# Algoritmos de aprendizaje y optimización

## Implementación y análisis
Noviembre 2025

**Autores:**

Alejandro Martínez Hermosa
Martín Serra Rubio

# Índice

# 1 Introducción

Para la realización de este análisis, hemos tenido que implementar los algoritmos de aprendizaje por refuerzo Q-Learning, Sarsa, Programación Dinámica y Montecarlo y un algoritmo genético en el juego de *Frozen Lake*, creado por OpenAI.

# 2 Explicación del juego

El juego es muy sencillo. Dado un tablero de dimensiones $n \times m$ hay que conseguir que el agente encuentre un camino desde su casilla (marcada como una silla) hasta la casilla objetivo (marcada con un regalo). Si se cae por uno de los lagos congelados o supera/llega a dar 100 pasos en un mapa $4 \times 4$ o 200 pasos en un mapa $8 \times 8$, el agente pierde.

Para poder llegar al objetivo, el agente se puede mover en cualquiera de las 4 posibles direcciones, representado directamente con un número entero para cada dirección del 0 al 3, aunque, debido a que el parámetro `is_slippery` está activado, el agente tendrá cierta probabilidad de realizar la acción en cualquiera de las otras dos direcciones paralelas. Estas dos acciones son equiprobables.

Para saber en qué casilla está, en cada movimiento se indica cuál es la posición del agente con un número entero representado por la siguiente fórmula:

**Fórmula de posición**

$$\text{posicion\_actual} = \text{fila\_actual} \cdot \text{numero\_columnas} + \text{columna\_actual}$$

Esta fórmula solo representa que cada casilla tiene un número entero de la forma que se indica en la imagen a continuación:


En cuanto a la recompensa obtenida, el agente obtendrá una recompensa de +1 en caso de llegar al objetivo. En caso contrario, la recompensa obtenida será de 0.

# 3 Decisiones de diseño

## 3.1 Algoritmos de aprendizaje por refuerzo

Primero hemos generado la matriz de probabilidad de estados inicializando todas las probabilidades a 0. En las filas hemos añadido los posibles estados, es decir, todas las casillas del tablero, y en las columnas hemos añadido los posibles movimientos, es decir, derecha, izquierda, arriba o abajo.

Su implementación ha sido bastante sencilla ya que, como he mencionado en el apartado de explicación del juego, los posibles estados están numerados en orden a partir del 0, igual que las acciones, por lo que ha sido muy sencillo implementar esto, debido a que los índices de los arrays que usamos con la librería `numpy` empiezan por 0, es decir, el primer elemento del array tiene el índice 0.

### 3.1.1. Q-Learning

La modificación de esta matriz se ha hecho mediante la siguiente fórmula, implementando así el algoritmo de Q-Learning:

**Función de Actualización Q-Learning**

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

En cuanto a la estructura del código, seguimos el esquema clásico de Q-Learning, mantenemos una tabla $Q$ que se actualiza después de cada interacción con el entorno. Implementamos un bucle de episodios donde el agente selecciona acciones mediante una política $\varepsilon$-greedy y ajusta los valores $Q$ usando la ecuación de actualización estándar. Además, organizamos el programa en funciones separadas para la inicialización, la actualización y la selección de acciones, lo que facilita la legibilidad y la reutilización del código.

### 3.1.2. Sarsa

La modificación de esta matriz se ha hecho mediante la siguiente fórmula, implementando así el algoritmo de Sarsa:

**Función de Actualización Sarsa**

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

En cuanto a la estructura del código, la implementación de SARSA sigue el enfoque on-policy, actualizando los valores $Q$ a partir de las acciones seleccionadas por la propia política del agente. El bucle de episodios integra la elección de acciones mediante una estrategia $\varepsilon$-greedy y la actualización continua de la tabla $Q$, manteniendo una organización modular para facilitar la claridad y la extensibilidad del algoritmo.

### 3.1.3. Monte Carlo

En cuanto a la estructura del código, el método Montecarlo se basa en generar episodios completos y actualizar los valores $Q(s,a)$ a partir del retorno promedio de todas las visitas. Implementamos la política $\varepsilon$-greedy de manera incremental y empleamos listas de trayectoria para registrar transiciones y calcular retornos, manteniendo el código organizado en funciones independientes.

**Función de Actualización de Montecarlo**

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

Se implementó una estructura de dos fases: primero se genera y registra el episodio completo en una lista temporal bajo una política $\epsilon$-greedy para garantizar la exploración, y después se hace un barrido hacia atrás para calcular el retorno acumulado (G). Siguiendo el pseudocódigo, se aplica la lógica First-Visit (verificando el historial para ignorar estados repetidos en el mismo episodio) y se actualiza la matriz Q mediante una media incremental (1/N) en lugar de una tasa fija, calculando así la media exacta de los retornos obtenidos.

### 3.1.4. Programación dinámica

Cuando el entorno tiene el atributo `is_slippery=True`, las acciones del agente se vuelven imprevisibles: moverse en una dirección determinada no garantiza llegar al estado deseado, sino que el agente tiene una probabilidad de acabar en un estado diferente.

La programación dinámica funciona asumiendo que el modelo del entorno es completamente conocido y determinista, es decir, que sabemos con seguridad qué estado se conseguirá después de cada acción. Cuando las transiciones no son deterministas, como ocurre con `is_slippery=True`, esto ya no funciona y hace que sea muy difícil calcular la política óptima exacta, ya que no hay un camino garantizado hacia la meta y la planificación basada en un modelo fijo se complica mucho.

## 3.2 Algoritmo Genético

El código del algoritmo genético está organizado en módulos: inicialización de la población, evaluación, selección, cruce y mutación. Cada función es independiente, lo que facilita ajustar parámetros y reutilizar el código

Cada individuo es un array de longitud 16 que representa una política que seguirá el agente, cada slot del array contiene un número $i \in \{0,1,2,3\}$ indicando su movimiento según el estado.

En cuanto a la función fitness($p$ : política, episodios : int) lo único que hace es que cada política "juegaün número *episodios* de veces y se evalúa cuán buena es dividiendo recompensas_totales/episodios.

Como añadido extra, hemos creado individuos **élite**, estos individuos son los mejores individuos de la generación actual y los guardamos para no perderlos. Con este modelo individuo/fitness, se repite el mismo proceso *generaciones* (parámetro de la función) veces, cada generación selecciona los élites, evalúa todos los individuos, y crea una lista de hijos seleccionando los padres según la función *fitness*, una vez con los hijos se aplican las mutaciones a estos y se unen con los élites obteniendo la nueva generación que será sometida a este ciclo otra vez.

# 4 Comparación de rendimientos obtenidos

## 4.1 Q-Learning

Para analizar el rendimiento de Q-Learning, hemos generado gráficos que muestran la probabilidad en tanto por 1 de éxito en cada episodio de los últimos 100 episodios medidos.

**¿Cómo afectan los parámetros a la gráfica?:**

**Parámetros Base del Experimento**

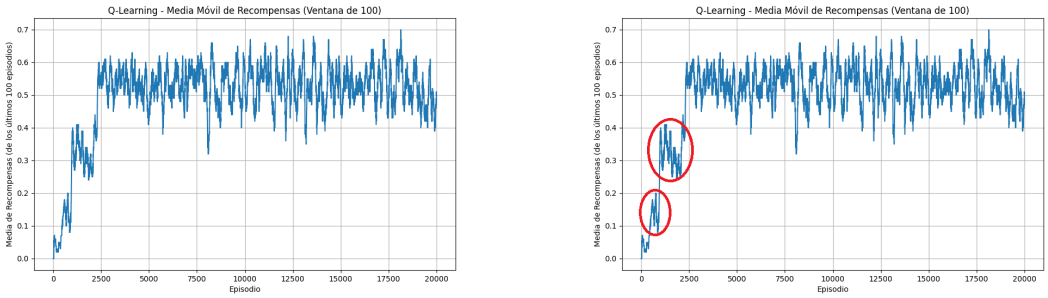
**¿Cómo afectan los parámetros?:**

- $\alpha$: es el valor que determina cuán suave es la curva, este lo que hace es dar más o menos importancia a los datos nuevos que llegan a la matriz.
- $\gamma$: es el valor que determina cuán alta puede llegar a ser la curva, gracias a este se tienen en cuenta las acciones futuras igual que las actuales.
- $\epsilon$: es el valor que determina cuán *greedy* es la política, un $\epsilon$ muy alto puede hacer que no sirva de nada lo que aprende y elija siempre movimientos aleatorios, mientras que un $\epsilon$ bajo puede hacer que encuentre caminos malos como soluciones buenas y se estanque, podría ser conveniente tener un $\epsilon_{decay}$ para poder ir modificando $\epsilon$ a medida que el agente aprende, ya que al inicio los datos no son muy fiables y debería ser más *greedy*.
- *episodios*: es el parámetro más sencillo de ver, si hay pocos episodios la política no llegará a aprender suficiente y se puede quedar en un máximo local o una zona mala, mientras que si hay demasiadas ejecuciones solo alargamos innecesariamente ya que puede haber llegado ya al máximo.

Según los diferentes valores de los parámetros $\alpha, \gamma, \epsilon$ y *episodios* de la función, se obtienen gráficas diferentes:

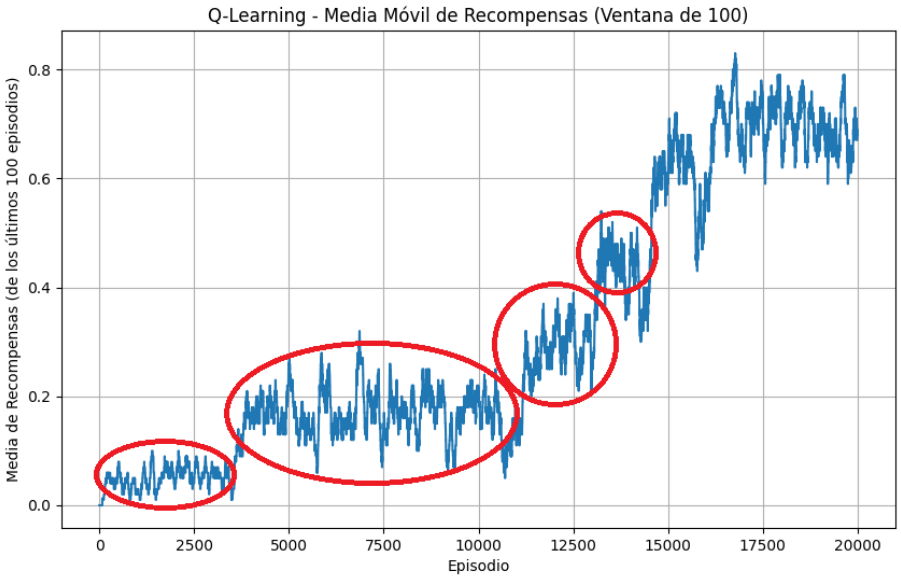
Este primer gráfico muestra una ejecución estándar de 20.000 episodios.

**Baseline Q-Learning ($\alpha = 0{,}1, \gamma = 0{,}99, \epsilon = 0{,}05$)**




**Análisis:** Se ve un claro aprendizaje. Gracias al $\epsilon = 0{,}05$ se evitan estancamientos en políticas mediocres (círculos rojos), estos estancamientos podían ser máximos locales de los que sale siendo greedy.

**Baseline Q-Learning ($\alpha = 0{,}05, \gamma = 0{,}99, \epsilon = 0{,}005$)**

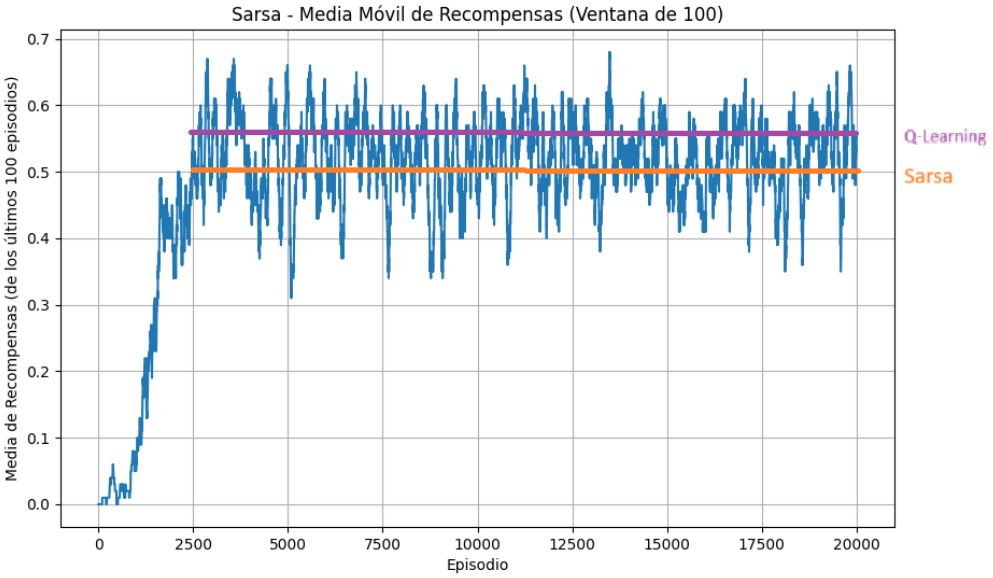



**Análisis:** Y con $\epsilon = 0{,}005$ le cuesta mucho más salir de máximos locales pero cuando perfecciona la política es mejor.

## 4.2 Sarsa

Haciendo la misma ejecución que con Q-Learning.

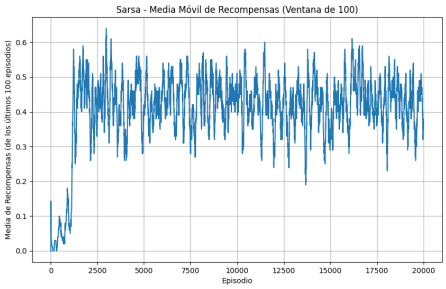
**Comparativa: Sarsa vs Q-Learning ($\alpha = 0{,}1$, $\gamma = 0{,}99$ y $\epsilon = 0{,}05$)**




**Conclusión:** La única diferencia tangible entre los algoritmos es que ***Q-Learning*** **tiene una pequeña superioridad** en cuanto a asegurar la llegada a la victoria (tiene un comportamiento más *greedy* respecto a los valores óptimos), mientras que Sarsa se mantiene ligeramente por debajo debido a su naturaleza conservadora (*on-policy*).

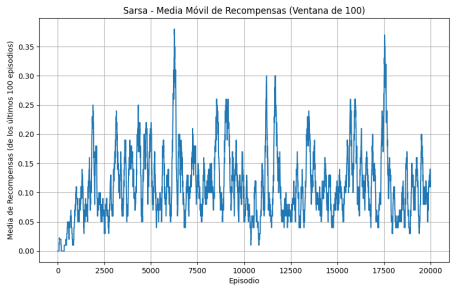
Jugando ahora con otros parámetros, si cambiamos $\alpha$ pasa lo siguiente:

**Efecto Alpha Alto ($\alpha = 0{,}3$)**




**Observación:** Se consiguen picos más drásticos y una gran inestabilidad.
Esto sucede porque se da demasiado valor a los datos nuevos, desestabilizando el conocimiento previo del agente.

**Efecto Gamma Bajo ($\gamma = 0{,}5$)**

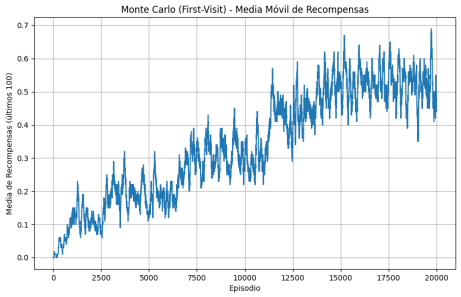



**Observación:** Se obtiene muy poca mejora.
Bajar la gamma hace que el agente sea "miope", ignorando las recompensas futuras (el regalo final) y centrándose solo en el paso inmediato, que no da puntos.

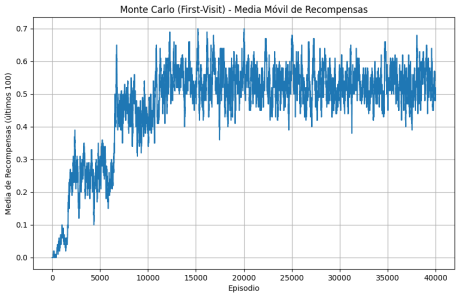
## 4.3 Monte Carlo

Haciendo uso del algoritmo de Monte Carlo (First-Visit)

**Evolución Monte Carlo: 20k vs 40k episodios ($\gamma = 0{,}99$ y $\epsilon = 0{,}05$)**



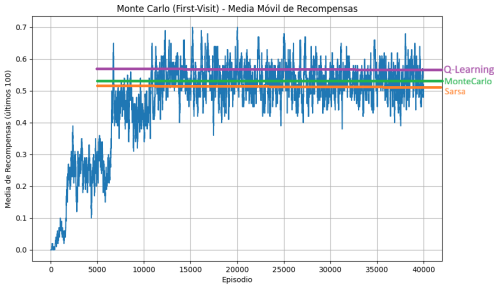

**20.000 Episodios:** Todavía en fase de aprendizaje.




**40.000 Episodios:** Estabilización conseguida.

Monte Carlo es un algoritmo que en general converge más lentamente, esto se debe a que para actualizar debe esperar al final de cada episodio, al contrario que los otros dos algoritmos que actualizan en cada movimiento hecho. Es por esto que para obtener un gráfico realmente comparable con los de *Q-Learning* y *Sarsa* tenemos que hacer un entrenamiento de episodios=40000.

**Comparación a lo largo del tiempo**

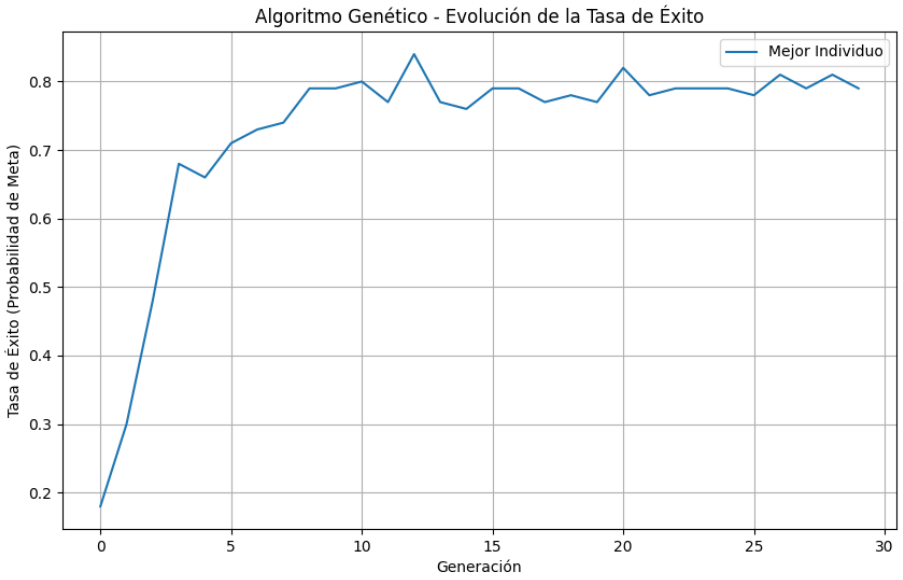



**Observación:** A largo plazo, Monte Carlo se iguala a Sarsa y a Q-Learning.

## 4.4 Algoritmo Genético

En el caso del Algoritmo Genético, al no tener episodios sino generaciones, el gráfico es menos denso. Cada generación cuenta con individuos que se evalúan y fusionan, si tomamos la mejor evaluación de los individuos de cada generación obtenemos el siguiente gráfico:

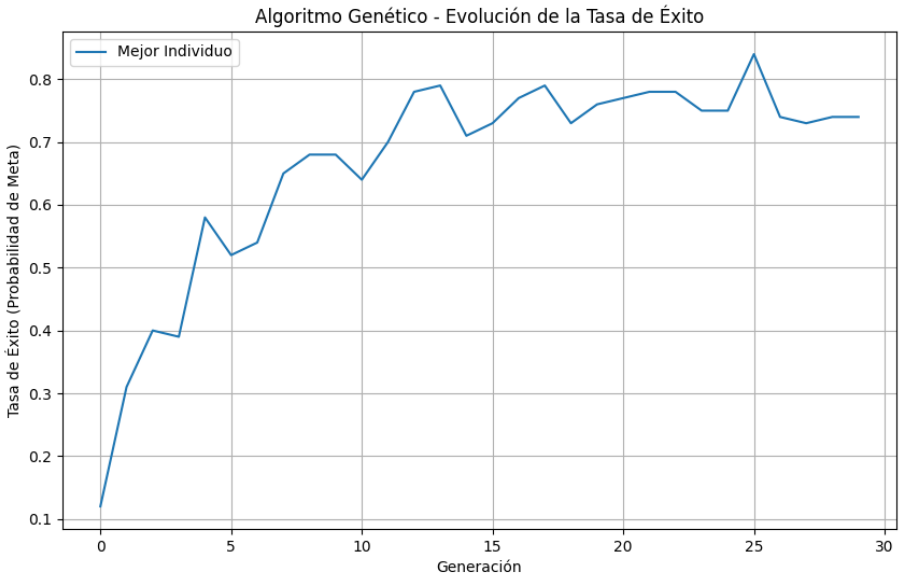
**Algoritmo Genético (*generaciones*=30,*tamano_poblacion*=100,mutacion=0.05,élites=5)**




**¿Es realmente tan bueno como parece?**

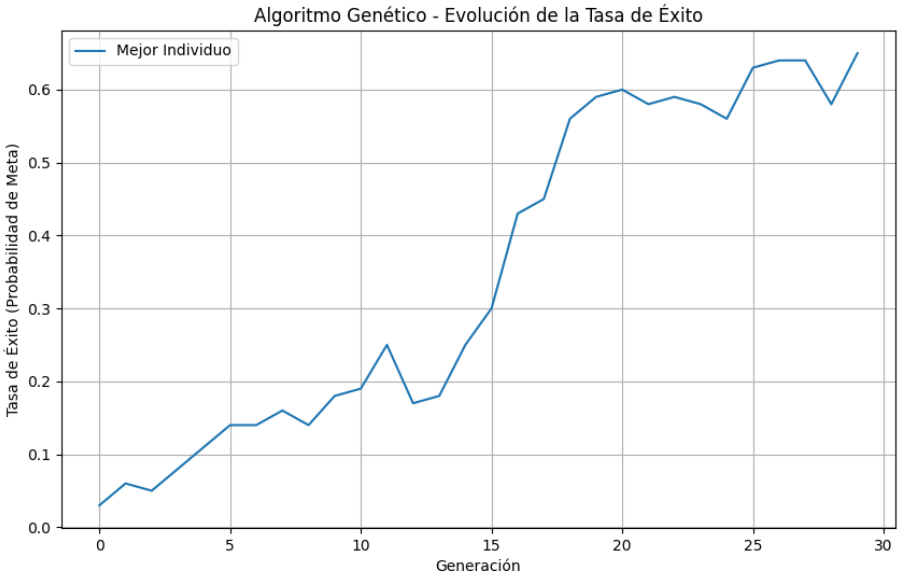
Este algoritmo puede parecer muy prometedor, a las pocas generaciones ya sube a un valor muy alto igual o mayor que *Q-Learning*, el problema del algoritmo genético es la poca escalabilidad, el problema del *Frozen-Lake* con un mapa 4x4 es muy pequeño.
Con 30 generaciones de 100 individuos cada una ya tarda una media de 2 minutos de entrenamiento
Por lo tanto, en un mapa de 100x100 tendríamos 10.000 estados, nuestros individuos serían arrays de *length*=10.000 con $policity[n] \in \{0,1,2,3\}$, la fusión entre políticas probablemente daría políticas malas y la función $fitness$ sería inescalable, tendría que ejecutar cada individuo muchas veces.

## Cambios en el parámetro *mutación*




**Análisis:** Cambiando el parámetro *mutación* por uno más alto (=0.3) la gráfica cambia al igual que si cambiáramos el parámetro $\epsilon$ en las otras, una subida más inestable y una cima que varía más a lo largo de las generaciones.

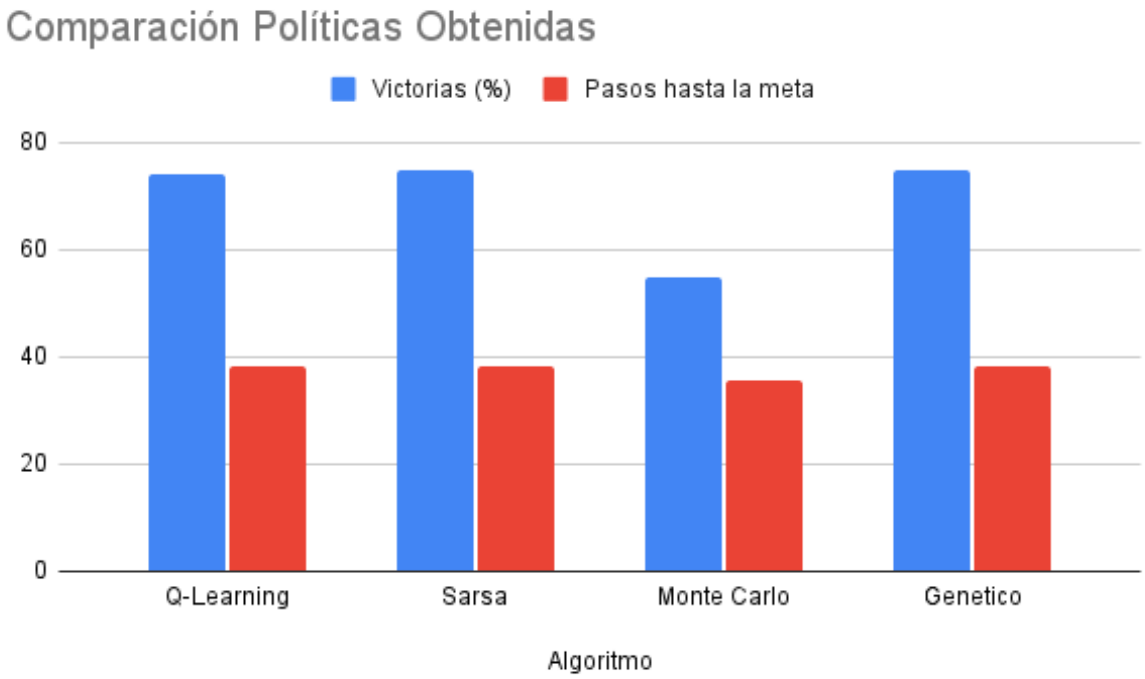
**Reducir el parámetro** *tamano_poblacion*




**Análisis:** Siguiendo con los parámetros del inicio, si ahora cambiamos la cantidad de políticas por generación (impuesta a 100 de primeras para asegurar) a una cantidad más pequeña, los gráficos son inconsistentes, depende mucho de la suerte de las políticas aleatorias iniciales

## 4.5 Comparaciones globales

Una vez comparados los diferentes algoritmos durante el entrenamiento, hemos cogido de cada uno 15 políticas entrenadas y las hemos evaluado, la evaluación consiste en ejecutarlas 100 veces con un máximo de 100 *steps*, al final se hace la media de veces que llega a la meta y los pasos medios de los casos que han llegado a la meta:




**Configuración Experimental**

**Aprendizajes por Refuerzo**

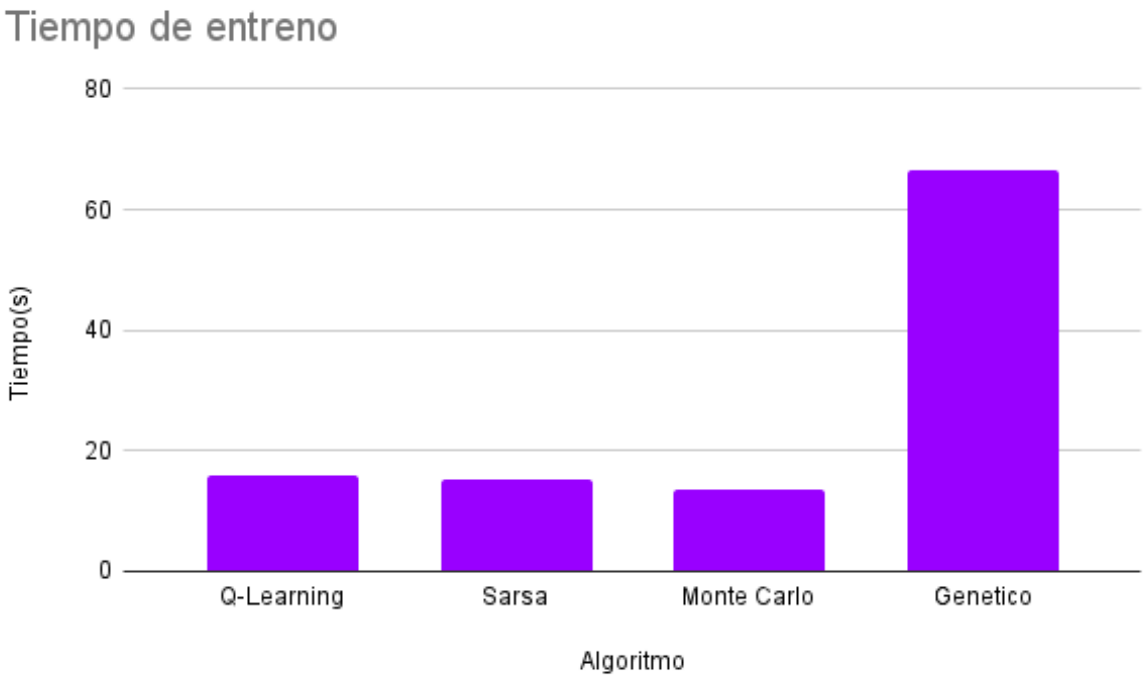
- **Episodios:** 20,000
- **Alpha ($\alpha$):** 0,1
- **Gamma ($\gamma$):** 0,99
- **Epsilon ($\epsilon$):** 0,05

**Algoritmo Genético**

- `tamano_poblacion`: 100
- `generaciones`: 30
- `prob_mutacion`: 0,05
- `elites`: 5

Se ve una clara minoría de victorias usando Monte Carlo, esto se debe a que crece mucho más lentamente que los otros algoritmos, con el doble de episodios tendría el mismo % que los otros.

Otra comparación que hemos hecho es el tiempo medio de entrenamiento de cada algoritmo con los mismos parámetros que antes, saliendo:




Todas tienen un tiempo parecido menos el algoritmo genético, que se dispara debido a la gran cantidad de evaluaciones que hace la función *fitness* y todas las generaciones creadas para perfeccionar la política.

# 5 Análisis de las políticas resultantes

## 5.1 Sarsa

*Sarsa* es un algoritmo *on-policy*, es decir, aprende utilizando la misma política que se emplea para actuar. Los indicadores principales son:

- **Recompensa acumulada:** muestra si el agente aprende a llegar a la meta. Se acostumbra a utilizar la media de varios episodios para suavizar la variabilidad.
- **Duración de los episodios:** medidas de cuántos pasos tarda el agente en completar un episodio; una duración menor con recompensa alta indica una política más eficiente.
- **Error de TD:** indica cuán bien se está ajustando la función de valor; cuando disminuye hacia cero, la política se estabiliza.
- **Exploración y explotación:** el porcentaje de acciones aleatorias disminuye a medida que el agente aprende, mostrando estabilidad de la política.

## 5.2 Q-Learning

*Q-Learning* es *off-policy*, actualiza los valores asumiendo que el agente siempre elegirá la mejor acción, independientemente de la política de exploración. Indicadores clave:

- **Recompensa acumulada y duración:** tiende a crecer más rápidamente que Sarsa, aunque puede presentar oscilaciones debidas a la actualización agresiva.
- **Convergencia de la tabla Q:** mide la estabilidad de los valores de las acciones; cuando se mantiene estable, la política ha convergido.
- **Política final:** se analiza la coherencia de las acciones óptimas y la tasa de éxito en el entorno.

## 5.3 Montecarlo

Los métodos de *Montecarlo* calculan los valores de acción a partir del retorno completo de los episodios. Indicadores relevantes:

- **Recompensa acumulada y duración:** similar a los otros métodos, aunque la variabilidad es más alta en episodios iniciales.
- **Valor medio por estado-acción:** la estimación se basa en la media de los retornos observados; la política se ajusta a medida que se acumulan más episodios.
- **Estabilidad de la política:** se mide por la consistencia de los valores y de las acciones escogidas, así como por la reducción de cambios en la política a lo largo del tiempo.

## 5.4 Algoritmo Genético

- **Políticas conservadoras:** las políticas generadas tienden a ser más seguras y menos arriesgadas, priorizando llegar a la meta por encima de la eficiencia.

- **Función *fitness*:** evalúa cada individuo ejecutando la política solo 100 veces y contando cuántas llegan a la meta, sin tener en cuenta la longitud del camino.
- **Ajuste de la política:** como resultado, las acciones seleccionadas buscan maximizar el éxito de completar el episodio, más que reducir el número de pasos.

## 6 Bibliografía


- Documentación Gymnasium Frozen_Lake
- Documentación Gymnasium Env
- Documentación Gymnasium Discrete