Estudio sobre Q-Learning y algunas de sus variaciones

José Manuel Rojas Granado
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
josrojgra@alum.us.es, jrojasgranado@gmail.com

Javier Solís García
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
javsolgar@alum.us.es, javiersgjavi@gmail.com

***Resumen*—El objetivo principal de este proyecto es aprender como funcionan los algoritmos de aprendizaje por refuerzo, más en concreto el algoritmo Q-Learning, además de conocer un poco sobre el estado del arte de este algoritmo.**

Para ello hemos hecho un pequeño estudio teórico comparativo del algoritmo Q-Learning y de algunas de sus variantes, así como una implementación propia de este y de su variación llamada *"SARSA"*. A lo largo de este documento se detallará dicha implementación y las distintas conclusiones derivadas de las pruebas.

***Palabras clave*—Inteligencia Artificial, Q-Learning, Aprendizaje por refuerzo, SARSA**

I. Introducción

Para dar contexto a nuestro proyecto empezaremos hablando del concepto *"aprendizaje por refuerzo"* dentro del ámbito de la inteligencia artificial. Este concepto está enmarcado dentro del campo del aprendizaje automático, que se dedica al estudio de algoritmos que son capaces de aprender de su entorno. Dentro de este campo existen principalmente dos tipos de algoritmos: aprendizaje supervisado y aprendizaje no supervisado.

El aprendizaje supervisado se refiere a aquellos algoritmos que parten de ejemplos, en los que se conocen las repuestas deseadas. A partir de estos ejemplos el algoritmo puede aprender y predecir las repuestas de ejemplos de los que no conoce sus respuestas.

Por otro lado tenemos el aprendizaje no supervisado, que son aquellos algoritmos en los que estamos interesados en aumentar la información ya disponible, y de posibles datos futuros, por ejemplo, dando una agrupación de los ejemplos según su similaridad (clustering), o simplificando las estructura de los mismos manteniendo sus características fundamentales (como en reducción de la dimensionalidad) (Caparrini, 2019) [1].

El aprendizaje por refuerzo no se engloba dentro de ninguno de estos dos tipos de algoritmos, ya que se suele considerar un tipo aparte, aunque se parece al aprendizaje supervisado. En este tipo de algoritmo se recibe algún tipo de feedback sobre la respuesta dada, si la respuesta es correcta es cuando el algoritmo empieza a parecerse al aprendizaje supervisado. Esto es porque en ambos casos el algoritmo recibe información de lo que es correcto. Pero es en las respuestas erradas donde se diferencian. Mientras que en el aprendizaje supervisado se le proporciona al algoritmo lo que debería haber respondido, en el aprendizaje por refuerzo solo se le informa que la respuesta es incorrecta y, normalmente, una cuantificación del error cometido.

En este proyecto, para el estudio del aprendizaje por refuerzo, se ha decidido hacer una implementación practica del algoritmo Q-Learning. Este algoritmo es comúnmente usado en sus variantes más simples para la resolución de laberintos, en nuestro caso lo usamos para, dado un tablero, encontrar el camino óptimo dadas una casilla de inicio y otra de fin.

Este algoritmo consiste en aprender a asignar valores de bondad a los pares estado-acción, siendo los valores más altos la opción óptima. El par estado-acción se define como: el estado actual, en nuestro caso la casilla del tablero donde nos encontramos, y la acción a tomar. Por ejemplo, nos encontramos en la segunda casilla y la acción a tomar es ir a la tercera casilla. Según esta definición, hay tantos pares estado-acción como estados haya multiplicados por el número de acciones que se pueden realizar en cada estado. Conforme el algoritmo va aprendiendo a asignar valores a los pares, este asignará los valores más altos a los pares que se acerquen más a la solución o estado final, que en nuestro caso es la casilla a la que deseamos llegar (Caparrini, 2019) [2].

A continuación, se detalla el procedimiento seguido a lo largo del desarrollo del proyecto, así como el análisis y resultados obtenidos tras realizar diversos experimentos, seguido de las conclusiones obtenidas.

II. Preliminares

A. Métodos empleados

En esta sección hablaremos de los aspectos teóricos del algoritmo Q-Learnning.

El algoritmo Q-Learning puede encontrar una política óptima, maximizando el valor esperado de la recompensa total sobre todos los pasos sucesivos, empezando por un estado inicial, para cualquier proceso de decisión de Markov finito ("Wikipedia Q-learning", 2020) [3].

Un proceso de decisión de Markov finito tiene las siguientes propiedades:

- El agente percibe un conjunto finito, S, de estados distintos en su entorno, y dispone de un conjunto finito, A, de acciones para interactuar con él.

- El tiempo avanza de forma discreta, y en cada instante de tiempo, t, el agente percibe un estado concreto, s_t, selecciona una acción posible, a_t, y la ejecuta, obteniendo un nuevo estado, $s_t + 1 = a_t(s_t)$.
- El entorno responde a la acción del agente por medio de una recompensa, o castigo, $r(s_t, a_t)$. Formalizaremos esta recompensa/castigo por medio de un número, de forma que cuanto mayor es, mayor es el beneficio.
- Tanto la recompensa como el estado siguiente obtenido no tienen por qué ser conocidos a priori por el agente, y dependen únicamente del estado actual y de la acción tomada. Es decir, antes de aprender, el agente no sabe qué pasará cuando toma una acción determinada en un estado particular. Precisamente, un buen aprendizaje es aquel que permite adelantarse al agente en las consecuencias de las acciones tomadas, reconociendo las acciones, que sobre estados concretos, le llevan a conseguir con más eficacia mayores recompensas, sus objetivos.

El algoritmo Q-Learning afronta este tipo de problemas con los siguientes elementos:

- Una matriz Q, que alberga los valores que el algoritmo asigna a cada par estado-acción. Esta matriz inicialmente está rellena de ceros y a medida que se ejecuta el algoritmo va modificando los valores.
- Una matriz R, que contiene las recompensas de cada par estado-acción, estas recompensas están definidas a priori y no se modifican cuando se ejecuta el algoritmo.
- Una representación del tablero, laberinto, grafo, etc., que se pueda discretizar en estados y acciones a tomar. Este sería la base del problema a resolver, por ejemplo, estamos en un tablero y queremos ir de la casilla "A" a la casilla "B" por el camino óptimo.

Por lo descrito anteriormente podemos ver que el algoritmo se dedica únicamente a asignar valores a la matriz Q, pero cómo lo hace. Para un par estado-acción su valor correspondiente, en la matriz Q, contiene la suma de todas las recompensas posibles. El problema es que esta suma podría ser infinita en caso de que no haya un estado terminal que alcanzar y, además, es posible que no queramos dar el mismo peso a las recompensas inmediatas que a las recompensas futuras, en cuyo caso se hace uso de lo que se llama un refuerzo acumulado con descuento: las recompensas futuras quedan multiplicadas por un factor, $\gamma \in [0, 1]$, de forma que cuanto mayor sea este factor, más influencia tienen las recompensas futuras en el valor Q del par analizado.

Para que los valores de la matriz Q se aproximen lo máximo posible hemos de proporcionar un método que sea capaz de calcular el valor final a partir de los valores inmediatos y locales.

Para ello:

- Si una acción en un estado determinado causa un resultado no deseado, hay que aprender a no aplicar esa acción en ese estado. Si una acción en un estado determinado causa un resultado sí deseado, hay que aprender a aplicar esa acción en ese estado.
- Si todas las acciones que se pueden tomar desde un estado determinado dan resultado negativo, es conveniente aprender a evitar ese estado. Es decir, no tomar acciones desde otros estados que nos lleven a él. Por contra, si cualquier acción en un determinado estado da un resultado positivo, se debe aprender que es conveniente llegar a él. Este hecho es lo que permite propagar la recompensa de un par (estado, acción) a los pares de los estados adyacentes.

Matemáticamente, podemos formalizar el cálculo de los valores Q por medio de la siguiente ecuación (1):

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \max\left(Q(s_t + 1, a_t + 1)\right) \quad (1)$$

Esto es, el valor de Q óptimo para un par (estado,acción) es la suma de la recompensa recibida cuando se aplica la acción junto al valor descontado del mejor valor Q, que se puede conseguir desde el estado alcanzado al aplicar esa acción. Para aproximar este cálculo, al principio del aprendizaje los valores Q se establecen a un valor fijo (puede ser aleatorio), a continuación el agente va tomando pares de (estado,acción) y anota cuánta recompensa recibe en ellos, entonces, actualiza el valor almacenado del valor Q de cada par considerando como ciertas las anotaciones tomadas de los otros pares (algunas de las cuales habrán sido aproximadas en pasos anteriores).

Un aspecto importante del aprendizaje por refuerzo es la restricción de que solo se actualizan los valores Q de acciones que se ejecutan sobre estados por los que se pasa, no se aprende nada de acciones que no se intentan. Pero puede ser interesante que, sobre todo al principio del aprendizaje, el agente intente una gama más amplia de acciones, para hacerse una idea de lo que funciona y lo que no. Más concretamente, en cualquier momento el agente puede elegir entre:

- Seleccionar una acción con el valor Q más alto para ese estado (explotación).
- Seleccionar una acción al azar (exploración).

Vamos a formalizar este proceso de decisión en términos de probabilidades:

La posibilidad más simple es elegir completamente al azar entre una opción y otra, pero usaremos una opción un poco más inteligente y asignaremos una probabilidad a cada acción en función del valor Q asociado, de forma que cuanto mayor sea el valor Q de esa acción, mayor será la probabilidad de ser elegida. De esta forma, incluso las acciones con valores Q más bajos tendrán opciones de ser elegidas. También tiene sentido hacer que la probabilidad de exploración dependa del tiempo que el agente lleva aprendiendo, de tal forma que, al principio, se favorece la exploración porque lo que el agente haya aprendido todavía no es confiable y porque todavía quedan muchas opciones que no han sido consideradas, y más tarde se potencia la explotación, porque la experiencia del agente hace que su conocimiento sea más confiable. Así, la ecuación de la probabilidad de seleccionar la acción at sería (2):

$$P(a_t) = \frac{e^{EtQ(s_t, a_t)}}{\sum_{a \in A_{st}} e^{EtQ(s_t, a)}} \quad (2)$$

donde E es una constante de explotación, y A_{st} representa todas las posibles acciones que se pueden tomar desde s_t. Gracias a la función exponencial, a medida que t aumenta, las acciones que tengan un valor Q más alto se hacen mucho más grandes, por lo que tendrán más probabilidad de ser elegidas.

Se puede comprobar que estos valores Q convergen al valor óptimo que estamos buscando. En caso de que el espacio de estados sea una cantidad finita y manipulable, no es necesario realizar toda la segunda parte de selección de acciones para equilibrar la exploración con la explotación, basta realizar un recorrido completo de todos los posibles pares e ir actualizando el valor de Q. La convergencia del método asegura que obtendremos el resultado deseado.

III. Metodología

En este apartado se explicará cómo hemos implementado el algoritmo de Q-Learning. La implementación se puede dividir en tres partes: carga de los datos, definición del algoritmo y representación de la solución y el rendimiento. Se ha añadido un apartado extra para explicar, de forma breve, el algoritmo *"SARSA"* que posteriormente compararemos con el algoritmo Q-Learning.

Para hacer la implementación hemos usado el lenguaje de programación python en su versión 3.7, y las librerías os, csv, numpy, matplotlib, networkx y random. Las dos primeras se usan para leer datos de un fichero externo, la de numpy para tener algunas herramientas para trabajar con números, matplotlib para dibujar gráficas, la de networkx para dibujar la solución como un grafo y la de random para hacer elecciones aleatorias.

Antes de empezar a explicar la implementación vamos a definir el problema al cual le hemos dado solución con este algoritmo: Dado un tablero con casillas numeradas del 0 al 8, se quiere averiguar el camino óptimo para llegar de la casilla 0 a la 6. El tablero tiene la siguiente distribución:

8	7	6
3	2	5
0	1	4

Tabla 1. Tablero de ejemplo

Se elegirá una representación del tablero, como grafo, matriz, etc. Se definirá la matriz R de recompensas con valores -1 para las transiciones no válidas (por ejemplo de la casilla 0 a la 4), valores 0 cuando la transición de una casilla a otra sea válida (por ejemplo, de 0 a 1,2 o 3) y la recompensa será 100 cuando alcancemos la casilla objetivo, 6.

Ahora se explicará la primera subdivisión de código que hemos escrito para implementar la solución a este problema.

A. Carga de datos

Lo primero que hicimos fue escoger una representación del tablero para poder trabajar con él. La representación la hicimos en forma de matriz, ya que conceptualmente casa mucho con la idea de tablero que se nos proporciona en el enunciado del problema, ya que al visualizar la matriz puedes comprobar las transiciones que se pueden hacer, las que no y hacer el recorrido. Además, en python es fácil tratar con las matrices, ya que son arrays bidimensionales a los que solamente consultaremos y actualizaremos sus datos.

Para cargar el tablero hemos definido la función *cargar-datos-txt*, que recibe como parámetro el fichero txt con los datos y los convierte en una matriz. Esta función también se utiliza para cargar una matriz de recompensas personalizada si así lo queremos, de lo contrario se generará una matriz de recompensas de forma automática.

Para cargar un tablero cualquiera lo único que necesitamos es indicarle al algoritmo, al comienzo de la ejecución, qué fichero queremos que use. El fichero debe estar ubicado en el directorio raíz del proyecto. Como ejemplo proporcionamos el fichero *"ejemplo.txt"*, que contiene el tablero del enunciado del problema propuesto anteriormente.

Como se ha mencionado anteriormente se puede representar cualquier tablero siempre que cumpla estas restricciones: el valor de las casillas debe ser un número entero comprendido entre cero y el número de casillas menos uno $(0, n * m - 1)$, siendo n el número de filas de la matriz del tablero y m las columnas; y no puede tener saltos de línea ni al principio ni al final del fichero. Además, este fichero debe estar ubicado en la carpeta raíz del proyecto.

Ahora que ya tenemos el tablero debemos definir la matriz de recompensas R. Podemos hacer que se autogenere, indicándole al algortimo con un 0, cuando se ejecute, o bien podemos crear un fichero en el directorio raíz del proyecto con las recompensas. Hay ya un fichero de ejemplo que se puede emplear llamado *"recompensas_enunciado.txt"*. En este fichero se indican todos los movimientos posibles entre las diferentes casillas, los que no lo son y cuál es la casilla de fin.

En la matriz de recompensas R las filas indican la casilla actual en la que nos encontramos y las columnas los movimientos posibles o imposibles. Por ejemplo, la fila 2 columna 3 indica el movimiento de la casilla 2 a la casilla 3. Ahora para saber si este movimiento es posible o no, debemos fijarnos en el valor de $R(2, 3)$, si es mayor o igual a 0 es que el movimiento está permitido y si es -1 es que no lo está. Si fuera el caso de que la casilla 3 es la casilla objetivo tendría que ser el valor más alto de la matriz.

La matriz R debe contener tantas filas y columnas como casillas haya en el tablero, para formar una matriz de $(n*m)*(n*m)$, siendo n el número de filas de la matriz que representa el tablero y m las columnas. Además también debe tener como recompensa, que indica un movimiento invalido, un valor de -1, y los movimientos que lleven a la casilla objetivo deben tener el valor más alto.

Si queremos que se autogenere debemos tener en cuenta que los únicos movimientos permitidos son arriba, abajo, izquierda y derecha, siempre que el movimiento lleve a otra casilla, no siendo así cuando la casilla actual está en una esquina o borde de la matriz. Además, solo se añadirán tres recompensas diferentes: -1 para los movimientos inválidos, 0 para los válidos y 100 para cuando ese movimiento lleve a la

casilla de fin. Sólo se permitirán como válidos los movimientos a casillas adyacentes.

Si decidimos poner nuestra propia matriz R de recompensas puede darse el caso de que el algoritmo encuentre una solución que contenga bucles, yendo de una casilla A y otra B varias veces. Si se da este caso, el algoritmo nos indicará que la solución contiene bucles y no nos dará ningún camino como solución o camino intermedio. Esto también suele pasar en las primeras épocas cuando se ejecuta con una matriz de recompensas auto-generada, pero también puede darse el caso de que todos los caminos intermedios y la solución nos den un camino con bucles si le indicamos muy pocas épocas o la matriz R es personalizada.

Aparte de estas dos matrices, el tablero y las recompensas, también necesitamos la matriz de aprendizaje Q, pero esta se genera de forma automática con la función *"inicializa_Q"*. La cual inicializa con ceros una matriz de las mismas dimensiones que la matriz de recompensas R.

Por último nos hace falta conocer las casillas de inicio y fin, los parámetros gamma, ϵ y α, y el número de épocas. Todos estos parámetros se nos requerirá que introduzcamos cuando ejecutemos el programa.

En cuanto a las épocas, indican al algoritmo el número de iteraciones que debe ejecutarse, cuantas más veces se ejecute más actualizará la matriz Q, y más posibilidades tendrá de encontrar más y mejores caminos. Y el factor de aprendizaje γ es el peso que queremos dar a las recompensas futuras. Cuanto mayor sea este factor más explorará el algoritmo, y, cuanto menor sea más importancia le dará a las recompensas inmediatas y menos explorará.

Los parámetros ϵ y α se explicarán más adelante para que sirven, pero haciendo un pequeño adelanto, estos se usan para una variación del algoritmo Q-Learnig.

B. Definición del algoritmo

En este proyecto se nos pedía que hicieramos dos implementaciones del algoritmo Q-Learning, una sin los parámetros ϵ y α, y otra con ellos. El primer algoritmo lo hemos llamado Q-Learning Fase 1 y al segundo Q-Learning Fase 2, en referencia al enunciado del proyecto.

Empezando por la fase 1, que es la que no tiene los parámetros ϵ y α, se ha implementado en el método llamado *q-learning*.

Vamos a ver el pseudocódigo y a analizarlo [Algoritmo 1].

En el psudocódigo vemos los parámetros de entrada que habíamos definido anteriormente.

En cuanto a la salida tenemos la matriz de aprendizaje Q, que es la que luego nos permitirá construir el camino más optimo y el vector rendimiento, que es una forma de medir cuanto aprende el algoritmo y como varía a lo largo de las épocas.

En el algoritmo lo primero que hacemos es inicializar con un vector vacío el rendimiento. Luego hacemos un bucle que se ejecuta tantas veces como épocas haya.

Dentro de este bucle es donde está el algoritmo Q-learning que hemos definido de la sección de *"Preliminares"*. Lo primero es seleccionar un estado actual aleatorio, que es la casilla del tablero donde nos encontramos, luego declaramos una variable que nos indica si es el primer movimiento que vamos a hacer en la época. Esto es, porque, si empieza en la casilla de fin y se ha especificado en la matriz de recompensas que se puede ir de la casilla de fin a la misma casilla de fin, ejecute esta acción antes de terminar la época, ya que de lo contrario no se ejecutaría el algoritmo en esta época porque ya habría llegado a la casilla objetivo, además de que los valores de Q estarían entre la recompensa mínima y la máxima definida en R.

Algoritmo 1 *q_learing*

Entrada: Matriz tablero; matriz R; matriz Q; casilla de fin; número de épocas; Parámetro γ.
Salida: Matriz Q; vector de rendimiento.

```
1: rendimiento ← []
2: primera-iteración ← cierto
3: para(cada época)
4: estado-actual ← selección-de-estado-aleatorio
5: mientras(estado-actual != fin o primera-iteración)
6: acción ← seleccionar-acción-aleatoria
7: máximo ← maxima recompensa del siguiente estado tras
   aplicar la acción
8: Q(estado, accion) ← R(estado, accion) + γ * maximo
9: estado-actual ← acción
10: primera-iteración ← falso
11: fin mientras
12: rendimiento ← rendimiento + calcula-rendimiento(Q)
13: fin para
14: devolver Q, rendimiento
```

Después se efectuá otro bucle, mientras no se llegue a la casilla de fin o bien sea la primera iteración de la época. Dentro de este bucle seleccionamos una acción aleatoria, de las posibles que se pueden hacer desde el estado actual. Luego seleccionamos la máxima recompensa posible del estado resultante de la acción anterior (siguiente estado) y la asignamos a la variable máximo. Después actualizamos la matriz Q pero solamente actualizamos el valor de Q para el par (estado actual, acción). Este valor se sustituye por el valor de la matriz de recompensas R para el el par (estado actual, acción) más el factor γ por la variable máximo.

Y por último, dentro de este bucle, llevamos a cabo el movimiento, que consiste en asignarle al estado actual el valor de la acción. Después, a la variable primera iteración le asignamos el valor *false*, ya que si ha llegado hasta aquí, significa que ha llegado al menos una vez al final de la primera iteración.

Ya una vez fuera del bucle *while* vamos añadiendo el cálculo del rendimiento al vector rendimiento, teniendo así un vector con el rendimiento de todas las épocas.

Cuando se ha ejecutado el algoritmo el número de épocas indicado, se devuelven tanto la matriz Q como el vector de rendimiento.

Esto que acabábamos de ver es el cuerpo principal del

algoritmo, pero aún hay algunas funciones que no se han explicado y que sin embargo tienen una gran importancia. Estas son funciones que el método del algoritmo delega para poder reutilizarlas en el otro algoritmo, además de separar el código de forma lógica en bloques. Esto hace que el código sea más fácil de depurar y de leer.

Este es el caso de la selección aleatoria de la casilla de inicio, la selección aleatoria de la acción, la máxima recompensa para el siguiente estado y el cómo se calcula el rendimiento. Aunque sepamos lo que hacen estas funciones no hemos explicado su implementación, por lo que nos vamos a detener para mostrarlo antes de explicar el algoritmo Q-Learnig Fase 2.

- Selección de una acción aleatoria: para implementar esta funcionalidad se ha creado una función llamada *"seleccionar-accion-aleatoria"* cuyo pseudocódigo es el siguiente:

Algoritmo 2 *seleccionar_accion_aleatoria*

Entrada: Estado actual; Matriz de recompensas R.
Salida: Acción a tomar.

1: aleatorio ← -1
2: **mientras**(aleatorio = -1)
3: acción ← número aleatorio ∈ [0,columnas de $R-1$]
4: aleatorio ← R(casilla actual, acción)
5: **fin mientras**
6: **devolver** acción

En este psudocódigo podemos ver la definición de la variable aleatorio con valor -1, este valor se modifica dentro del bucle con la recompensa de aplicar al estado actual una acción aleatoria, en el paso 4. La acción a tomar se escoge con un método de selección aleatoria entre un rango de números que nos proporciona la librería numpy. Este nos devuelve un número aleatorio entre 0 y el número que le indiquemos, en este caso el número de filas de la matriz de recompensas, que es el mismo número que casillas tiene el tablero. Una vez elegida una acción a tomar, la variable aleatorio cambia de valor como dijimos al principio, y el bucle *while* evalúa si la recompensa para el par estado-acción es igual a -1, si es el caso se vuelve a ejecutar el bucle seleccionando otra acción aleatoria para ese estado. Esto es así hasta que encuentra un par estado-acción que su recompensa es diferente de -1, de esta forma nos aseguramos de que las acciones seleccionadas siempre son válidas.

- Selección aleatoria de la casilla de inicio: en este caso el código es tan simple que no necesita de pseudocódigo para entenderlo. Haciendo uso de la librería numpy, que devuelve un numero aleatorio entre 0 y el numero indicado, se escogen dos números aleatorios que como máximo tengan el mismo valor que el número de filas y el número de columnas de la matriz del tablero, creando las variables i,j respectivamente. El estado es la del valor para los índices i,j en la matriz del tablero data ($data[i][j]$).
- Elección de la máxima recompensa para el siguiente estado: para ello simplemente se escoge el máximo valor de las recompensas para las acciones posibles del estado siguiente. Pero esta recompensa se escoge en la matriz de aprendizaje Q, no en la de recompensas. De modo que conforme pasen las épocas el algoritmo irá aprendiendo que para ciertos estados se obtiene una mayor recompensa, ya que desde ellos se pueden llevar acabo acciones que ofrecen recompensas mayores que desde otros estados. Por ejemplo, desde la casilla 2 obtengo la misma recompensa al ir a las casillas 4 y 3, pero desde la casilla 3 puedo ejecutar una acción que me da una mayor recompensa que desde la casilla 4. Teniendo esto en cuenta el algoritmo asignará una mayor recompensa al par estado-acción Q(2,3) que al Q(2,4).
  Todo esto se da si el factor de aprendizaje γ tiene un valor alto, para valores bajos esto deja de tener peso en el algoritmo y se centra en las recompensas más inmediatas.
- Cálculo del rendimiento: para el calculo del rendimiento hemos implementado otra función en la que delegamos este cálculo:

Algoritmo 3 *calcula_rendimiento*

Entrada: Matriz de aprendizaje Q.
Salida: Valor del rendimiento.

1: suma ← 0
2: máximo ← 1
3: i ← 0
4: **para**(cada fila de la matriz Q)
5: suma ← suma + suma de los valores de $Q[i]$
6: **si**(máximo $< \max(Q[i])$)
7: máximo ← $\max(Q[i])$
8: **fin si**
9: i ← i + 1
10: **fin para**
11: **devolver** suma

El cálculo del rendimiento no es más que el sumatorio de todos los valores de la matriz Q dividido por el máximo de sus valores y multiplicado por cien. Para hacer este calculo definimos la variable suma donde se va acumulando la suma de los valores de las filas de Q, y la variable máximo, que es donde vamos almacenando los valores máximos de cada fila. El bucle recorre todas las filas de la matriz Q y, dentro de este se suman los valores de Q, además de modificar el valor del máximo si se ha encontrado un mayor valor dentro de Q, dándonos como resultado el máximo absoluto de la matriz. Por último, se devuelve directamente la suma.

Ahora que ya sabemos como funciona y como se ha implementado el algoritmo Q-learning Fase 1, vamos a ver el Fase 2 apoyándonos en este.

El algoritmo Q-Learning Fase 2 es igual al anterior a excepción de los parámetros de entrada ϵ y α, y de la función de seleccionar la acción aleatoria, que se ha implementado de otra forma.

El algoritmo se ha implementado en el método llamado *"q_learning_F2"*. El parámetro ϵ se utiliza en la nueva función de seleccionar la acción aleatoria, y el parámetro α para modificar el valor de ϵ en cada iteración de cada época. El valor de estos parámetros los decide el usuario a la hora de lanzar el algoritmo, cuyos valores siempre han de valer entre 0 y 1.

Cabe destacar que en cada época se restablece el valor de ϵ. Aquí podemos ver el pseudocódigo de la función *"q_learning_F2"*:

Algoritmo 4 *q_learning_F2*

Entrada: Matriz tablero; Matriz R; Matriz Q; Casilla de fin; Número de épocas; Parámetro γ; Parámetro ϵ, Parámetro α.
Salida: Matriz Q; Valor del rendimiento.

```
1: rendimiento ← []
2: para( cada época)
3: estado_actual ← selección_estado_aleatorio
4: primera_iteración ← cierto
5: ϵ-prima ← ϵ
6: mientras( estado_actual != fin o primera_iteración)
7: acción ← seleccionar_acción_aleatoria_F2
8: máximo ← max(Q[acción])
9: Q(estado, acción) ← R(estado, acción) + γ∗máximo
10: primera_iteración ← falso
11: ϵ-prima ← ϵ-prima∗α
12: fin mientras
13: rendimiento ← rendimiento.añadir(calcula_rencimiento(Q))
14: fin para
15: devolver Q, rendimiento
```

A simple vista parece el mismo algoritmo que el de la fase 1, pero vamos a poner de relieve las diferencias.

La primera, y más obvia son los parámetros de entrada ϵ y α, la segunda es la utilización de los mismos y la tercera es la llamada a la función *"seleccionar_acción_aleatoria_F2"*.

En cuanto a la utilización de los nuevos parámetros solo cabe añadir que se utilizan, el parámetro ϵ para calcular la acción aleatoria dentro de la función que se ha mencionado, y el parámetro α para modificar el valor de ϵ como se ha explicado con anterioridad. Además se utiliza *epsilo-prima*, asignándole el valor $\epsilon * \alpha$ para que se vaya modificando dentro del bucle (paso 6), y no modifique el valor original, y poder restablecer el valor original al inicio de cada época.

Pasemos a ver y explicar el pseudocódigo de la nueva función de seleccionar una acción aleatoria, que se ha implementado con el método llamado *"seleccionar_acción_aleatoria_F2"* (*Algoritmo 5*).

El pseudocódigo mostrado aquí difiere mucho de la función *seleccionar_acción_aleatorio* que vió, ahora vamos a explicar qué hace esta función y cómo afecta el parámetro ϵ.

Lo primero que hacemos es definir un número aleatorio entre 0 y 1 para compararlo con el parámetro ϵ. Si el valor del parámetro es más grande se llama a la función *seleccionar_acción_aleatorio* y el algoritmo devuelve la acción que esta devuelve a su vez.

Algoritmo 5 *seleccionar_acción_aleatoria_F2*

Entrada: Matriz de aprendizaje Q; Estado actual; Matriz de recompensas R; Parámetro ϵ.
Salida: Acción a tomar.

```
1: aleatorio ← número aleatorio ∈ [0, 1]
2: si(aleatorio < ϵ)
3: acción ← seleccionar_acción_aleatoria
4: si no
5: acciones_posibles ← []
6: acciones ← R[estado]
7: contador ← 0
8: para( cada acción)
9: si( acción >= 0)
10: acciones_posibles←acciones_posibles.añadir(contador)
11: fin si
12: contador ← contador + 1
13: fin para
14: no_es_máximo ← cierto
15: mientras(no_es_máximo)
16: aleatorio_máximo ← elección_aleatoria(acciones_posibles)
17: si( Q[estado][aleatorio_máximo] = máximo)
18: no_es_máximo ← falso
19: fin si
20: fin mientras
21: acción ← aleatorio_máximo
22: fin si
23: devolver suma
```

Si no es así debemos escoger la acción que más recompensa tenga para el estado actual, dentro de la matriz de aprendizaje Q. Para ello debemos obtener las acciones posibles y dentro de ellas escoger la de mayor valor. Si hubiera varias con este mismo valor deberemos escoger una aleatoria entre ellas.

Para almacenar las acciones posibles definimos la variable *acciones_posibles* (paso 5), las acciones válidas e inválidas para estado actual en la variable *acciones* y un contador, que nos dice el índice de la columna de las acciones que son posibles dentro del bucle que le sigue.

Dentro de este bucle comprobamos si la acción es posible, comprobando que el valor de la acción es mayor o igual a 0. Si es posible se añade el índice (*contador*) de la acción al vector *acciones_posibles*. Al final de cada iteración se incrementa el contador.

Cuando finaliza el bucle se asigna a la variable máximo el mayor valor de la matriz Q para el *estado_actual*. Luego se define una variable *booleana no_es_máximo* a *true*, y se ejecuta un bucle mientras esta variable mantenga su valor.

Cuando se ejecuta este bucle se le asigna a la nueva variable *aleatorio_máximo* una de las acciones posibles, que se calcularon anteriormente, de forma aleatoria. Si el valor de la matriz Q para el par estado-acción, que forman el estado actual y la acción que contiene esta nueva variable, coincide con el valor de la variable *máximo* entonces, la variable

no_es_máximo pasa a valer *false* sino, se ejecuta el bucle hasta que se cumpla esta condición. Por último, una vez fuera del bucle, se devuelve el valor de la variable *aleatorio_máximo*.

Con los nuevos parámetros lo que se consigue es que: en el algoritmo el componente de aleatoriedad vaya descendiendo, dándose el fenómeno de que cuantas más iteraciones tenga una época menos probabilidad haya de que se exploren caminos nuevos, dando prioridad a las recompensas inmediatas. Esto se debe a que esta componente depende del factor ϵ, de si es más grande que un número aleatorio. El azar sigue jugando un papel importante pero cada vez menos, puesto que en cada iteración es menos probable que un número aleatorio sea más pequeño que ϵ.

Y con esto daríamos por terminada la explicación del algoritmo *Q_Learning* fase 2.

C. Representación de la solución y rendimiento

Primero vamos a definir cuál es la solución a este problema y luego hablaremos de su cálculo y representación.

La solución al problema, dado un tablero, una casilla de inicio y una casilla de fin, es un subconjunto de casillas ordenado por orden de selección. Es decir, si se empieza en la casilla 1 y se escoge ir sucesivamente a las casillas 2, 3 y 4, se representarán estas casillas en ese orden. Estas casillas serán las que cuya suma de recompensas, en la matriz de aprendizaje Q, sea máxima.

La solución más intuitiva sería la del camino más corto de la casilla de inicio a la de fin, pero eso puede ser falso. En el caso de que se autogenere la matriz de recompensas R esta definición sería acertada, pero sino, si la definimos nosotros puede no serlo.

Pongamos un ejemplo: Quiero ir de la casilla 0 a la 9, y el camino más corto es el 0-3-6-8-9. Pero quiero que el recorrido pase por la casilla 2.

En este caso le puedo dar una recompensa más alta a la casilla 2 que al resto, pero menor que a la casilla de fin. Entonces el algoritmo aprenderá que ir a la casilla 2 tiene una recompensa alta y que debe pasar por ella.

Ahora que ya hemos visto en qué consiste la solución vamos a ver como se ha implementado la función *"calcular_camino"* y como se representa, por lo que la solución sería un camino que no tiene por qué ser el más corto, pero si ha de pasar por 2.

El pseudocódigo de esta función es el que podemos ver a continuación [Algoritmo 6].

Como vemos esta función para calcular el camino solo necesita la matriz de aprendizaje Q, la casilla de inicio y la de final. Sin embargo no tiene ninguna variable de salida, ya que lo que devuelve es un mensaje que se muestra en pantalla.

La función empieza definiendo variables auxiliares, para ir almacenado las casillas que conforman el camino, así como la casilla por la que vamos pasando hasta llegar al final.

También tenemos inicializadas las variables que guardan la representación del camino y la cobertura, que por defecto es *true*.

Algoritmo 6 calcula-camino

Entrada: Matriz de aprendizaje Q; casilla de inicio; casilla de fin.

```
posición ← inicio
camino ← []
cobertura ← cierto
representación ← ""
mientras(posición != fin)
máximo ← max(Q[posición])
siguiente ← índice de máximo en (Q[posición])
posición ← siguiente
si(posición ∈ camino)
cobertura ← falso
rompe el bucle
fin si
imprime "El algoritmo no ha encontrado un camino sin bucles"
camino ← añadir siguiente a camino
fin mientras
represención ← conjunto de casillas en orden de adición separadas por "→"
si(cobertura)
imprime representación del camino
fin si
```

La variable cobertura viene a significar si el algoritmo es capaz de encontrar un camino o si por el contrario se queda en bucle y nunca llega a la casilla final. Este puede ser el caso si hacemos una mala asignación de recompensas en la matriz R antes de ejecutar el algoritmo, si no escogemos la generación automática de R. Esto se debe a que si la recompensa por ir de la casilla 2 a la 3 es más alta que el resto de acciones posibles y cuando nos encontramos en la casilla 3 la recompensa de ir a la casilla 2 es también la recompensa más alta, se provoca un bucle en que el algoritmo avanza y retrocede continuamente. Esto también suele pasar durante las primeras épocas, ya que la matriz de aprendizaje Q se ha actualizado pocas veces y sus valores aún no se han ponderado bien.

Luego de inicializar estas variables hacemos un bucle *while* hasta que la casilla actual sea la de fin. Dentro de este bucle buscamos el valor máximo en la matriz Q para el par *estado-actual-acción*. Luego, cogemos la casilla a donde nos conduce esta acción seleccionada y la asignamos como posición actual.

Luego hacemos la comprobación de si la casilla actual está ya en el camino, y si es así imprimimos un mensaje de que no se ha podido encontrar un camino y cambiamos el valor de la variable cobertura a false además de salir del bucle. Esto es para evitar una situación como la que se describe anteriormente, en la que nos quedamos en un bucle.

Si esto no pasa, simplemente añadimos la casilla actual al camino. Es aquí donde se termina el bucle *while*.

Después de este bucle se construye la representación del camino y si la variable cobertura es igual *true* entonces pintamos el camino junto a un mensaje.

Esta representación del camino es la que se utiliza tanto

para los caminos intermedios, el camino encontrado en cada época, y en el camino de la última época, que es la solución.

Para este último camino además se ha implementado una función que pinta el tablero como un grafo, que tiene los nodos de distintos colores para indicar el camino. Para hacer este grafo se ha utilizado la librería networkx, la cual nos proporciona las herramientas que necesitamos.

Puesto que esta representación es un añadido, y que además, su código es bastante simple no nos detendremos a analizar el pseudocódigo, simplemente describiremos lo que hacen las funciones que implementan esta funcionalidad:

- crea-vertices: crea tantos vértices como casillas hay en el tablero.
- crea-aristas: crea una arista entre cada casilla, si la acción para moverse de una casilla A a una B está permitida.
- crea-colores: si la casilla está en el camino y es la de inicio o fin, colorea de amarillo los nodos correspondientes a estas casillas, sino colorea de rojo el nodo. Y si la casilla no está en el camino colorea de azul el nodo de esa casilla
- *muestra_grafo_camino*: si el camino no contiene bucles, entonces crea un grafo con los nodos y las aristas anteriores y lo pinta por pantalla.

El grafo quedaría tal que así:

```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 6

Leyenda:
inicio/fin = amarillo
camino escogido = rojo
resto de casillas = azul
```

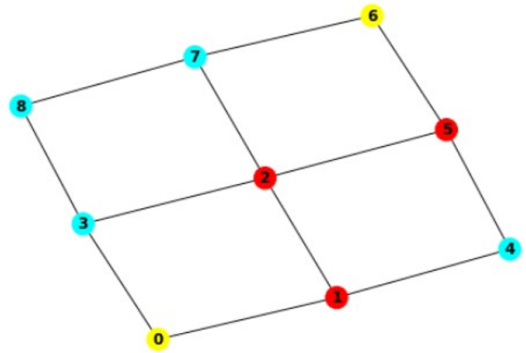
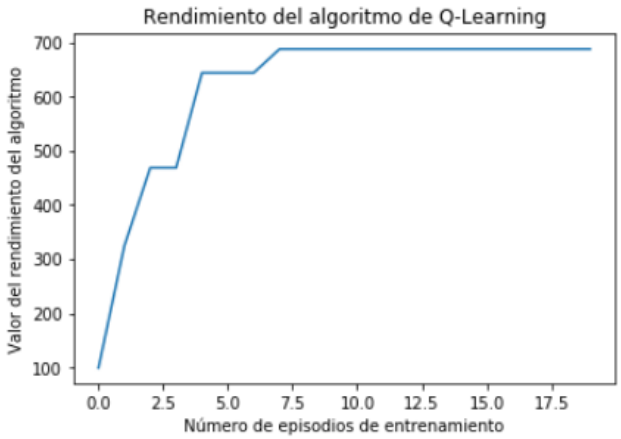


Fig.1 Resultado de la ejecución

Una vez representado el camino solo nos queda representar el rendimiento (el cálculo se ha explicado anteriormente), para ello hemos usado la librería matplotlib y en concreto la funciónes pyplot. Con esta librería conseguimos pintar una gráfica como la que se vé a continuación:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 6
```

Fig.2 Rendimiento obtenido tras la ejecución

El código para generar la gráfica es bastante simple y no merece la pena analizarlo, solo hay que indicarle el título, las etiquetas del eje x e y, he indicarle un vector de valores, que es el rendimiento del algoritmo, ya sea el fase 1 o fase 2, que se extrae de la matriz de aprendizaje Q en cada época (como se explica en el cálculo del rendimiento). Y abajo de la gráfica podemos ver la representación del camino óptimo.

D. Implementación del algoritmo "SARSA"

Para la implementación del algoritmo *"SARSA"* hemos utilizado como guía algunas páginas web y artículos que estudian este algoritmo entre otros (Suárez, 2020)[12].

Primero vamos a ver el pseudocódigo del algoritmo y a analizarlo:

Algoritmo 7 SARSA

Entrada: Matriz de aprendizaje Q; casilla de inicio; casilla de fin.
Salida: Matriz Q; Vector de rendimiento.

1: rendimiento ← []
2: **para**(cada época)
3: estado_actual ← selección_estado_aleatorio
4: primera_iteración ← **cierto**
5: **mientras**(estado_actual != fin o primera_iteración)
6: acción ← *seleccionar_acción_aleatoria(desde estado)*
7: estado2 ← acción
8: acción2 ← *seleccionar_acción_aleatoria(desde acción)*
9: Q(estado, acción) ← Q(estado, acción) + $\alpha[R$(estado, acción) + γQ(estado2, acción2) − Q(estado, acción)]
10: estado ← acción
11: primero ← **falso**
12: **fin mientras**
13: rendimiento ← rendimiento + *calcula_rendimiento(Q)*
14: **fin para**
15: Q, rendimiento

Como vemos se parece mucho al algoritmo Q_Learning fase 1, pero hay varias diferencias. La primera de todas es que no hay una elección de la máxima recompensa para el estado siguiente, en su lugar se calcula una acción aleatoria que da lugar a un segundo estado y a otra segunda acción aleatoria (pasos 6, 7 y 8).

Y la segunda diferencia es la fórmula de actualización de la matriz Q. En esta fórmula vemos que se utiliza un parámetro α para ponderar el aprendizaje de recompensas, ya que la recompensas inmediatas de la matriz R se multiplican por este parámetro, y se pondera doblemente la recompensa del par estado-acción 2 para luego restarle la recompensa del par actual.

Esto último suena un poco raro así que vamos a detenernos un poco. Cuando decimos que se pondera doblemente es porque la recompensa para el par estado-acción 2 está multiplicado tanto por el parámetro γ como por el parámetro α, teniendo así menos peso en la ecuación, ya que estos siempre toman valores entre 0 y 1.

El resto del pseudocódigo es igual que en el algoritmo Q_Learning, de hecho reutilizan las funciones *"selección-estado-aleatorio"* y *"seleccionar-acciónaleatoria"*.

IV. Resultados

En esta sección vamos a exponer primero las pruebas realizadas y luego comentaremos el resultado de las mismas para poder hacer comparaciones entre ellas. Pero primero vamos a hacer una guía rápida de cómo reproducir las pruebas.

Vamos a ejecutar, por ejemplo, el algoritmo Q_Leaning fase 1 sobre el tablero de ejemplo que nos propone el enunciado:

Primero abrimos el notebook *"Trabajo IA"*, que es donde se ha hecho la implementación del algoritmo. Una vez abierto ejecutamos todas las celdas y nos vamos a la parte de abajo del notebook a la sección llamada *"Main"*. Esta es la parte que ejecuta el programa. Si le hemos dado a ejecutar todas las celdas nos irán apareciendo, debajo de este bloque, un formulario para que indiquemos los datos de entrada al algoritmo. Es aquí donde debemos indicarle los siguientes parámetros:

- Si queremos cargar nuestra matriz de recompensas le indicamos un 1, o si queremos que se autogenere un 0.
- Casilla de inicio.
- Casilla objetivo
- Número de épocas
- El valor del factor de aprendizaje $\gamma \in [0, 1]$.
- El nombre del fichero donde está el tablero que se quiere cargar (sin extensión).
- Si hemos especificado que queremos cargar nuestra matriz de recompensas, ahora debemos indicarle el nombre del fichero (sin extensión).
- Si queremos que se muestren los caminos intermedios le indicamos un 1, sino un 0.
- El algoritmo que queremos que ejecute, si queremos ejecutar el Q_Learning fase 1 (sin los parámetros α y ϵ) le indicamos un 1, si queremos que ejecute el algoritmo Q_Learning fase 2 le indicamos un 2 y si queremos ejecutar el algoritmo *"SARSA"* le indicamos un 3.
- El valor del parámetro ϵ y α, ambos entre 0 y 1 (los decimales se indican con un punto no con una coma).

A partir de ahora vamos a enumerar las pruebas realizadas:

A. Prueba 1:

Para la primera prueba le indicaremos al algoritmo los siguientes datos de entrada:

Autogenerar R	0
Casilla de inicio	0
Casilla objetivo	6
Epìsodios de entrenamiento	20
γ	0'5
Nombre del fichero del tablero	ejemplo
Caminos intermedios	1
Algoritmo	1

Tabla 2. Parámetros ejemplo 1

Como resultado de estos datos de entrada tenemos como respuesta:

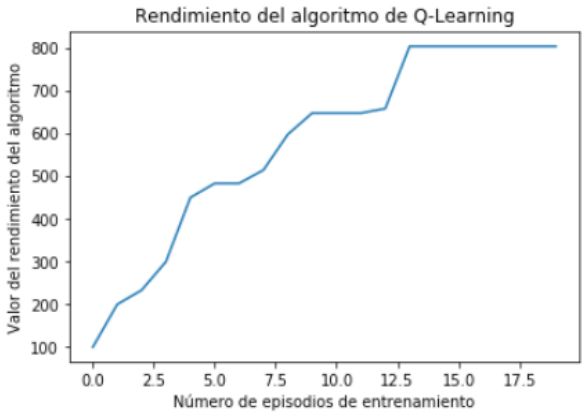


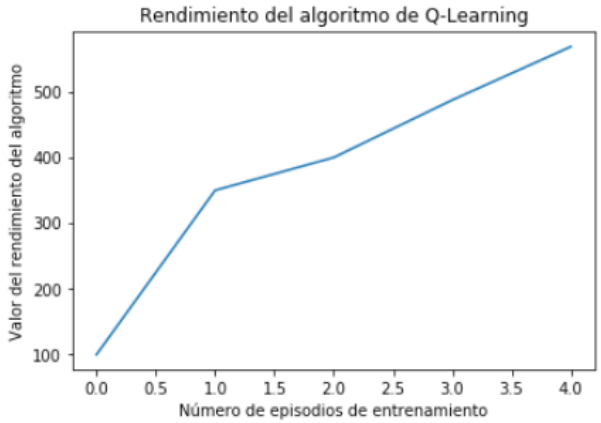
Fig.3 Rendimiento obtenido tras la ejecución de la prueba 1

Además de los caminos intermedios:

- De la época 0 a la 4, incluida, no se ha encontrado un camino sin bucles.
- De la época 5 a la 11, incluida, el camino óptimo es: 0 $\rightarrow$ 3 $\rightarrow$ 2 $\rightarrow$ 5 $\rightarrow$ 6
- De la época 12 a la 19 el camino óptimo es el que se puede ver en la figura anterior [Fig.3].

B. Prueba 2:

Ahora vamos a ver el mismo ejemplo pero con 5 épocas:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 6
```

Fig.4 Rendimiento obtenido tras la ejecución de la prueba 2

C. *Prueba 3:*

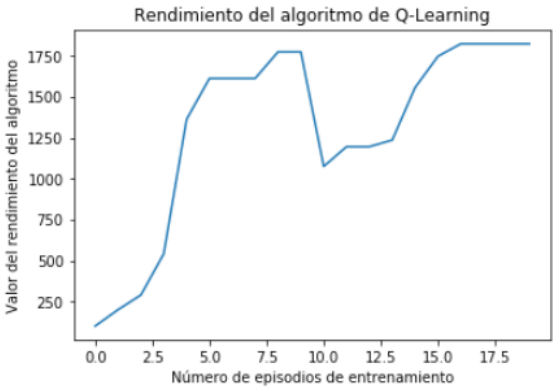
Ahora vamos a ver el mismo ejemplo pero con 50 épocas:


```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 6
```

Fig.5 Rendimiento y Q obtenido tras la ejecución de la prueba 3

D. *Prueba 4:*

Para un factor de aprendizaje γ igual a 0.9 y fijando el número de épocas a 20 tenemos el siguiente resultado:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 6
```

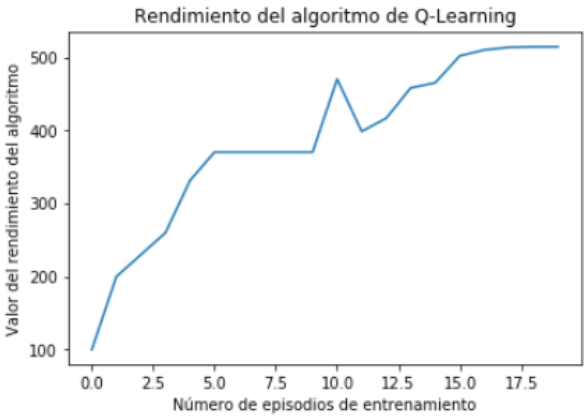
Fig.6 Rendimiento obtenido tras la ejecución de la prueba 4

En los caminos intermedios se ha obtenido que:

- Hasta la época 4 no se ha encontrado camino sin bucles.
- De la época 4 hasta la 13 el camino es el que se vé en la imagen.
- En las épocas 14 y 15 explora dos caminos diferentes.
- De la 16 a la 20 consigue un camino sin bucles.

E. *Prueba 5:*

Ahora pasemos a ver un factor de aprendizaje más bajo, de 0.3:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 6
```

Fig.7 Rendimiento obtenido tras la ejecución de la prueba 5

En este caso los caminos intermedios han sido:

- De 0 a 3 épocas no se ha encontrado camino sin bucles.
- De la época 4 a la 16 sigue el camino : $0 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 6$
- Y desde la 17 a la 19 consigue el camino sin bucles.

A partir de ahora y en las pruebas sucesivas se usará el tablero contenido en el fichero *"ejemplo2.txt"*, que es el siguiente:

0	1	4	9	13
3	2	5	10	14
8	7	6	11	12

Tabla 3. Tablero de *"ejemplo2.txt"*

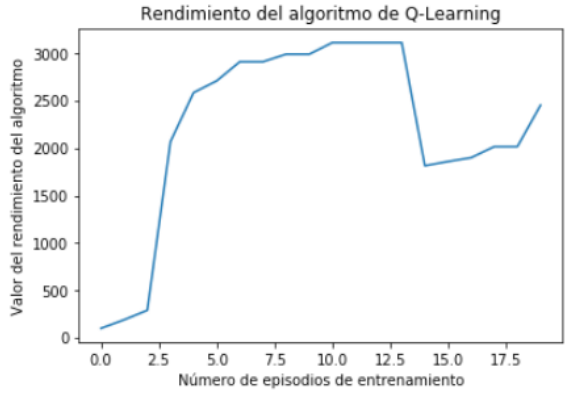
Y los parámetros serán:

Autogenerar R	0
Casilla de inicio	0
Casilla objetivo	14
Epìsodios de entrenamiento	20
γ	0'5
Nombre del fichero del tablero	ejemplo2
Caminos intermedios	1
Algoritmo	1

Tabla 4. Parámetros ejemplo 5

F. Prueba 6:

Para un γ de 0,9 tenemos que:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 10 -> 14
```

Fig.8 Rendimiento obtenido tras la ejecución de la prueba 6

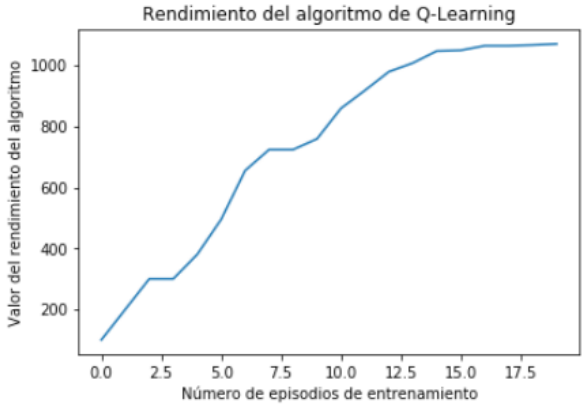
- De la época 0 a la 2 no se ha encontrado un camino sin bucles.
- En la época 3 el camino encontrado ha sido : 0 → 1 → 2 → 5 → 6 → 11 → 12 → 14
- En la época 4 el camino encontrado ha sido: 0 → 1 → 4 → 9 → 13 → 14
- De la época 5 a la 9: 0 → 1 → 4 → 9 → 10 → 14
- Y de la época 10 a la 19 la que podemos ver en la siguiente figura junto con el rendimiento [Fig. 8].

G. Prueba 7:

Ahora vamos a ver el mismo ejemplo pero con factor γ igual a 0.5:

- De la época 0 a la 10 no se ha encontrado un camino sin bucles.
- De la época 11 a la 13 y de las épocas 16 a 19 se ha encontrado el camino: 0 → 3 → 2 → 5 → 10 → 14
- En la época 4 el camino encontrado ha sido: 0 → 1 → 4 → 9 → 13 → 14
- En las épocas 14 y 15 se ha encontrado el camino: 0 → 1 → 4 → 9 → 13 → 14

Y el rendimiento ha sido:




```
El camino óptimo es:  0 -> 1 -> 4 -> 5 -> 10 -> 14
```

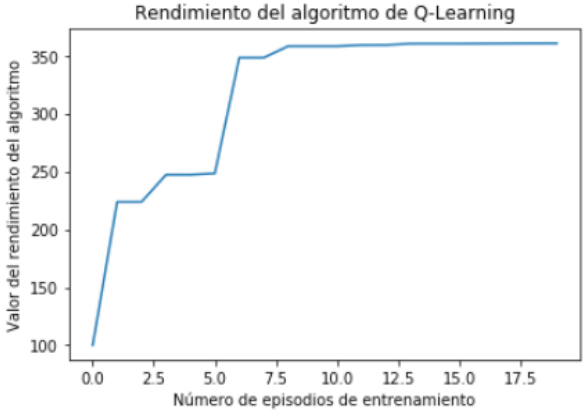
Fig.9 Rendimiento obtenido tras la ejecución de la prueba 7

H. Prueba 8:

Por último vamos a ver que supone indicarle un factor de aprendizaje γ de 0.1:

- De la época 0 a la 2 no encuentra ningún camino sin bucles.
- De la época 3 a la 12 y de la 17 y 18, el camino encontrado es: 0 → 1 → 4 → 5 → 10 → 14
- Del 13 al 16 el camino encontrado es: 0 → 1 → 4 → 9 → 13 → 14
- En la época 19 el camino es: 0 → 1 → 2 → 5 → 10 → 14

Y el rendimiento ha sido:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 10 -> 14
```

Fig.10 Rendimiento obtenido tras la ejecución de la prueba 8

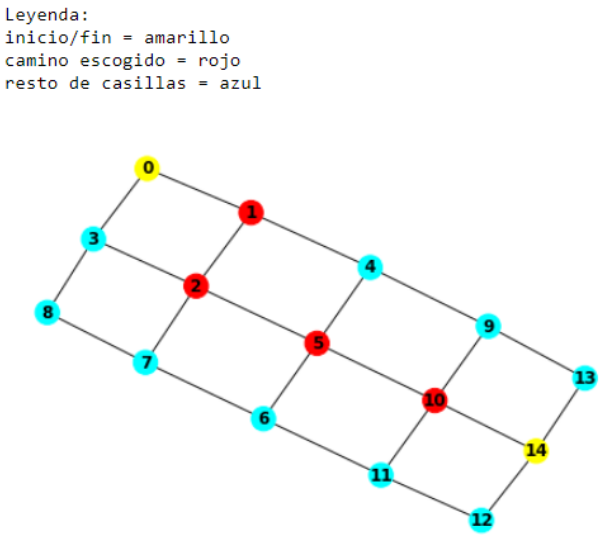


Fig.11 Camino obtenido tras la ejecución de la prueba 8

I. Prueba 9:

Vamos a coger el tablero de *"ejemplo2"* y vamos a cambiar las recompensas de modo que el camino más óptimo en vez de pasar por la diagonal del tablero pase por el borde. Para ello pondremos recompensas en los pares estado-acción, que marquen el camino deseado que sean más altas de la matriz R.

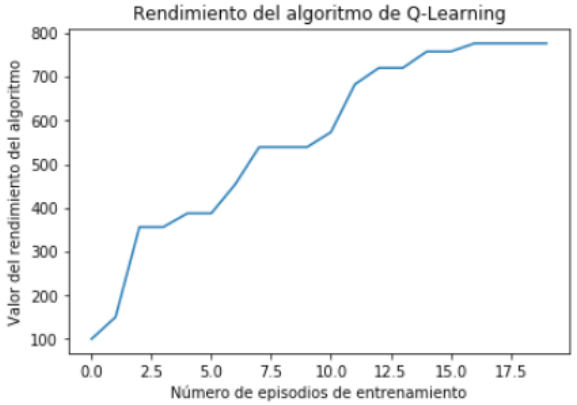
-1	10	-1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
0	-1	0	-1	20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	0	-1	0	-1	0	-1	0	-1	-1	-1	-1	-1	-1	-1
0	-1	0	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1
-1	0	-1	-1	-1	0	-1	-1	-1	30	-1	-1	-1	-1	-1
-1	-1	0	-1	0	-1	0	-1	-1	-1	0	-1	-1	-1	-1
-1	-1	-1	-1	-1	0	-1	0	-1	-1	-1	0	-1	-1	-1
-1	-1	0	-1	-1	-1	0	-1	0	-1	-1	-1	-1	-1	-1
-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	0	-1	-1	-1	-1	-1	0	-1	-1	40	-1
-1	-1	-1	-1	-1	0	-1	-1	-1	0	-1	0	-1	-1	100
-1	-1	-1	-1	-1	-1	0	-1	-1	-1	0	-1	0	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	100
-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	100
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	0	0	100

Fig.12 Matriz de recompensas R de tablero *"ejemplo2"* cambiada

Se usarán los siguientes parámetros de entrada:

Autogenerar R	1
Casilla de inicio	0
Casilla objetivo	14
Epìsodios de entrenamiento	20
γ	0'5
Nombre del fichero del tablero	ejemplo2
Caminos intermedios	1
Algoritmo	1

Tabla 5. Parámetros ejemplo 9




```
El camino óptimo es:  0 -> 1 -> 4 -> 9 -> 10 -> 14
```

Fig.13 Rendimiento obtenido tras la ejecución de la prueba 9

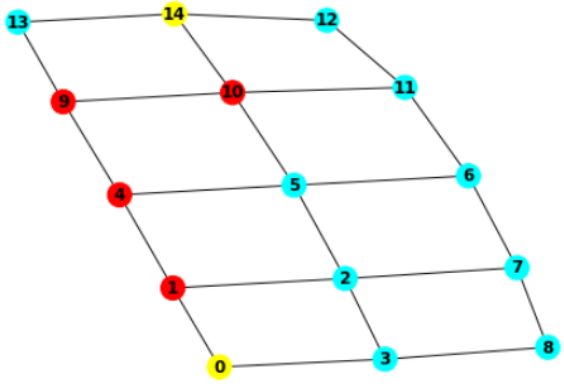


Fig.14 Camino obtenido tras la ejecución de la prueba 9

En cuanto a los caminos intermedios, estos son los que se han obtenido:

- En la época 0 y 1 no se encontró un camino sin bucles.
- En la época 2 se encontró el camino: $0 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow 14$
- Y de la época 3 hasta la 19 ya encuentra el camino óptimo: $0 \rightarrow 1 \rightarrow 4 \rightarrow 9 \rightarrow 13 \rightarrow 14$

Ahora vamos a empezar a hacer pruebas con el algoritmo Q-Learning fase2. Esta vez vamos a concentrarnos en pruebas en las que variaremos los nuevos parámetros aquí incluidos (ϵ y α).

J. Prueba 10:

Para esta prueba emplearemos los siguientes parámetros:

Autogenerar R	0
Casilla de inicio	0
Casilla objetivo	14
Epìsodios de entrenamiento	20
γ	0'5
Nombre del fichero del tablero	ejemplo2
Caminos intermedios	1
Algoritmo	2
ϵ	0'9
α	0'8

Tabla 6. Parámetros ejemplo 10

Los resultados han sido:

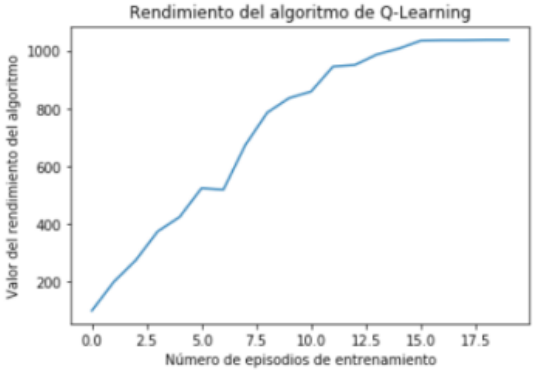


Fig.15 Rendimiento obtenido tras la ejecución de la prueba 10

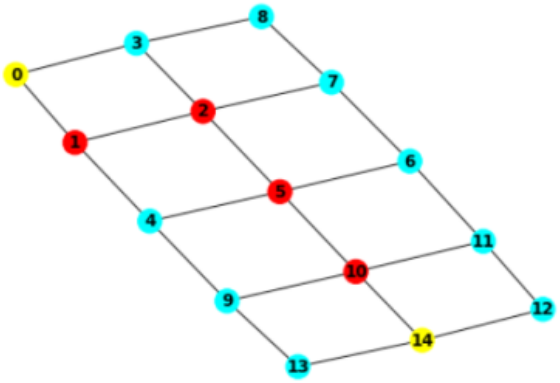


Fig.16 Camino obtenido tras la ejecución de la prueba 10

- De la época 0 a la 7 no se ha encontrado un camino sin bucles.
- En la 8 y 9 el camino ha sido: 0 → 1 → 4 → 9 → 10 → 14
- En la 10 ha sido: 0 → 1 → 4 → 5 → 10 → 14
- De la 11 a la 13 ha sido: 0 → 1 → 4 → 9 → 13 → 14
- Y de la 14 a la 19: 0 → 1 → 2 → 5 → 10 → 14

K. Prueba 11:

Para esta prueba utilizaremos los mismos parámetros que en la prueba anterior excepto el valor de α, que pasa a valer 0.4.

Los resultados son:

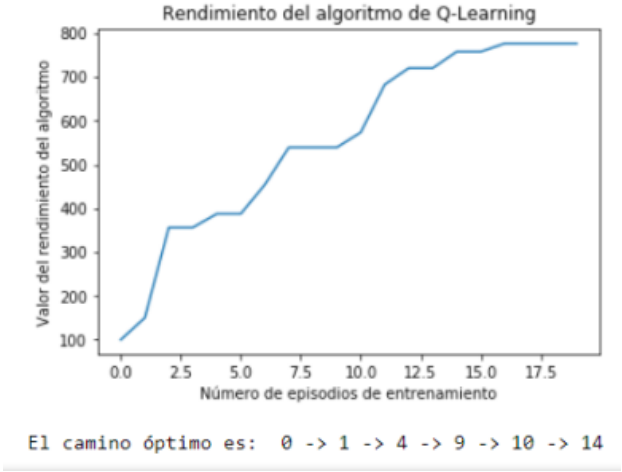


Fig.16 Rendimiento obtenido tras la ejecución de la prueba 9

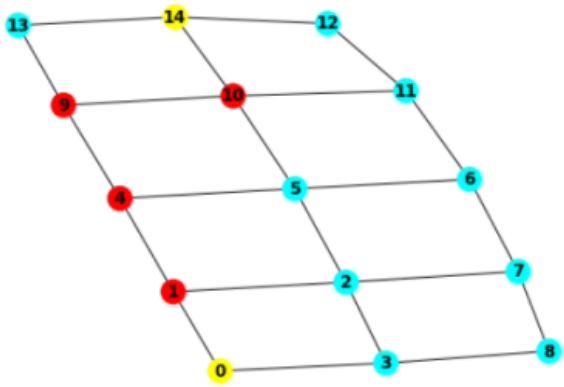


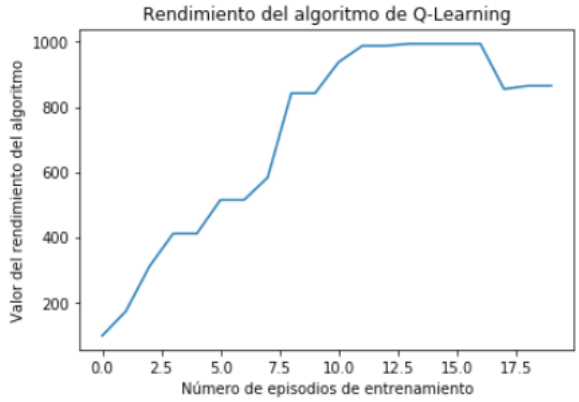
Fig.17 Camino obtenido tras la ejecución de la prueba 11

- De la época 0 a la 6 no se ha podido encontrar un camino sin bucles.
- De la 8 a la 11 el camino encontrado ha sido: 0 → 1 → 4 → 9 → 13 → 14
- De la 12 a la 19 ha sido: 0 → 1 → 4 → 9 → 10 → 14

L. Prueba 12:

En esta prueba utilizaremos los mismos parámetros que en la prueba anterior excepto el valor de α, que pasa a valer 0.1.

Los resultados han sido:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 10 -> 14
```

Fig.18 Rendimiento obtenido tras la ejecución de la prueba 12

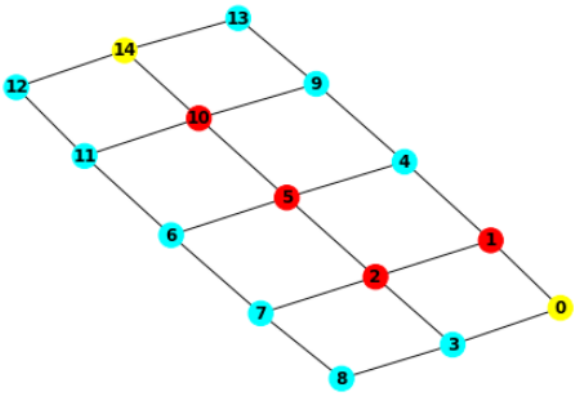


Fig.19 Camino obtenido tras la ejecución de la prueba 12

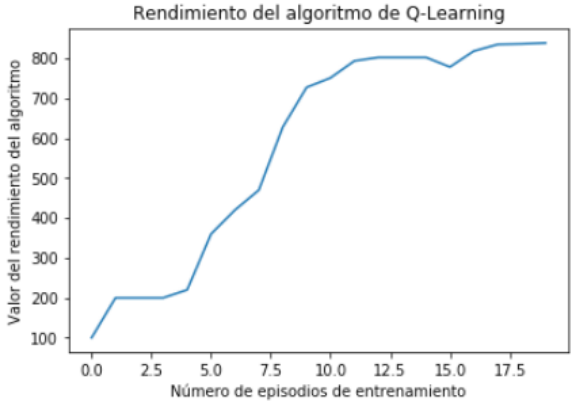
Los caminos intermedios han sido:

- De la época 0 a la 7 no se ha podido encontrar un camino sin bucles.
- De la 8 a la 19 ha sido: $0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 10 \rightarrow 14$

M. Prueba 13:

Para esta prueba se ha fijado el parámetro ϵ a 0.5, ya que en la prueba 10 ya tenemos un valor alto para este, y α se fijará con el mismo valor que en la prueba 10.

Los resultados han sido:




```
El camino óptimo es:  0 -> 3 -> 2 -> 5 -> 10 -> 14
```

Fig.20 Rendimiento obtenido tras la ejecución de la prueba 13

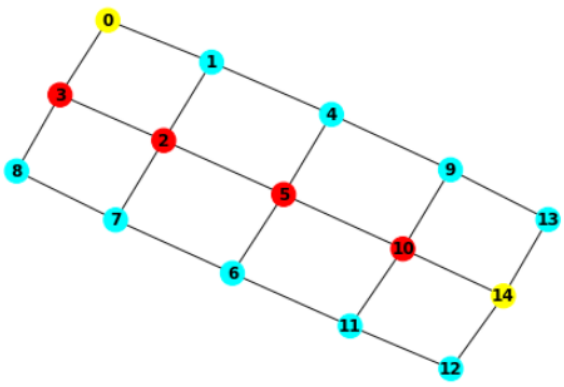


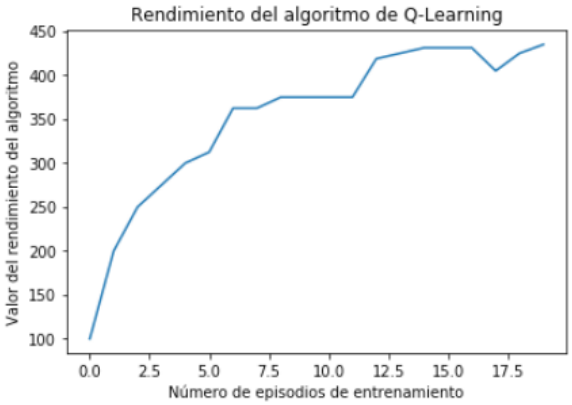
Fig.21 Camino obtenido tras la ejecución de la prueba 13

Los caminos intermedios han sido:

- De la época 0 a la 15 no se ha podido encontrar un camino sin bucles.
- De la 16 a la 19 el camino encontrado ha sido: $0 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 10 \rightarrow 14$

N. Prueba 14:

Para esta prueba se ha fijado ϵ en 0.1 y los resultados son:




```
El camino óptimo es:  0 -> 1 -> 4 -> 9 -> 13 -> 14
```

Fig.22 Rendimiento obtenido tras la ejecución de la prueba 14

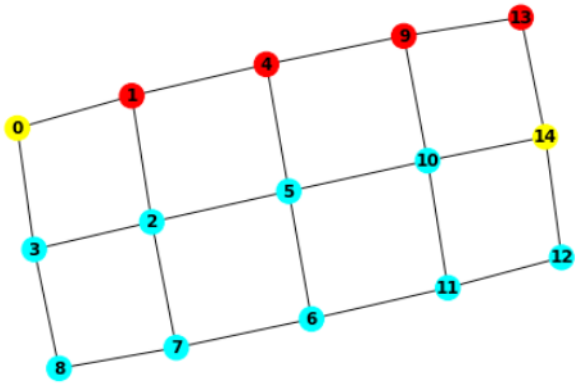


Fig.23 Camino obtenido tras la ejecución de la prueba 14

Los caminos intermedios han sido:

- De la época 0 a la 12 no se ha podido encontrar un camino sin bucles.
- De la 13 a la 19 el camino encontrado ha sido: $0 \rightarrow 1 \rightarrow 4 \rightarrow 9 \rightarrow 13 \rightarrow 14$

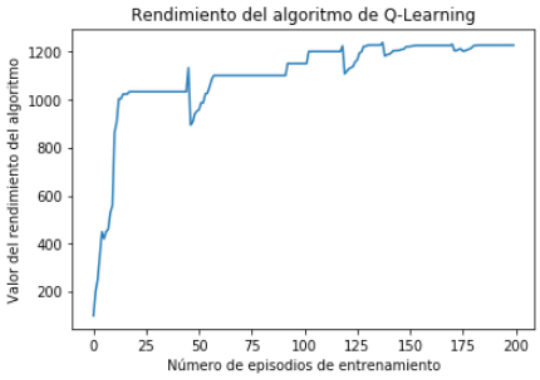
O. Prueba 15:

En esta prueba vamos ajecutar el algoritmo Q-Learning fase 1, pero esta vez con muchas épocas de entrenamiento. Los parámetros usados son:

Autogenerar R	0
Casilla de inicio	0
Casilla objetivo	14
Epìsodios de entrenamiento	200
γ	0'5
Nombre del fichero del tablero	ejemplo2
Caminos intermedios	1
Algoritmo	1

Tabla 6. Parámetros ejemplo 10

Los resultados son:




```
El camino óptimo es:  0 -> 1 -> 2 -> 5 -> 10 -> 14
```

Fig.24 Rendimiento obtenido tras la ejecución de la prueba 15

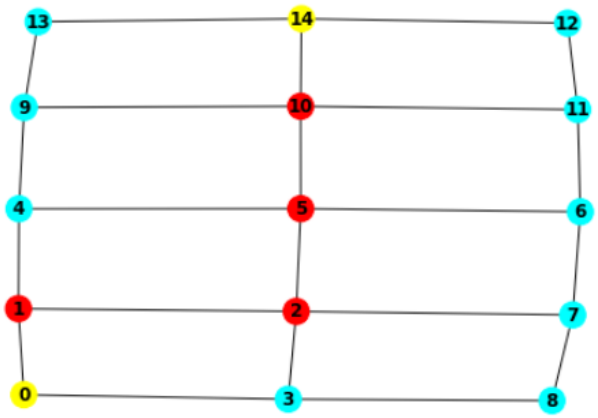


Fig.25 Camino obtenido tras la ejecución de la prueba 15

P. Prueba 16:

Esta vez se va a ejecutar el algoritmo *"SARSA"*, con los mismos parámetros que en la prueba anterior. Los resultados han sido:


```
El camino óptimo es:  0 -> 3 -> 2 -> 5 -> 10 -> 14
```

Fig.26 Rendimiento obtenido tras la ejecución de la prueba 16

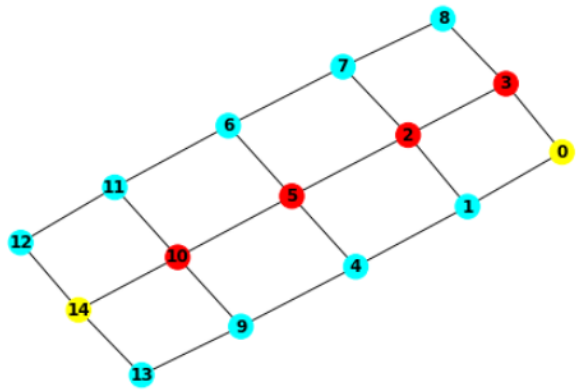
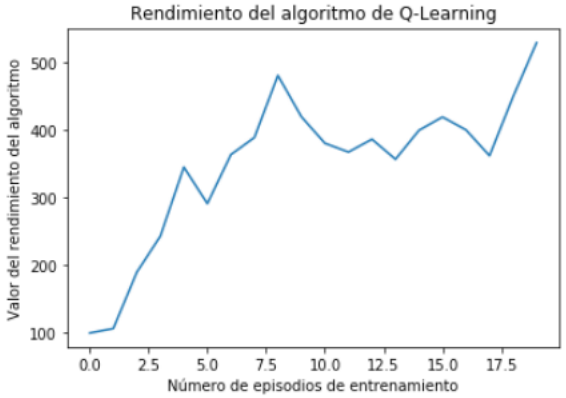


Fig.27 Camino obtenido tras la ejecución de la prueba 16

Q. Prueba 17:

Esta vez ejecutaremos el algoritmo *"SARSA"* de nuevo pero con 20 épocas y mostrando los caminos intermedios. Los resultados han sido:




```
El camino óptimo es:  0 -> 1 -> 4 -> 5 -> 10 -> 14
```

Fig.28 Rendimiento obtenido tras la ejecución de la prueba 17

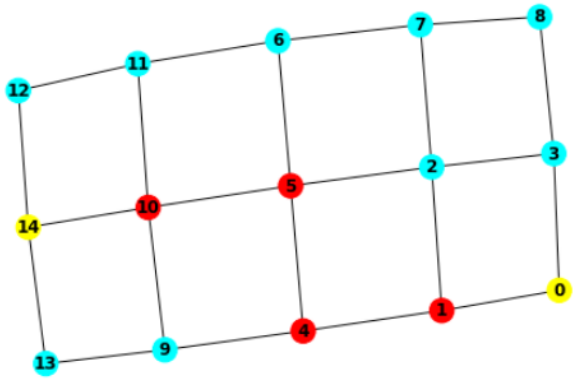


Fig.29 Camino obtenido tras la ejecución de la prueba 17

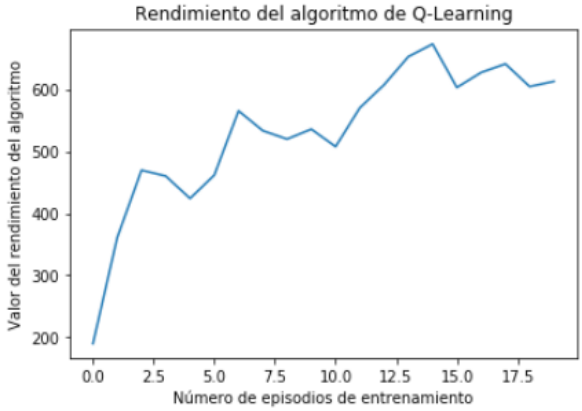
Los caminos intermedios han sido:

- De la época 0 a la 5, y en la 8, 15 y 16 no se ha podido encontrar un camino sin bucles.
- En la 6 y 7 el camino encontrado ha sido: 0 → 3 → 2 → 5 → 10 → 14
- En la 9, 10 y 17 el camino encontrado ha sido: 0 → 1 → 4 → 9 → 13 → 14
- En la 18 y 19 el camino encontrado ha sido: 0 → 1 → 4 → 5 → 10 → 14

R. Prueba 18:

Se ejecutará *"SARSA"* con los mismos parámetros que la prueba anterior pero esta vez indicándole que use la matriz de recompensas del fichero *"recompensas2.txt"*.

Los resultados han sido:




```
El camino óptimo es:  0 -> 1 -> 4 -> 9 -> 13 -> 14
```

Fig.30 Rendimiento obtenido tras la ejecución de la prueba 18

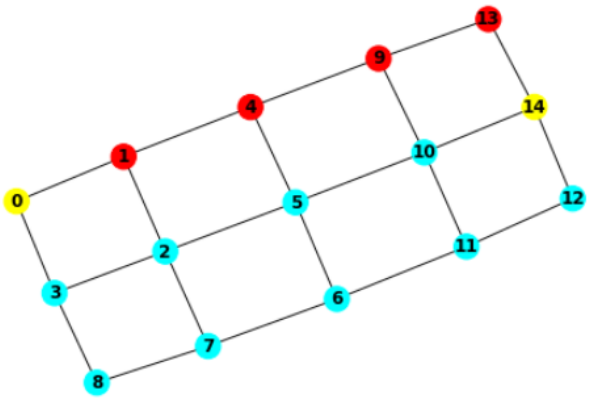


Fig.31 Camino obtenido tras la ejecución de la prueba 18

Los caminos intermedios han sido:

- En la época 0 no se ha podido encontrar un camino sin bucles.

- De la 2 a la 19 el camino encontrado ha sido: $0 \rightarrow 1 \rightarrow 4 \rightarrow 9 \rightarrow 13 \rightarrow 14$

Estas han sido todas las pruebas realizadas, ahora vamos a pasar analizarlas. Algunas de las pruebas serán analizadas de forma individual y otras serán analizadas en conjunto con otras, ya que su análisis no tendría más valor que en comparación con otras pruebas. Para analizar las pruebas de una forma lógica y ordenada dividiremos el análisis en bloques, donde analizaremos un aspecto o comportamiento de los algoritmos según los parámetros de entrada. También haremos una comparativa entre los algoritmos.

1) Variación del número de épocas:
   En este bloque analizaremos el comportamiento de los algoritmos Q-Learning fase 1 y fase 2.
   Empezando por la prueba 1, tenemos que esta es una prueba a modo de ejemplo para ver el funcionamiento del programa y el como se muestran los resultados. Lo que nos interesa de esta prueba es compararla con las pruebas 2 y 3.
   En estas pruebas lo que vemos son los resultados de ir variando el número de épocas, y lo que se intenta demostrar es que cuantas más épocas se le den al algoritmo es más probable que se encuentre el camino óptimo.
   El número de épocas toma los valores de 20, 5 y 50. Y lo que se observa es que siempre se llega al mismo camino óptimo. Podemos suponer que esto es debido a que el tablero es muy pequeño, por lo que no se necesitan muchas épocas para que la matriz de aprendizaje Q tenga unos pesos adecuados para encontrar este camino.
   Como en la prueba 1, y en muchas otras de las pruebas realizadas, en las primeras épocas no se ha encontrado un camino sin bucles, esto es porque, la matriz Q aún no ha asignado unos pesos propicios.
   De estas pruebas también podemos extraer que el rendimiento no está directamente relacionado con que se encuentre una solución óptima. En la prueba 1, con 20 épocas podemos ver que tenemos un rendimiento que avanza de forma escalonada hasta estancarse. En la prueba 2, con 5 épocas, vemos que el rendimiento es menor y que siempre está aumentando y sin embargo se llega a la misma solución. Y en la prueba 3, con 50 épocas, tenemos un rendimiento un poco mayor que en la prueba 1 y que llegado cierto punto empieza a fluctuar y a aumentar de forma más lenta.
   Mirando también otras pruebas, como la 13, vemos que en un número importante de sus primeras épocas no se encuentra un camino sin bucles. Y sin embargo vemos que durante las mismas ha aumentado el rendimiento. También ha habido casos, que aquí no se encuentran documentados, en los que obteniendo un rendimiento semejante a otras pruebas no se ha encontrado ningún camino sin bucles.

2) Variación del factor de aprendizaje γ:
   En este bloque se analiza como afecta el factor de aprendizaje γ en el algoritmo Q-Learning fase 1, y por consiguiente también en el fase 2. Esto es porque la sentencia de actualización para ambos algoritmos es la misma. Aunque hay que tener muy presente que aunque afecte de la misma manera, los resultados no serían los mismos si ejecutáramos las pruebas en ambos algoritmos, ya que en el fase 2 entran en juego los parámetros α y ϵ, lo que hace que a igualdad de parámetros los resultados puedan diferir.
   En las pruebas de la 4 a la 8, incluida, se ha ido variando el parámetro gamma para ver cómo afecta al comportamiento del algoritmo, también se toma en consideración la prueba 1 en este apartado. En las pruebas 1, 4 y 5 se utiliza el tablero del enunciado del proyecto (del fichero *"ejemplo.txt"*), y en el resto se utiliza un tablero un poco más grande (*"ejemplo2.txt"*) para acentuar las diferencias.
   En estas pruebas el parámetro γ toma los valores 0.9, 0.5, 0.3 y 0.1. Y lo que vemos con el tablero de ejemplo es que con gamma igual a 0.9 (prueba 4) se han explorado varios caminos, de hecho se encontró el camino de la solución, exploró otro camino y luego volvió al de la solución. En las pruebas 1 y 5, con γ igual a 0.5 y 0.3 respectivamente, vemos que han explorado más de un camino pero que cuando llegan al camino óptimo ya no exploran más.
   Con las pruebas 6, 7 y 8 pasa algo parecido, vemos que con el factor igual a 0.9 se tienen 5 caminos diferentes; con el factor igual a 0.5 se tienen 3; y con 0.1 se exploran 4.
   Si bien con la última prueba parece que no haya relación entre el factor γ y el número de caminos encontrados, porque encuentra más caminos con un factor de 0.1 que de 0.5, es que hay que tener en cuenta que este algoritmo tiene un factor de aleatoriedad a la hora de elegir la casilla desde donde empieza a explorar en cada época y en las acciones a tomar. Es por esto que, aunque haya excepciones, la tendencia que podemos observar en el algoritmo es que cuanto mayor sea el factor de aprendizaje más posibilidad hay de que se exploren nuevos caminos.

3) Variación de recompensas en la matriz R.
   En este bloque solo se ha hecho una prueba, la 9, con una matriz de recompensas R personalizada para compararla con las pruebas cuya matriz R se autogenera.
   Para ello se confeccionó el fichero *"recompensas2.txt"* para el tablero *"ejemplo2.txt"*, este fichero tiene un camino marcado para que el algoritmo lo siga. Para hacerlo hemos puesto recompensas más altas a los pares estado-acción que queremos que el algoritmo pondere más, excepto que aquellos que llevan a la casilla objetivo.
   Los pares estado acción que hemos recompensado de más han sido los que llevaban a la secuencia $0 \rightarrow 1 \rightarrow$

$4 \rightarrow 9 \rightarrow 13 \rightarrow 14$. Y como resultado hemos obtenido este camino cómo el óptimo. Pero lo interesante de ello es que lo ha encontrado de forma muy rápida, ya que desde la tercera época encontró este camino y no exploró ningún otro.

Cabe destacar que también se han hecho algunas pruebas que no están incluidas en este documento, como por ejemplo, el de dar una recompensa más alta a los pares que llevaran solo a una determinada casilla, a casillas que estuvieran muy lejos de la solución pero que no tuvieran un camino guiado hacia ni hacia ella misma, ni a la casilla objetivo como en el caso anterior o bien que tuvieran dos caminos diferentes igualmente recompensados.

En estos dos primeros casos lo que solía pasar es que el algoritmo o bien no pasaba por las casillas indicadas o bien los caminos obtenidos contenían bucles, siendo en muy pocas las ocasiones en las que el algoritmo determinaba un camino que pasara por esa casilla y no hubiera bucles.

En el último caso lo que obteníamos es que el algoritmo exploraba a veces ambos caminos y devolvía uno de ellos como óptimo, y otras simplemente exploraba uno de ellos.

Como vemos el comportamiento del algoritmo no es muy interesante en estas pruebas, ya que, dependen más del factor aleatorio que de los parámetros de entrada.

4) Variación del parámetro α:
   Este bloque analiza como varía el comportamiento del algoritmo Q-Learning fase 2 con la variación del parámetro α. Para este análisis nos fijaremos en las pruebas 10, 11 y 12, donde el parámetro tomará los valores 0.8, 0.4 y 0.1 respectivamente.

   En estas pruebas lo que se observa claramente es que cuanto más bajo es el valor de α menos caminos explora, encontrando con el valor más alto 5 caminos diferentes, con el intermedio 3 y con el más bajo 2.

   Las pruebas aquí expuestas son las más representativas, aunque como el algoritmo tiene un factor de aleatoriedad puede darse el caso de que se exploren los mismos caminos con el valor de α más alto y más bajo. Pero esto no es lo que nos interesa analizar, ya que esto serían la minoría de los casos, lo que nos interesa es la tendencia que sigue el algoritmo con esta variación en α.

   En cuanto al rendimiento de este algoritmo no hay mucho que decir, ya que, arroja un rendimiento similar al que ofrecería el Q-Learning fase 1, y en las tres pruebas se obtienen unas gráficas de rendimiento similares, con la salvedad de que en la prueba 12 en las últimas épocas el rendimiento baja un poco, pero nada que no se haya visto hasta ahora.

5) Variación del parámetro ϵ:
   En este bloque vamos a analizar el impacto del parámetro ϵ sobre el algoritmo Q-Learning fase 2. Para ello nos fijaremos en las pruebas 10, 13 y 14, las cuales tienen un valor de ϵ de 0.9, 0.5 y 0.1 respectivamente, y en todas se ha fijado el parámetro α en 0.8.

   Lo que vemos es que en la prueba 10 se exploran 5 caminos diferentes y en las pruebas 13 y 14 solamente 2. Además a este hecho hay que sumar que en ambas se tarda casi el doble de épocas en encontrar un camino que no contenga bucles. Y, aunque no es muy relevante, vemos que en la prueba 14, con ϵ igual a 0.1, se ha obtenido un rendimiento bastante más bajo que las otras dos pruebas. Esto se debe a que al explorar muchas veces el mismo camino no actualiza la matriz Q con los mismos valores, que llevarían a caminos menos óptimos pero que aumentarían el rendimiento.

   En estas pruebas podemos ver el mismo comportamiento que con las pruebas del bloque anterior, cuanto más bajo es el valor del parámetro menos caminos explora. Esto es porque si bajamos el valor del parámetro α estamos bajando también, de forma indirecta, el valor del parámetro ϵ. Hay que recordar que el parámetro α multiplica a ϵ en cada iteración del algoritmo para una misma época (en cada época nueva el valor de ϵ se restablece), bajando su valor en cada iteración.

   Pero la diferencia entre bajar el valor de ϵ y α, es que cuando se baja el valor de ϵ el algoritmo tarda mucho más en encontrar un camino sin bucles. Podemos observarlo en las pruebas 11 y 12, donde bajando el valor de α se obtienen los primeros caminos tras 6 y 4 épocas, y que en las pruebas 13 y 14 se tardan 15 y 12 épocas. Esto se debe a que al principio de las épocas, con un ϵ mayor, se tiene más probabilidad de explorar más, aunque el parámetro α haga que el otro parámetro disminuya muy rápidamente. Sin embargo con un ϵ bajo, desde el principio de las épocas, apenas se tienen probabilidades de explorar nuevos caminos y esto se agrava conforme pasan las iteraciones de las épocas por el parámetro α.

6) Comparación del algoritmo Q_Learning con el algoritmo *"SARSA"*:
   En este bloque vamos a comparar estos dos algoritmos, y para ello nos fijaremos en las pruebas 7, 15, 16, 17 y 18.

   La primera comparación que haremos será con las pruebas 7 y 17, que nos muestran el comportamiento de ambos algoritmos a igualdad de parámetros (el 17 corresponde a *"SARSA"* y el 7 a Q-Learning). En ambos vemos que se ha encontrado un camino óptimo, en la prueba 17 se exploran 4 caminos mientras que en la prueba 7 se exploran 3. La principal diferencia de *"SARSA"* con el algoritmo Q_Learning, es que *"SARSA"* antepone la exploración a encontrar el camino con la mayor recompensa.

   Esto se ve claramente en la fluctuación del rendimiento, en las gráficas de las pruebas con Q-Learning vemos (pruebas 7 y 15), que aunque a veces fluctúe,

tiende siempre a subir hasta quedarse estancado. Y en *"SARSA"* vemos que, además de que su rendimiento es menor, fluctuá mucho (pruebas 16 y 17). Esto se ve muy claramente si comparamos las gráficas de rendimiento de las pruebas 15 y 16, en las que se le han fijado el número de épocas a 200.

La última prueba que vemos, la 18, es la de *"SARSA"* con la matriz de recompensas personalizada (*"recompensas2.txt"*). En esta prueba podemos ver que, aunque el rendimiento fluctúa igual que en las otras pruebas, solamente explora dos caminos diferentes y el primero de ellos contiene bucles y corresponde a la época 0 solamente. Se ha de recalcar que esta prueba se ha ejecutado varias veces con idéntico resultado. Esto pone en relieve que, aunque este algoritmo tienda a explorar más, su propósito es por encima de todo encontrar un camino óptimo, por lo que si este está marcado lo encontrará de forma recurrente hasta que acabe su ejecución.

V. Conclusiones

A modo de resumen, este trabajo tiene como objetivo el estudio y comprensión de los algoritmos de aprendizaje por refuerzo. Para ello se ha hecho un estudio teórico y una implementación del algoritmo Q-Learning y también del algoritmo *"SARSA"*, que es una variante del anterior. Además para entender bien el funcionamiento del Q-Learning se ha hecho una implementación llamada fase 1, en el que se ha implementado el algoritmo según nos dice la literatura existente, y una variación de este con los parámetros α y ϵ con el propósito de ver la importancia del factor aleatorio del algoritmo. Además, el algoritmo *"SARSA"* se ha implementado para ver alguna de las variantes que se nombran en el anexo teórico y hacer una pequeña comparación con el algoritmo original.

La conclusión a la que hemos llegado después de haber realizado las pruebas que se mencionan en este documento y las que se han hecho durante el desarrollo del código y que no están incluidas, es que:

1) El algoritmo siempre tiene una componente de aleatoriedad a la hora de explorar caminos.
2) El factor de aprendizaje γ sirve para regular el peso de las recompensas futuras, teniendo más probabilidades de explorar nuevos caminos cuanto mayores sean estos pesos.
3) Los parámetros α y ϵ son capaces de aumentar y disminuir el factor aleatorio del algoritmo para explorar caminos
4) El número de épocas es importante solo hasta cierta cantidad, para que al algoritmo le dé tiempo a aprender. A partir de cierto número no importa que aumenten.
5) La matriz de recompensas R puede influir directamente en el comportamiento del algoritmo, haciendo que solo explore un camino o solo los deseados, puede que no afecte en nada dependiendo de cómo se confeccione la matriz.
6) El rendimiento pude ayudarnos a ver si el algoritmo sigue aprendiendo pero no está directamente relacionado con que el algoritmo encuentre caminos.
7) El algoritmo *"SARSA"* es una variante que tiende a obtener más información de caminos alternativos aunque estos no sean óptimos.

Y para terminar las conclusiones vamos a proponer algunas mejoras que se podrían incluir en futuros proyectos, como son: el estudio más detallado del algoritmo *"SARSA"* así como su comparación con otros algoritmos del anexo teórico y también la inclusión de pruebas con datos que representen problemas reales

VI. Anexo. Marco teórico

El algoritmo de Q-Learning fue introducido por Christopher John Cornish Hellaby Watkins en 1989 en la tesis de su doctorado titulada "Learning from delayed rewards" [4], pero este algoritmo ha ido evolucionando a lo largo del tiempo y sigue siendo vigente en la actualidad. Recientemente, en 2014, Google patentó una variación de este algoritmo llamada "Q-Learning profundo" el cual puede llegar a jugar, a los juegos de la Atari 2600, al mismo nivel que jugadores humanos profesionales ("Wikipedia Q-learning", 2020) [3]. Esto es un claro ejemplo de que Q-Learning es un algoritmo que ha sufrido varias modificaciones a lo largo del tiempo, por lo que haremos un repaso a algunas de sus variaciones, para intentar comprender un poco mejor el estado del arte de este algoritmo en la actualidad.

"Q-Learning profundo", o *"DQN"* por sus siglas en inglés, es una variación del algoritmo Q-Learning que hace uso de redes neuronales convolucionales con el objetivo de poder extraer información de imágenes, para poder luego aplicar Q-Learning ("Wikipedia Q-learning", 2020) [3].

"SARSA" es una de las versiones más famosas del algoritmo de Q-Learning. Esta variante se utiliza para suplir el problema, que existe en el Q-Learning, que consiste en que como el algoritmo tenderá a escoger solo los caminos óptimos y tendremos un total desconocimiento de que ocurre en los caminos que no son "buenos" (Aguado, 2015) [7]. El único cambió que se aplicaría a la función de actualización de Q es que, en vez de escoger la acción óptima, se escogerá la acción actual que se ha tomado para tener el estado (Agusto y Guerrero, 2009) [5] (3).

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha[r(s_{t+1}) + \gamma Q(s_{t+1}, a_{t+1})] \quad (3)$$

"DYNA - Q" es un algoritmo mezcla el aprendizaje por refuerzo directo, y el aprendizaje por refuerzo indirecto. Al usar el aprendizaje por planificación, el modelo genera recompensas ante las acciones simuladas, de forma que el algoritmo va encontrando unos valores de recompensas, pero como estos valores se están obteniendo a través de un modelo que simula la realidad, luego se usa el aprendizaje por refuerzo directo para someter al algoritmo a un escenario real y que se enfrente a problemas que no se contemplaban en el entorno simulado. Cuando se usa el aprendizaje por refuerzo directo, el algoritmo

interactúa con un entorno real para obtener recompensas y actualizar el modelo interno a uno que si es fiel a la realidad, es decir, ajusta los valores de la matriz Q, esto permite que el modelo mantenga el conocimiento obtenido anteriormente, pero, que también contemple los problemas que se puedan encontrar en la realidad. Al mezclar estos dos métodos se consigue obtener la matriz Q óptima de forma rápida y eficiente (Saury, 2019) [6]. Como función de actualización de la matriz Q se usa la siguiente (4).

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max(Q(s',a') - Q(s,a))] \quad (4)$$

"Q-Learning doble" es una variante propuesta para corregir el problema de la aplicación del algoritmo Q-Learning en entornos ruidosos. El problema se produce porque Q-Learning evalúa la futura recompensa máxima, obtenida por aplicar una acción, en la misma función Q de la cual se obtiene la política de la acción actualmente seleccionada. Esto puede provocar que el algoritmo sobreestime las recompensas que se pueden obtener al aplicar una acción en estos entornos ("Wikipedia Q-learning", 2020) [3] (5).

$$Q^A_{t+1}(s_t,a_t) \leftarrow Q^A_t(s_t,a_t) + \alpha_t(s_t,a_t)(r_t + \gamma Q^B_t(s_{t+1}, \max(Q^A_t(s_{t+a},a) - Q^A_t(s_t,a_t)))) \quad (5)$$

Este algoritmo se llego a mezclar con el algoritmo *"DQN"*, llegando a mejorar el rendimiento de este último. Al algoritmo que mezclaba estas dos técnicas se le llamó *"doble DQN"* ("Wikipedia Q-learning", 2020) [3].

Una vez que hemos repasado algunas de las variantes del algoritmo de Q-Learning, revisaremos cuáles son sus funciones en la actualidad.

Hoy en día el campo de los videojuegos es un área ideal para aplicar y realizar estudios sobre la Inteligencia Artificial. Este campo es muy popular, lo cual repercute en una inmensa cantidad de datos que analizar; una gran variedad, lo que aporta un montón de problemas diferentes que intentar abarcar, y además cada vez son más complejos, lo que hace que constantemente haya que realizar mejoras sobre los modelos existentes, para que puedan aportar soluciones más consistentes (Adonahi, 2019) [8].

Este campo necesita una gran diversidad de modelos diferentes, que abarcan ámbitos como son el *"Machine Learning"*, el *"Tree Search"* o el *"Path finding"*. Además de esto, también se incrementa la investigación sobre Inteligencia Artificial en los videojuegos, porque se ha demostrado, que al crear mejores modelos que sepan entender al jugador y personalizar parámetros como la dificultad o la forma de interactuar, en función de cada persona, se consigue una mejor inmersión y disfrute del jugador (Adonahi, 2019) [8]. Aunque por lo general los modelos que se usan están preparados para abordar un único juego, actualmente se investiga para encontrar algoritmos capaces de generalizar y aprender a afrontar diferentes problemas (Adonahi, 2019) [8].

En concreto, los algoritmos basados en Q-Learning, son los más usados en investigaciones de IA para los videojuegos, y como mencionamos antes, ya se han llegado a usar para crear Inteligencias Artificiales capaces de jugar a los juegos de la Atari 2600. Estos algoritmos se pusieron a prueba en 2014 contra jugadores profesionales reales, y se encontró que el mismo modelo, *"DQN"* pudo aprender a jugar a 46 juegos diferentes, de los cuales en 26 logró mejorar la habilidad de los jugadores reales. Además de eso, el algoritmo *"SARSA"* es muy vigente en los juegos de lucha (Adonahi, 2019) [8].

Aun así, los videojuegos no es el único campo en que se utilizan estos tipos de algoritmos, pues Q-Learning es un algoritmo que se puede usar en una gran diversidad de áreas como pueden ser el almacenamiento de energía en redes de potencia, la gestión de los semáforos que regulan el tráfico, la robótica y las finanzas.

En cuanto al almacenamiento de energía se refiere, los algoritmos de aprendizaje por refuerzo se utilizan para optimizar las técnicas que evalúan la seguridad de los sistemas de generación eléctrica. Estos algoritmos están centrados en la detección de problemas en los sistemas de control de la energía (garychl, 2018) [9].

El problema de los semáforos se abordó en el paper "Reinforcement learning-based multi-agent system for network traffic signal control" (Arel, Liu, Urbanik y Kohls, 2010), los investigadores intentaron resolver el problema de los atascos de tráfico. Para ello hicieron un entorno simulado en el cual se pusieron 5 agentes, que controlaban las señales de tráfico de una intersección de 5 carreteras, y con la aplicación de *"DQN"* se obtuvo una mejora sustancial respecto a los métodos tradicionales usados para la gestión de tráfico (garychl, 2018) [9].

La robótica también es un amplio campo en el que se puede trabajar con algoritmos de aprendizaje por refuerzo. Estos algoritmos, se suelen unir con las redes neuronales convolucionales, para que un robot pueda comprender su entorno, y en base a eso, tomar decisiones en función de un modelo obtenido, por ejemplo con Q-Learning, para que este robot sea capaz de realizar unas acciones que le otorguen una recompensa. Un ejemplo, son las manufacturas, en las que podemos ver grandes cantidades de brazos mecánicos, que se encargan de gestionar una línea de montaje de coches, como muestra la empresa de Tesla, en la que más de 160 robots hacen la mayor parte del trabajo con el objetivo de reducir los riesgos o defectos de fabricación. Otro caso, sería la gestión de inventario, pues actualmente ya existen fabricas en las que se implementan robots que siguen una cadena de suministros y son capaces de transportar paquetes a lo largo de la fábrica, con el objetivo de reducir el tiempo de espera para reponer el stock de un producto (Techlabs, 2017) [11].

En lo referente a las finanzas, los algoritmos basados en Q-Learning se usan para analizar los posibles resultados que se pueden obtener al aplicar diferentes estrategias de inversión. Esto se realiza con el fin de poder optimizar los beneficios a la hora de realizar inversiones, pues estos modelos podrán tener en cuenta diferentes factores, como pueden ser los precios del mercado o los riesgos asociados a una inversión (Techlabs, 2017) [11].

Por último, también sería interesante abordar un último tema, y es que ya que sabemos cuál es el estado del arte del aprendizaje por refuerzo, y más concretamente del algoritmo Q-Learning y sus derivados, lo que nos queda por descubrir es cuál es su futuro.

Aunque en general los algoritmos de aprendizaje por refuerzo actualmente siguen teniendo problemas que impiden su uso de forma mas fácil y generalizada, conforme vayan mejorando en los próximos años, se espera que puedan llegar a ser útiles e influyentes en diversos campos como puede ser la ayuda y trato con humanos, el entendimiento de las consecuencias obtenidas tras aplicar diferentes acciones (garychl, 2018) [9] o la creación de modelos en videojuegos que puedan llegar a cooperar de forma eficiente (Adonahi, 2019) [8].

Por ejemplo, puede llegar a ser factible, que en un futuro tratemos con asistentes virtuales que consideren cuales son las mejores acciones que pueden realizar, en consecuencia a acciones que nosotros realicemos previamente, con la finalidad de alcanzar metas comunes (garychl, 2018) [9].

En cuanto al análisis de las consecuencias de las acciones, tampoco es difícil imaginar un futuro en el que modelos de Q-Learning simulen hechos ya ocurridos, para entender qué provocan las diferentes interacciones entre los agentes involucrados. Por ejemplo, situándonos en el ámbito del fútbol, podríamos lograr entender de que forma influye una estrategia escogida por el equipo para afrontar el partido, y como esto afecta al resultado final (garychl, 2018) [9].

Por último, en lo referente a la creación de modelos capaces de cooperar de forma eficiente, hay que explicar que, en la actualidad, aunque existen diferentes modelos de Inteligencia Artificial que ya pueden llegar a comportarse de forma bastante realista, lo que no se ha conseguido todavía es crear una inteligencia que pueda llegar a cooperar entre varios agentes para enfrentarse en equipo contra el jugador, realizando diferentes estrategias para intentar sobreponerse a su rival. Por otro lado, el ámbito de la cooperación con el jugador todavía no es todo lo bueno que se desearía, por lo que en el futuro se espera que estos modelos basados en Q-Learning sean capaces de solucionar estos problemas y proporcionar una mejor experiencia a los jugadores (Adonahi, 2019) [8].

En conclusión, hemos realizado un recorrido por el estado actual del mundo del aprendizaje por refuerzo, concretamente por los algoritmos de Q-Learning y sus variantes, adicionalmente, hemos conocido cuáles son sus aplicaciones reales actualmente, y nos hemos aventurado a intentar comprender cuál será el futuro que les depara.

Referencias

[1] Caparrini, F. (2019). Aprendizaje Supervisado y No Supervisado - Fernando Sancho Caparrini. Retrieved 7 June 2020, from http://www.cs.us.es/ fsancho/?e=77

[2] Caparrini, F. (2019). Aprendizaje por refuerzo: algoritmo Q Learning - Fernando Sancho Caparrini. Retrieved 7 June 2020, from http://www.cs.us.es/ fsancho/?e=109

[3] Q-learning. (2020). Retrieved 7 June 2020, from https://es.wikipedia.org/wiki/Q-learning

[4] Cornish Hellaby Watkins, C. (1989). Learning from delayed rewards [Ebook] (p. 241). Cambridge. Retrieved 7 June 2020, from http://www.cs.rhul.ac.uk/ chrisw/new_thesis.pdf

[5] Agusto, H., & Guerrero, P. (2009). Retrieved 7 June 2020, from shorturl.at/bsDI1

[6] Saury, M. (2019). Planificación de trayectorias de sistemas multi-robot en entornos desconocidos. Retrieved 7 June 2020, from http://zaguan.unizar.es/record/85911/files/TAZ-TFG-2019-085.pdf

[7] Aguado, G. (2015). Aplicación de técnicas de aprendizaje automático sobre juegos. Retrieved 7 June 2020, from shorturl.at/chp24

[8] Adonahi, A. (2019). Reinforcement Learning en los videojuegos. Retrieved 7 June 2020, from shorturl.at/FHP48

[9] garychl. (2018). Applications of Reinforcement Learning in Real World. Towards Data Science. Retrieved 8 June 2020, from https://towardsdatascience.com/applications-of-reinforcement-learning-in-real-world-1a94955bcd12

[10] Arel, Liu, Urbanik, & Kohls. (2010). Reinforcement learning-based multi-agent system for network traffic signal control. Retrieved 8 June 2020, from http://web.eecs.utk.edu/ ielhanan/Papers/IET_ITS_2010.pdf

[11] Techlabs, M. (2017). Reinforcement Learning and Its Practical Applications. Retrieved 8 June 2020, from https://chatbotsmagazine.com/reinforcement-learning-and-its-practical-applications-8499e60cf751

[12] Suárez, J. (2020). Entorno de pruebas y demostración de algoritmos de IA para juegos y simulaciones. Retrieved 16 June 2020, from https://upcommons.upc.edu/bitstream/handle/2117/117491/126534.pdf?sequence=1&