Aprendizaje Automático Por Refuerzo

Ezequiel Portillo
Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática
Sevilla, España
ezeporjur@us.es

Belén Garrido
Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática
Sevilla, España
belgarlop3@us.es

I. INTRODUCCIÓN

El aprendizaje por refuerzo viene dado por las recompensas dadas por las acciones que el algoritmo realice. Para estudiar cómo funciona este tipo de aprendizaje, hemos utilizado un pequeño proyecto de inicio para entender mejor cómo afrontar nuestro objetivo principal.

Este proyecto, llamado proyecto inicial [1], consta de una serie de líneas de código y que analizamos para entender cómo funciona cada una de sus partes. De esta forma, nos hemos centrado en identificar los elementos que intervienen en la representación que se da del entorno del problema, describir el funcionamiento del algoritmo y, por último, interpretar el papel que juega cada parámetro de este.

A. *Elementos que intervienen en el entorno*

1) *Estados:* Es la situación en la que se encuentra el agente en el entorno dado. Para completar el problema empezaremos en un estado y a través de distintas acciones se deberá de llegar a un "estado final". Estos estados pueden tener consecuencias en la decisión del algoritmo debido a las recompensas otorgadas por pasar por esos estados.

2) *Acciones:* En el entorno en que nos encontramos, una acción es la decisión que toma el algoritmo de realizar una acción u otra. Para llevar a cabo una acción, se necesita estar en un estado, y tras ejecutar dicha acción, termina en otro (o en el mismo estado en algunos casos). Existen de 1 a N probabilidades de ejecutar unas acciones u otras, que, además, son elegidas aleatoriamente.

3) *Probabilidades:* Para elegir cada una de las acciones que puede tomar nuestro algoritmo, cada una de estas tienen asociada una probabilidad de forma que la suma de todas estas deberá ser 1. Estas probabilidades se pueden ir alterando a medida que provoquemos que el algoritmo aprenda, induciendo a que el agente alcance la solución más óptima posible.

4) *Recompensas*: La recompensa es la encargada de guiar al algoritmo, indicándole lo bien o mal que se está ejecutando. En cada uno de los estados por los que pasa el algoritmo encontramos una recompensa que va facilitando la resolución del problema. Nuestra prioridad, además de optimizar el camino a recorrer, es maximizar la recompensa total del algoritmo. La recompensa total se obtiene con la suma total de las recompensas obtenidas en cada estado.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$$

En nuestro entorno también es importante tener en cuenta la diferencia entre las recompensas presentes y las futuras, para no cometer el error que lleva a la suma hasta el infinito, provocando así que nunca se encuentre nuestra solución. Por ello, utilizamos la variable γ que se encarga de minimizar y maximizar el impacto de la recompensa según las acciones que realice el algoritmo.

5) *Política:* Es una función que determina qué acción se debe escoger en un estado determinado. El objetivo es que la política consiga maximizar la recompensa final, además de encargarse de indicar la probabilidad de realizar una acción cuando el agente se encuentra en un estado determinado.

$$\sum_a \pi(s, a) = 1$$

B. *Funcionamiento del algoritmo*

El algoritmo que se encuentra en el proyecto de inicio consta de 2 partes principalmente.

- *Entorno:* se encarga, a grandes rasgos, de generar las líneas de código necesarias para que el algoritmo cree el mapa a recorrer, es decir, establece los datos de entrada del problema, como la posición inicial, posición final, etc. También realiza la acción que decida el algoritmo, es decir, desplazar a nuestro agente por el mapa según la acción escogida. Finalmente, muestra las modificaciones que se van realizando en el mapa para que el usuario pueda comprobar cómo funciona el algoritmo de forma más visual y fácil.
- *Algoritmo Q-Learning:* en esta parte del proyecto está la propia utilización del algoritmo de aprendizaje por refuerzo que usa los distintos métodos creados en el entorno y aplica Q-Learning. También se declaran algunos parámetros necesarios para el algoritmo y la tabla de valores de acciones.

A continuación, se describe el funcionamiento del algoritmo junto a los distintos métodos que lo componen.

Tras declarar las variables de inicio del entorno, se crea una tabla de forma aleatoria de dimensión igual al número de casillas que tiene nuestro mapa. Cada una de las filas de la

tabla contiene una serie de números entre 0 y 1, que sirven al algoritmo para elegir la acción a realizar. El número de columnas de la tabla viene dado por el número de acciones diferentes que el algoritmo puede realizar, que, en nuestro caso, son: ir arriba, abajo, derecha e izquierda. Básicamente, cada estado encuentra en esta tabla la posibilidad de escoger una acción u otra según el número de la celda en que se encuentre.

A continuación, se declara una serie de valores que necesitaremos para el algoritmo:

- *"Epochs"*: se refiere al número de iteraciones que realizará nuestro algoritmo.
- *"Gamma"*: es la variable utilizada para controlar el impacto de la recompensa.
- *"Epsilon" y "Decay"*: sirven para indicar y controlar que cuanto más aprenda el algoritmo menos acciones aleatorias deberá elegir.

Tras todo esto, el algoritmo comienza a ejecutar iteraciones (epochs). En cada una de ellas, el algoritmo reinicia los valores de recompensa y estado, además, se visualiza el mapa que este momento tiene el algoritmo (en caso de ser al comienzo, éste sería el mapa inicial). La iteración continúa con la decisión de tomar una acción aleatoria o si, al contrario, esta será según los valores encontrados en la tabla anteriormente nombrada. Esta serie de decisiones son tomadas por la variable épsilon. Si el algoritmo escoge una opción aleatoria, esta será escogida entre todas las posibles. Mientras que, si la opción elegida no es aleatoria, la acción será escogida según el valor más alto que se encuentre en la fila del mapa. Por ejemplo, si el agente se encuentra en la posición (estado) 7 del mapa, el algoritmo recorrerá la q-tabla hasta la fila 7 y comprobará cuál de los números es el mayor. Supongamos que el mayor número es el de la primera columna, entonces, el algoritmo habrá decidido que la acción que se va a tomar es la 0, la cual está relacionada con la acción de ir a la izquierda en el mapa.

Se calcula el nuevo estado en el que se encuentra el algoritmo en ese momento, la recompensa obtenida y si está en el estado final (si esto es así, será la última vuelta de esa iteración). Gracias a la recompensa obtenida y al uso de la variable gamma (que controla los valores del mapa), obtenemos el nuevo valor de la celda. Así, conseguimos que los valores en la tabla cambien, y que el algoritmo, en sus últimas iteraciones, cuando ya no seleccione acciones aleatorias, vaya seleccionando la acción que le proporciona mayor recompensa.

De esta forma, conseguimos maximizar la recompensa. También es de destacar que en el proyecto de inicio, solo obtenemos una recompensa distinta a 0 al finalizar el algoritmo, lo que conlleva a que se maximice su llegada desde el estado inicial al final.

Finalmente, se calcula el nuevo estado actual y se reduce la variable épsilon gracias a la variable decay que, como ya se dijo antes, se encarga de que el algoritmo elija menos acciones aleatorias según vaya avanzando en el número de iteraciones.

C. Papel de cada parámetro

- *self.height:* se encarga de indicar la altura del mapa.
- *self.width:* se encarga de indicar el ancho del mapa.
- *self.posX:* se encarga de indicar la posición en el mapa, en el eje X, del estado en el que se encuentra actualmente.
- *self.posY:* se encarga de indicar la posición en el mapa, en el eje Y, del estado en el que se encuentra actualmente.
- *self.endX:* se encarga de indicar la posición en el mapa, en el eje X, del estado final que se ha elegido.
- *self.endY:* se encarga de indicar la posición en el mapa, en el eje Y, del estado final que se ha elegido.
- *self.actions:* es una lista que contiene las distintas acciones que puede realizar el algoritmo. En este caso, utiliza acciones numeradas del 1 al 4 para llevar a cabo las acciones: arriba, abajo izquierda y derecha.
- *self.stateCount:* cuenta el número de estados diferentes que tiene nuestro mapa y por los que puede ejecutarse el algoritmo.
- *self.actionCount:* cuenta el número de acciones que ha ejecutado el algoritmo en cada una de las iteraciones.
- *self.done:* es una variable booleana que indica si nos encontramos en el estado final requerido o no.
- *action:* en esta variable se guarda el número (entre 1 y 4 en este caso) que indica la decisión tomada para realizar la acción. Esta puede venir tanto de la elección aleatoria como de la tabla de estado/acción.
- *nextState:* en esta variable se guarda el siguiente estado en el que se encuentra el algoritmo tras haber realizado una determinada acción.
- *reward:* en esta variable se almacena la recompensa obtenida al pasar por un estado en concreto, y que servirá para modificar la tabla de estado/acción.
- *qtable:* es una tabla con filas y columnas en las que se almacenan una serie de números aleatorios que se van modificando de forma que el algoritmo, tras varias iteraciones, aprende cuáles son las acciones que debe ejecutar para realizar el recorrido más óptimo.
- *epochs:* número total de iteraciones que va a realizar nuestro algoritmo.
- *gamma:* es una de las variables más importantes, ya que nos ayuda a modificar el impacto de la recompensa según el algoritmo recorre más casillas. En el caso del proyecto inicial, gamma no varía, mientras que altera el número máximo de la tabla. Para la realización del ejercicio, la utilizaremos para hacer que la recompensa disminuya conforme más pasos del agente sobre

el mapa, además de para indicar que el impacto de la recompensa sea menor. De esta forma, conseguimos evitar la suma infinita de recompensas.

- *epsilon:* es una variable que se va reduciendo en cada acción ejecutada en una iteración y que se encarga de disminuir la probabilidad de que el algoritmo escoja una acción aleatoria.
- *decay:* número que, multiplicado por épsilon, sirve para calcular la cantidad restada a epsilon en cada acción que se ejecuta.
- *state:* es el estado actual en el que se encuentra el algoritmo antes de realizar la acción oportuna
- *steps:* número de acciones que realiza nuestro algoritmo antes de alcanzar al estado final en cada una de las iteraciones que se realizan.

II. IMPLEMENTACIÓN DEL ENTORNO DEL PROBLEMA Y ADAPTACIÓN DEL ALGORITMO

Para realizar el problema propuesto, hemos utilizado, como se sugería, la implementación base que se nos proporcionaba a raíz del proyecto inicial.

Podemos dividir esta implementación en 3 partes. Por un lado, la implementación del nuevo entorno, incluyendo un entorno de prueba personalizado para el problema dado. Por otro lado, las modificaciones en el algoritmo de aprendizaje con refuerzo y, finalmente, la implementación realizada con Tkinter para mostrar el mapa generado y la solución de este.

1. Implementación del entorno

En el código que se nos ha proporcionado en el proyecto inicial, nos encontramos con el método de __init__ y reset que se encargan de inicializar los valores necesarios para la creación del mapa y, también, de reiniciar este en cada una de las iteraciones que va a realizar el algoritmo. En este caso, estos dos métodos han sido modificados en menor medida, ya que, la única actualización realizada en ellos ha sido la de los parámetros de entrada que se le proporciona al algoritmo. Esto es debido a que el mapa del problema es diferente al del proyecto inicial y comienza en una posición distinta.

A continuación, encontramos el método step que se encarga de realizar la acción que elige el algoritmo. Se ha reutilizado el trozo de código que contiene a los if/else ya que, al intentar modificarlo, no se han visto cambios significativos en la solución y, por tanto, no nos interesaba. No obstante, algo que sí que se ha modificado ha sido el cálculo del próximo estado y el atributo "done" (aunque este último no influye en gran medida).

El cambio más significativo lo encontramos en la recompensa debido a que el algoritmo base solo daba recompensa = 1 cuando llegaba al final. En cambio, nosotros hemos dado un valor distinto a cada una de las casillas teniendo en cuenta cuales poseían más valor o menos según el enunciado (este valor está medido según a cuáles de las casillas debíamos darles prioridad frente a otras para pasar por ellas) [2]. Siguiendo el enunciado, hemos deducido que: dado que el mayor valor era para las casillas marrones claras o llanuras (en nuestro mapa, grises), estas reciben 9 de recompensa y son representadas por el número 3; en prioridad le siguen las casillas verdes o bosques, que reciben 6 de recompensa, representadas por el número 2; luego, están las casillas marrones o montañas, con 3 de recompensa que se representan con el número 1; y, por último, las casillas azules o agua, donde la recompensa es de -30 y que son representadas por el número 0. Además, hemos añadido una restricción negativa de -20 a las casillas de agua. De esta forma, tras realizar varios experimentos, comprobamos que, debido a esta restricción negativa de -20 de recompensa, el algoritmo hace todo lo posible por no atravesarlas a menos que sea estrictamente necesario (un callejón sin salida o la propia casilla de inicio). Las demás casillas, como ya hemos descrito, sí que tienen un valor positivo dependiendo de su prioridad. Este valor ha sido calculado gracias a distintos experimentos que nos han ayudado a ajustar estas recompensas.

El método que selecciona una acción aleatoria se ha mantenido. Sin embargo, el método de render que muestra por consola el mapa en tiempo real y el estado por el que el agente está pasando, ha sido modificado de forma severa. Tanto es así que hemos generado 5 métodos nuevos.

Se ha implementado el método "randomlista" que se encarga de generar, mediante de una lista de valores del 0 al 3, una lista aleatoria de números con la cual podemos crear un mapa totalmente aleatorio.

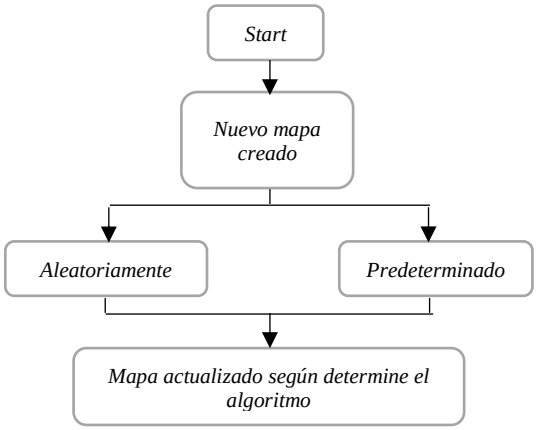
Esta lista aleatoria se utiliza en el método de "crearMapa", que se encarga de generar un mapa que se escribe en la consola al igual que el antiguo render. No obstante, en este caso, mostramos un valor que nos ayuda, de manera visual, a diferenciar los tipos de casillas. Esto nos resultó muy útil, ya que es mucho más fácil ver cómo trabajaba el algoritmo y qué caminos va siguiendo.

Del mismo modo, encontramos el método "crearMapaPrueba" que se encarga de crear el mapa específico del problema que se nos ha planteado. Genera, de la misma forma que el método "crearMapa", el mapa, pero, en este caso, la lista no es aleatoria, si no que es una lista creada a mano con los parámetros que necesitamos.

Por otro lado, encontramos el método "modificaMapa". Este método es el encargado de ir dibujando en el mapa creado una A (nuestro agente) que nos muestra, de manera más visual y sencilla, en qué estado se encuentra el algoritmo en cualquier momento de la ejecución. Esto nos ha facilitado la comprobación de la ejecución del algoritmo, dejando nos apreciar en qué momento fallaba. Está implementado de una forma muy parecida al "creaMapa".

Por último, encontramos el método "fin" cuya única función es la de dibujar la A al final del recorrido del algoritmo.

Proceso ejecutado en la iteración x al crear el mapa



Algo llamativo de nuestra implementación es que, a diferencia de la metodología utilizada en el proyecto inicial que utiliza las cadenas de caracteres para representar el mapa a recorrer, nosotros nos hemos decantado por la utilización de listas de números. Esta decisión fue tomada debido a la versatilidad que ofrece el lenguaje de Python para trabajar con listas frente a la que ofrece para trabajar con cadenas de caracteres.

2. Adaptación del algoritmo

El esqueleto base del algoritmo se ha mantenido cambiando pequeños detalles en comparación con los que ha habido en el entorno. No obstante, estos han sido mucho más significativos en pos de hallar el objetivo principal del problema, y con él, la solución más óptima.

La inicialización del entorno y de las q-tablas es la misma que la utilizada en el proyecto inicial.

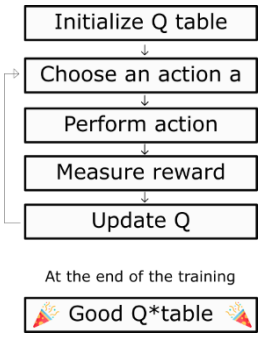
Los parámetros tienen la misma funcionalidad que la ya implementada con anterioridad, aunque estos han sido modificados según encajaban mejor con el resultado esperado. Especialmente, gamma ha sido el más modificado ya que un pequeño cambio en este suponía que las recompensas tuvieran una prioridad mucho más o mucho menos importante.

El comportamiento del algoritmo en sí es el mismo que el utilizado en el proyecto inicial. Primero, se crea el mapa inicial, ya sea el de prueba o uno aleatorio.

A continuación, utilizamos el método “reset” que ya empleamos en el proyecto inicial para reiniciar el algoritmo en cada mapa. A su vez, usamos nuestro método de modificar el mapa para ver en tiempo real cómo cambia este en la consola. Por otro lado, hemos mantenido la selección de la acción de forma aleatoria además de la utilización de la q-tabla para obtener las recompensas y estados. En general, la gran parte de esta implementación en concreto no ha sido modificada respecto al proyecto inicial.

Seguido a esto, encontramos un par de condiciones “if” que se encargan de dibujar correctamente el mapa, además de mostrar una ventana interactiva con el mapa (utilizando la librería de tkinter).

A continuación, encontramos la parte más importante: la línea de código que calcula el nuevo valor en la q-tabla de la acción que se ha elegido. Las actualizaciones que se han hecho en esta línea de código repercuten bastante en la ejecución del algoritmo. Tras varios experimentos, lo más óptimo que encontramos fue que este valor se calculara a partir del número actual de la q-tabla multiplicado por 0.9 (si esto no se hace, el algoritmo tiende a entrar en un bucle infinito en el que no avanza). A esta multiplicación se le suma la recompensa obtenida en el estado. Esta recompensa está ya multiplicada exponencialmente por gamma.



De esta forma, también evitamos la recreación del problema del bucle infinito con la recompensa (si la recompensa tuviera siempre el mismo valor, el algoritmo tiende a entrar en un bucle en el que solo maximiza la recompensa, obviando lo demás), pues hacemos que cuantos más pasos realice el algoritmo para alcanzar el final, menos valor tengan las recompensas que obtiene. Es decir, si el agente atraviesa una casilla de valor 3 mientras el algoritmo ejecuta la segunda acción, esta proporcionará casi su valor total mientras que, si el agente la atraviesa en la ejecución de la decimocuarta acción, esta recompensa se verá severamente reducida. Este cálculo evita, como se ha mencionado antes, que el algoritmo entre en un bucle infinito y no termine nunca de ejecutar la primera iteración.

El valor asignado a gamma es muy difícil de deducir ya que, dependiendo del mapa creado puede que le favorezca más uno u otro. Todo esto es debido a que depende del tamaño del mapa, de las casillas que se generan en el mapa, de cómo están distribuidas estas…

El conjunto de las condiciones enumeradas hace notar que esta sea la parte más delicada y al mismo tiempo más importante de la implementación llevada a cabo.

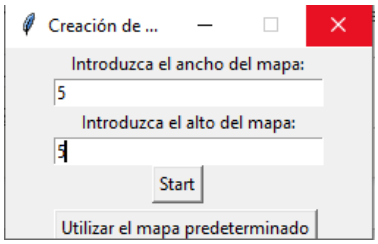
Para terminar con la ejecución del algoritmo, este calcula el nuevo estado a visitar por el agente y la nueva variable epsilon. Esta parte no ha sido modificada respecto a la ofrecida por el proyecto inicial.

3. Implementación Tkinter

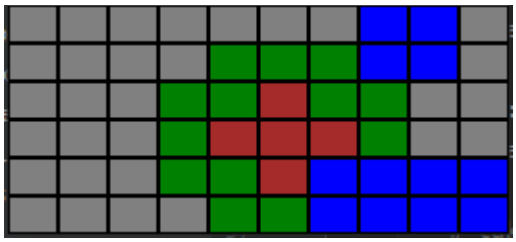
Con el fin de facilitar al usuario la comprobación de los resultados obtenidos por el algoritmo, se han implementado varios métodos encargados de construir una imagen del mapa creado además de mostrar la secuencia más óptima que ha encontrado este para resolver el problema. Esto se ha realizado gracias a Tkinter, un binding de la biblioteca gráfica Tcl/Tk del lenguaje Python. Tkinter se considera un estándar para las interfaces graficas de usuario (GUI) en Python [3].

En primer lugar, utilizando Tkinter hemos creado un pequeño formulario en el que el usuario puede añadir tanto el

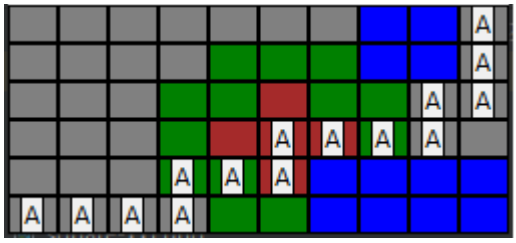
alto del mapa a crear como el ancho, además de facilitarle la oportunidad de seleccionar el mapa de prueba otorgado por el problema [4]. Una vez que el usuario presione el botón de 'Start' comenzará a ejecutarse el algoritmo teniendo en cuenta los datos introducidos.



A continuación, en los métodos de creación de los mapas, se han calculado las posiciones en X y Y que tienen los distintos valores que van a ser mostrados. Gracias a Tkinter se ha implementado una ventana emergente en la cual se dibujan cuadrados (estados) con distintos colores [5]. De esta forma, es mucho más fácil de ver y más intuitivo percibir lo que hace el algoritmo, especialmente si lo usa alguien que no ha realizado la implementación.



Además, en los métodos de modificación del mapa y de fin se ha implementado una función que sitúa una A (en representación del agente) en cada uno de los cuadrados (estados) que recorre nuestro algoritmo mediante acciones. De este modo, se nos enseña cuál es el camino recorrido por el algoritmo en la iteración más óptima.



La visualización del recorrido que es solución en un principio quedó implementada para cada una de las iteraciones que hacía el algoritmo, pero la potencia que necesitan los ordenadores para ir actualizando el mapa durante un periodo largo de tiempo es muy alta. Es por ello que, debido a esta limitación tecnológica, finalmente solo se traza el recorrido del algoritmo en su iteración más óptima.

Para poder llevar a cabo estos dos procesos de Tkinter (formulario y visión del mapa, junto a solución), hemos tenido que duplicar la implementación del algoritmo, pues cada proceso de Tkinter requería de un algoritmo para sí mismo.

III. EXPERIMENTOS

Para garantizar la correcta implementación del algoritmo, hemos llevado a cabo, durante bastante tiempo, la realización de diferentes pruebas con el fin de ajustar los parámetros del algoritmo. Con estas pruebas hemos comprobado que un pequeño cambio en un número de una ecuación del algoritmo puede suponer que este aprenda mucho peor. Los parámetros que intervienen en gran medida en la eficiencia del algoritmo son principalmente: el parámetro gamma y las recompensas de los estados y, en menor medida: los parámetros como el alto y el ancho de del mapa o las iteraciones que hace el algoritmo. Por tanto, nuestras pruebas se centran en actualizar el parámetro gamma y las recompensas, y a continuación observar cómo afecta al comportamiento del algoritmo.

1. Experimento: *No utilizar el parámetro gamma al calcular el nuevo valor de la tabla.*

   - **Valores alterados**:

   Gamma: nula.

   - **Análisis de los resultados**:

Al no utilizar el parámetro gamma para multiplicarlo por la recompensa obtenida en cada acción y, conseguir así que su valor se reduzca según se van realizando las acciones, encontramos que el algoritmo prioriza las casillas que tienen mejor recompensa. Esto provoca que, en la mayoría de las ocasiones, el algoritmo entre en un bucle infinito o casi infinito, lo que supone que el algoritmo lleve a cabo demasiadas acciones dentro de una misma iteración y que nunca acabe esta.

   - **Conclusión obtenida**:

Debemos de añadir un valor no nulo al parámetro de gamma obligatoriamente para que el algoritmo funcione correctamente.

2. Experimento: *El parámetro gamma se reduce muy rápidamente.*

   - **Valores alterados**:

   Gamma: gamma^x (siendo x un número muy alto)

   - **Análisis de los resultados**:

Al elevar el parámetro gamma a número muy alto, provocamos que, por cada iteración, este parámetro se reduzca demasiado. Por lo tanto, la recompensa obtenida se convierte en un número muy pequeño tras realizar pocas acciones. Tanto es así, que el propio algoritmo ignora las recompensas. Esto causa que en las últimas acciones antes de llegar al final del recorrido haya muy poca diferencia entre la recompensa que hay en una casilla u otra. En muchas ocasiones esto induce a que el algoritmo decida atravesar casillas con recompensas negativas puesto que la diferencia de recompensas es mínima, convirtiendo así ese recorrido en uno no óptimo.

   - **Conclusión obtenida**:

Debemos de controlar cómo de rápido se reduce el parámetro gamma de forma que, aunque al final del recorrido la recompensa vaya a ser menor, no llegue a convertirse en nula.

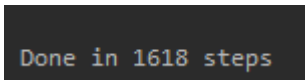
3. Experimento: *El parámetro gamma no se reduce.*

- **Valores alterados**:

Gamma: gamma (nunca altera su valor)

- **Análisis de los resultados**:

Si el parámetro gamma no se actualiza, nunca se reduce la recompensa al realizar varias acciones. Esto provoca que el algoritmo no tenga la necesidad de llegar al final lo antes posible y, en muchas ocasiones, entre en un bucle en el que recorre siempre las mismas casillas para obtener una recompensa alta, realizando así muchas acciones antes de completar el recorrido.



Por ejemplo, como vemos en la imagen si lo ejecutamos de esta forma se crean iteraciones de más de 1000 acciones hasta llegar a la casilla final.

- **Conclusión obtenida**:

Es totalmente necesario reducir el parámetro gamma para que el algoritmo pueda optimizar el número de acciones a realizar antes de alcanzar la meta.

4. Experimento: *Otorgar recompensas positivas a los estados.*

- **Valores alterados**:

Reward: valores entre 0 y 3, dependiendo de la casilla que se trate.

- **Análisis de los resultados**:

Volviendo al algoritmo utilizado en el proyecto inicial, recordamos que todas sus recompensas eran 0 salvo la casilla final, que era 1. Esto provocaba que la prioridad del algoritmo fuese alcanzar el final del recorrido obviando todo lo demás. Actualizando esta metodología valorizando estas recompensas imponemos al algoritmo a aprender que intente alcanzar el final lo antes posible, pero teniendo en cuenta la recompensa total y que, finalmente sea él el que decida cuál es el recorrido más óptimo para alcanzar el final.

- **Conclusión obtenida:**

Debemos de otorgar recompensas distintas según estados por los que pase el agente, aunque sin dejar de tener en cuenta cuáles son los valores que ayudan al correcto funcionamiento del algoritmo. Especialmente hay que tener en cuenta las casillas cuya recompensa es negativa y evitarlas a toda costa.

5. Experimento: *Otorgar recompensas muy altas.*

- **Valores alterados**:

Reward: valores muy elevados

- **Análisis de los resultados**:

Al hacerle tanto hincapié a las recompensas y otorgarles una prioridad por encima de la adecuada, podemos provocar que el algoritmo decida entrar en un bucle para conseguir la máxima recompensa posible obviando tanto la posibilidad de un recorrido más óptimo como el hecho de terminar el recorrido en sí.

- **Conclusión obtenida**:

Es muy importante calcular bien cuáles son las recompensas más óptimas, especialmente no darles demasiada prioridad ya que pueden hacer de nuestro algoritmo un algoritmo inútil.

6. Experimento: *Añadir recompensas negativas por las zonas que quieres evitar.*

- **Valores alterados**:

Reward: añadimos valor negativo en las casillas que queremos evitar.

- **Análisis de los resultados**:

Al añadir recompensas negativas a las casillas que queremos evitar, en este caso, las casillas de agua, nos encontramos con que el algoritmo hace todo lo posible por evitarlas. Especialmente si le otorgamos a estas casillas una recompensa negativa muy alta.

- **Conclusión obtenida**:

Es útil utilizar la recompensa negativa para evitar las casillas azules, aunque se debe comprobar cuál es el valor más apropiado al igual que pasaba con las recompensas positivas.

Es importante controlar el valor de la recompensa negativa ya que, si utilizamos un valor muy alto, puede provocar que el algoritmo no pase nunca por esas casillas y el recorrido no pueda completarse de forma óptima.

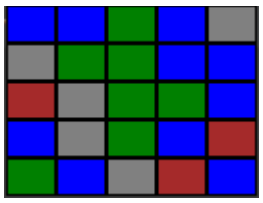
7. Experimento: *Crear un mapa donde obligatoriamente para llegar al final hay que pasar por casillas con recompensa negativa.*

- **Valores alterados**:

-

- **Análisis de los resultados**:

Por ejemplo, para verlo de forma más gráfica nos referimos a un mapa como este:



El algoritmo tiene que cruzar por las casillas con recompensa negativa ya que necesita alcanzar el final del recorrido. Algunas veces esto produce que el algoritmo ejecute más acciones de las óptimas para así conseguir sumar valores positivos a la recompensa total.

- **Conclusión obtenida**:

La recompensa negativa no cohíbe al algoritmo de atravesar un cierto estado. Esto nos resulta útil de cara a la creación de mapas aleatorios que contengan muchas casillas con recompensa negativa o que las tengan antes de alcanzar el final, como el mostrado anteriormente.

8. Experimento: *Cambiar los epochs (iteraciones) que hace el algoritmo.*

- **Valores alterados**:

Epochs: aumentamos las iteraciones del algoritmo.

- **Análisis de los resultados**:

En algunas ocasiones con 50 iteraciones, el algoritmo ofrecía un resultado muy lejos de la solución óptima. Es por ello por lo que decidimos aumentar el número de iteraciones a 100, disminuyendo la probabilidad de que esto ocurra, ya que sobre la iteración 60-70 ya el algoritmo ha aprendido el recorrido más óptimo.

- **Conclusión obtenida**:

Es mejor aumentar las iteraciones, aunque esto haga que el algoritmo termine más tarde de ejecutarse. A cambio conseguimos una mejor solución.

9. Experimento: *Cambiar el tamaño del mapa y crearlo de forma aleatoria.*

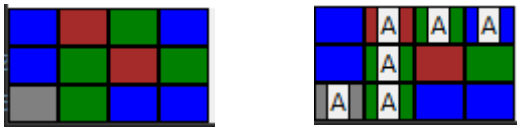
- **Valores alterados**:

Self.height: aumentarlo o disminuirlo.

Self.weight: aumentarlo o disminuirlo.

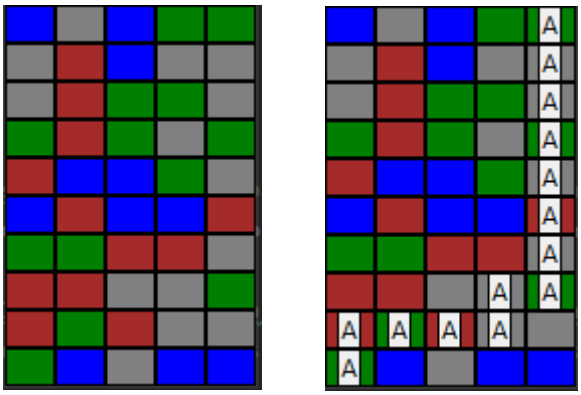
- **Análisis de los resultados**:

En primer lugar, se hizo una prueba con un mapa pequeño para comprobar si el algoritmo funcionaba bien.



Como podemos comprobar, el algoritmo evita las casillas azules (con recompensa negativa) e intenta llegar al final de la forma más rápida posible sumando las recompensas más altas. En resumen, vemos como el algoritmo actúa de forma correcta.

A continuación, probamos con un mapa con mayor dimensión que provoque más trabajo al algoritmo a la hora de devolver la solución más óptima.



Comprobamos que vuelve a evitar las casillas azules (recompensa negativa) y encuentra el único recorrido por el que puede atravesar las casillas sin reducir su recompensa total. Además, lo consigue utilizando el menor número de acciones posibles.

Tras esto, continuamos probando distintos mapas con diferentes características. Según vamos probando estos mapas nos percatamos de que es difícil que los parámetros que hemos elegido correspondan perfectamente a los esperados en todos. No obstante, en la mayoría de los casos, el algoritmo realiza un recorrido bastante semejante al más óptimo.

- **Conclusión obtenida**:

Hay que seleccionar unos parámetros que concuerden con la mayoría de los mapas que vamos a generar y no solo para el mapa base que es de 10x6. Como se ha demostrado en los ejemplos descritos anteriormente, hemos utilizador los valores que, según los diferentes experimentos realizados, nos han proporcionado resultados bastante satisfactorios.

10. Experimento: *Reducir la recompensa en cada acción que ejecute el algoritmo.*

- **Valores alterados**:

Reward:reducción según el número de acciones ejecutadas.

- **Análisis de los resultados**:

Al reducir la recompensa obtenida por acción, el algoritmo interpreta que, cuantas más acciones ejecute, menos valor de recompensa adquiere y que, por tanto, debe minimizar el recorrido a seguir.

- **Conclusión obtenida**:

Podemos utilizar esta técnica, junto al aumento de gamma de forma exponencial, para reducir las acciones que ejecuta el algoritmo.

IV. CONCLUSIONES

Concluimos la resolución del problema dado con la seguridad suficiente de que el algoritmo funciona como debe, ya que hemos corregido tanto la posibilidad de que el algoritmo entre en bucle para maximizar las recompensas, como que no termine el recorrido proporcionado. No obstante, cada mapa tiene asociadas una serie de propiedades que dificultan el correcto funcionamiento del algoritmo. Esto es así debido a que usamos un método de creación de mapa totalmente aleatorio, por lo que no sabemos a ciencia cierta cómo se va a comportar el algoritmo con este, aunque, por lo general, se realiza correctamente.

Además, garantizamos que se pueda comprobar su funcionamiento a través de la visualización del mapa a recorrer y su posterior solución, lo que es más visual y práctico de cara al usuario sin experiencia alguna en programación.

La realización de este problema, con su correspondiente implementación, ha sido posible, en gran parte, gracias al documento de Q-Learning ofrecido por la asignatura, ya que nos ha ayudado a entender cómo funciona el algoritmo de aprendizaje por refuerzo en profundidad y con detalles. Además, la visualización del ejemplo que se presenta en este a través de su implementación en Python nos ha permitido comprobar cómo el algoritmo aprendía.

La introducción a este tipo de problemas nos ha proporcionado cierta curiosidad acerca del funcionamiento de algoritmos de aprendizaje por refuerzo como, por ejemplo, el usado en videojuegos.

La forma de observar este problema tal y como si se tratase de un videojuego nos ha facilitado su implementación, ya que es un ámbito que tratamos con frecuencia y que se nos hace amigable. Esto nos ha ayudado también a la hora de visualizar cómo sería una interfaz gráfica de usuario, y es por ello que la hemos implementado.

Finalmente, este problema nos ha resultado muy interesante, e incluso nos ha animado a investigar más acerca de Q-Learning, aprendizaje automático y cómo funciona.

V. BIBLIOGRAFÍA

[1] https://medium.com/datadriveninvestor/math-of-q-learning-python-code-5dcbdc49b6f6 utilizado para desenvolverse con el aprendizaje por refuerzo además de resolver el objetivo inicial.

[2] https://planetachatbot.com/introducci%C3%B3n-al-aprendizaje-por-refuerzo-f910d669d077 utilizado para establecer y entender mejor cuáles son las condiciones que hay que contemplar para que el algoritmo se ejecute correctamente. En concreto, qué tipo de recompensas proporcionar al algoritmo.

[3] https://www.tutorialspoint.com/python3/python_gui_programming.htm utilizado en general para interpretar Tkinter.

[4] https://es.stackoverflow.com/questions/160008/datos-introducidos-mediante-formulario-hecho-en-tkinter-no-se-insertan-en-la-bas utilizado para entender cómo introducir un formulario utilizando Python y Tkinter. Se ha adaptado el código a nuestras necesidades.

[5] https://stackoverflow.com/questions/14349526/creating-a-game-board-with-python-and-tkinter utilizado para entender cómo representar un mapa dinámico utilizando Python. Este aporte ha servido únicamente como inspiración.