Aprendizaje por refuerzo en robótica móvil

Jesús Fernández Rodríguez
Universidad de Sevilla
Sevilla, España
jesferrod1, jfr16sevilla@gmail.com

Enrique Pérez Milla
Universidad de Sevilla
Sevilla, España
enrpermil, enrique.perez.milla@gmail.com

***Resumen*—El objetivo principal de este trabajo es aplicar de una forma práctica, y lo más real posible los métodos estudiados a lo largo de la asignatura, concretamente, en el temario correspondiente a "Aprendizaje por refuerzo". Las conclusiones que hemos sacado de este proyecto es que el área de aprendizaje por refuerzo es muy interesante en cuanto a las tecnologías que tenemos hoy en día y son aplicables a problemas reales, a pesar del gran coste computacional que pueden llegar a tener estos métodos que hemos aplicado.**

***Palabras clave*—Inteligencia Artificial, Coste computacional, Q-Learning, Montecarlo, SARSA,...**

I. Introducción

En este caso, el proyecto trata sobre un robot el cual está situado en un mapa donde encontramos una serie de obstáculos, mapas los cuales hemos diseñado nosostros mismos (como explicamos a continuación). Se nos pide hacer llegar al robot desde un punto de origen (establecido al azar) hasta un punto de destino evitando el avance del robot hacia una posición 'clasificada' como obstáculo.

Para conseguir dicho objetivo, en primer lugar hemos diseñado dos mapas mediante matrices de ocupación (matriz binaria donde 0 significa espacio libre, y 1 significa presencia de obstáculo). El robot debe realizar acciones para llegar a un destino que ha sido previamente definido al diseñar el mapa, de modo que podamos obtener una política que nos asocie una determinada acción para cada posible estado, de modo que el camino a seguir por el robot desde el punto de inicio hasta el destino sea el más óptimo. El camino que hemos recorrido para obtener los resultados comienza con el diseño del mapa, seguido de la implementación de los algoritmos de aprendizaje por refuerzo que se nos indica y que previamente hemos estudiado en la asignatura (Q-Learning y Montecarlo), así como el algoritmo SARSA, que es similar al algoritmo Q-Learning, además de algunas funciones auxiliares para la implementación de el algoritmo de Montecarlo y SARSA.

Por último hemos hecho pruebas variando los parámetros de los algoritmos implementados para estudiar como cambian las políticas resultantes de aplicarlos.

Fig. 1. Representación visual del primer mapa creado

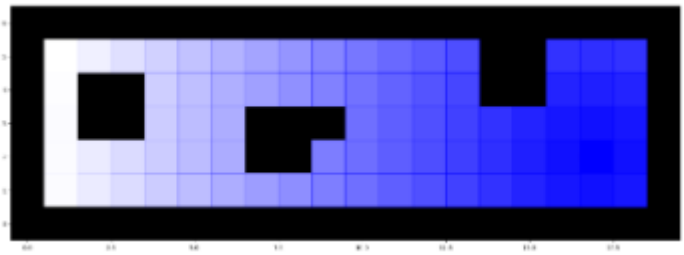

Fig. 2. Visualización de recompensas del primer mapa

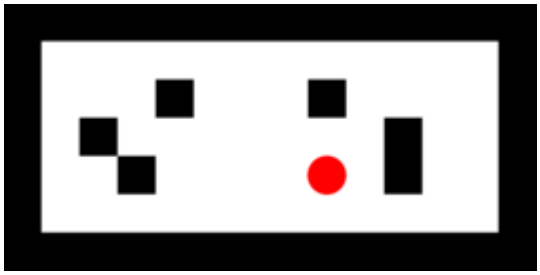

Fig. 3. Representación visual del segundo mapa creado

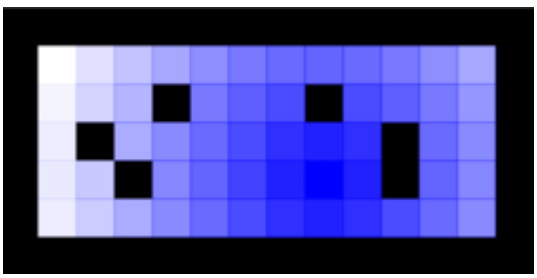

Fig. 4. Visualización de recompensas del segundo mapa

II. Preliminares

Para conseguir el objetivo del proyecto, se nos indicaba en las instrucciones que hiciéramos uso de los algoritmos Q-Learning, Monte-Carlo y SARSA. De modo que los hemos implementado como se indica a continuación.

A. Métodos empleados

El método Q-Learning ha sido implementado haciendo uso de la biblioteca mdptoolbox, la cual se nos proporciona. El método Montecarlo ha sido de implementación propia. El método SARSA al igual que Montecarlo, es de implementación propia.

B. Trabajo Relacionados

Ninguno de los dos autores de este trabajo hemos hecho otro estudio relacionado con este previamente.

III. Metodología

A continuación se detalla cada uno de los métodos que hemos usado a lo largo de este trabajo. Para que se pueda entender de forma clara lo vamos a poner estructurado por cada uno de los métodos.

1) *Q-Learning Algoritmo Q-Learning:* El algoritmo Q-learning es un método de aprendizaje por refuerzo que busca encontrar la mejor acción a tomar en cada estado para maximizar la recompensa acumulada a largo plazo. Funciona actualizando iterativamente una tabla Q(s, a), donde s representa un estado y a una acción. La tabla Q almacena el valor esperado de la recompensa acumulada si se toma la acción a en el estado s. Conforme realizamos iteraciones del algoritmo, Q-learning converge hacia una política óptima de decisión. Hemos implementado este método usando la biblioteca mdptoolbox [1]. Los parámetros de esta función son:
   - La matriz de transicciones del sistema, que ya está previamente definida en el cuadernillo RoboticsRL [4].
   - La matriz de recompensas del sistema, que también está ya previamente definida en el cuadernillo RoboticsRL [4].
   - Número de iteraciones del algoritmo, si no lo indicas toma por defecto el valor 10000, cuanto mayor sea este número más preciso será el algoritmo, pero también tendrá un mayor costo computacional.
   - Factor de descuento, que en nuestro caso escogimos como valor para las pruebas 0.9 .
2) *Algoritmo Monte-Carlo:* El algoritmo Monte Carlo en aprendizaje por refuerzo estima los valores de las acciones basándose en las recompensas promedio obtenidas de múltiples episodios completos. Se ejecutan muchas simulaciones (episodios), y al final de cada uno se calcula la recompensa acumulada. Estos valores se promedian para actualizar la estimación del valor de cada estado o acción, sin necesidad de un modelo del entorno. A través de este proceso, el algoritmo converge hacia una política óptima evaluando el desempeño real de las acciones a lo largo de varios episodios. Para el método de montecarlo de cada visita eliminaremos el condicional que vemos en la línea 8.

Algorithm 1 Algoritmo Q-Learning [2]

1: Inicializamos arbitrariamente $q(s, a)$
2: **repeat**
3: elegir aleatoriamente s no terminal
4: **repeat**
5: elegir a según política ϵ-voraz derivada de q
6: realizar acción a y observar R y s'
7: $q(s, a) \leftarrow q(s, a) + \alpha(R + \gamma \max q(s', a') - q(s, a))$
8: $s \leftarrow s'$
9: **until** s es terminal
10: **until** se cumple la condición de parada
11: **return** política voraz derivada de q

Fig. 5. Algoritmo Q-Learning

Hemos realizado una implementación propia de este algoritmo, para ello utilizamos una función auxiliar para crear la secuencia que acaba en un estado terminal (que en el caso de este problema es el destino). Para generar esta secuencia inicial partimos de un estado inicial aleatorio en el que no haya colisión y una acción aleatoria que haga que desde el primer estado no pasemos a un estado en el que haya una colisión. Luego aplicamos las acciones que dice la política hasta llegar al estado terminal. En caso de que durante el cálculo de esta secuencia colisionemos, se repetirá el cálculo de la secuencia inicial tomando de nuevo un estado inicial y acción inicial aleatorios. Además, hemos implementado una función alternativa para obteners recompensas que hemos usado en el método montecarlo, que devuelve valor -1000 si hay colisión , -1 si el estado no es el destino y 0 si el estado es el estado destino.

Para ejecutar nuestra implementación del algoritmo de Montecarlo se requieren los siguientes parámetros:

- Política inicial, desde la cual se crea el primer episodio y desde la cual iremos modificándola según las iteraciones del algoritmo. Para este problema hemos utilizado la política greedy como política inicial
- Número de iteraciones del algoritmo, nuestra implementación es muy costosa computacionalmente, por tanto, hemos elegido un número pequeño de iteraciones , 1000 , para realizar las pruebas de este algoritmo.
- Factor de descuento, que en nuestro caso escogimos como valor para las pruebas 0.9 .
- Primera visita, que es un booleano , que si toma el valor true , se ejecutará el algoritmo de montecarlo de primera visita, y si toma el valor false, se

ejecutará el algoritmo de Montecarlo de cada visita.

Algorithm 2 Montecarlo de primera visita con inicios exploratorios [2]

Inicializar arbitrariamente $\pi(s) \in A(s)$, $\forall s \in S$
Inicializar arbitrariamente $q(s,a) \in R$, $\forall s \in S, a \in A(s)$
$Racum(s,a) \leftarrow$ lista vacía, $\forall s \in S, a \in A(s)$
repeat
  Elegir aleatoriamente s_0 no terminal y $a_0 \in A(s_0)$
  Generar $s_0, a_0, R_0, s_1, a_1, R_1, \ldots, s_T, R_T, s_{T+1}$ $\{(s_{T+1}$ terminal, $a_i = \pi(s_i)$ para $i > 0$, $R_i = R(s_i, a_i)$ para $i \geq 0)\}$
  for $t = 0, \ldots, T$ **do**
    if s_t, a_t es la primera vez que ocurre en la secuencia **then**
      $U \leftarrow \sum_{i=t}^{T} \gamma^{i-t} R_i$
      Añadir U a $Racum(s_t, a_t)$
      $q(s_t, a_t) \leftarrow$ media de los valores de $Racum(s_t, a_t)$
      $\pi(s_t) \leftarrow \arg\max_{a \in A(s_t)} q(s_t, a)$
    end if
  end for
until se cumple la condición de parada
return π

Fig. 6. Montecarlo de primera visita con inicios exploratorios

3) *Algoritmo SARSA:* El algoritmo SARSA (State-Action-Reward-State-Action) es un método de aprendizaje por refuerzo que actualiza los valores Q basándose en la acción realmente tomada en el siguiente estado. En cada paso, se observa el estado actual (s), se elige una acción (a), se recibe una recompensa (r), se observa el siguiente estado (s') y la siguiente acción (a'). La actualización de Q(s, a) se realiza utilizando esta secuencia (s, a, r, s', a') , incorporando tanto la recompensa inmediata como el valor de la siguiente acción elegida, promoviendo un aprendizaje más acorde con la política actual.
   Hemos realizado una implementación propia de este algoritmo, para ello utilizamos la misma función auxiliar que usamos para el algoritmo de Montecarlo que calcula la secuencia inicial, pero a la cual cambiamos el valor de un parámetro booleano de esta función que hace que no tome una primera acción aleatoria, si no la que dicte la política que se va actualizando en cada iteración del algoritmo.
   Como función para obtener la recompensa hemos usado la proporcionada en el cuadernillo RoboticsRL [4]
   Para ejecutar nuestra implementación del algoritmo SARSA se requieren los siguientes parámetros:
   - Política inicial, desde la cual se crea el primer episodio y desde la cual iremos modificándola según las iteraciones del algoritmo. Para este problema hemos utilizado la política greedy como política inicial
   - Número de iteraciones del algoritmo, esta implementación, aunque menos costosa computacionalmente que la del algoritmo de Montecarlo, sigue siendo costosa, por tanto, hemos elegido un número pequeño de iteraciones , 1000 , para realizar las pruebas de este algoritmo.
   - Factor de descuento, que en nuestro caso escogimos como valor para las pruebas 0.9 .
   - Factor de aprendizaje, que en nuestro caso escogimos como valor para las pruebas 0.5 .

Algorithm 3 Algoritmo SARSA [3]

Inicializar arbitrariamente $q(s,a)$
for cada episodio **do**
  Inicializar el estado s
  Elegir a según política ϵ-voraz derivada de q
  repeat
    Tomar acción a, observar recompensa R y siguiente estado s'
    Elegir a' según política ϵ-voraz derivada de q
    $q(s,a) \leftarrow q(s,a) + \alpha(R + \gamma q(s', a') - q(s,a))$
    $s \leftarrow s'$
    $a \leftarrow a'$
  until s es un estado terminal
end for
return política voraz derivada de q

Fig. 7. Algoritmo SARSA

IV. Resultados

En esta sección se detallarán tanto los experimentos realizados como los resultados conseguidos:

- *Experimentos para el primer mapa*:
  1) *Q-Learning*: Para este mapa hemos elegido un numero no muy alto de iteraciones porque al aumentarlo puede hacer que la ejecucións sea demasiado larga. Hemos introducido la matriz de transicciones del sistema y de recompensas que nos facilita el cuadernillo RoboticsRL [4]. Hemos realizado tres pruebas variando el valor del facotr de descuento que toma los valores 0.5 en la primera prueba, 0.1 en la segunda y 0.9 en la tercera. Como se puede observar en las figuras 8 , 9 y 10, se obtiene resultados algo "caóticos" debido al insuficiente número de iteraciones, como se podrá comprobar más adelante, en los experimentos del segundo mapa, más reducido, los resultados mejoran si se aumenta el número de iteraciones. En esta prueba se obserban cambios notables tras varias el valor del factor de descuento.

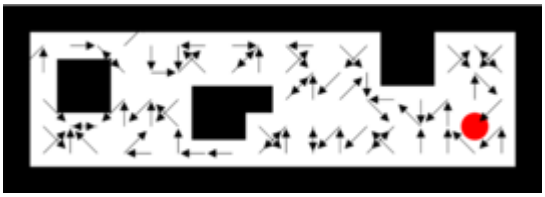

Fig. 8. Politica resultante ejecución q-learning con factor de descuento 0.5 en el primer mapa.

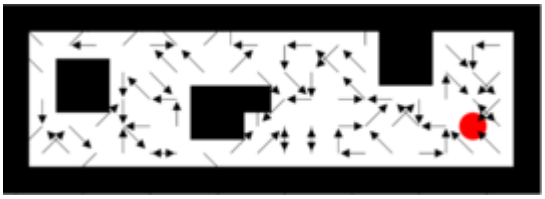

Fig. 9. Politica resultante ejecución q-learning con factor de descuento 0.1 en el primer mapa.

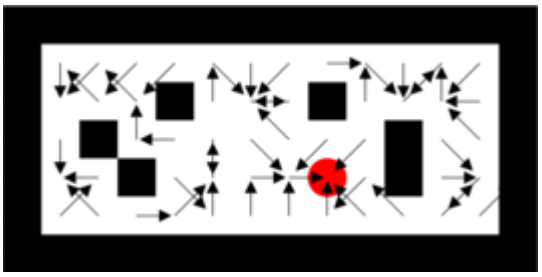

Fig. 10. Politica resultante ejecución q-learning con factor de descuento 0.9 en el primer mapa.

2) *Montecarlo*: Este algoritmo es muy costoso computacionalmente, por tanto, hemos establecido en 1000 el número de iteraciones, la política greedy como política inicial, 0.9 como factor de descuento e hicimos una primera prueba en la que establecimos el parámetro primera visita en true , lo que hace que se ejecute el algorimo de Montecarlo de primera visita, y una segunda prueba con el parámetro primera visita en false , lo que hace que se ejecute el algorimo de Montecarlo de cada visita. En ambos inicializamos los q values de todos los pares posibles estado acción arbitrariamente con el valor 0. Como se puede observar en las figuras 11 y 12, se obtienen valores similares, posiblemente, debido al número bajo de iteraciones del algoritmo.

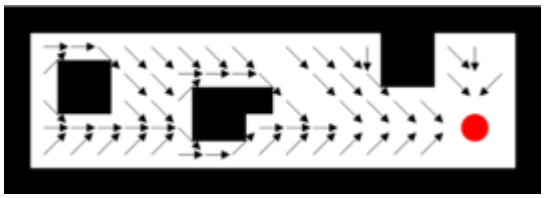

Fig. 11. Politica resultante ejecución montecarlo de primera visita en el primer mapa.

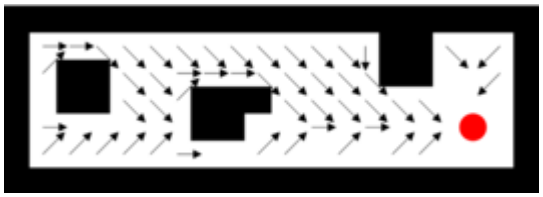

Fig. 12. Politica resultante ejecución montecarlo de cada visita en el primer mapa.

3) *SARSA*: Hemos hecho tres ejecuciones de este algoritmo, en la que lo único que hemos variado es el factor de descuento y mantenido el resto de valores constantes para ver como esto afecta a la política resultante de ejecutar este algoritmo, en todas las ejecuciones hemos establecido como parámetros la política greedy como política inicial, 1000 el número de iteraciones y 0.5 como factor de aprendizaje. En la primera ejecución hemos utilizado un valor intermedio , 0.5, como factor de descuento, y en las siguientes dos ejecuciones hemos usado 0.1 y 0.9 como factor de descuento para estudiar como varia en estos extremos. Como se puede observar en las figuras 13, 14 y 15, de nuevo no se aprecian grandes diferencias, esto es , posiblemente, debido al reducido número de iteraciones.

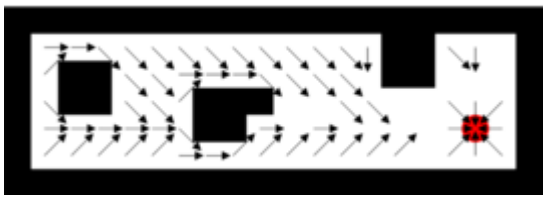

Fig. 13. Politica resultante ejecución SARSA con factor de descuento 0,5 en el primer mapa.

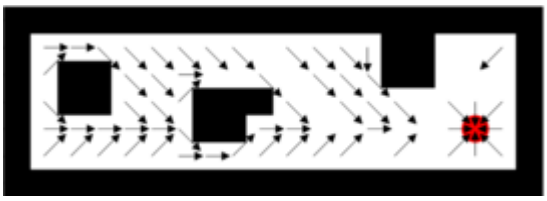

Fig. 14. Politica resultante ejecución SARSA con factor de descuento 0,1 en el primer mapa.

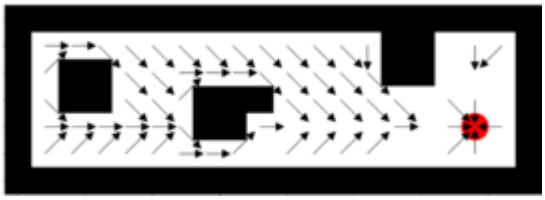

Fig. 15. Politica resultante ejecución SARSA con factor de descuento 0,9 en el primer mapa.

- *Experimentos para el segundo mapa*:
  1) *Q-Learning*: Para este mapa hemos hecho dos tests de q-learning,para la primera introducimos un número ”bajo” de iteraciones , 100000, la matriz de transicciones del sistema y de recompensas que nos facilita el cuadernillo RoboticsRL [4], como factor de descuento introducimos 0,9. Y para la segunda , al ser de menor dimensión hemos elegido un elevado número de iteraciones para intentar refinar al máximo posible la política de la función de q-learning de la biblioteca mdptoolbox, para ello hemos pasado como parámetro de número de iteraciones 100000000 (este elevado número dió lugar a una larga ejecución). Como se puede observar en las figuras 16 y 17, aunque los resultados no son perfectos se aprecia una mejoría en la política con un elevado número de iteraciones.

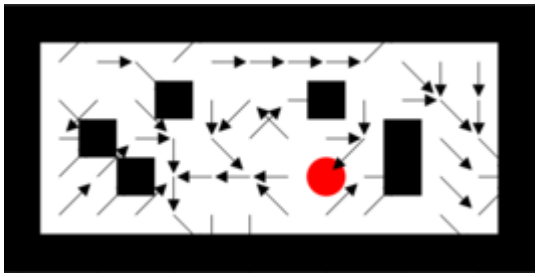

Fig. 16. Politica resultante ejecución q-learning 100000 iteraciones (número ”reducido”) en el segundo mapa.

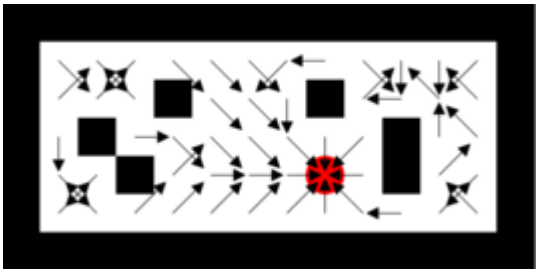

Fig. 17. Politica resultante ejecución q-learning 100000000 iteraciones (número elevado) en el segundo mapa.

  2) *Montecarlo*: Aunque el este mapa tiene menor dimensión el algoritmo que hemos implementado es muy costoso computacionalmente, por tanto, hemos establecido en 1000 el número de iteraciones, la política greedy como política inicial, 0.9 como factor de descuento e hicimos una primera prueba en la que establecimos el parámetro primera visita en true , lo que hace que se ejecute el algorimo de Montecarlo de primera visita, y una segunda prueba con el parámetro primera visita en false , lo que hace que se ejecute el algorimo de Montecarlo de cada visita. En ambos inicializamos los q values de todos los pares posibles estado acción arbitrariamente con el valor 0. Como se puede apreciar en las figuras 18 y 19, no hay una gran variación entre ambas ejecuciones, esto puede deberse principalmente a que hay un número reducido de iteraciones.

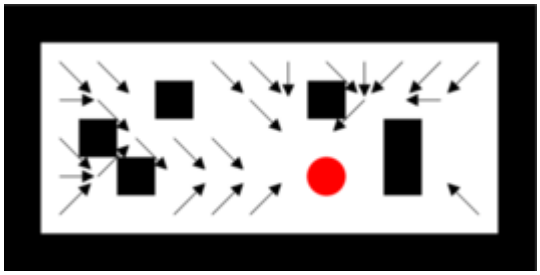

Fig. 18. Politica resultante ejecución montecarlo de primera visita en el segundo mapa.

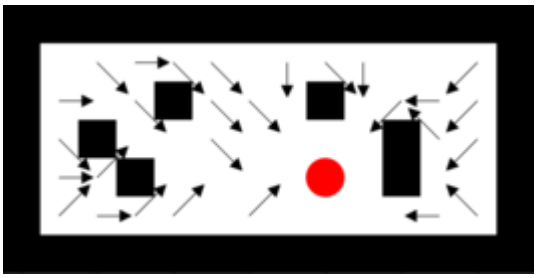

Fig. 19. Politica resultante ejecución montecarlo de cada visita en el segundo mapa.

  3) *SARSA*: Hemos realizado tres ejecuciones de este algoritmo, en la que lo único que hemos variado es el factor de aprendizaje y mantenido el resto de valores constantes para ver como esto afecta a la política resultante de ejecutar este algoritmo, en todas las ejecuciones hemos establecido como parámetros la política greedy como política inicial, 1000 el número de iteraciones y 0.9 como factor de descuento. En la primera ejecución hemos utilizado un valor intermedio , 0.5, como factor de aprendizaje, y en las siguientes dos ejecuciones hemos usado 0.1 y 0.9 como factor de aprendizaje para estudiar como varia en estos extremos. Comos se puede apreciar en las figuras 20,21 y 22, aunque se aprecian cambios no son muy sustanciales, esto puede ser debido principalmente a que no se han ejecutado con un numero elevado de iteraciones.

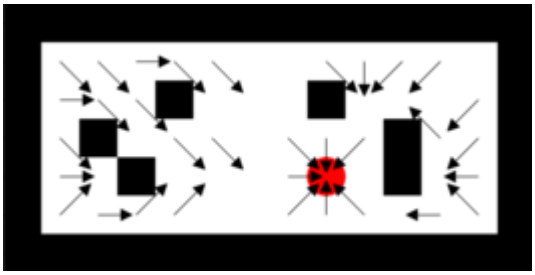

Fig. 20. Politica resultante ejecución SARSA con factor de aprendizaje 0,5 en el segundo mapa.

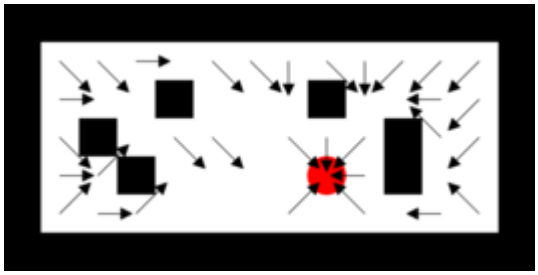

Fig. 21. Politica resultante ejecución SARSA con factor de aprendizaje 0,1 en el segundo mapa.

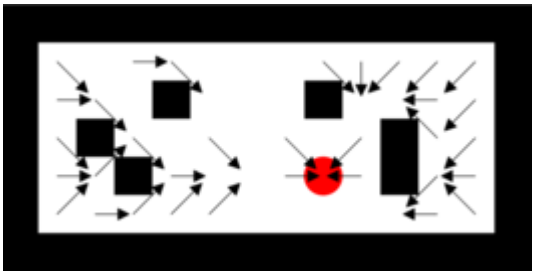

Fig. 22. Politica resultante ejecución SARSA con factor de aprendizaje 0,9 en el segundo mapa.

V. Conclusiones

Este estudio nos ha permitido observar la aplicación de diversos algoritmos de aprendizaje por refuerzo en un problema real, además de profundizar en la comprensión de los algoritmos tratados a lo largo del trabajo.

A partir de los experimentos realizados, hemos concluido que el aprendizaje por refuerzo es considerablemente costoso en términos computacionales, lo cual implica la necesidad de implementaciones de algoritmos altamente optimizadas , que seguirán siendo costosas cuando se aumente el número de iteraciones, o el uso de computadores de gran potencia.

Referencias

[1] https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html

[2] Diapositivas de Teoria Aprendizaje por refuerzo. Departamento de Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla.

[3] Richard S. Sutton y Andrew G. Barto. Reinforcement Learning: An Introduction. MIT Press, 2018. SARSA: ON-POLICY TD CONTROL

[4] https://www.cs.us.es/docencia/aulavirtual/mod/forum/discuss.php?d=760