Resolución de Path-Finding mediante el algoritmo Q-learning

Andrés Aguilar Alhama
Dpto. Ciencias de la Computación e I. Artificial
Universidad de Sevilla
Sevilla, España
andagualh@alum.us.es

Iván Hernández Rodríguez
Dpto. Ciencias de la Computación e I. Artificial
Universidad de Sevilla
Sevilla, España
ivaherrod1@alum.us.es

***Resumen*—En este proyecto se presentan los resultados obtenidos al aplicar el algoritmo de aprendizaje por refuerzo Q-learning a diferentes ejemplos usando el lenguaje de programación Python y haciendo uso de las librerías pertinentes. También se estudia como afectan los distintos parámetros que se le pasan al algoritmo en la ejecución del mismo y como cambian los resultados dependiendo del valor de estos.**

Finalmente se añade como anexo un marco teórico sobre Aprendizaje por Refuerzo, así como también de Q-learning y sus variantes y las distintas aplicaciones de este en la vida real. Asimismo hablaremos de algunos estudios en los que se puede ver la importancia de este algoritmo.

***Palabras clave*—Inteligencia Artificial, aprendizaje por refuerzo, q-learning, tablero, dimensiones, constante de aprendizaje, agente, cerebro, matriz Q, ratio de exploración.**

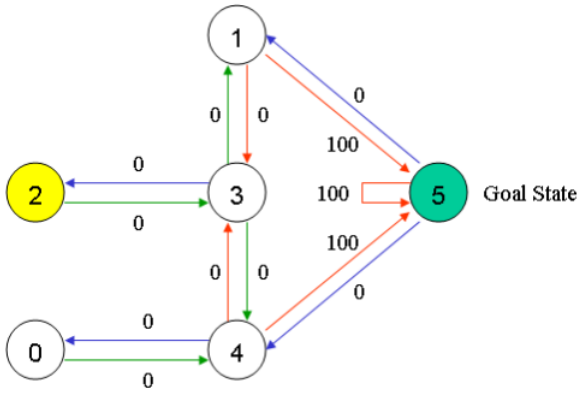


Figura 1. Representación del ejemplo sobre habitaciones y puertas.

I. Introducción

El algoritmo Q-learning es una técnica de aprendizaje por refuerzo cuyo objetivo es aprender ciertos matices para que en un futuro sepa qué acción tomar bajo unas circunstancias. Es por esto que el algoritmo Q-learning se suele usar como un algoritmo de *path-finding*, ya que recorre los distintos caminos y es capaz de reconocer cual de ellos le conviene más. Los elementos que precisamos para hacer esto posible es una matriz que defina los diferentes estados y otra que defina la recompensas de los mismos. También precisaremos de un estado inicial y un estado objetivo. Otra representación posible sería con grafos, donde los nodos representan los diferentes estados y las aristas las posibles acciones.

Para familiarizarnos con el algoritmo se ha indagado en el ejemplo propuesto [1], en el cual se propone una casa con 5 habitaciones conectadas por puertas y donde nuestro objetivo será ir desde la habitación 0 a la 5 por el camino que mejor recompensa otorgue. En él los estados vienen representados por cada habitación, mientras que las acciones serán los posibles movimientos del agente de una habitación a otra. Podemos observar una representación en grafo de dicho problema en la figura 1.

Una vez controlamos los diferentes aspectos técnicos del problema, se intento aplicar el algoritmo a distintos problemas, aumentando la complejidad en cada uno de ellos. Finalmente, se diseñó un algoritmo que es flexible a la hora de imponerle restricciones, tales como trampas que tendrá que aprender a esquivar, transiciones inválidas que no podrán darse en el recorrido o casillas con recompensas intermedias. De esta forma el algoritmo tendrá que aprender a tratar con las diferentes casillas.

Para introducir los datos se ha trabajado en una interfaz gráfica que trabaja sobre la librería Tkinter del lenguaje Python. Los datos que se introducen se someten a una posterior validación, lo cual se explica convenientemente en la misma interfaz, la cual indica que hay que introducir en cada campo, y en caso de que sea más complejo que un número también indica el formato. Otras librerías de Python usadas son numpy, random, matplotlib, Python RegEx y math, para controlar diversas operaciones matemáticas, la generación de números aleatorios y las validaciones que requieren expresiones regulares.

Por lo tanto, el documento queda estructurado en secciones donde se detallarán los distintos aspectos del proyecto. En primer lugar, se definen unas preliminares para la resolución, la metodología seguida para la resolución del problema en sus diferentes fases, los resultados de los experimentos realizados, las conclusiones de los mismos y finalmente un marco teórico

en el que se enmarcan el aprendizaje por refuerzo, el algoritmo Q-learning y sus variantes, y las aplicaciones de este en la vida real.

II. Preliminares

En esta sección explicaremos los distintos métodos que se han usado para resolver el problema, así como el diseño ideado. También detallaremos algunos trabajos relacionados que resuelven ciertos problemas usando la misma técnica.

II-A. Métodos empleados

Para aplicar el algoritmo Q-Learning ha sido necesario implementar una búsqueda en espacio de estados, teniendo así en cuenta el estado inicial y el estado final; en las fases más complejas del experimento también se ha de tener en cuenta las casillas especiales por las que se puede pasar así como recortar los posibles vecinos para establecer transiciones imposibles.

El objeto a recorrer es una matriz de unas dimensiones dadas por el usuario, para ello se ha hecho uso de las clases SimpleTableInput y Framegeneration, una solución proporcionada por Bryan Oakley [2] que crean una ventana de interfaz de usuario con la librería TkInter para introducir una matriz de las dimensiones dadas y devuelven un array que contiene los valores de cada fila de la matriz. A estas clases se han realizado diversos cambios, la mayoría de ellos en pos de validar los elementos introducidos en la matriz.

Posteriormente, se introducen los valores de los parámetros en una ventana de interfaz de usuario creada con soporte de la librería TkInter que se usarán: Número de Episodios, Estado Inicial, Estado Objetivo y γ; en fases más complejas se añade el ratio de exploración "ϵ", el ratio de caída "α", restricciones a las acciones posible, casillas de recompensa y casillas de penalización. Estos parámetros son validados de dos formas: En tiempo real o en transiciones. Las validaciones en tiempo real son aquellas que validan mientras se introduce el valor [21], mientras que el otro tipo corresponde a las validaciones que se realizan tras pulsar un botón.

Dada la matriz a recorrer y los parámetros, se forma a partir de ella una matriz de recompensa que contiene las acciones posibles para cada estado, que se dividen en filas para estos últimos, y las recompensas o penalizaciones si las hubiera. Para generar correctamente una matriz de recompensa con las acciones posibles es necesario realizar una búsqueda de vecinos en la matriz de entrada [6]. Nuestro método para la generación de la matriz de recompensa se basa en la solución propuesta por Surya K. [3].

Tras la introducción de la matriz y la introducción de los parámetros pasamos a la ejecución de la fase elegida según la opción escogida:

- Para la fase 1, hacemos la búsqueda de vecinos y la generación de la matriz de recompensa tal y como se ha descrito. Para esta fase sólo requerimos los parámetros Estado Inicial, Episodios, Estado Objetivo y Gamma.
- Para la fase 2, seguimos una dinámica similar a la fase 1, la principal diferencia es asignar los parámetros del ratio de exploración ϵ y el ratio de caída α
- La fase 3 está dividida en la fase 3-1, para funcionar sólo con los parámetros de la fase 1, y la fase 3-2 para funcionar con los parámetros de la fase 2. La fase 3 añade las casillas de recompensa, las casillas de penalización y las restricciones de acción.

II-B. Trabajo Relacionado

Con el objetivo de familiarizarnos con el aprendizaje por refuerzo, más concretamente con el algoritmo Q-learning, hemos buscado apoyo en otros problemas o problemas similares donde se aplica este algoritmo:

- Tutorial de Path-Finding mediante Q-Learning [1]
- Ejemplo de Path-Finding mediante Q-Learning en Java [4]
- Ejemplo de Path-Finding mediante Q-Learning ϵ-greedy [5]

III. Metodología

Esta sección esta dividida en las siguientes categorías:

- Clases Auxiliares de generación de Matriz
- Métodos Auxiliares del Backend
- Métodos Auxiliares del Frontend
- Métodos de Recompensa
- Algoritmos principales
- Backend de la Aplicación Principal
- Frontend de la Aplicación Principal

1. Comenzamos con las **clases auxiliares** que se encargan del frontend de la entrada de matriz. Las clases empleadas para esta labor con **SimpleTableInput y Framegeneration** como se menciona en el apartado de métodos empleados. A continuación se describirán los métodos que se emplean en estas clases:
   - **SimpleTableInput, init**: Inicializa la clase y obtiene los valores de filas, columnas y la ventana de TkInter. El código establece el código de validación para los campos de Entry y crea un doble bucle con un índice para crear filas*columnas Entries y añadirlas a un objeto de Tkinter de tipo Grid. Posteriormente se configura el Grid usando las columnas para que estén bien dispuestas en la interfaz de usuario. También se establecen aquí las instrucciones para introducir la matriz.
   - **SimpleTableInput, get**: Recorre los valores de los objetos que se han introducido en los entries y se añaden a una lista de arrays. Se ha añadido aquí la validación para evitar introducir valores repetidos o campos vacíos, recorriendo la lista de arrays con un doble bucle para obtener los valores en un sólo array y posteriormente usando el método unique de la librería numpy, que obtiene una lista con los números que son únicos en el array creado anteriormente. Posteriormente comprobamos que la longitud de

esa lista sea igual al valor filas*columnas, en caso de que no lo sea el método devuelve False. Si la validación es superada, el método devuelve una lista de arrays con los valores de la matriz introducida en el orden que haya preferido el usuario.

- **SimpleTableInput, validate**: Es el método de validación en tiempo real para las Entries de TkInter, comprueba que sólo se introducen números enteros o valores vacíos en los campos Entry. En caso de que se intente introducir un valor con decimales, una letra o un número entero superior a filas*columnas, la interfaz no lo permitirá y emitirá un sonido para indicarlo.
- **Framegeneration, init**: Se encarga de crear una tabla para alojar el Grid proporcionado por la clase SimpleTableInput, así como un botón de submit. Este método también crea el frame de la ventana.
- **Framegeneration, onsubmit**: Este método describe que ocurrirá al pulsar el botón de Submit de la interfaz a la hora de introducir la matriz. En caso de que la validación de SimpleTableInput.get no se haya superado, invocará al método de Qlearn "**errorWindow**", también parará la ejecución del código posterior. Si la validación ha sido superada llamará la método de Frontend "Qlearn.parguiçon la matriz de entrada y el objeto de la ventana.

2. Continuamos con los **métodos auxiliares del Backend** de la aplicación. Estos son los métodos auxiliares que controlan operaciones que no son vistas por el usuario final. Añadir que a partir de este punto todos los métodos pertenecerán a la clase **Qlearn**:

   - **stringToArrayInt**: Recibe un string y una ventana de TkInter. El método se encarga de convertir en array de valores enteros el string recibido con el formato adecuado. También se realiza aquí la validación de números repetidos, en caso de existir un número repetido se llama al método "**errorWindow**çon la ventana que recibimos en la entrada del método. El método devuelve un array de números enteros o un boolean False en caso de fallar la validación.
   - **applyRestrictions**: Recibe un array de estilo tupla que contiene los estados del tablero y sus vecinos, un string de un formato dado (Estado-Estado) y una ventana de TkInter. El método desglosa el string acorde al formato dado para convertirlo en una lista de arrays que indican que dos estados vecinos se han restringido, como si se hubiese añadido un muro entre ellos. También realizamos aquí la validación que comprueba si un valor de los introducidos en el string no puede estar contenido en un tablero de las dimensiones dadas. Si no se supera la validación se llama al método "**errorWindow**çon la ventana recibida y se devuelve un False. Si se supera la validación se modifica el array de vecinos con las restricciones indicadas y se devuelve como salida del método.
   - **find neighbours**: Este método recibe una lista de arrays que contiene la matriz que forma el tablero. El primer doble bucle se encarga de obtener los valores de la matriz que están en los movimientos permitidos: Arriba, Abajo, Izquierda, Derecha y las esquinas que limitan con el estado que se está evaluando. Finalmente se genera un array de estilo tupla formada por primero el estado y después sus vecinos de los movimientos permitidos. Este método esta basado en una solución ideada por maxrudometkin [6] y modificado para añadir nuevos movimientos posibles y cambiar la salida. En la figura 2 podemos ver el pseudo-código, en el cual se observa el flujo para elaborar la tupla con la relación de vecinos, la entrada arr es la matriz tablero.

find neighbours(arr)
Entrada: arr
Salida: final neighbors
$finalneighbors \leftarrow$ **Inicializar array**
Desde $i \leftarrow$ **hasta longitud lista** arr
  Desde $j \leftarrow$ **hasta longitud de** $arr[i]$
    $n \leftarrow$ **Inicializar array**
    $lon \leftarrow$ **Longitud arr[i]-1**
    $len \leftarrow$ **Longitud arr**
    Si $i = 0$ **o** $i = len$ **o** $j = 0$ **o** $j = lon$
      Si $i \neq 0$
        $n \leftarrow arr[i-1][j]$ **Casilla Superior**
      Si $j \neq lon$
        $n \leftarrow arr[i][j+1]$ **Casilla Derecha**
      Si $i \neq lon$
        $n \leftarrow arr[i+1][j]$ **Casilla Inferior**
      Si $j \neq 0$
        $n \leftarrow arr[i][j-1]$ **Casilla Izquierda**
      Si $i \neq 0$ **y** $j \neq lon$
        $n \leftarrow arr[i-1][j+1]$ **Diagonal Sup. Der.**
      Si $i \neq 0$ **y** $j \neq 0$
        $n \leftarrow arr[i-1][j-1]$ **Diagonal Sup. Izq.**
      Si $i \neq lon$ **y** $j \neq lon$
        $n \leftarrow arr[i+1][j+1]$ **Diagonal Inf. Der.**
      Si $i \neq lon$ **y** $j \neq 0$
        $n \leftarrow arr[i+1][j-1]$ **Diagonal Inf. Izq.**
    En cualquier otro caso
      Introducir en $n \leftarrow arr[i-1][j]$ **C.Sup.**
      Introducir en $n \leftarrow arr[i][j+1]$ **C.Der.**
      Introducir en $n \leftarrow arr[i+1][j]$ **C.Inf.**
      Introducir en $n \leftarrow arr[i][j-1]$ **C.Izq.**
      Introducir en $n \leftarrow arr[i-1][j-1]$ **Di.Sup.Izq.**
      Introducir en $n \leftarrow arr[i-1][j+1]$ **Di.Sup.Der.**
      Introducir en $n \leftarrow arr[i+1][j-1]$ **Di.Inf.Izq.**
      Introducir en $n \leftarrow arr[i+][j+1]$ **Di.Inf.Der.**
  $finalneighbors \leftarrow$ **Tupla[Value, n]**
Devolver $finalneighbors$

Figura 2. Búsqueda de Vecinos en el tablero

- **rendimiento**: Este método calcula una medida del rendimiento empleando la suma de todos los valores de la matriz, dividiéndolos entre el más alto y multiplicando por 100 para obtenerlo en forma de porcentaje.
- **maximum**: Dado un estado, un array con las acciones posibles para este estado y la matriz q; evalúa que valor de Q para las acciones dadas es mayor y lo devuelve.
- **getAction**: Dado un estado, los estados máximos posibles y la matriz recompensa, obtiene las acciones que son posibles realizar acorde a la matriz recompensa, aquellas que en la matriz recompensa no tengan un valor de -100, en la Fase 3, o -1, en la Fase 1 y 2; que indica que esa casilla está fuera del rango de movimientos.
- **normalize** : Normaliza la matriz Q dada usando el valor máximo que contenga y redondea los valores resultantes a su techo.

3. Los siguientes métodos se refieren a los **métodos auxiliares del Frontend**, aquellos métodos que influyen sobre lo que el usuario final ve:
   - **passToFase1**: Controla la validación de los valores necesarios para realizar la Fase 1. En caso de detectar un valor no válido llama al método "**errorWindow**". Este método recibe los valores de Episodios, Objetivo, Gamma y Estado Inicial. Da paso al método que gestiona la realización de la fase 1.
   - **passToFase2**: Controla la validación de los valores necesarios para realizar la Fase 2. En caso de detectar un valor no válido llama al método "**errorWindow**". Este método recibe los mismos valores que "**passToFase1**"pero añadiendo Epsilon y Alfa. Da paso a método que gestiona la realización de la fase 2.
   - **passToFase3**: Controla la validación de los valores necesarios para realizar la Fase 3. Muy similar a los dos descritos anteriormente pero añade la validación mediante expresiones regulares para los strings que contienen las restricciones, las casillas de recompensa y las casillas de penalización. Da paso al método que gestiona la realización de la fase 3.
   - **passToFase32**: Controla la validación de los valores necesarios para realizar la Fase 3-2. Es igual al método anterior pero también valida los valores de Epsilon y Alfa. Da paso al método que gestiona la realización de la fase 3-2.
   - **validateint**: Método que valida en tiempo real los números enteros introducidos en Entries de TkInter. Sólo permite escribir números enteros y dejar el campo vacío.
   - **validatefloat**: Método que valida en tiempo real los números con decimales introducidos en Entries de TkInter. Sólo permite escribir números en un rango de 0 a 1 y dejar el campo vacío.
   - **render**: Muestra la matriz Q dispuesta con filas que representan estados y columnas que representan acciones.
   - **shortestpath**: Dadas una matriz q, un estado inicial y un estado final, se devuelve el camino más corto según los valores de la matriz q.
   - **errorWindow**: Muestra un mensaje de error al usuario indicándole que ha introducido valores incorrectos.
4. A continuación se definen los dos métodos que se encargan de la **generación de la matriz reward** a partir de una matriz tablero dada:
   - **reward**: Genera una matriz reward basada en la casilla objetivo, las dimensiones de la matriz tablero

y la relación de estado-vecinos. En el siguiente pseudo-código podemos observar que flujo sigue el método, donde $goalstate$ es el estado objetivo, $maxstates$ es el número de estados máximos de la matriz tablero, $maxactions$ es el número posible de acciones que se pueden realizar en el tablero y $neighbors$ es la relación de vecinos contenida en un objeto estilo tupla. Podemos ver el pseudo-código del método en la figura 3.

reward$(goalstate, states, actions, neighbors)$
Entrada: goal state, max states, max actions, neighbors
Salida: matriz[max states, max actions]
1 $r \leftarrow$ **Crear una matriz(max states, max actions)**
2 $r \leftarrow$ **r * -1**
3 **Para los todos los elementos** row **de** $neighbors$
4 $\quad r[row] \leftarrow 0$
5 $\quad r[row] \leftarrow r[row :: -1]$
6 $\quad r[row] \leftarrow 0$
7 **Desde** $c \leftarrow$ **hasta** $states$
8 $\quad$ **Desde** $c \leftarrow$ **hasta** $actions$
9 $\quad\quad$ **si** c **es igual a** $goalstate$ **y** $r[(c, c2)] \leq -1$
10 $\quad\quad\quad r[(c, c2)] \leftarrow 100$
11 $r[(goalstate), (goalstate)] \leftarrow 100$
12 **Devolver** r

Figura 3. Generación de Matriz Recompensa

- **reward2**: Método similar al anterior pero que también recibe un array con las casillas de recompensa y de penalización. Para ello sólo es necesario recorrer esos arrays y asignar los valores de bonificación y penalización cuando sea correspondiente tal y como se hace para asignar las acciones objetivo en el método. El valor de recompensa asignado a las casillas de penalización es -50 y el valor de recompensa asignado a las casillas de bonificación es de +10 "**reward**".

5. Aquí hablaremos sobre los **algoritmos principales** que definen el comportamiento del código, existen dos métodos que implementan algoritmos Q-learning similares:
   - **evaluate**: Parte del algoritmo de entrenamiento que va iterando por distintos episodios, en cada episodio el agente comenzará desde un estado inicial aleatorio y decidirá un estado alcanzable al azar como su siguiente estado. Posteriormente aplicará la siguiente formula para calcular el valor que el cerebro matriz Q almacenará para la acción escogida:

$$Q(s, a) = R(s, a) + \gamma * Max[Q(a, aa)] \quad (1)$$

Para obtener el valor máximo que el siguiente estado tiene almacenado en Q se emplea el método auxiliar "**maximum**". $R(s, a)$ se refiere al valor que la matriz recompensa tiene almacenado para el estado actual y la acción correspondiente al siguiente estado propuesto. En el siguiente pseudo-código observamos el procedimiento realizado, dónde q es la Matriz Q cerebro, cs es Estado Actual, st es el número máximo de estados, ac es el número máximo de acciones, γ es el ratio de futuro y r es la matriz recompensa. Podemos observar el método en pseudo-código en la figura 4.

evaluate$(q, cs, st, ac, \gamma, r)$
Entrada: q, cs, st, ac, γ, r
Salida: q actualizada, cs
1 $pa \leftarrow$**Acciones posibles para** cs
2 $ns \leftarrow$ **Elemento aleatorio de** ps
3 $fs \leftarrow$ **Acciones posibles para** ns
4 $learn \leftarrow \gamma * Max[Q(ns, fs)]$
5 $q[(cs, ns)] \leftarrow r[(cs, ns)] + \gamma * learn$
6 $cs \leftarrow ns$

Figura 4. Algoritmo Principal Evaluate

- **train**: Comenzamos inicializando objetos de medición del algoritmo: El tiempo y los puntos de rendimiento. Se itera por episodios y se selecciona un estado inicial al azar. Mientras el estado en el que nos encontramos no sea el final, se ejecuta la segunda parte del algoritmo "**evaluate**". Este método nos devuelve el cerebro actualizado y el próximo estado. Cuando hemos llegado al estado final, realizamos una iteración extra para asegurar una convergencia correcta. Una vez acabada esta última iteración se vuelve a establecer como estado actual un estado inicial aleatorio y se comienza un nuevo episodio. Se realizará el número de episodios que haya sido especificado en los parámetros recibidos. Cada número de episodios múltiplo de 5, se imprimirá en consola la matriz Q sin normalizar y se calculará la medida de rendimiento mediante el método: "**rendimiento**". A continuación observamos el pseudocódigo que describe el funcionamiento del algoritmo, dónde q es la Matriz Cerebro Q, st es el número de estados máximos, ac es el número de acciones máximas, ep es el número de episodios, is es el estado inicial, gs es el estado objetivo, γ es el ratio de aprendizaje y r es la matriz recompensa. Se han excluido los elementos de medición en el pseudo-código (figura 5) para aumentar la legibilidad.

train$(q, st, ac, ep, is, gs, \gamma, r)$
Entrada: q, st,ac, ep,is,gs, γ, r
Salida: q actualizada
Para los todos los i **del rango** ep
    $cs \leftarrow$ **estado aleatorio**
        Mientras $cs \neq gs$
            $q, cs \leftarrow$ **ejecutar evaluate**
        Si $cs = gs$
            $q, cs \leftarrow$ **ejecutar evaluate**

Figura 5. Algoritmo Principal Train

- **train2**: Algoritmo similar al anterior, pero introduce el ratio de exploración ϵ y el ratio de aprendizaje α. La adición de ϵ trata de aportar un componente de aleatoriedad a la hora de explorar el tablero. La selección entre política de exploración y política de camino óptimo depende del valor de ϵ y α, siendo esta última el componente que reduce el valor de ϵ para converger hacia una política de camino óptimo como la que se sigue en "**train**". En este algoritmo también se imprimen en consola el rendimiento y la matriz Q en cada episodio múltiplo de 5. En el siguiente pseudocódigo observamos el flujo del algoritmo, donde q es la matriz cerebro Q, st es el número máximo de estados, ac es el número máximo de acciones, ep es el número de episodios a ejecutar, is es el estado inicial del agente, gs es el estado objetivo, γ es el ratio de aprendizaje, r es la matriz de recompensas, α es el ratio de caída para ϵ y ϵ es el ratio de exploración. Se han excluido los elementos de medición del flujo del pseudo-código (figura 6) para mejorar la legibilidad.

train2$(q, st, ac, ep, is, gs, \gamma, r, \alpha, \epsilon)$
Entrada: q, st,ac, ep,is,gs, γ, r,α, ϵ
Salida: q actualizada
$pa \leftarrow$ **Crear array vacío**
Para los todos los i **del rango** ep
    $cs \leftarrow is$
    $ep\epsilon \leftarrow \epsilon$
        Mientras $cs \neq gs$
            $q, cs, ep\epsilon \leftarrow$**ejecutar evaluateGreedy**
        Si $cs = gs$
            $q, cs, ep\epsilon \leftarrow$**ejecutar evaluateGreedy**

Figura 6. Algoritmo principal train2

- **evaluateGreedy**: Al igual que el método "**evaluate**"de la fase 1, se encarga de realizar la evaluación de movimientos que puede tomar el agente. La principal diferencia es la presencia de la política ϵ-greedy. Dicha política determina con el valor de ϵ si se debe explorar de forma aleatoria o se debe evaluar usando el valor máximo almacenado en la matriz Q cerebro. En el siguiente pseudo-código de la figura 7, podemos observar su funcionamiento, dónde q es la Matriz Q cerebro, cs es el Estado Actual, st son los Estados Máximos posibles, ac son las Acciones Máximas posibles, γ es el ratio de futuro, ϵ es el ratio de exploración y α es el ratio de aprendizaje:

evaluateGreedy$(q, cs, st, ac, \gamma, r, \epsilon, \alpha)$
Entrada: q, cs, st, ac, γ, r, ϵ,α
Salida: q actualizada, cs, ϵ
$\epsilon index \leftarrow$**Valor Aleatorio rango[0,1]**
$pa \leftarrow$**Acciones posibles para** cs
$ns \leftarrow$ **Elemento aleatorio de** ps
$fs \leftarrow$ **Acciones posibles para** ns
$learn \leftarrow$ **Inicializar entero**
Si $\epsilon index < ep\epsilon$
    $ranfs \leftarrow$**Valor de fs aleatorio**
    $learn \leftarrow q[ns][ranfs]$
En cualquier otro caso
    $learn \leftarrow \gamma * Max[Q(ns, fs)]$
$q[(cs, ns)] \leftarrow r[(cs, ns)] + \gamma * learn$
$cs \leftarrow ns$
$\epsilon \leftarrow \epsilon * \alpha$

Figura 7. Algoritmo Principal EvaluateGreedy

6. Los métodos de **fase backend** son aquellos que reciben los datos de la interfaz de usuario y los procesa como variables de Python.
    - **fase1**: Procesa los datos de las Entry de TkInter al tipo adecuado, obtiene la relación de vecinos usando el método "**find neighbours**", crea la matriz Q con las dimensiones (estados,acciones), obtiene la matriz de recompensa mediante el método "**reward**z llama al algoritmo principal. Al final del proceso hace la llamada al método que normaliza la matriz q: "**normalize**z crea una ventana de TkInter con el estado final de la matriz Q.
    - **fase2**: Realiza una labor similar a "**fase1**"pero también procesa los ratios epsilon y alfa.
    - **fase3**: Es algo más complejo que los anteriores, este método es igual a **fase1** pero valida que los datos introducidos en las restricciones, casillas de recompensa y casillas de penalización no estén vacíos. En caso de que se haya introducido una casilla que no puede estar presente en dichos datos, los métodos auxiliares correspondientes devuelven un boolean False y activan la condición if que para la ejecución del método.
    - **fase32**: Al igual que "**fase3**", procesa los datos y valida los mismos parámetros. La diferencia es que al igual que "**fase2**.[es] que procesa los ratios epsilon y alfa.
7. Por último tenemos los **métodos frontend**, son aquellos que generan las ventanas de la interfaz de usuario dónde se introducen los parámetros:

- "**pargui**": Se encarga de generar la ventana de TkInter dónde se introducen los parámetros Episodios, estado inicial, estado objetivo, gamma, epsilon, alfa, restricciones, casillas de recompensa y casillas de penalización. Tiene un botón para ejecutar cada fase, que funcionarán siempre y cuando se introduzcan los parámetros que requiere cada fase.
- "**magui**": Genera la ventana donde se introduce el frame de Tkinter generado por las clases "**SimpleTableInput**z "**Framegeneration**".
- "**startgui**": Genera la ventana donde se introducen las dimensiones de la matriz tablero.

IV. Resultados

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

Las ejecuciones de la batería de experimentación se han realizado sobre un equipo con una CPU Intel® Core™ i5-8300H, frecuencia base de reloj 2,30GHz y frecuencia turbo de reloj 4,00GHz; la GPU empleada es la HD Intel® 630, con unas frecuencias base y turbo de 350MHz y 1.00GHz respectivamente. El equipo cuenta con 8GB de memoria RAM DDR4.

Destacar que los tiempos medidos en este apartado se miden desde el comienzo de la ejecución del algoritmo principal hasta el final de este, los algoritmos principales serán los mostrados en la Figura 5 o Figura 6 dependiendo de la operación escogida. En caso de no referirse a este tiempo se especificará. Encontramos la justificación para medir sólo el tiempo del algoritmo debido a la presencia de la interfaz gráfica, la cual podría variar el tiempo dependiendo de lo rápido que el usuario introduzca los datos de forma correcta.

El contexto de la experimentación realizada supone un tablero de un juego de mesa básico. Se han realizado experimentos en 3 fases atendiendo a la complejidad y parámetros añadidos:

- Fase 1: Es la fase más básica y simple, el tablero de juego es aquel proporcionado en el documento de la propuesta de trabajo. Los movimientos que puede el agente realizar es hacia sus casillas anexas sin limitaciones, incluyendo las diagonales. El agente no podrá saltar casillas de ninguna manera. La casilla de inicio es 0 y la casilla que marca el final es 6, la **recompensa por alcanzar el objetivo es 100**. En esta fase de la experimentación queremos evaluar el impacto del ratio de aprendizaje γ sobre el comportamiento del agente, también buscamos observar aquí el impacto de tiempo que tiene el número de episodios sobre el tiempo.
- Fase 2: Una vez evaluado el impacto de la fase γ**,** introducimos el segundo algoritmo que tiene en cuenta el ratio de exploración ϵ y el ratio de caída α. Este nuevo algoritmo tiene implementada una política ϵ**-greedy**, al final de cada movimiento el valor de ϵ se multiplica por α reduciendo su valor. Con estos dos parámetros, queremos hacer que el agente sea capaz de explorar otros caminos alternativos al más corto.
- Fase 3: Con un nuevo tablero de mayores dimensiones, queremos intentar introducir casillas que afecten al resultado final del juego. Estas casillas serán la de **Boost**, que **suma 10 puntos**, y la de **Trampa**, que **resta 50 puntos**. También se introducen transiciones prohibidas por las que el agente no podrá realizar algunos movimientos. Con esta fase de la experimentación pretendemos evaluar si nuestro agente es capaz de determinar como obtener una mayor puntuación final.

En todas las fases se tiene como objetivo encontrar el camino más óptimo del tablero en el menor tiempo posible. En los experimentos vamos a centrarnos en observar como, al cambiar los valores de los distintos parámetros (nº episodios, gamma, epsilon...) o las dimensiones del tablero, se altera el tiempo de ejecución del algoritmo. También observaremos como al modificar dichos parámetros cambia la forma de trabajar del algoritmo, produciéndose cambios en el rendimiento total e incluso en los caminos óptimos elegidos.

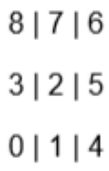


Figura 8. Tablero para fase 1 y fase 2

Para el tablero del ejemplo inicial, el cual podemos observar en la figura 8, comprobamos que al introducirlo nos devuelve una matriz Q correcta con el camino más corto. Tal y como observamos en la figura 9, la cual viene dada por la matriz Q que devuelve el tablero del ejemplo, podemos observar que elige el camino más corto que es, en este caso, el más óptimo al carecer de recompensas intermedias.Es por esto que si nos situamos en el estado 0, siguiendo los valores más altos en la matriz Q, el camino más óptimo será [0, 2, 6], tal y como devuelve el algoritmo. Si ignoráramos la acción 2, observamos que tanto la acción 1 como la 3 representan caminos igual de óptimos, ya que tienen el mismo valor en la matriz Q. Por lo tanto si eligiéramos una de estas dos acciones, los caminos alternativos serían [0,1,5,6], [0,1,2,6], [0,3,7,6] ó [0,3,2,6].

+		0	1	2	3	4	5	6	7	8
0	|	0	65	80	65	0	0	0	0	0
1	|	65	0	80	64	65	80	0	0	0
2	|	65	65	0	64	65	80	100	80	65
3	|	64	65	80	0	0	0	0	80	64
4	|	0	65	80	0	0	80	0	0	0
5	|	0	65	80	0	65	0	100	80	0
6	|	0	0	80	0	0	80	100	80	0
7	|	0	0	80	64	0	80	100	0	65
8	|	0	0	80	64	0	0	0	80	0

Camino más corto: [0, 2, 6]

Figura 9. Matriz Q para el ejemplo

Para la fase 1 nos hemos centrado en modificar el nº de episodios, el factor gamma y las dimensiones del tablero intro-

Episodios	Gamma	Dimensiones	Tiempo (ms)
50	0.8	3x3	70.41
100	0.8	3x3	149.64
250	0.8	3x3	263.39
500	0.8	3x3	458.36
1000	0.8	3x3	876.21
2500	0.8	3x3	2015.35
5000	0.8	3x3	3928.11
5000	0.8	5x5	50127.38

Tabla I
FASE 1 CON GAMMA ALTO

Episodios	Gamma	Dimensiones	Tiempo (ms)
50	0.2	3x3	77.75
100	0.2	3x3	114.82
250	0.2	3x3	193.33
500	0.2	3x3	508.20
1000	0.2	3x3	996.02
2500	0.2	3x3	2294.62
5000	0.2	3x3	4234
5000	0.2	5x5	47670

Tabla II
FASE 1 CON GAMMA BAJO

ducido. De esta forma podemos observar independientemente como afectan los distintos valores a la hora de ejecutar el algoritmo.

En primer lugar, tal y como podemos observar claramente en la tabla 1, existe una relación proporcional entre el nº de episodios y el tiempo, de forma que al aumentar el nº de episodios lo hace también el tiempo de ejecución del algoritmo. Por otra parte observamos que al aumentar las dimensiones del tablero, se produce un aumento exponencial en el tiempo de ejecución, dado que tanto la matriz de recompensas como la matriz Q aumentarán su tamaño considerablemente, provocando que los bucles del algoritmo tarden más en ejecutarse y buscar el camino óptimo.

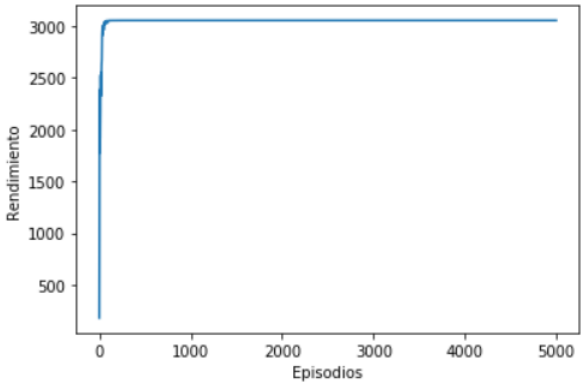


Figura 10. Gráfica de rendimiento total en fase 1 con gamma 0.8 para 5000 episodios

Respecto al rendimiento, podemos observar en la figura 10 y la figura 11 como, al realizarse los primeros episodios (normalmente suele ocurrir en los primeros 200 episodios, dependiendo de las dimensiones del tablero) alcanza rápidamente el rendimiento máximo, manteniéndose este constante en el resto de episodios una vez alcanzado. En la figura 11 se observa más detalladamente el ascenso del rendimiento en los primeros 50 episodios, manteniéndose ya constante para el resto de episodios una vez alcanzado el máximo.

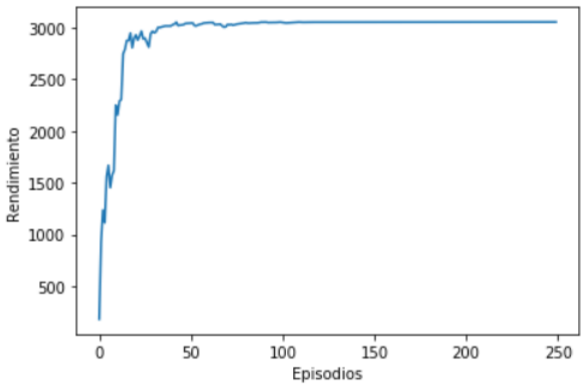


Figura 11. Gráfica de rendimiento total en fase 1 con gamma 0.8 para 250 episodios

Por otra parte, para valores del parámetro gamma inferiores, vemos que el tiempo no se ha visto realmente alterado al cambiar el valor de gamma, cambiando en un 7.79 % para 5000 episodios. Esto se debe a que el parámetro gamma solo influye en el algoritmo a la hora de escoger cierto camino, teniendo más en cuenta el agente las recompensas futuras con valores de gamma cercanos a 1 y teniendo más en cuenta las recompensas inmediatas con valores de gamma cercanos a 0.Los cambios mínimos que se observan en el tiempo se deben a un factor de aleatoriedad.

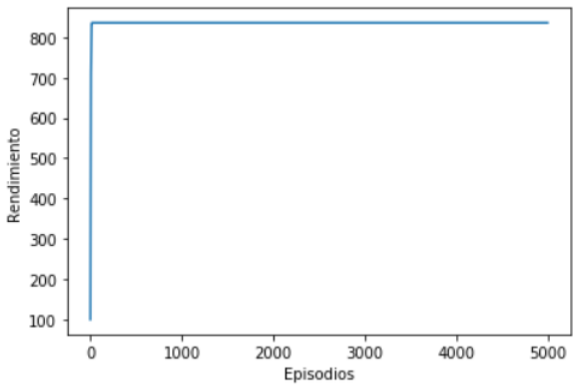


Figura 12. Gráfica de rendimiento total en fase 1 con gamma 0.8 para 5000 episodios

Episodios	Gamma/Epsilon/Alpha	Dimensiones	Tiempo (ms)
3000	0.8 / 0.9 / 0.9	3x3	2956.16
3000	0.8 / 0.9 / 0.1	3x3	2876.61
3000	0.8 / 0.1 / 0.9	3x3	3009
3000	0.8 / 0.1 / 0.1	3x3	3016
3000	0.2 / 0.9 / 0.9	3x3	2989.13
3000	0.2 / 0.9 / 0.1	3x3	3426.07
3000	0.2 / 0.1 / 0.9	3x3	2916.62
3000	0.2 / 0.1 / 0.1	3x3	2950.45

Tabla III
FASE 2

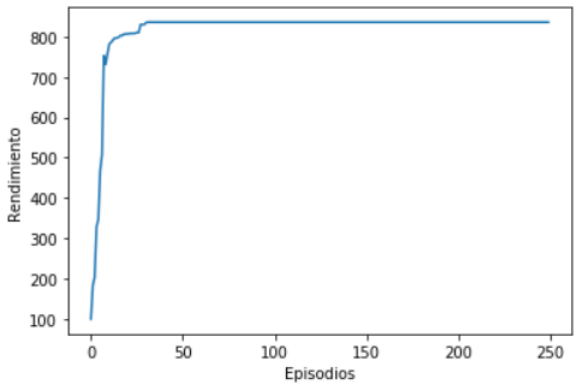


Figura 13. Gráfica de rendimiento total en fase 1 con gamma 0.2 para 250 episodios

En la fase 2, queremos evaluar el impacto de las políticas ϵ-greedy y la evolución del valor ϵ en función de Alpha. Para ello se ha realizado la ejecución del mismo tablero que en la fase 1 con el mismo estado inicial y final, con dos γ distintos: 0.8 y 0.2. Hemos variado el valor de ϵ y α probando las siguientes configuraciones: ϵ alto y α alto, ϵ alto y α bajo, ϵ bajo y α alto, ϵ bajo y α bajo.

En la tabla III observamos los tiempos de ejecución del algoritmo usando las configuraciones descritas:

Observamos que los tiempos de ejecución del algoritmo principal no son afectados por ϵ ni α, dado que estos sólo influyen en la evaluación de los valores que se almacenan en la matriz Q cerebro.

La política ϵ-greedy reina sobre la posibilidad de selección de valores aleatorios cuanto más grande sea ϵ, por ello es lógico deducir que durante el entrenamiento podrá realizar acciones no óptimas. Las figuras 14 y 15 muestran la medida de rendimiento para ϵ y α altos y para un ϵ bajo y α alto.

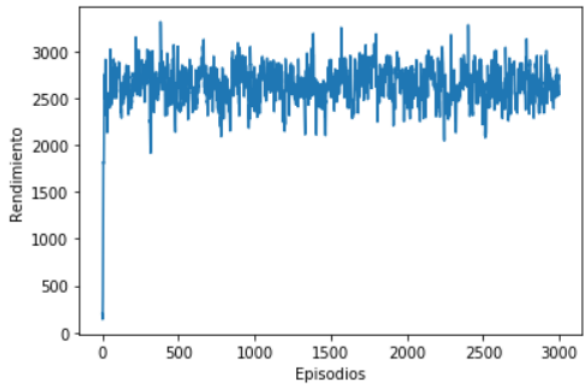


Figura 14. Gráfica de rendimiento total en fase 2 con gamma 0.8 para 5000 episodios y un valor de 0.9 para epsilon y alpha.

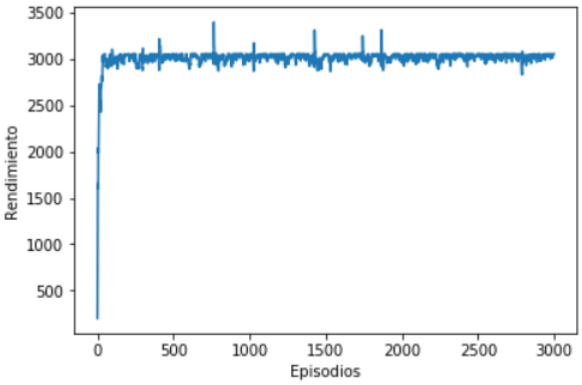


Figura 15. Gráfica de rendimiento total en fase 2 con gamma 0.8 para 5000 episodios y un valor de 0.1 para epsilon y 0.9 alpha.

Como observamos cuanto mayor es ϵ, más baila el rendimiento a lo largo de los episodios. En la figura 14 con un valor de ϵ alto, se observan las caídas de rendimiento que se dan cuando se escoge una opción aleatoria. En la figura 15 vemos como ϵ se mantiene más o menos constante, aunque se pueden observar pequeños picos superiores de rendimiento.

¿Por qué se dan picos superiores? Es posible que en un episodio determinado haya encontrado un camino mejor explorando, pero como no ha explorado esa posibilidad lo suficiente a lo largo de diversos episodios no será capaz de reconocer un camino más corto.

En la Figura 16 observamos un agente entrenado con los parámetros ϵ 0.1 y α 0.9 y γ 0.8, y en ella podemos ver que para el estado inicial 0 las acciones 1 y 3 tienen un valor mayor en q que la acción 2 aún cuando ir de 0 a 2 implica el camino más corto, por tanto determinamos que ha evaluado más los caminos que emplean las acciones 1 y 3 desde el estado 0 pero es posible que los picos superiores de rendimiento correspondan a episodios donde ha explorado usando la acción 2.

+	0	1	2	3	4	5	6	7	8
0	0	64	52	64	0	0	0	0	0
1	64	0	80	64	64	80	0	0	0
2	52	64	0	64	64	80	100	80	64
3	64	64	80	0	0	0	0	80	64
4	0	64	80	0	0	80	0	0	0
5	0	64	80	0	64	0	100	80	0
6	0	0	80	0	0	80	100	80	0
7	0	0	80	64	0	80	100	0	64
8	0	0	80	64	0	0	0	80	0

Camino más corto: [0, 1, 2, 6]

Figura 16. Matriz Q para Epsilon 0.1 y Alpha 0.9

Si le damos más espacio para explorar, como sería en un caso con los parámetros ϵ 0.9, γ 0.8 y α 0.9, el camino encontrado correspondería al más corto. Esto ocurre porque cuanto más explore nuestro agente, mejor podrá determinar otros posibles caminos existentes.

En la Figura 17 observamos como con un valor alto de ϵ encontramos el camino más corto con bastante diferencia a los demás aunque se pueden observar que caminos que deberían tener misma posibilidad de llegar al objetivo tienen valores distintos, esto se da debido al componente de aleatoriedad que introduce la política ϵ-greedy, si el agente puede explorar más es posible que no determine con precisión absoluta todos los caminos.

+	0	1	2	3	4	5	6	7	8
0	0	54	80	64	0	0	0	0	0
1	64	0	38	60	60	41	0	0	0
2	60	64	0	48	60	52	100	60	64
3	60	35	80	0	0	0	0	48	39
4	0	48	44	0	0	75	0	0	0
5	0	54	80	0	60	0	94	75	0
6	0	0	80	0	0	39	94	80	0
7	0	0	48	52	0	75	100	0	60
8	0	0	80	48	0	0	0	41	0

Camino más corto: [0, 2, 6]

Figura 17. Matriz Q para Epsilon 0.9 y Alpha 0.9

La observación de estos resultados, nos indica que ϵ influye enormemente sobre el rendimiento del algoritmo. Mientras que aumentar su valor nos permite determinar nuevos caminos que han podido ser ignorados en favor del más rápido conocido, la aleatoriedad que introduce puede que no determine el valor real de todos los caminos.

Sobre α, sólo afecta a cuán rápido disminuye ϵ dentro de un episodio, por lo que una gráfica de rendimiento con un ϵ alto y un α bajo, tenderá hacía una parecida a la de un ϵ bajo pasadas unas pocas iteraciones ya que el episodio abandonará la política de exploración rápidamente.

Para la fase 3 se ha diseñado un tablero, el cual se puede observar en la figura 16, propuesto por los autores con recompensas intermedias, las cuales consistirán de una casilla de mejora "Boost" (representada por una estrella) y una casilla trampa (representada por un rayo). También habrá paredes (representadas en rojo), las cuales impedirán que se puedan realizar algunas transiciones entre casillas.

Al aplicar el algoritmo de la fase 1 al tablero propuesto en la fase 3, nos encontramos con los siguientes resultados:

Figura 18. Tablero 4x4 propuesto para la fase 3

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	4	0	0	2	-36	0	0	0	0	0	0	0	0	0	0
1	9	0	-20	0	2	-36	20	0	0	0	0	0	0	0	0	0
2	0	4	0	100	0	-36	20	20	0	0	0	0	0	0	0	0
3	0	0	-20	100	0	0	20	20	0	0	0	0	0	0	0	0
4	9	4	0	0	0	-36	0	0	1	1	0	0	0	0	0	0
5	9	4	-20	0	2	0	20	0	1	0	0	0	0	0	0	0
6	0	4	-20	100	0	-36	0	20	0	0	0	12	0	0	0	0
7	0	0	-20	100	0	0	20	0	0	0	0	12	0	0	0	0
8	0	0	0	0	2	-36	0	0	0	1	0	0	1	1	0	0
9	0	0	0	0	2	0	0	0	1	0	1	1	1	1	3	0
10	0	0	0	0	0	0	0	0	0	1	0	0	0	1	3	3
11	0	0	0	0	0	0	20	20	0	0	0	0	0	0	3	3
12	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0
14	0	0	0	0	0	0	0	0	0	1	1	12	0	0	0	3
15	0	0	0	0	0	0	0	0	0	0	1	12	0	0	3	0

Camino más corto: [12, 8, 4, 0, 1, 6, 3]

Figura 19. Matriz Q al aplicar al tablero el algoritmo de la fase 1 con gamma 0.2

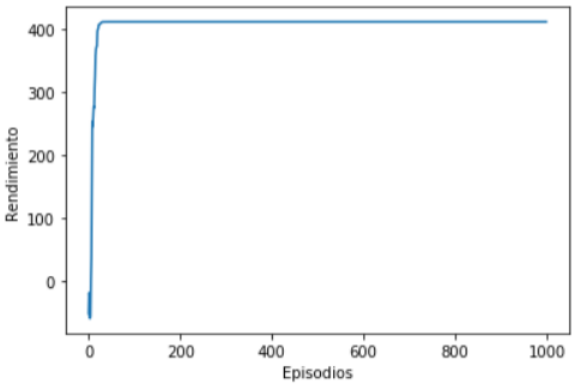


Figura 20. Gráfica de rendimiento al aplicar al tablero el algoritmo de la fase 1 con gamma 0.2

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	64	0	0	52	54	0	0	0	0	0	0	0	0	0	0
1	54	0	70	0	52	54	80	0	0	0	0	0	0	0	0	0
2	0	64	0	100	0	54	80	80	0	0	0	0	0	0	0	0
3	0	0	70	100	0	0	80	80	0	0	0	0	0	0	0	0
4	54	64	0	0	54	0	0	44	43	0	0	0	0	0	0	0
5	54	64	70	0	52	0	80	0	44	0	0	0	0	0	0	0
6	0	64	70	100	0	54	0	80	0	0	0	66	0	0	0	0
7	0	0	70	100	0	0	80	0	0	0	0	66	0	0	0	0
8	0	0	0	0	52	54	0	0	0	43	0	0	35	35	0	0
9	0	0	0	0	52	0	0	0	44	0	43	0	35	35	53	0
10	0	0	0	0	0	0	0	0	0	43	0	0	0	35	53	53
11	0	0	0	0	0	0	80	80	0	0	0	0	0	0	53	53
12	0	0	0	0	0	0	0	0	44	43	0	0	0	35	0	0
13	0	0	0	0	0	0	0	0	44	43	43	0	35	0	0	0
14	0	0	0	0	0	0	0	0	0	43	43	66	0	0	0	53
15	0	0	0	0	0	0	0	0	0	0	43	66	0	0	53	0

Camino más corto: [12, 8, 5, 6, 3]

Figura 21. Matriz Q al aplicar al tablero el algoritmo de la fase 1 con gamma 0.8

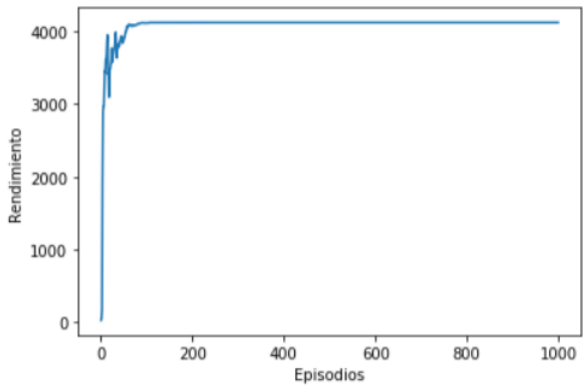
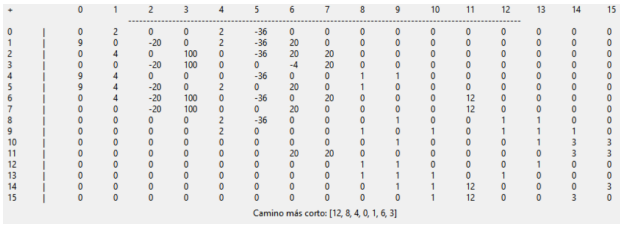


Figura 22. Gráfica de rendimiento al aplicar al tablero el algoritmo de la fase 1 con gamma 0.8

Como se puede observar en los valores de las matrices Q de las figuras 19 y 21, hemos obtenido un camino mucho más óptimo al usar un gamma más bajo, ya que como podemos comprobar en la figura 21 el camino no es correcto, pues nos hace pasar por la trampa colocada en la casilla 5. Esto se debe a que al tener un valor de gamma muy cercano a 1, no está teniendo en cuenta las recompensas intermedias, sino que solo valora llegar al objetivo lo antes posible. Sin embargo, no existe un solo camino posible tan óptimo como [12, 8, 4, 0, 1, 6, 3], sino que tal y como vemos en la figura 19 en la fila 12 se nos muestran 2 acciones más que son posibles e igual de óptimas (mismo valor en Q) que 8. Estas son 9 y 13, ya que al llevar a cabo estas acciones nos lleva por otro camino por el que también tenemos una casilla Boost, estos serían los caminos [12, 9, 14, 11, 7, 3] y [12, 13, 10, 15, 11, 7, 3]. También podemos observar en la figura 20 que alcanza el rendimiento máximo en un número menor de episodios que con un gamma cercano a uno, lo cual se puede ver también en la figura 22. Por lo tanto, para resolver el problema del tablero propuesto en la fase 3, se debería fijar un gamma cercano a 0. De esta forma, el algoritmo aprenderá que las recompensas intermedias son importantes a la hora de escoger un camino y aprenderá así a esquivar las trampas y a pasar por las casillas Boost.

Una vez determinado que es necesario un γ bajo, para que el agente aprenda a esquivar las trampas, realizamos pruebas añadiendo la política ϵ-greedy que se emplea en el algoritmo de la fase 2. Estas pruebas se realizarán sólo con un gamma bajo, para que el agente tenga en cuenta las trampas y las casillas Boost. Como conclusión de los experimentos de la fase 2 determinamos que α aún siendo un factor importante sólo es relevante para reducir el valor de ϵ dentro de un episodio. Por ello, se harán pruebas usando sólo un valor de ϵ alto y otro bajo.

En las Figuras 23 y 24 observamos los resultados obtenidos para un valor de ϵ de 0.9 y un α de 0.9. Comprobamos en la captura de la matriz Q obtenida el camino más beneficioso explorado: [12,8,4,0,1,6,3]. La gráfica de rendimiento muestra grandes fluctuaciones debidas a las razones ya explicadas en la fase 2: Si el ratio de exploración ϵ es alto la evaluación para las acciones de la matriz Q usa estados aleatorios, lo que hace variar enormemente el rendimiento entre distintos episodios.



+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	2	0	0	2	-36	0	0	0	0	0	0	0	0	0	0
1	9	0	-20	0	2	-36	20	0	0	0	0	0	0	0	0	0
2	0	4	0	100	0	-36	20	20	0	0	0	0	0	0	0	0
3	0	0	-20	100	0	0	-4	20	0	0	0	0	0	0	0	0
4	9	4	0	0	0	-36	0	0	1	1	0	0	0	0	0	0
5	9	4	-20	0	2	0	20	0	1	0	0	0	0	0	0	0
6	0	4	-20	100	0	-36	0	20	0	0	0	12	0	0	0	0
7	0	0	-20	100	0	0	20	0	0	0	0	12	0	0	0	0
8	0	0	0	0	2	-36	0	0	0	1	0	0	1	1	0	0
9	0	0	0	0	2	0	0	0	1	0	1	0	1	1	1	0
10	0	0	0	0	0	0	0	0	0	1	0	0	1	1	3	3
11	0	0	0	0	0	0	20	20	0	0	0	0	0	0	3	3
12	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0
14	0	0	0	0	0	0	0	0	0	1	1	12	0	0	0	3
15	0	0	0	0	0	0	0	0	0	0	1	12	0	0	3	0

Camino más corto: [12, 8, 4, 0, 1, 6, 3]

Figura 23. Matriz Q y camino más corto para ϵ 0.9, 3000 episodios

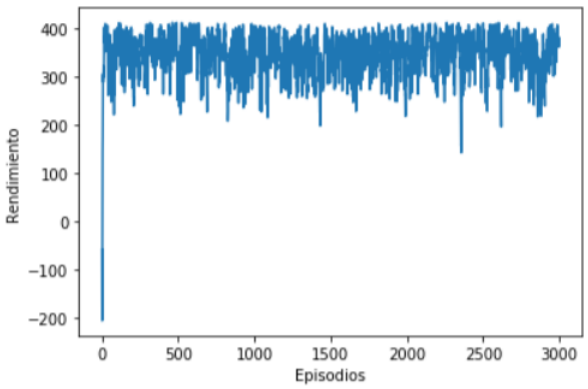
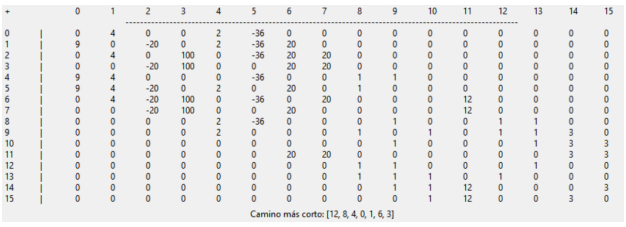


Figura 24. Gráfica de Rendimiento para ϵ 0.9, 3000 episodios

En las Figuras 25 y 26 vemos los resultados obtenidos al experimentar usando γ 0.2, ϵ 0.1 y α 0.9. La figura que contiene la matriz Q muestra que el camino más beneficioso es [12,8,4,0,1,6,3] al igual que el anterior, aunque se puede observar que se pueden tomar los caminos alternativos [12,9,14,11,7,3] o [12,9,14,11,6,3] que parten de tomar la acción 9 desde el estado 12 dado que las acciones 8 y 9 para el estado 12 tienen el mismo valor en la matriz Q. Este es un caso que no se daba para un ϵ de 0.9, dado que por el factor de aleatoriedad es posible que no hubiese investigado la acción 9 lo suficiente.

Observamos que al igual que en la fase 2, la gráfica para un ϵ bajo tiende a oscilar menos debido a su baja exploración.



+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	4	0	0	2	-36	0	0	0	0	0	0	0	0	0	0
1	9	0	-20	0	2	-36	20	0	0	0	0	0	0	0	0	0
2	0	4	0	100	0	-36	20	20	0	0	0	0	0	0	0	0
3	0	0	-20	100	0	0	20	20	0	0	0	0	0	0	0	0
4	9	4	0	0	0	-36	0	0	1	1	0	0	0	0	0	0
5	9	4	-20	0	2	0	20	0	1	0	0	0	0	0	0	0
6	0	4	-20	100	0	-36	0	20	0	0	0	12	0	0	0	0
7	0	0	-20	100	0	0	20	0	0	0	0	12	0	0	0	0
8	0	0	0	0	2	-36	0	0	0	1	0	0	1	1	0	0
9	0	0	0	0	2	0	0	0	1	0	1	0	1	1	3	0
10	0	0	0	0	0	0	0	0	0	1	0	0	0	1	3	3
11	0	0	0	0	0	0	20	20	0	0	0	0	0	0	3	3
12	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
13	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0
14	0	0	0	0	0	0	0	0	0	1	1	12	0	0	0	3
15	0	0	0	0	0	0	0	0	0	0	1	12	0	0	3	0

Camino más corto: [12, 8, 4, 0, 1, 6, 3]

Figura 25. Matriz Q y camino más corto para ϵ 0.1, 3000 episodios

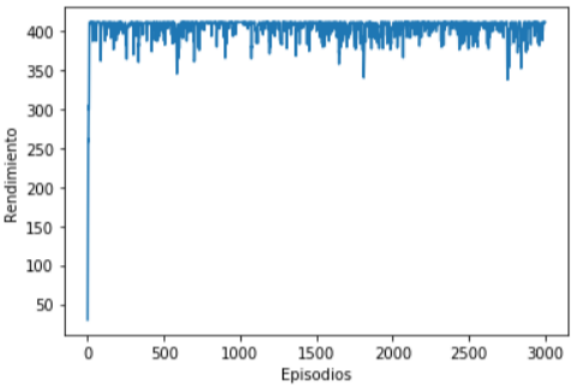


Figura 26. Gráfica de Rendimiento para ϵ 0.1, 3000 episodios

Comparando las Figuras 23 y 25, sólo notamos pocos cambios significativos en el estado 0 para el que la acción 1 tiene más valor en la segunda figura que en la primera o que algunas acciones no se han siquiera probado. Estas diferencias se corresponden con lo comprobado en la fase 2.

Comparar las figuras 24 y 26, confirma que lo estudiado durante la fase 2 también se da en este nuevo tablero con recompensas intermedias. Es por eso que la gráfica de rendimiento para un ϵ alto varía más que una para un ϵ bajo.

V. Conclusiones

En resumen, tal y como hemos ido observando a lo largo del documento, se ha estudiado como al aplicar el algoritmo de aprendizaje por refuerzo Q-learning en un ámbito de path-finding, se obtienen caminos más o menos óptimos al usar distintos valores para los parámetros del algoritmo y a su vez, como influyen estos valores a la hora de que el agente aprenda los diferentes caminos.

Sobre la implementación, el hecho de haber introducido una interfaz gráfica completamente validada ha aumentado notablemente la carga de trabajo y el tamaño del desarrollo de la batería experimental. La presencia de la interfaz gráfica nos ha forzado a dividir el código en Frontend: operaciones visibles al usuario; y Backend: operaciones no visibles al usuario.

Respecto al algoritmo Q-learning, hemos podido observar como el factor gamma provoca que para valores cercanos a 1 el agente tenga más en cuenta las recompensas futuras, mientras que para valores cercanos a 0 el agente tiene más en cuenta las recompensas inmediatas a la hora de escoger un camino. El factor epsilon hace que el algoritmo tenga una política más o menos orientada a la exploración de otros caminos, de forma que esta se favorece para valores de epsilon cercanos a 1. Por otro lado, el factor alpha es un multiplicador de epsilon, que hace que este cambie en cada episodio, haciendo que deje de explorar más rápidamente para valores cercanos a 0 o que continue más tiempo con la política de exploración para valores cercanos a 1. A la hora de introducir recompensas intermedias concluimos que hay que usar un valor bajo de γ para que el agente las tenga en cuenta, de otra forma el agente valorará la llegada al objetivo sin tener mucho en cuenta las recompensas intermedias. La política ϵ-greedy en el contexto de las recompensas intermedias afecta de la misma manera que lo hace en la fase 2.

Para terminar, este proyecto se podría mejorar en un futuro optimizando el código, de forma que al introducir un tablero de dimensiones muy grandes, el tiempo de ejecución del algoritmo no se vea tan afectado. Otras posibles mejoras serían una interfaz gráfica que devuelva los posibles caminos en forma de grafo y no en forma de una cadena de texto como actualmente, o el ajuste de los valores de las recompensas intermedias en la propia interfaz.

VI. Mejoras

Los autores proponen las siguientes mejoras a la hora de realizar el proyecto:

- El uso de una plantilla LaTeX, para la cual los autores se han apoyado en algunos apuntes que explican el uso de ciertos métodos y símbolos matemáticos [8] [9].

VII. Anexo. Marco Teórico

Tal y como pasa con los seres vivos, los algoritmos también son capaces de aprender tras realizar cierta acción como respuesta a los estímulos que reciben desde el entorno que trabajan. De esta forma se puede decir que se ha aprendido del entorno, al igual que lo haría una persona con la que usáramos algún tipo de psicología conductista (figura 27).

Figura 27. Psicología del conductismo

De este razonamiento podemos entender el aprendizaje automático, rama de la Inteligencia Artificial que persigue como objetivo desarrollar técnicas que permitan a los ordenadores aprender con ciertas acciones.

Existen dos tipos principales de aprendizaje automático: en primer lugar, el **Aprendizaje Supervisado**, y por otro lado el **Aprendizaje No Supervisado**. Sin embargo, podemos considerar un tercer tipo de aprendizaje junto a estos dos tipos, el cual es conocido como **Aprendizaje por Refuerzo**. En este tipo, el algoritmo recibirá algún tipo de valoración acerca de lo correcta que es su respuesta. De esta forma cuando su respuesta es correcta, el aprendizaje por refuerzo se parece al aprendizaje supervisado, ya que en ambos casos el aprendiz recibe información sobre que respuesta es más apropiada. Aunque estas aproximaciones difieren cuando se trata de una respuesta errónea.

Esto es diferente ya que, en el aprendizaje supervisado se le dice exactamente al aprendiz que es lo que debería haber respondido, mientras que en el Aprendizaje por Refuerzo solo se le informa de que la respuesta ha sido errónea y cómo de perjudicial ha sido cometer dicho error.

En el ejemplo que comentábamos anteriormente, cuando aplicamos psicología conductista a una persona, si hace bien cierta acción esta recibirá una recompensa, mientras que si no lo hace recibirá algún castigo. La repetición de esta tarea producirá un refuerzo de los comportamientos que son considerados como acertados respecto de los comportamientos que reciben un castigo.

El aprendizaje por Refuerzo se compone de un agente, un conjunto de estados S, un conjunto de acciones por estado A. Al llevar a cabo una acción a que pertenece a nuestro conjunto de acciones A, el agente pasará al siguiente estado a. El haber ejecutado dicha acción proporcionará cierta recompensa numérica al agente. Se puede apreciar un esquema de este tipo de aprendizaje en la Figura 28.

El agente perseguirá el objetivo de maximizar la recompensa total. Esto se puede conseguir sumando la máxima recompensa alcanzable, en estados futuros, a la recompensa por alcanzar su estado actual. De esta forma, el agente no tiene por qué conocer a priori la recompensa o el estado siguiente. Por lo que, antes de aprender, el agente no sabe las consecuencias que tendrá tomar una acción determinada en cierto estado. Por esto, podemos entender como un buen aprendizaje aquel que permite al agente adelantarse a las consecuencias de tomar cierta acción, de forma que pueda investigar las acciones disponibles que más recompensa le aportarán o cuáles le penalizarán en su camino al objetivo.

Es por esto, que se puede definir el objetivo del aprendizaje por refuerzo como extraer qué acciones deben ser elegidas en los diferentes estados de forma que se maximice así la recompensa. De esta forma, lo que buscamos es que el agente aprenda una política, la cual entendemos como una utilidad que nos dice en cada estado que acción es más beneficiosa tomar. Podemos dividir la política del agente en dos componentes diferentes: en primer lugar, como de óptimo cree el agente que es una cierta acción sobre un cierto estado, y luego, cómo el agente es capaz de recordar que estados le beneficiarán más y cuáles le perjudicarán.

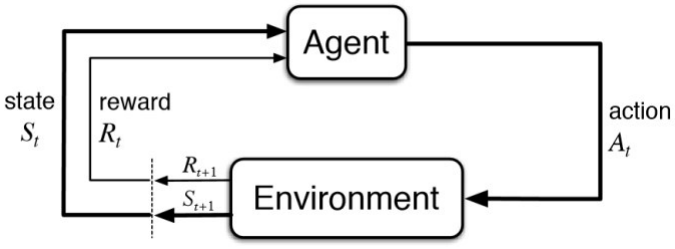


Figura 28. Esquema de Aprendizaje por Refuerzo

Uno de los algoritmos que se basa en el Aprendizaje por Refuerzo es el algoritmo Q-learning [10], cuyo objetivo es aprender una serie de normas a partir de las cuales un agente decida que acción debería tomar bajo ciertas circunstancias. Dicho algoritmo no requiere de un modelo del entorno y es capaz de manejar problemas con un comportamiento no determinista. Dado un cierto proceso de decisión de Markov finito, el algoritmo Q-learning es capaz de encontrar un camino óptimo al estado objetivo de forma que el valor de recompensa total, aquellas que vamos sumando recorriendo el camino óptimo, sea el máximo posible; teniendo en cuenta las diferentes acciones que se pueden tomar desde el estado en el que el agente empieza.

En el algoritmo Q-learning, el valor de un par (estado, acción) de la matriz Q contiene la suma tanto de las recompensas inmediatas, como de las futuras. Pero, en el caso de que no hubiera un estado final que alcanzar, este valor podría ser infinito, por lo que para solucionar dicho problema hacemos uso de un aprendizaje acumulado por descuento, donde multiplicamos las recompensas futuras por un factor $\gamma \in [0, 1]$. De esta forma, cuanto mayor sea el factor gamma, mayor influencia tendrán las recompensas futuras en el valor Q de dicho par.

Matemáticamente hablando, el cálculo de los valores de la matriz Q vendrán dados por la siguiente ecuación donde s=state, a=action, n=next action y aa=all actions:

$$Q(s, a) = R(s, a) + \gamma * Max[Q(n, aa)] \tag{2}$$

Debemos tener en cuenta el matiz de que el aprendizaje por refuerzo solo actualiza los valores Q de acciones que se usan sobre estados por los que se ha pasado, pero no se aprende nada del resto de acciones que no se han usado. Es por esto que se recomienda que al principio se use un valor de gamma más pequeño, de esta forma el agente intenta más acciones, por lo que se dará cuenta de que caminos funcionan mejor.

Algunas variantes de este algoritmo Q-learning [11] son: Q-learning profundo, Q-learning doble, Q-learning tardío o SARSA [7]. Este último al igual que Q-learning, es un método de diferencia temporal. Aunque ambos métodos intentan aprender los valores optimos de Q para todos los pares (estado,acción), la forma en que ambos métodos aproximan Q es diferente. Mientras que Q-learning es un método sin política y su fórmula matemática aproxima directamente Q, SARSA si utiliza una política que aproxima Q mediante un proceso denominado *iteración de política generalizada*. Esto quiere decir que SARSA estima Q para la política P que está siguiendo el agente, mientras que a la vez cambia P para hacerla más *greedy* con respecto a Q.

En el ámbito real, los algoritmos de aprendizaje por refuerzo como Q-learning son realmente útiles, dándole solución a algunos problemas muy comunes y a otros más atípicos. A continuación trataremos algunas de las aplicaciones más útiles de estos algoritmos, los cuales suelen ser usados en ámbitos de *path finding*, búsqueda de caminos en inglés. Es por esto que una de sus aplicaciones es solucionar atascos debidos a los semáforos. En algunos estudios sobre entornos simulados [12] hemos podido observar como, al usar un

algoritmo de aprendizaje por refuerzo multi-agente, es posible buscar los caminos que reducen al máximo los atascos. En este caso, los estados serían el flujo relativo de tráfico en cada carril, mientras que las acciones disponibles para el agente serían las distintas combinaciones de fases y la función de recompensa vendría dada en relación a la reducción de tiempo conseguida respecto de las demás acciones.

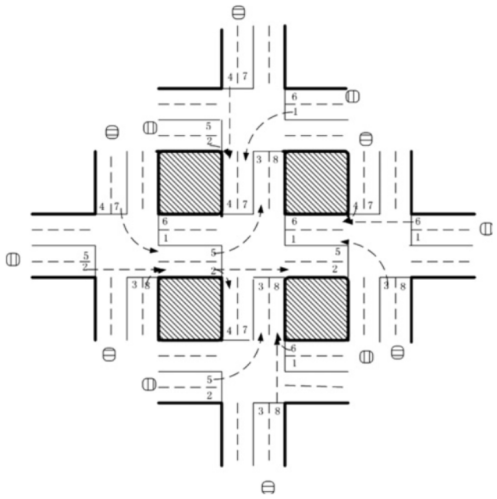


Figura 29. Ejemplo de aplicación de un algoritmo RL en una intersección de semáforos.

Otra aplicación de dichos algoritmos sería la robótica, donde ya se ha usado para enseñar a un robot a diseñar un camino respecto de ciertos estímulos que recibe del entorno, así como a esquivar objetos que se pueda encontrar en el camino [13].

Por otro lado, el aprendizaje por refuerzo también ha sido usado a la hora de configurar sistemas web [14], donde llegamos a encontrar más de 100 parámetros configurables que requieren a un experto y numerosas pruebas. En algunos estudios se ha mostrado una reconfiguración autónoma de dichos parámetros en sistemas basados en máquinas virtuales[insertar referencia]. Los autores fijaron los estados como los parámetros de configuración del sistema dónde las acciones podían aumentar, disminuir o mantener cada uno de los parámetros y las recompensas venían dadas como la diferencia ente el tiempo de respuesta objetivo y el tiempo de respuesta dado.

Asimismo, hemos podido ver aplicaciones de aprendizaje por refuerzo también en química [15], donde se ha aplicado para optimizar reacciones químicas[introducir referencia]. En dicho estudio los estados venían dados por el conjunto de condiciones experimentales (temperatura, presión atmosférica...), las acciones por el conjunto de todas las acciones que puedan cambiar las condiciones experimentales y la recompensa por una función del estado. Esta aplicación concreta demuestra lo beneficioso que puede ser el aprendizaje por refuerzo para reducir la carga de trabajo en pruebas de ensayo y error en un entorno estable.

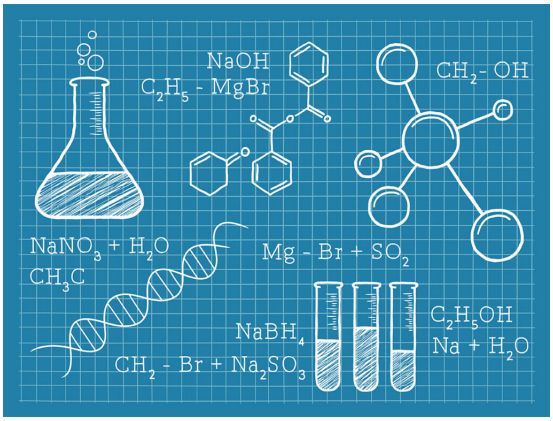


Figura 30. Entorno químico.

Por último, hablaremos de uno de los campos donde el aprendizaje por refuerzo ha supuesto una auténtica revolución: El mundo de los videojuegos. En él hemos podido observar algoritmos de aprendizaje por refuerzo en, prácticamente, todos los juegos que implementan una inteligencia artificial, llegando a conseguir que el algoritmo resuelva algunas jugadas o conseguir un rendimiento que un humano no podría conseguir. Los más famosos deben ser los algoritmos AlphaGo [16] y AlphaGo Zero [17], los cuales hemos podido observar como se usaban en juegos de estrategia en tiempo real como StarCraft II [18]. Recientemente hemos podido ver el proyecto OpenAI Five [19], que usa una Inteligencia Artificial que emplea Aprendizaje por Refuerzo para jugar al videojuego Defense of the Ancients 2. Este proyecto ha conseguido poner en jaque incluso a algunos de los mejores jugadores profesionales del juego [20].

Referencias

[1] Step-By-Step Tutorial of Q-learning. Example given by IA department, http://mnemstudio.org/path-finding-q-learning-tutorial.htm

[2] Bryan Oakley, solución al recorrer una matriz de ciertas dimensiones dadas por el usuario, https://stackoverflow.com/a/18986884/9746316

[3] Surya Kari, Método para generación automática de la matriz recompensa. https://www.linkedin.com/pulse/simple-q-learning-algorithm-python-surya-kari

[4] Ejemplo de código de Path-Finding con Q-learning en lenguaje Java, http://mnemstudio.org/ai/path/q_learning_java_ex1.txt

[5] Kyle Kastner, Ejemplo de código de path-finding mediante política e-greedy, https://gist.github.com/kastnerkyle/d127197dcfdd8fb888c2

[6] Ejemplo de método para encontrar los vecinos de un vértice en el tablero. https://stackoverflow.com/questions/22550302/find-neighbors-in-a-matrix5udhge

[7] Apuntes sobre el algoritmo SARSA. https://www.academia.edu/11392805/SARSA_BB_Un_algoritmo_on_policy_para_Sistemas_Clasificadores

[8] Apuntes sobre latex y fórmulas matemáticas. http://metodos.fam.cie.uva.es/ latex/apuntes/apuntes3.pdf

[9] Apuntes sobre latex y conceptos de excritura matemática.

[10] Artículo sobre algoritmo Q-learning. http://www.cs.us.es/ fsancho/?e=109

[11] Enlace al articulo de wikipedia de Q-learning y algunas de sus variantes. https://es.wikipedia.org/wiki/Q-learning#Variantes

[12] Artículo sobre aplicación de Q-learning en un entorno de semáforos. http://web.eecs.utk.edu/ ielhanan/Papers/IET_ITS_2010.pdf

[13] Artículo sobre aplicación de Q-learning en un entorno de robótica. https://www.ias.informatik.tu-darmstadt.de/uploads/Publications/Kober_IJRR_2013.pdf

[14] Artículo sobre aplicación de Q-learning en un entorno de configuración de un sistema web. http://ranger.uta.edu/ jrao/papers/ICDCS09.pdf
[15] Artículo sobre aplicación de Q-learning en un entorno de química. https://pubs.acs.org/doi/full/10.1021/acscentsci.7b00492
[16] Artículo sobre aplicación de Q-learning en videojuegos. AlphaGo. https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf
[17] Artículo sobre aplicación de Q-learning en videojuegos. AlphaGo Zero. https://deepmind.com/blog/article/alphago-zero-starting-scratch
[18] Artículo sobre aplicación de Q-learning en el videojuego Starcraft2. https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii
[19] Artículo sobre el uso de Aprendizaje por refuerzo en el videojuego Dota2. https://arxiv.org/abs/1912.06680
[20] Noticia en la que se muestra las dificultades de algunos profesionales para superar a la IA desarrollada. https://www.theverge.com/2019/4/13/18309459/openai-five-dota-2-finals-ai-bot-competition-og-e-sports-the-international-champion
[21] Steven Rumbalski, sobre la validación de Widgets en TkInter. https://stackoverflow.com/questions/4140437/interactively-validating-entry-widget-content-in-tkinter