APRENDIZAJE POR REFUERZO EN ROBÓTICA MÓVIL

Álvaro Ruiz Gutierrez
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
alvruigut@alum.us.es

Lidia Jiménez Soriano
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
lidjimsor@alum.us.es

***Resumen*—El objetivo de este trabajo es aplicar técnicas de aprendizaje por refuerzo para la planificación de rutas de un robot móvil en un entorno con obstáculos. Se explorarán y compararán diferentes algoritmos para encontrar una política óptima que permita al robot navegar desde una posición inicial hasta una posición final, minimizando el riesgo de colisiones con los obstáculos presentes.**

Los resultados indican que cada algoritmo tiene ventajas y desventajas específicas dependiendo del escenario. Nos encontraremos con algoritmos que serían muy eficaces a la hora de buscar una rápida convergencia en problemas con grandes dimensiones, y otros que muestran una mayor precisión en la toma de decisiones en problemas con dimensiones más reducidas. Esto conlleva a que la elección de un algoritmo u otro dependerá del problema a resolver y de las características del entorno en el que se desenvuelva el robot.

***Palabras clave*—Inteligencia Artificial, Off-policy, On-policy, Aprendizaje por Refuerzo, Q-Learning, Monte-Carlo, SARSA, Robótica, Explorativo, Explotativo.**

I. Introducción

El aprendizaje por refuerzo es una rama de la inteligencia artificial que se centra en cómo los agentes deben tomar decisiones en un entorno para maximizar alguna noción de recompensa acumulada [1]. En robótica, estos algoritmos son utilizados para enseñar a los robots a navegar y realizar tareas de manera autónoma.

Este proyecto se centra en la aplicación de tres algoritmos de aprendizaje por refuerzo: Q-Learning, Monte-Carlo y SARSA, en un entorno de robótica simulado. Estos algoritmos se seleccionaron por su popularidad y diversidad en la aproximación al aprendizaje.

La problemática abordada consiste en la navegación de un robot en una cuadrícula con obstáculos, un escenario clásico que permite evaluar el desempeño de los algoritmos en términos de eficiencia y efectividad. El objetivo es desarrollar y comparar estrategias de navegación autónoma que permitan al robot moverse de manera segura y eficiente desde una posición inicial hasta una posición objetivo en el entorno simulado.

Para cada escenario, se ha generado imágenes representativas para mostrar visualmente algunos datos de interés como la política obtenida tras abordar cada problema, las recompensas asociadas a cada escenario, o la implementación de un robot que simularía el camino a seguir desde un estado inicial aleatorio hasta otro estado final, tomando las políticas definida por cada algoritmo en cada episodio de prueba.

La estructura del documento es la siguiente: En la sección II se describen los métodos empleados y se revisan trabajos relacionados. La sección III detalla la metodología utilizada para la implementación y pruebas de los algoritmos. En la sección IV se presentan los resultados obtenidos y su análisis. Finalmente, en la sección V se exponen las conclusiones y propuestas para futuros trabajos.

II. Preliminares

En esta sección se hace una breve introducción de las técnicas empleadas y también se mencionan, algunos trabajos relacionados.

A. Métodos empleados

En este trabajo se emplearon tres algoritmos de aprendizaje por refuerzo:

Detalles de la política Q-Learning

Q-Learning es un algoritmo off-policy que usa valores Q para mejorar el comportamiento del agente. Aprende de experiencias generadas por una política distinta a la que optimiza, eligiendo siempre la mejor opción en cada actualización.

Fórmula de actualización de valores:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot \Big(r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a) \Big)$$

donde:

$Q(s,a)$: Valor Q del estado s y acción a
α : Tasa de aprendizaje $(0 < \alpha \leq 1)$
r : Recompensa recibida después de tomar la acción a en el estado s
γ : Factor de descuento $(0 \leq \gamma \leq 1)$
s' : Estado tras de tomar la acción a en el estado s
a' : Acción que maximiza $Q(s',a')$

Detalles de la política Monte-Carlo

Es un algoritmo que actualiza las estimaciones de sus valores, basándose en la recompensa acumulada al final de cada episodio.

Fórmula de actualización de valores (para $t = 0, \ldots, T$):

$$U(s_t) \leftarrow U(s_t) + \alpha(s_t) \cdot (R_t + \gamma \cdot U(s_{t+1}) - U(s_t))$$

donde:

$U(s_t)$: Valor estimado de U para el estado s_t
$\alpha(s_t)$: Tasa de aprendizaje para el estado s_t $(0 < \alpha(s_t) \leq 1)$
R_t : Recompensa obtenida después de tomar la acción en el estado s_t
γ : Factor de descuento $(0 \leq \gamma \leq 1)$
$U(s_{t+1})$: Valor estimado de U para el estado siguiente después de tomar la acción en el estado s_t

Detalles de la política SARSA

Es un algoritmo on-policy, similar a Q-Learning. La principal diferencia se encuentra al actualizar la tabla Q-values. Esta utiliza la acción realizada por la política actual para conocer el valor Q.

Fórmula de actualización de valores:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot (r + \gamma \cdot Q(s', a') - Q(s, a))$$

donde:

$Q(s, a)$: Valor Q del estado s y acción a
α : Tasa de aprendizaje $(0 < \alpha \leq 1)$
r : recompensa recibida después de tomar la acción a en el estado s
γ : Factor de descuento $(0 \leq \gamma \leq 1)$
s' : Estado resultante después de tomar la acción a en el estado s
a' : Acción tomada en el estado s' de acuerdo con la política actual

B. Trabajo Relacionado

Numerosos estudios han comparado estos algoritmos en diferentes contextos. Sutton y Barto [1] ofrecen una descripción exhaustiva de Q-Learning y SARSA. Otros trabajos han explorado la eficiencia de Monte-Carlo en entornos complejos [2]. En robótica, se ha demostrado que la elección del algoritmo puede afectar significativamente el desempeño del robot, especialmente en entornos dinámicos [3].Varios estudios han contribuido a la comprensión del uso de estos algoritmos mediante su implementación en programas, lo cual permite analizar el rendimiento de un agente en un entorno virtual de pruebas [4].

III. Metodología

Inicialmente, diseñamos cuatro mapas de entorno formado por cuadrículas con obstáculos, basándonos en el ejemplo dado. En estos mapas, los 1 representan obstáculos y los 0 indican espacios libres por donde puede moverse el robot.

Posteriormente, creamos una clase problema donde reutilizamos funciones definidas previamente en el archivo `RoboticsRL.ipynb`. Además, implementamos funciones auxiliares necesarias para la creación de los algoritmos, tales como:

- `obtiene_indice_estado(estado, mapa)`: Devuelve el índice del estado en el mapa dado un estado en forma de tupla.
- `obtiene_estado_desde_indice(índice, mapa)`: Devuelve la tupla del estado dado el índice en el mapa.

Luego creamos archivos `.ipynb`, uno para cada algoritmo analizado (`Montecarlo.ipynb`, `Sarsa.ipynb`, `MonteCarlo_cada_visita.ipynb` y `QLearning.ipynb`). En estos archivos, ejecutamos las pruebas pertinentes para cada uno sobre los distintos mapas y así poder llegar a conclusiones mediante sus resultados.

Seguidamente, creamos un archivo `.py` para cada algoritmo, donde definimos las funciones necesarias para el correcto funcionamiento de cada uno:

1) Se inicializaron las variables en el constructor `__init__` siguiendo el modelo específico de cada algoritmo.
2) Creamos la función `entrenar`, encargada de realizar el proceso detallado para iterar y actualizar los valores necesarios para cada algoritmo.
3) Implementamos funciones adicionales como: label= ·
   - `es_estado_terminal(estado, destino)`: Verifica si el estado seleccionado es el destino declarado.
   - `obtener_politica(self)`: Devuelve la política obtenida en un formato más legible.
   - `selecciona_accion(self, estado)`: Aplica una política epsilon-greedy para seleccionar una acción dado un estado.
   - `selecciona_estado(self, matriz)`: Aplica una política epsilon-greedy para seleccionar un estado basado en una matriz de probabilidades de transición.

Finalmente, ejecutamos las pruebas requeridas en los archivos `.ipynb` y evaluamos los resultados obtenidos para llegar a conclusiones que discutiremos más adelante.

Los pseudocódigo para la implementación de los algoritmos son los siguientes:

Pseudocódigo QLearning

1) Inicializar arbitrariamente $q(s,a) \in \mathbb{R}, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.
2) $q(\text{terminal}, a) \leftarrow 0, \forall a \in \mathcal{A}(\text{terminal})$.
3) repetir
4) elegir aleatoriamente s no terminal
5) repetir
6) elegir $a \in \mathcal{A}(s)$ según política greedy de q
7) realizar acción a y observar R y s'
8) $q(s,a) \leftarrow q(s,a) + \alpha(R + \gamma \max_{a' \in \mathcal{A}(s')} q(s',a') - q(s,a))$
9) $s \leftarrow s'$
10) hasta que s es terminal
11) hasta que se cumple la condición de parada
12) devolver política ϵ-greedy derivada de q

Pseudocódigo SARSA

1) Inicializar arbitrariamente $q(s,a) \in \mathbb{R}, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.
2) $q(\text{terminal}, a) \leftarrow 0, \forall a \in \mathcal{A}(\text{terminal})$.
3) repetir
4) elegir aleatoriamente s no terminal
5) repetir
6) elegir $a \in \mathcal{A}(s)$ según política ϵ-greedy derivada de q
7) realizar acción a y observar R y s'
8) $q(s,a) \leftarrow q(s,a) + \alpha(R + \gamma q(s',a') - q(s,a))$
9) $s \leftarrow s'$
10) hasta que s es terminal
11) hasta que se cumple la condición de parada
12) devolver política ϵ-greedy derivada de q

Pseudocódigo MonteCarlo Cada Visita

1) Inicializar arbitrariamente $\pi(s) \in \mathcal{A}(s), \forall s \in \mathcal{S}$.
2) Inicializar arbitrariamente $q(s,a) \in \mathbb{R}, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.
3) $R_{accum}(s,a) \leftarrow$ lista vacía, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.
4) repetir
5) elegir al azar s_0 no terminal y $a_0 \in \mathcal{A}(s_0)$
6) generar $s_0, a_0, R_0, \ldots, s_T, a_T, R_T, s_{T+1}$ (s_{T+1} terminal, $a_i = \pi(s_i)$ para $i > 0$, $R_i = R(s_i, a_i)$ para $i \geq 0$)
7) para cada $t = 0, \ldots, T$ hacer
8) $U \leftarrow \sum_{i=t}^{T} \gamma^{i-t} R_i$
9) añadir U a $R_{accum}(s_t, a_t)$
10) $q(s_t, a_t) \leftarrow$ media de $R_{accum}(s_t, a_t)$
11) $\pi(s_t) \leftarrow \arg\max_{a \in \mathcal{A}(s_t)} q(s_t, a)$
12) hasta que se cumple la condición de parada
13) devolver π

Pseudocódigo MonteCarlo Primera Visita

1) Inicializar arbitrariamente $\pi(s) \in \mathcal{A}(s), \forall s \in \mathcal{S}$.
2) Inicializar arbitrariamente $q(s,a) \in \mathbb{R}, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.
3) $R_{accum}(s,a) \leftarrow$ lista vacía, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$.
4) repetir
5) elegir aleatoriamente s_0 no terminal y $a_0 \in \mathcal{A}(s_0)$
6) generar $s_0, a_0, R_0, \ldots, s_T, a_T, R_T, s_{T+1}$ (s_{T+1} terminal, $a_i = \pi(s_i)$ para $i > 0$, donde: $R_i = R(s_i, a_i)$ para $i \geq 0$)
7) para cada $t = 0, \ldots, T$ hacer
8) si s_t, a_t es la primera vez que ocurre
9) $U \leftarrow \sum_{i=t}^{T} \gamma^{i-t} R_i$
10) añadir U a $R_{accum}(s_t, a_t)$
11) $q(s_t, a_t) \leftarrow$ media de $R_{accum}(s_t, a_t)$
12) $\pi(s_t) \leftarrow \arg\max_{a \in \mathcal{A}(s_t)} q(s_t, a)$
13) hasta que se cumple la condición de parada
14) devolver π

Fig. 1. Pseudocódigos de los algoritmos de aprendizaje por refuerzo.

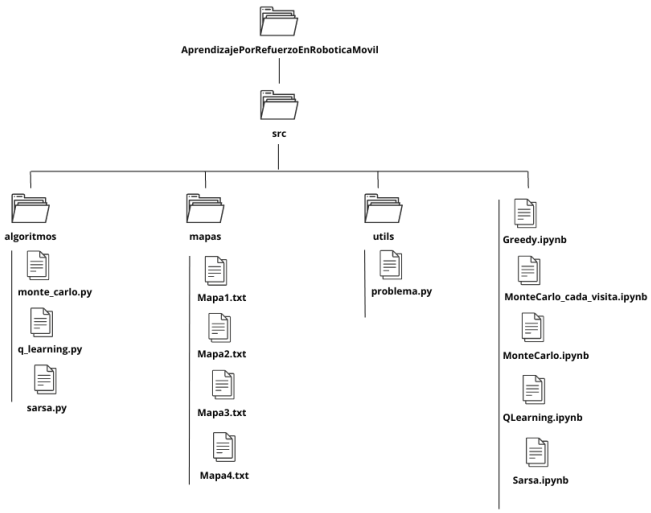

Fig. 2. Estructura del proyecto.

Para una mejor comprensión de la estructura de nuestro proyecto, a continuación se presenta una imagen que muestra la organización de las carpetas y archivos, de forma que se facilite el desarrollo y mantenimiento del proyecto.

IV. Resultados

En esta sección, examinaremos detenidamente los resultados obtenidos al aplicar cada algoritmo a dos mapas de tamaños opuestos, ajustando sistemáticamente los parámetros utilizados en cada caso. Nuestro objetivo es evaluar la efectividad y la eficiencia de cada algoritmo en diferentes escenarios. Para ello, compararemos los resultados obtenidos con cada una de las políticas.

Greedy

A continuación, se muestran las acciones obtenidas en el mapa 1 usando una política ambiciosa o *greedy*, la cual asigna a cada estado la acción que más acercaría al robot al objetivo (sin colisionar), y un posible camino recorrido por un robot partiendo desde un estado inicial aleatorio.

ESCENARIO 1

A continuación, se mostrarán los resultados obtenidos en las pruebas realizadas sobre el mapa 1.

TABLA I
Parámetros y Valores Seleccionados

Parámetros	**Valores**
Factor de descuento (γ)	0.9
Factor de aprendizaje (α)	0.2
Probabilidad de exploración (ϵ)	0.5
Número iteraciones	20000
Número pasos	25

Para evaluar los algoritmos, hemos optado por tomar los siguientes valores:

- Factor de descuento (γ): Toma el valor 0.9 de forma que se maximize la recompensa y los resultados obtenidos sea más eficaces.
- Factor aprendizaje (α): Toma el valor 0.2 para que el agente estabilice la velocidad de aprendizaje de forma que no sea muy rápida. Nótese que para el algoritmo de MonteCarlo no procede el uso de este parámetro.
- Probabilidad de exploración (ϵ): Toma el valor de 0.5 para que el agente explore y aprenda de manera equilibrada.
- Número de iteraciones: Toma el valor de 20000 de forma que el agente tenga un número suficiente de iteraciones para aprender las acciones óptimas en cada estado que se acercan al estado final.
- Número de pasos: Toma el valor de 25 de forma que el agente finalice un episodio en un número de pasos determinado y no se quede en un bucle infinito en caso de no llegar al destino.

Con estos valores se espera que el agente tenga un aprendizaje más lento pero estable, y que pueda aprender las acciones óptimas en cada estado para llegar al destino, en un número de iteraciones y pasos de acuerdo a las dimensiones del mapa.

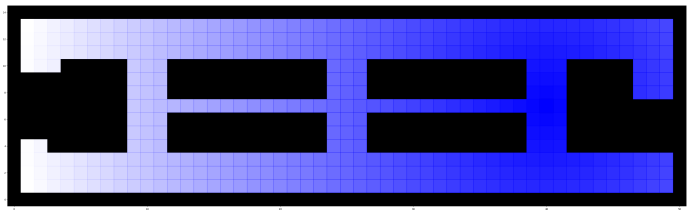

Fig. 5. Recompensa obtenida en el mapa 1

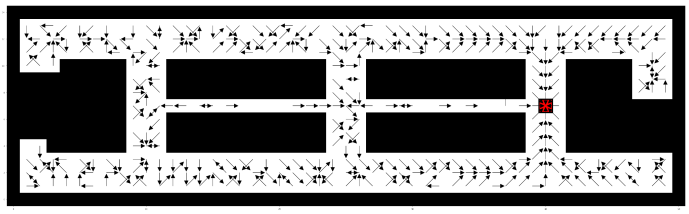

Fig. 6. Algoritmo QLearning sobre el mapa 1

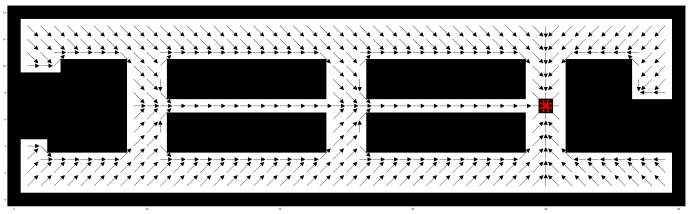

Fig. 3. Política greedy mapa 1

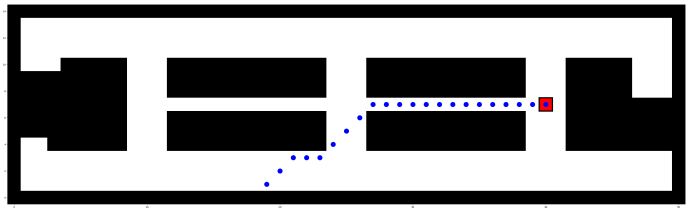

Fig. 4. Recorrido del robot con política greedy mapa 1

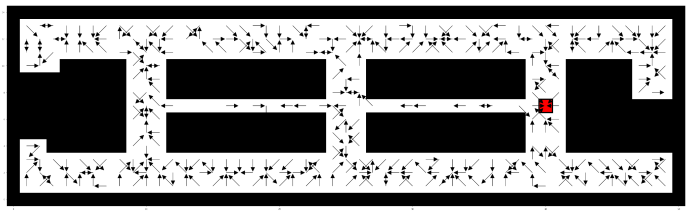

Fig. 7. Algoritmo SARSA sobre el mapa 1

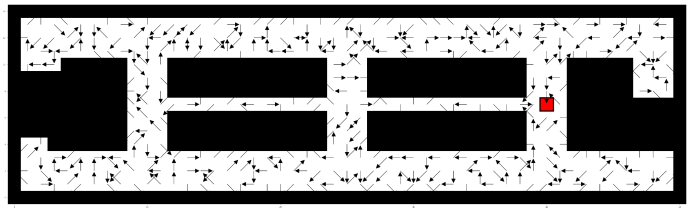

Fig. 8. Algoritmo Monte Carlo primera visita sobre el mapa 1

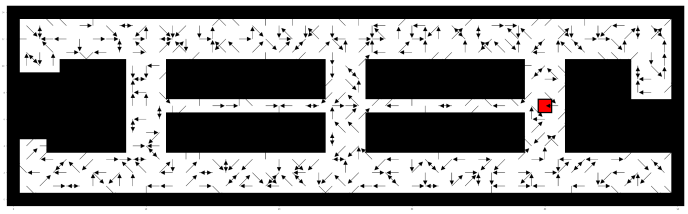

Fig. 9. Algoritmo Monte Carlo cada visita sobre el mapa 1

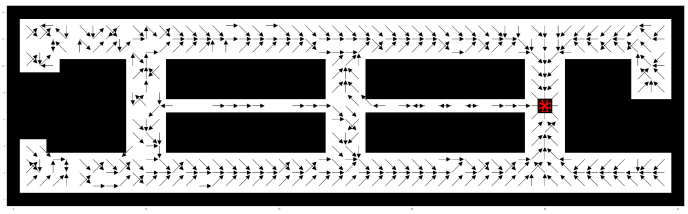

Fig. 10. Algoritmo QLearning sobre el mapa 1

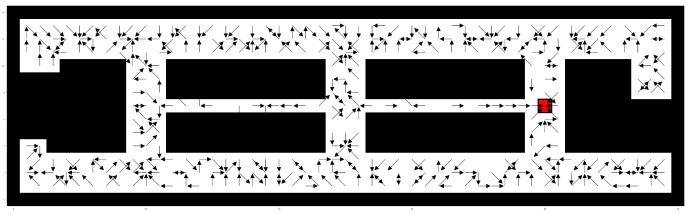

Fig. 11. Algoritmo SARSA sobre el mapa 1

TABLA II
PARÁMETROS Y VALORES SELECCIONADOS

Parámetros	**Valores**
Factor de descuento (γ)	0.9
Factor de aprendizaje (α)	0.5
Probabilidad de exploración (ϵ)	0.5
Número iteraciones	20000
Número pasos	25

Seguidamente, se han realizado las pruebas con los siguientes valores:

- Factor de descuento (γ): Toma el valor 0.9 de forma que se maximize la recompensa y los resultados obtenidos sean más eficaces.
- Factor aprendizaje (α): Toma el valor 0.5 de forma que el agente busque un aprendizaje más forzado y rápido pero más inestable. Nótese que para el algoritmo de MonteCarlo no procede el uso de este parámetro.
- Probabilidad de exploración (ϵ): Toma el valor de 0.5 para que el agente explore y aprenda de manera equilibrada.
- Número de iteraciones: Toma el valor de 20000 de forma que el agente tenga un número suficiente de iteraciones para aprender las acciones óptimas en cada estado que se acercan al estado final.
- Número de pasos: Toma el valor de 25 de forma que el agente finalice un episodio en un número de pasos determinado y no se quede en un bucle infinito, en caso de no llegar al destino.

Con estos valores se espera que el agente tenga un aprendizaje más rápido y que pueda aprender las acciones óptimas en cada estado para llegar al destino, aunque sea de forma más inestable.

ESCENARIO 4

A continuación, se mostrarán los resultados obtenidos en las pruebas realizadas sobre el mapa 3.

TABLA III
PARÁMETROS Y VALORES SELECCIONADOS

Parámetros	Valores
Factor de descuento (γ)	0.9
Factor de aprendizaje (α)	0.5
Probabilidad de exploración (ϵ)	0.5
Número iteraciones	25000
Número pasos	50

Nota: Para el algoritmo de MonteCarlo, se ha tenido que tomar un valor menor de 5000 debido al gran coste computacional que supone.

Para evaluar los algoritmos, hemos optado por tomar los siguientes valores:

- Factor de descuento (γ): Toma el valor 0.9 de forma que se maximize la recompensa y los resultados obtenidos sean más eficaces.
- Factor aprendizaje (α): Toma el valor 0.5 de forma que el agente busque un aprendizaje más forzado y rápido pero más inestable. Nótese que para el algoritmo de MonteCarlo no procede el uso de este parámetro.
- Probabilidad de exploración (ϵ): Toma el valor de 0.5 para que el agente explore y aprenda de manera equilibrada.
- Número de iteraciones: Toma el valor de 20000 de forma que el agente tenga un número suficiente de iteraciones para aprender las acciones óptimas en cada estado que se acercan al estado final. En el caso de MonteCarlo de cada visita, se ha tenido que tomar un valor menor de 5000 debido al gran coste computacional que supone.
- Número de pasos: Toma el valor de 25 de forma que el agente finalice un episodio en un número de pasos determinado y no se quede en un bucle infinito en caso de no llegar al destino.

Con estos valores se espera que el agente tenga un aprendizaje más lento pero estable, y que pueda aprender las acciones óptimas en cada estado para llegar al destino, en un número de iteraciones y pasos de acuerdo a las dimensiones del mapa.

Como hemos mencionado anteriormente en el escenario 1, los valores de los parámetros factor de descuento y probabilidad de exploración no varían, ya que son los valores óptimos para este tipo de problemas. En cuanto al número de pasos y al número de iteraciones, son valores mayores debido a las dimensiones del mapa. Hemos tratado de ajustar el número de pasos de manera que el robot tenga la oportunidad de aprender el camino óptimo hacia el destino, y un incremento en las iteraciones para que el agente tenga más formas de aprender y converger. Por último, hemos realizado las pruebas solo cambiando el valor del factor de aprendizaje y se concluye lo mismo que en el escenario 1, para un valor de 0.2 el agente aprende de forma más lenta y estable, mientras que para un valor de 0.5 el agente aprende de forma más rápida pero menos estable.

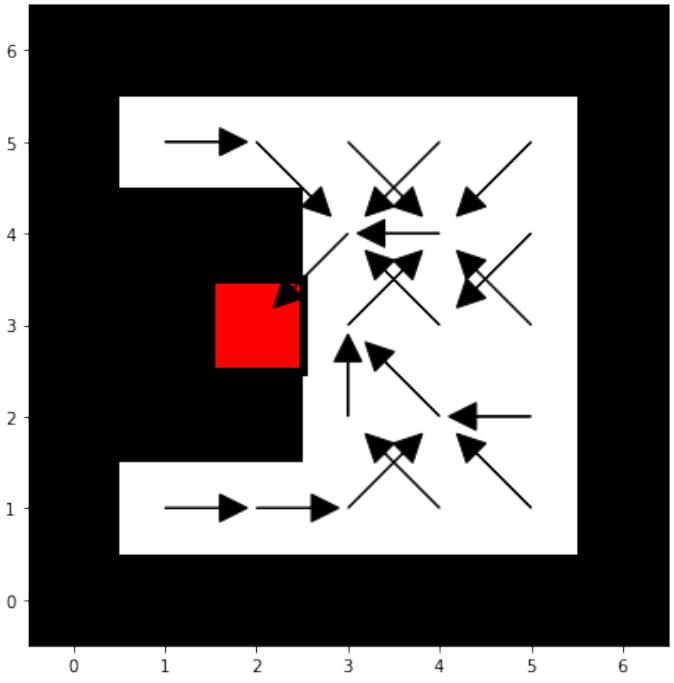

Fig. 12. Algoritmo QLearning sobre el mapa 4

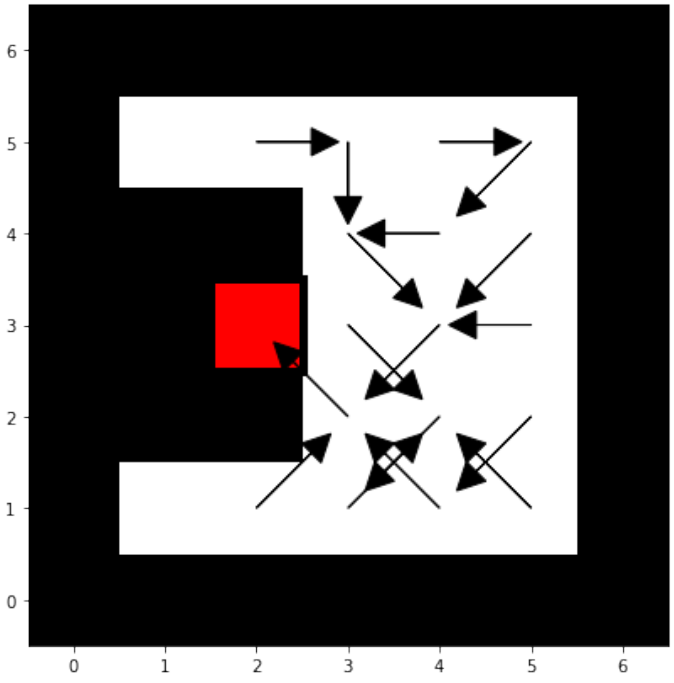

Fig. 13. Algoritmo SARSA sobre el mapa 4

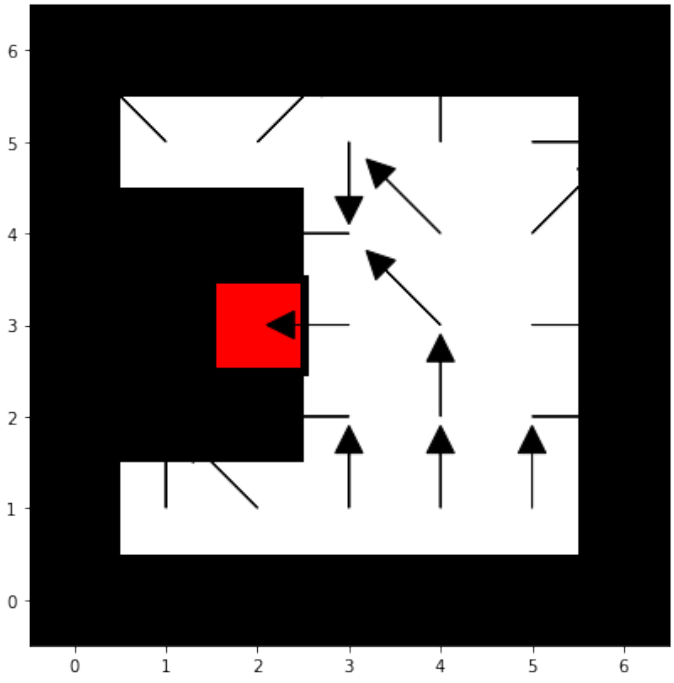

Fig. 14. Algoritmo Monte Carlo primera visita sobre el mapa 4

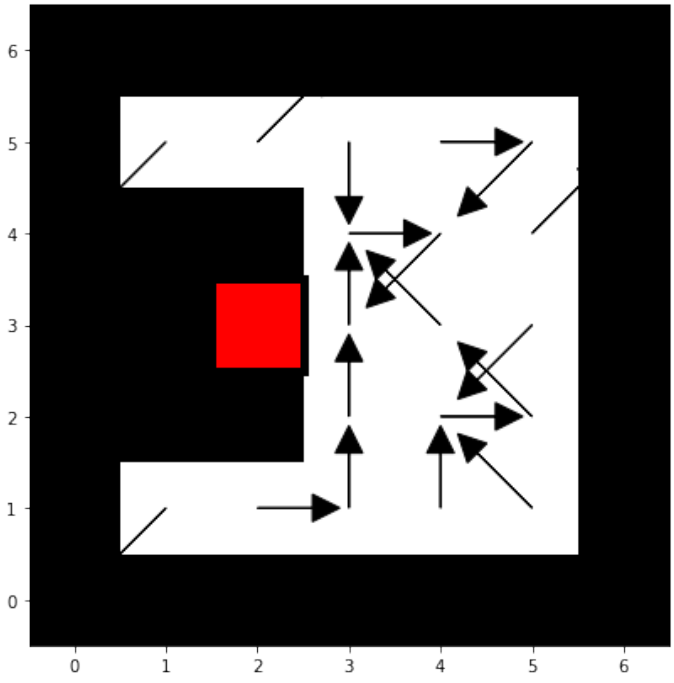

Fig. 15. Algoritmo Monte Carlo cada visita sobre el mapa 4

Notase que en este escenario, el algoritmo de Montecarlo para ambos caso ha obtenido unos resultados menos óptimos que cualquier otro. Esto se debe al número de iteraciones y pasos que hemos seleccionado, ya que el coste computacional que supone es muy elevado y no hemos podido realizar un número de iteraciones y pasos suficientes para que el agente aprenda de forma eficaz.

Para concluir, QLearning ha sido el algoritmo que ha obtenido mejores resultados en este escenario. Podemos observar para este algoritmo que asignándole una tasa de aprendizaje estándar 0.5, el agente es capaz de llegar al destino desde cualquier punto de partida.

V. Conclusiones

Se han aplicado cuatro algoritmos a dos mapas de entornos, uno más simple y otro más complejo, con el objetivo de encontrar una política óptima que permita al robot navegar desde una posición inicial hasta una posición final, minimizando el riesgo de colisiones con los obstáculos presentes.

Los resultados obtenidos muestran que cada algoritmo tiene ventajas y desventajas específicas dependiendo del escenario. MonteCarlos mostró una rápida convergencia en entornos con grandes dimensiones, siendo la de primera visita la más rápida de todas, pero una mala opción en términos de eficiencia para escenarios de menor tamaño. Con respecto a sus resultados, en ambos escenarios mostraron una baja precisión en su política. Por el contrario, QLearning y SARSA fueron más robusto en cada uno de los escenarios, siendo algo más costoso en los mapas de mayor tamaño pero con una buena precisión en sus acciones.

En general, los algoritmos de aprendizaje por refuerzo son una herramienta poderosa para la planificación de rutas de robots móviles en entornos con obstáculos. Sin embargo, es importante considerar las limitaciones de cada algoritmo y ajustar los parámetros según las características del entorno. Para problemas más complejos, que requiera una convergencia rápida, MonteCarlos sería una buena opción, mientras que en cualquier otro escenario, QLearning sería ligeramente más adecuado que SARSA.

Algunas mejoras futuras podrían ser la implementación de la técnica 'epsilon decay'[5]. Esta consiste en disminuir la probabilidad de exploración del agente a medida que éste adquiere más conocimiento del entorno, de forma que cada vez tome un carácter más explotativo y escoja acciones que maximicen las recompensas aprendidas anteriormente. Esto ayudaría a mejorar la eficiencia de los algoritmos de forma significativa. Además, se podría explorar la implementación de otros algoritmos de aprendizaje por refuerzo, como DQN [6], (parecido a Q-learning aplicando redes neuronales profundas), para comparar su rendimiento con los algoritmos estudiados en este trabajo.

Referencias

[1] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed, Pearson, 2010.

[2] Y. LeCun, Y. Bengio, G. Hinton. "Deep Learning", Nature, vol. 521, 2015, pp. 436-444.

[3] Página web del curso IA de Ingeniería del Software **www.cs.us.es/docencia/aulavirtual/course/**.
[4] Página web que implementa en Python la resolucion de algunos problemas de aprendizaje por refuerzo usando algoritmos como SARSA **barcelonageeks.com/aprendizaje-reforzado-de-sarsa/**.
[5] Ejemplos y aclaraciones sobre epsilon decay **https://aakash94.github.io/Reward-Based-Epsilon-Decay/**.
[6] Artículo sobre DQN **https://markelsanz14.medium.com/**.