execution of 20,000 episodes.

Baseline Q-Learning ($\alpha = 0.1, \gamma = 0.99, \epsilon = 0.05$)



Analysis: Clear learning is visible. Thanks to $\epsilon = 0.05$, stagnation in mediocre policies (red circles) is avoided; these stagnations could be local maximums from which it exits by being greedy.

Baseline Q-Learning ($\alpha = 0.05, \gamma = 0.99, \epsilon = 0.005$)

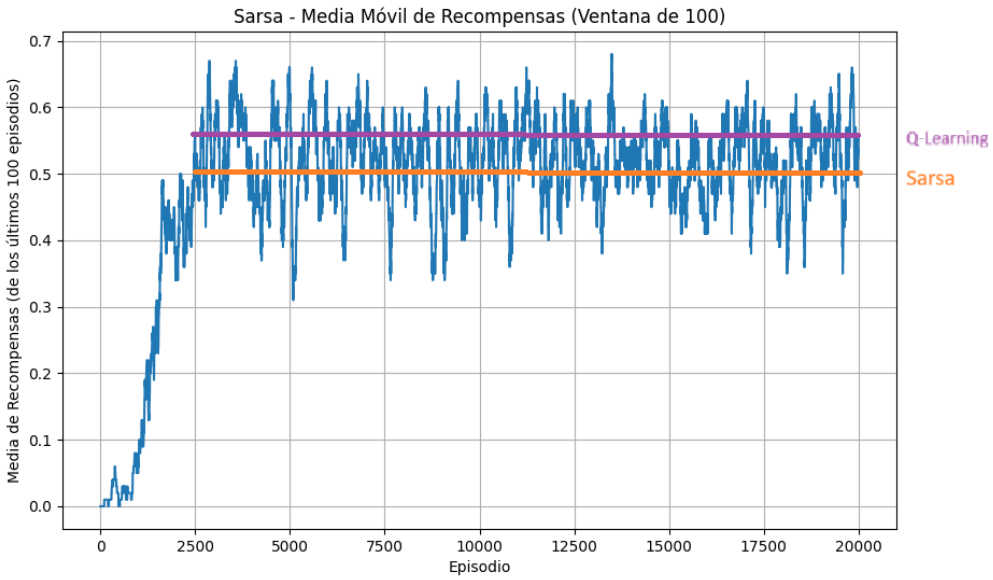


Analysis: And with $\epsilon = 0.005$ it struggles much more to exit local maximums, but when it perfects the policy, it is better.

4.2 Sarsa

Performing the same execution as with Q-Learning.

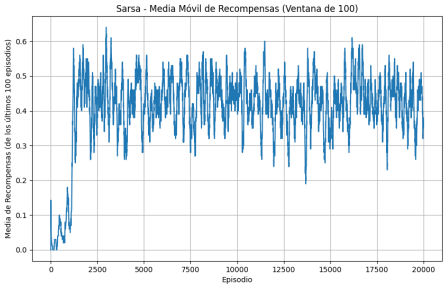
Comparison: Sarsa vs Q-Learning ($\alpha = 0.1$, $\gamma = 0.99$ and $\epsilon = 0.05$)



Conclusion: The only tangible difference between the algorithms is that ***Q-Learning* has a slight superiority** in terms of ensuring victory (it has a more *greedy* behavior regarding optimal values), while Sarsa remains slightly below due to its conservative nature (*on-policy*).

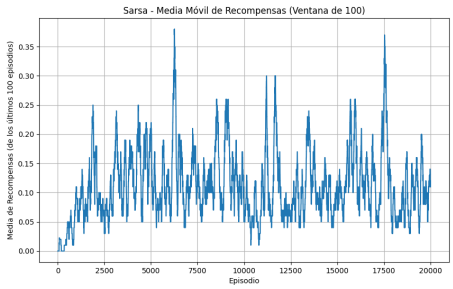
Playing now with other parameters, if we change α, the following happens:

High Alpha Effect ($\alpha = 0.3$)



Observation: More drastic peaks and great instability are achieved.
This happens because too much value is given to new data, destabilizing the agent's previous knowledge.

Low Gamma Effect ($\gamma = 0.5$)

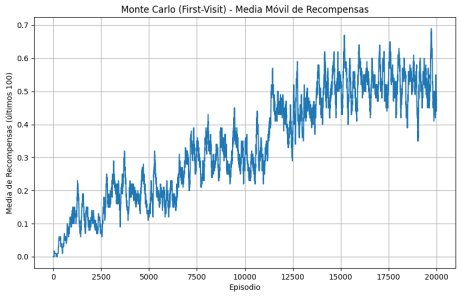


Observation: Very little improvement is obtained.
Lowering gamma makes the agent "myopic", ignoring future rewards (the final gift) and focusing only on the immediate step, which gives no points.

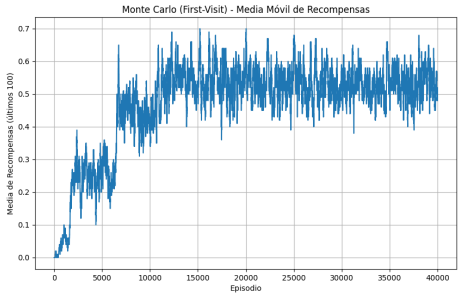
4.3 Monte Carlo

Using the Monte Carlo algorithm (First-Visit)

Monte Carlo Evolution: 20k vs 40k episodes ($\gamma = 0.99$ and $\epsilon = 0.05$)



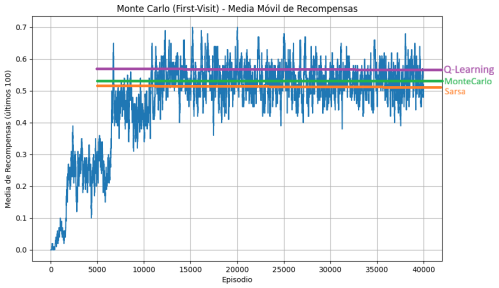
20,000 Episodes: Still in the learning phase.



40,000 Episodes: Stabilization achieved.

Monte Carlo is an algorithm that generally converges more slowly; this is because to update, it must wait for the end of each episode, unlike the other two algorithms which update with each movement made. That is why to obtain a graph truly comparable with those of *Q-Learning* and *Sarsa*, we have to perform a training of episodes=40000.

Comparison over time



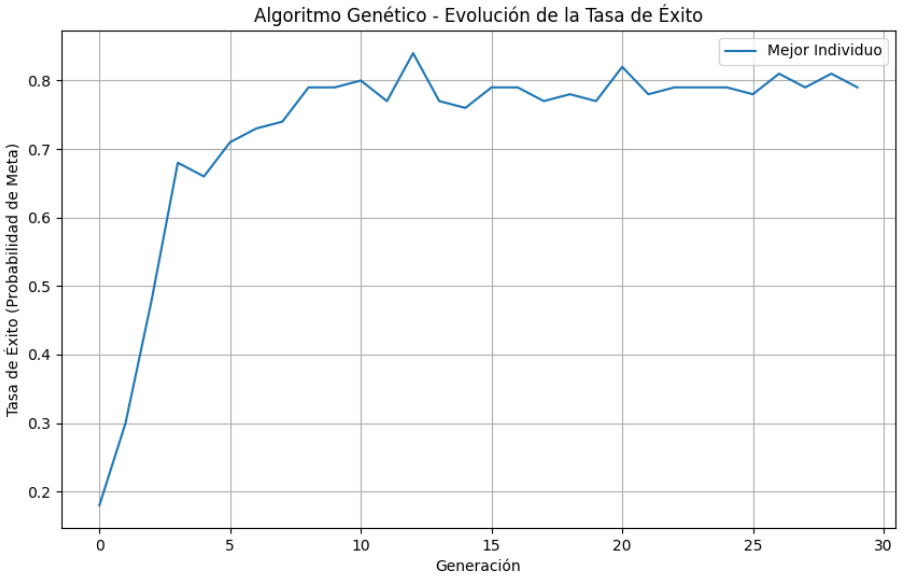
Observation: In the long term, Monte Carlo equals Sarsa and Q-Learning.

4.4 Genetic Algorithm

In the case of the Genetic Algorithm, since it does not have episodes but generations, the graph is less dense. Each generation has individuals that are evaluated and merged; if we take the best

evaluation of the individuals of each generation, we obtain the following graph:

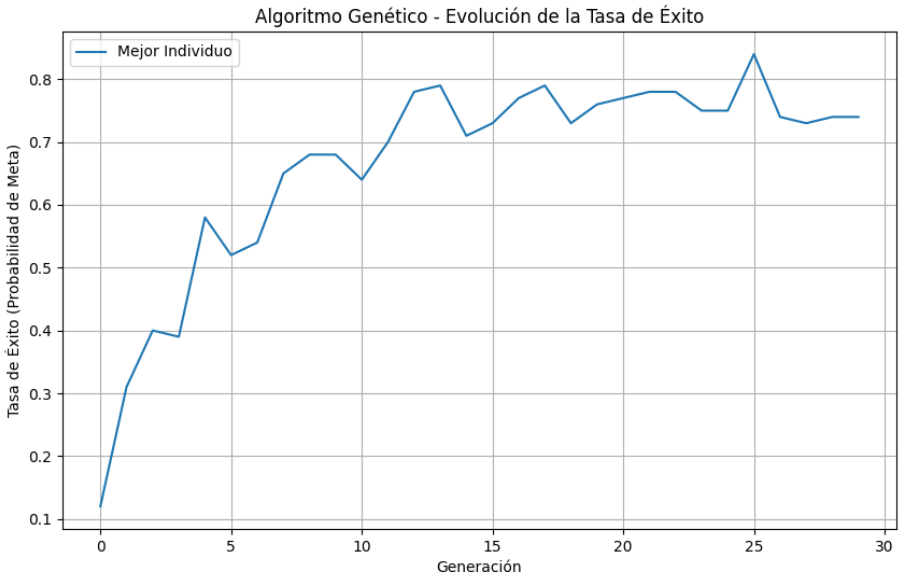
Genetic Algorithm (*generations*=30,*population_size*=100,mutation=0.05,elites=5)



Is it really as good as it seems?

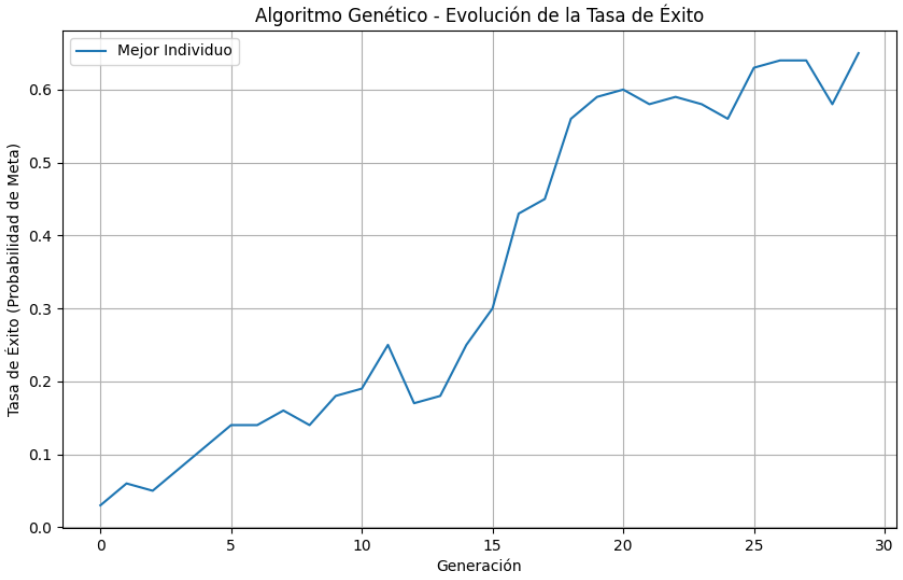
This algorithm might seem very promising; within a few generations, it rises to a very high value equal to or greater than *Q-Learning*. The problem with the genetic algorithm is poor scalability; the *Frozen-Lake* problem with a 4x4 map is very small.
With 30 generations of 100 individuals each, it already takes an average of 2 minutes of training.
Therefore, on a 100x100 map, we would have 10,000 states, our individuals would be arrays of *length*=10,000 with $policy[n] \in \{0, 1, 2, 3\}$, the merger between policies would probably yield bad policies, and the $fitness$ function would be unscalable, having to execute each individual many times.

Changes in the *mutation* parameter



Analysis: Changing the *mutation* parameter to a higher one (=0.3), the graph changes just as if we changed the ϵ parameter in the others: a more unstable rise and a peak that varies more throughout the generations.

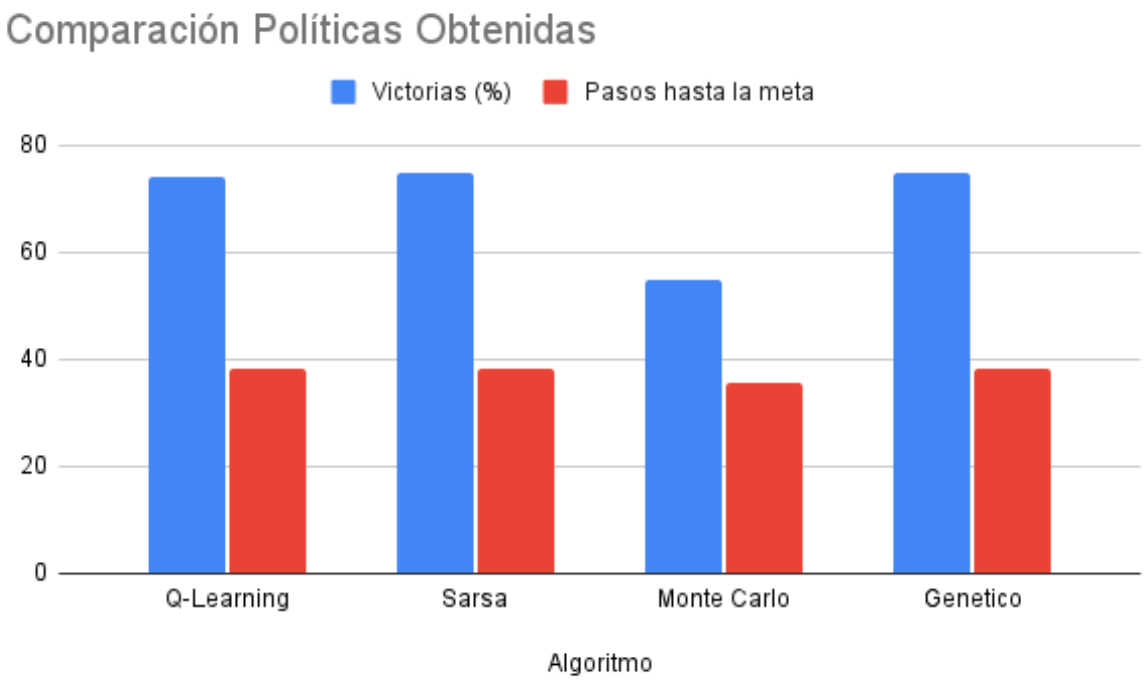
Reducing the *population_size* parameter



Analysis: Continuing with the initial parameters, if we now change the amount of policies per generation (imposed at 100 initially to ensure) to a smaller amount, the graphs are inconsistent; it depends a lot on the luck of the initial random policies.

4.5 Global Comparisons

Once the different algorithms were compared during training, we took 15 trained policies from each and evaluated them. The evaluation consists of executing them 100 times with a maximum of 100 *steps*; in the end, the average number of times it reaches the goal and the average steps of the cases that reached the goal are calculated:



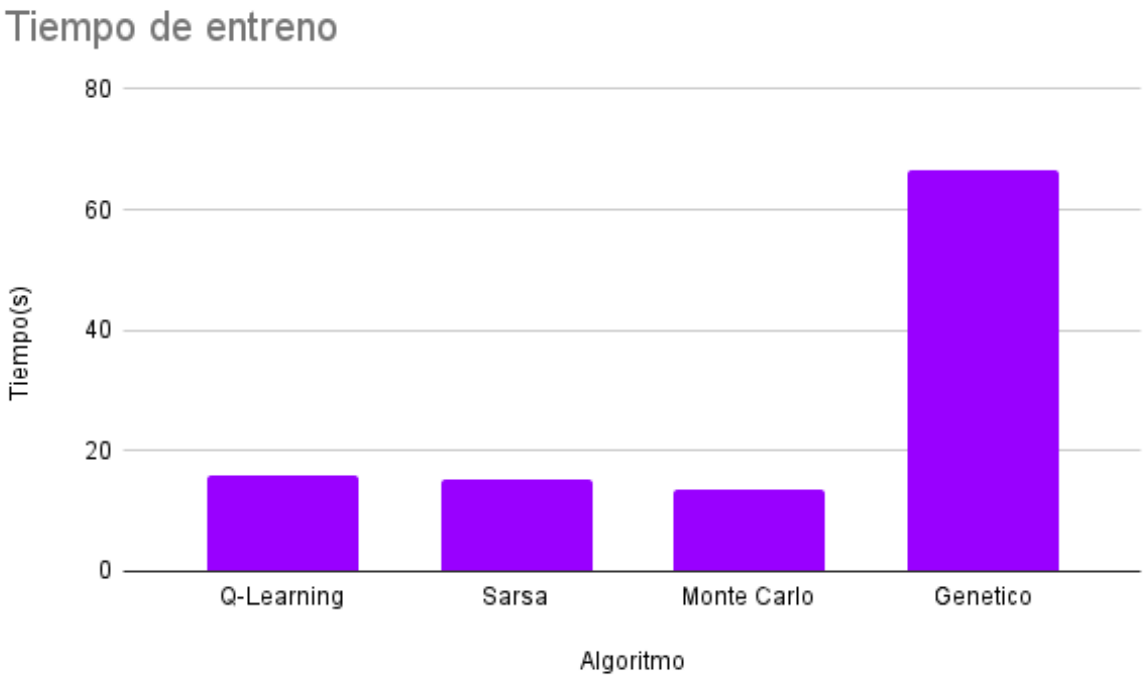
Experimental Setup

Reinforcement Learning
- **Episodes:** 20,000
- **Alpha (α):** 0.1
- **Gamma (γ):** 0.99
- **Epsilon (ϵ):** 0.05

Genetic Algorithm
- `population_size`: 100
- `generations`: 30
- `prob_mutation`: 0.05
- `elites`: 5

A clear minority of victories is seen using Monte Carlo; this is because it grows much more slowly than the other algorithms. With double the episodes, it would have the same % as the others.

Another comparison we made is the average training time of each algorithm with the same parameters as before, resulting in:



All have a similar time except the genetic algorithm, which skyrockets due to the large number of evaluations performed by the *fitness* function and all the generations created to perfect the policy.

5 Analysis of Resulting Policies

5.1 Sarsa

Sarsa is an *on-policy* algorithm, meaning it learns using the same policy used to act. The main indicators are:

- **Cumulative reward:** shows if the agent learns to reach the goal. It is customary to use the average of several episodes to smooth out variability.
- **Episode duration:** measures how many steps the agent takes to complete an episode; shorter duration with high reward indicates a more efficient policy.
- **TD Error:** indicates how well the value function is adjusting; when it decreases towards zero, the policy stabilizes.
- **Exploration and exploitation:** the percentage of random actions decreases as the agent learns, showing policy stability.

5.2 Q-Learning

Q-Learning is *off-policy*; it updates values assuming the agent will always choose the best action, regardless of the exploration policy. Key indicators:

- **Cumulative reward and duration:** tends to grow faster than Sarsa, although it may present oscillations due to aggressive updating.
- **Convergence of the Q-table:** measures the stability of action values; when it remains stable, the policy has converged.
- **Final policy:** the coherence of optimal actions and the success rate in the environment are analyzed.

5.3 Monte Carlo

Monte Carlo methods calculate action values from the complete return of episodes. Relevant indicators:

- **Cumulative reward and duration:** similar to other methods, although variability is higher in initial episodes.
- **Average value per state-action:** the estimation is based on the average of observed returns; the policy adjusts as more episodes are accumulated.
- **Policy stability:** measured by the consistency of values and chosen actions, as well as the reduction of changes in the policy over time.

5.4 Genetic Algorithm

- **Conservative policies:** the generated policies tend to be safer and less risky, prioritizing reaching the goal over efficiency.
- ***Fitness* function:** evaluates each individual by executing the policy only 100 times and counting how many reach the goal, without taking into account path length.
- **Policy adjustment:** as a result, selected actions seek to maximize the success of completing the episode, rather than reducing the number of steps.

6 Bibliography

- Gymnasium Frozen_Lake Documentation
- Gymnasium Env Documentation
- Gymnasium Discrete Documentation