PEC 3 – Aprendizaje en vídeojuegos

Presentación

En esta PEC trabajaremos con los conceptos relacionados con aprendizaje que se han visto en los materiales, especialmente con el aprendizaje por refuerzo.

Competencias

En este enunciado se trabajan las siguientes competencias generales de máster:

- Capacidad para aplicar el pensamiento creativo y generar nuevas soluciones.
- Capacidad para el aprendizaje autónomo.
- Capacidad para dominar las herramientas aplicables al desarrollo de videojuegos, según las tendencias tecnológicas.
- Capacidad para el uso efectivo de los lenguajes de programación y las metodologías para el desarrollo de videojuegos.
- Capacidad para analizar y diseñar los elementos y los principios de funcionamiento de un videojuego.
- Capacidad para diseñar los componentes asociados con la creación de una experiencia de juego.
- Capacidad para representar elementos visuales y sus interacciones de manera eficiente.
- Entender la relación existente entre la Inteligencia Artificial y los otros componentes del videojuego.
- Saber cómo mover los elementos en un mundo virtual, tanto de forma individual como colectiva.

Objetivos

Al acabar esta PEC el alumno sabrá aplicar aprendizaje por refuerzo a juegos sencillos, así como a modificar parámetros para adecuarlos al tipo de juego o agente que se desee conseguir.

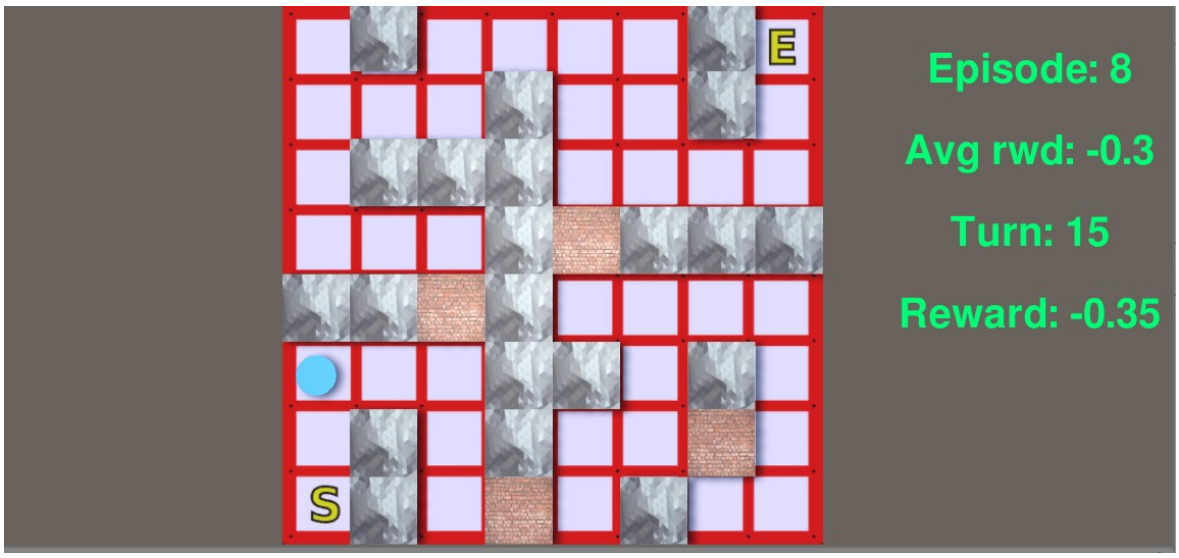
Descripción de la PEC a realizar

Actividades

Para desarrollar las actividades hay que leer los conceptos de aprendizaje por refuerzo (*reinforcement learning, RL*) de los materiales (pág. 62).

Instalación y ejecución del entorno de pruebas

El objetivo de esta actividad es la instalación del paquete de Unity titulado “Maze”, que contiene una escena, los *scripts* y los objetos necesarios para el juego que se ve en la figura:



El juego consiste en conseguir que el jugador (cilindro azul), partiendo de la casilla S, llegue a la salida del laberinto, casilla E. El jugador puede:

- Moverse en las 4 direcciones cardinales, es decir no puede moverse en diagonal
- Caminar por las casillas libres
- Destruir las casillas de ladrillo, que entonces se convierten en casillas libres

Las casillas de piedra no se pueden destruir ni se puede caminar sobre ellas. Tampoco se puede salir del tablero, como es lógico.

El jugador es controlado por la IA, pero a diferencia de otros planteamientos de IA, en este caso no le damos ninguna indicación de cómo resolver el laberinto,

sino que simplemente vamos a dejar que la IA vaya aprendiendo con la práctica: jugando, y viendo cuál es el resultado de cada acción. Es lo que se conoce como **aprendizaje por refuerzo**.

Para empezar a trabajar:

- Cread un proyecto nuevo e importad el paquete *Maze*.
- Comprobad que el juego funciona correctamente.
- El objeto *GameManager* ofrece una serie de parámetros de configuración del juego; probad a reducir el valor "Turn Duration" para que el juego vaya más rápido y se aprecie antes el aprendizaje conseguido.

Aplicación de RL en este juego

En primer lugar, habréis observado que el juego ya incluye el código para hacer el RL, y de hecho si lo ejecutáis notaréis que el agente va aprendiendo a recorrer el laberinto y alcanzar la salida. El objetivo de esta PEC no es que programéis código de aprendizaje sino que entendáis el código dado, cómo se ha aplicado al juego, cómo se puede configurar para conseguir otros comportamientos, y finalmente que apliquéis algún cambio al proyecto.

En el aprendizaje por refuerzo se suele dividir el aprendizaje en episodios de entrenamiento (*episodes*), lo que en el juego corresponde con las "partidas", que simplemente son secuencias de movimientos hasta que el jugador alcanza la salida o bien ha utilizado 100 turnos (movimientos del jugador).

El agente solo puede realizar cuatro acciones: moverse hacia la izquierda, hacia la derecha, hacia arriba o hacia abajo. No puede quedarse quieto. Las posibles acciones y sus recompensas son:

- Moverse a una casilla libre: -0.05
- Moverse a una casilla de ladrillo (y por tanto romperla): -0.2
- Intentar moverse a una casilla de piedra (no se ejecuta el movimiento, simplemente choca): -0.7
- Intentar salir de los límites del tablero (tampoco se ejecuta): -0.8
- Alcanzar la salida: +10

El código del juego consta de dos *scripts*: uno que gestiona el juego y coordina los diferentes elementos, y otro que gestiona el aprendizaje que lleva a cabo el

agente. El código tiene abundantes comentarios para que podáis seguir lo que hace.

Recordad que en RL el agente va aprendiendo la recompensa que obtendrá si, en el estado S del juego, ejecuta la acción A. Poco a poco irá viendo cuál es la acción más fructífera en cada estado.

El estado de este juego se define por:

1. La posición del jugador. Hay 8x8=64 casillas, por tanto 64 posiciones diferentes. Se podría optimizar ya que no es posible colocarse sobre algunas casillas (piedra)
2. El estado intacto/destruido de los bloques de ladrillo. Hay 4 bloques de ladrillo y cada uno tiene dos estados posibles, por tanto hay $2^4 = 16$ combinaciones posibles

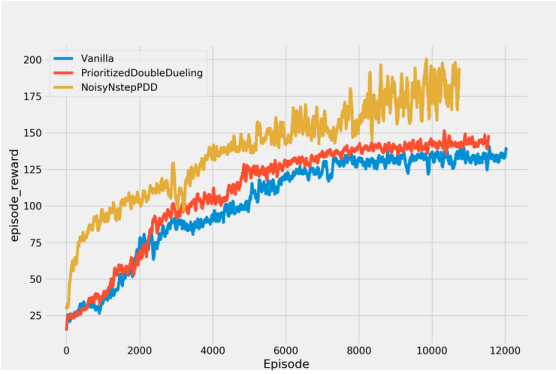
Los estados posibles del juego son todas las combinaciones de ambos factores, por tanto hay 64x16=1024 estados posibles del juego (algunos no se darán nunca pero no hay problema, solo que usan memoria).

Así, la tabla de aprendizaje (tabla Q) es una tabla de dimensiones estados x acciones, es decir de 1024 x 4 elementos, en la que se va almacenando la recompensa obtenida al tomar la acción *j* (columna) cuando el juego se encontraba en el estado *i* (fila).

De esa manera, poco a poco el agente va eligiendo las acciones mejor recompensadas dado el estado actual.

Actividad 1: monitorización del aprendizaje

La principal herramienta para entender lo que ocurre durante el proceso de entrenamiento son las curvas de aprendizaje, en las que se muestra la recompensa obtenida en cada episodio, y así se puede ver si el agente va mejorando, si se ha estancado, etc.



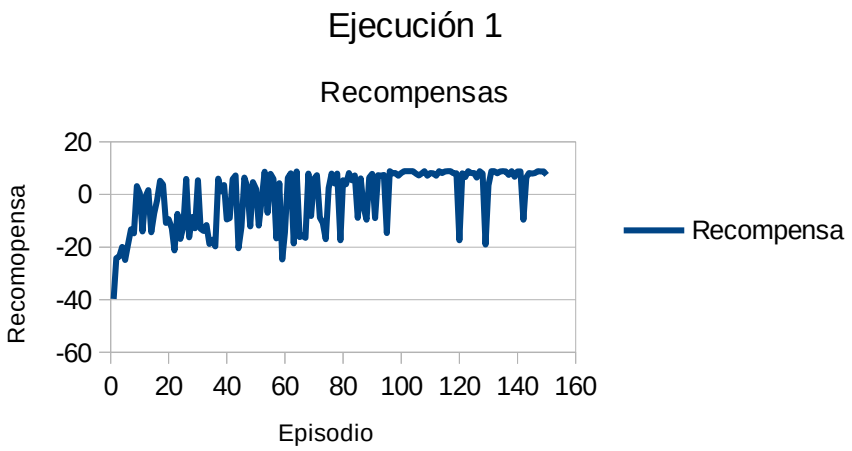
Realiza las siguientes tareas:

a) Modifica Maze para poder visualizar la curva de aprendizaje del agente. Elige el número de episodios. Recuerda que puedes acelerar los turnos para poder ejecutar más episodios. Sugerencia: guarda los pares #episodio, recompensa en un archivo y a continuación utiliza una hoja de cálculo para leerlos y generar la gráfica.

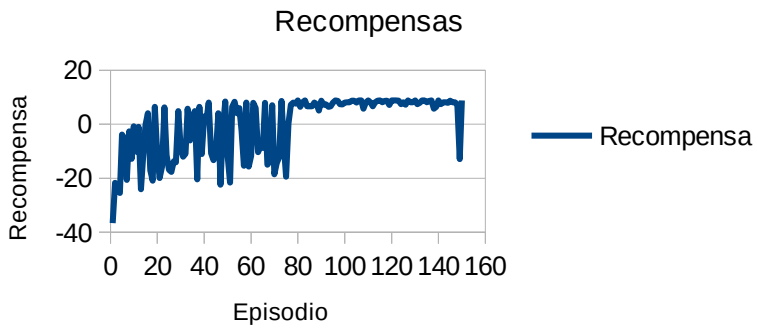
b) Analiza la curva obtenida. Puedes ejecutar varias veces el programa desde cero y comparar los resultados.

c) ¿Hay subidas y bajadas en la recompensa a lo largo de los episodios? Si es así, ¿a qué crees que es debido?

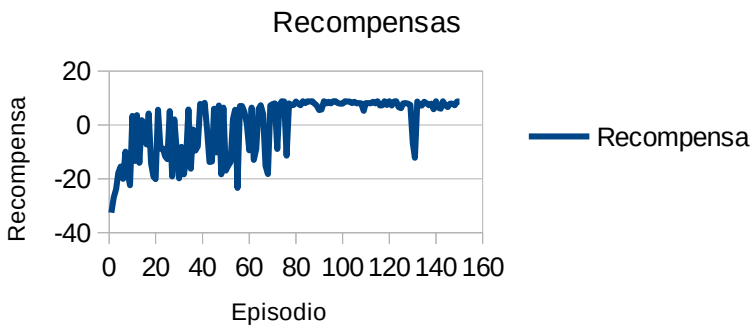
Curvas de aprendizaje:



Ejecución 2



Ejecución 3



Ejecución 4

Análisis de las curvas:

En las ejecuciones realizadas para este análisis se ha utilizado la configuración inicial del unitypackage maze, y el número de episodios elegido es 150. En las curvas de aprendizaje obtenidas se observa que, después de unos 80-100 episodios, el agente ha aprendido a llegar a la salida del laberinto. Esto se debe a que se ha identificado la mejor acción en un número suficiente de estados como para que el agente pueda llegar al final de la salida en casi todos los episodios siguientes.

En cuanto a las grandes subidas y bajadas antes de que la curva de aprendizaje se estabilice, se deben a la diferencia de recompensa de llegar o no llegar al final del laberinto antes de que se acabe el episodio. Gracias a estas subidas y bajadas se pueden identificar los episodios en los que el agente ha llegado a la salida por las recompensas obtenidas mayores de 0.

Finalmente en cuanto a las grandes bajadas que suceden después de que la curva de aprendizaje se estabiliza, se deben a la probabilidad de que el agente tome acciones aleatorias. Estas acciones aleatorias llevan al agente a un nuevo estado sin explorar haciendo que éste no llegue al final del laberinto en algunas ocasiones.

Código:

```
// Save reward in file Actividad 1
if (episode <= 150 && episode > 0)
{
    using (System.IO.StreamWriter file =
        new  System.IO.StreamWriter(@"C:\Users\manur\Desktop\UOC\IAV\PEC3\
Datos\data.csv", true))
    {
        file.WriteLine($"{episode};{episodeReward}");
    }
}
if (episode == 150)
{
    using (System.IO.StreamWriter file =
        new  System.IO.StreamWriter(@"C:\Users\manur\Desktop\UOC\IAV\PEC3\
Datos\time.txt", true))
    {
        file.WriteLine($"Tiempo de ejecución: {Time.time}");
    }
    Debug.Log("Data saved.");
}
```

Para poder visualizar las curvas se han escrito los pares episodio, recompensa en un archivo csv. Una vez guardados, los pares se leen en una hoja de cálculo de excel y se crea un diagrama para visualizar la curva de aprendizajes. Además el tiempo de ejecución es escrito en un archivo de texto una vez escrito el último par episodio, recompensa seleccinado.

Actividad 2: límite de turnos

En la versión de Maze proporcionada hay un máximo de turnos por episodio, de forma que si el agente no alcanza la salida en ese número de turnos se acaba el episodio con la recompensa que tuviera hasta ese momento.

Elimina ese límite y analiza el efecto en el aprendizaje. Ten en cuenta las curvas de aprendizaje y también el tiempo de ejecución requerido, ya que sin límite de turnos cada episodio puede durar mucho más.

Indica cuál de las dos estrategias consideras más adecuada.

Continúa la práctica usando el número de turnos máximo.

Código:

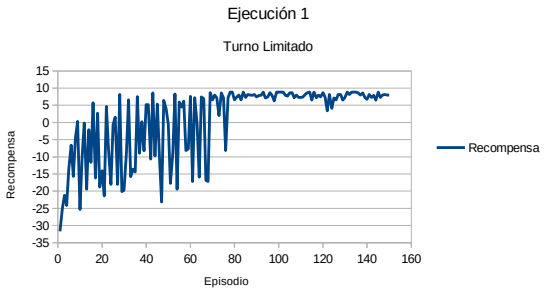
```
bool episodeEnds = goalReached //|| turn >= episodeTurns Actividad 2
```

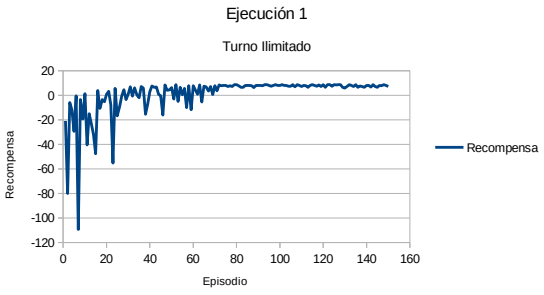
Para quitar el límite de turnos se ha quitado la condición de los limites de turno a la hora de determinar si el episodio ha terminado.

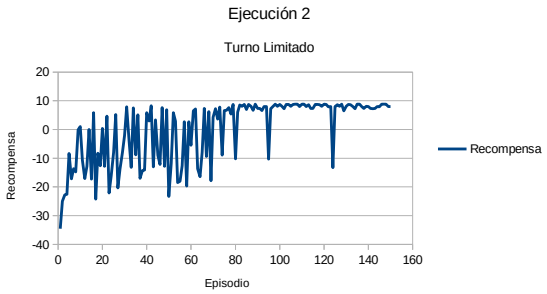
Tiempos de ejecución para 150 episodios con un límite de turnos de 100:

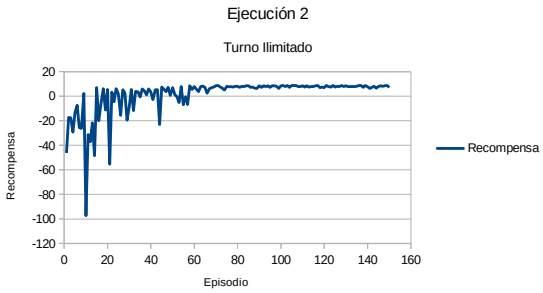
- Tiempo de ejecución 1 limitado: 93,48154 segundos
- Tiempo de ejecución 2 limitado: 98,6505 segundos
- Tiempo de ejecución 1 sin límite: 124,0468 segundos
- Tiempo de ejecución 2 sin límite: 121,966 segundos

Curvas de aprendizaje:









Análisis:

En estas ejecuciones se ha vuelto a utilizar la configuración inicial de unitypackage maze. Las curvas de aprendizaje de la izquierda son de ejecuciones en las que el límite de turnos es de 100. La curvas de la derecha, en cambio, corresponden a ejecuciones sin límite de turnos.

Si se observan los valores de las recompensas, se puede apreciar que el rango de valores es mayor en las ejecuciones sin límite de turnos. Las recompensas de las ejecuciones con los turnos limitados se encuentran en el intervalo (-40, 10) aproximadamente. En cambio, las recompensas de las ejecuciones sin límite se encuentran en el intervalo (-120, 10) aproximadamente.

A pesar de que en todos los episodios de las ejecuciones de turnos ilimitados el agente llega a la salida, algunas de las recompensas son mucho peores que las de las ejecuciones de turnos limitados. Esto se debe a que el valor absoluto de la acumulación de recompensas negativas es mucho mayor que la recompensa recibida al llegar a la salida, de lo que concluimos que en estos episodios se han realizado una gran cantidad de turnos antes de encontrar la salida.

Comparando la evolución de las curvas se observa que en el caso de las ejecuciones sin límite de turno se necesitan menos episodios para que la curva de aprendizaje se estabilice. Y comparando los tiempos de ejecución se observa que la ejecución sin límite de turno tarda más tiempo en realizar los 150 episodios. Esto es debido a que sin límite de turnos el agente recorre más estados y lleva a cabo más acciones en los episodios que con los turnos limitados. Por ello, se tarda más tiempo en terminar los episodios y se llena antes la tabla de aprendizaje.

En cuanto a cuál de las estrategias es más adecuada en este caso, opto por la estrategia sin límite de turnos, porque llena mejor la tabla de aprendizaje evitando que no sucedan bajadas una vez estabilizada la curva de aprendizaje.

Actividad 3: ladrillos

Como primer cambio en el juego y para profundizar en el cálculo del espacio de estados, realiza las siguientes tareas:

a) Añade otro bloque de tipo ladrillo en la posición que desees. Ajusta el cálculo del número de estados posibles y el tamaño de la tabla Q.

b) Comprueba que el agente resuelve esta versión del laberinto.

c) Compara la curva de aprendizaje actual con la obtenida en la actividad 1.

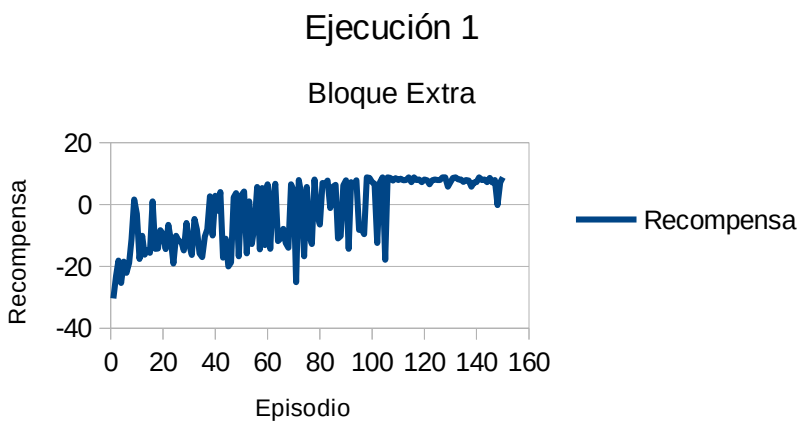
d) ¿Cómo cambiaría el tamaño de la tabla Q a medida que se añadieran nuevos bloques de ladrillo? ¿Y si se añadieran bloques de roca?

Código:

```
// Actividad 3
board_int = new int       [ROWS, COLS] {
    {0, 2, 0, 1, 0, 2, 0, 0},
    {0, 2, 0, 2, 0, 0, 1, 0},
    {0, 0, 0, 2, 2, 0, 2, 0},
    {2, 2, 1, 2, 0, 0, 0, 0},
    {0, 0, 0, 2, 1, 2, 2, 2},
    {1, 2, 2, 2, 0, 0, 0, 0},
    {0, 0, 0, 2, 0, 0, 2, 0},
    {0, 2, 0, 0, 0, 0, 2, 3}
};
```

Para añadir un bloque de ladrillo se ha puesto el valor 1 en una de las posiciones de board_int. De esta forma el propio algoritmo toma en cuenta los cambios en las dimensiones de la tabla Q y en el cálculo del estado actual.

Curva de aprendizaje:



Análisis:

En comparación con las curvas de aprendizaje de la actividad 1, al haber aumentado el número de estados posibles, el agente tarda más episodios en llenar la tabla de aprendizaje y estabilizar la curva de aprendizaje.

Añadiendo el nuevo bloque de ladrillos, el número de estados se duplica y pasa de 1024 a 2048. Así que al añadir un nuevo bloque de ladrillos el número de estados/filas de la tabla Q se duplica.

Siendo n el número de bloques de ladrillos actual:

Número de estados = $64*2^n$

Número de estados añadiendo un bloque de ladrillos = $64*2^{n+1} = (64*2^n)*2$

Por otro lado añadir un bloque de roca no tienen ningún efecto sobre el tamaño de la tabla Q.

Actividad 4: enemigo

Se introduce un nuevo elemento: un enemigo que se mueve aleatoriamente (un paso cada turno), de manera que si el jugador y el enemigo chocan (están en la misma casilla), el jugador muere, termina el episodio y recibe una recompensa negativa (a determinar por vosotros).

El enemigo no puede destruir bloques de ladrillo y solo se puede mover por las casillas libres.

Para resolver esta actividad debes:

- Programar el enemigo
- Explicar cómo cambia el cálculo de estados y modificar el juego para que lo tenga en cuenta
- Entrenar el agente en este nuevo entorno y observar el aprendizaje. Analiza las diferencias respecto al entorno anterior

Incluye un vídeo del resultado.

Código enemigo:

```
// The agent starts at position 0, 0
agentRow = 0;
agentCol = 0;
// The enemy starts at position 7, 0
enemyRow = 7;
enemyCol = 0;

// Renew the goal reached flag and player reached flag
goalReached = false;
playerReached = false;

agent.gameObject.transform.position = new Vector3 (agentCol, 0.0f,
agentRow);
enemy.transform.position = new Vector3(enemyCol, 0.0f, enemyRow);
```

En la función StarEpisode de la clase Game, la posición del enemigo inicia en 7,0 y se reinicia el flag playerReached que determina si el enemigo y el agente han chocado.

```
// 2) The agent performs the action and the reward is computed
float reward   = MoveAgent(action);
if (agentRow == enemyRow && agentCol == enemyCol && !playerReached)
{
        reward += deathReward;
        playerReached = true;
}
// The enemy perform random action
MoveEnemy(UnityEngine.Random.Range(0,4));
if (agentRow == enemyRow && agentCol == enemyCol && !playerReached)
{
        reward += deathReward;
        playerReached = true;
}
```

En la función NewTurn de la clase Game, el enemigo se mueve después del agente. Además se comprueba después de cada movimiento si el enemigo y el agente han chocado, y en caso de que hallan chocado se aplica la recompensa deathReward y se activa el flag playerReached.

```
float MoveEnemy(int action)
{
// 1) Determine destination cell
int destRow = enemyRow;
int destCol = enemyCol;

switch (action)
{
case 0:
        destCol++;  // Right
        break;

case 1:
        destRow++;  // Up
        break;

case 2:
        destCol--;  // Left
        break;

case 3:
        destRow--;  // Down
        break;
}

// 2) Discard out-of-limits attempts; do not move and end action
if (destCol < 0 || destRow < 0 || destCol >= COLS || destRow >= ROWS)
```

```
{
        return -1;
}


// 3) Detect destination cell type, act accordingly
int destCell = board_int[destRow, destCol];

switch (destCell)
{
case 0:         // Walkable
        enemyRow = destRow;
        enemyCol = destCol;
        break;

case 1:         // Breakable:
        if (!board_go[destRow, destCol].activeSelf)
        {
                // If block broken move
                enemyRow = destRow;
                enemyCol = destCol;
        }
        break;
case 2:         // Hard wall: do not move
        break;

case 3:         // Goal reached
        enemyRow = destRow;
        enemyCol = destCol;
        break;
}

// Execute the movement
enemy.transform.position = new Vector3(enemyCol, 0.0f, enemyRow);
return 0;
}
```

La función para mover al enemigo (MoveEnemy) se ha creado en la clase Game. MoveEnemy es igual que la función para mover el agente (MoveAgent), salvo por algunas modificaciones. El enemigo no puede romper bloques de ladrillo, así que solo se mueve a este tipo de casillas cuando el bloque de ladrillo esta roto. Además puede moverse a cualquier casilla caminable y a la casilla del final del laberinto. Por último, a diferencia de MoveAgent, MoveEnemy no calcula ningún tipo de recompensa.

Cálculo del número de estados:

Añadiendo el enemigo, ahora hay que tener en cuenta sus posiciones posibles como se ha hecho con las del agente. Por lo tanto, se añaden las 64 posiciones posibles del enemigo al càlculo del número de estados:

Número de estados = estados de bloques de ladrillo * posiciones posibles agente * posiciones posibles enemigo = 16*64*64 = 65.536

Código del cálculo del número de estados:

Cambio realizado en función GameSetup de la clase Agent

```
states      = rows * cols * rows * cols * Mathf.FloorToInt(Mathf.Pow(2, breakables));
// con enemigo 65.536 estados pow(2, rows*cols)
```

Cálculo del estado actual sin enemigo:

Para calcular el estado actual en la versión sin enemigo tenemos los siguientes elementos:

- Estados bloques de ladrillos: Los estados posibles de los bloques de ladrillo interpretados como una array de valores binarios donde si el bloque está roto el valor es 1 y si no lo está es 0. Este array binario es convertido en número entero.
- Posiciones posibles del agente: Columnas * Filas = 8 * 8 = 64
- Posición actual del agente: La posición actual del agente se convierte en un único número entero sumando su columna actual y la multiplicación entre su fila actual y el número de columnas del tablero.

Posición actual del agente = fila actual * número de columnas + columna actual

Donde “fila actual” ϵ [0,8), “número de columnas” = 8, y “columna actual” ϵ [0,8)

El cálculo del estado actual es el siguiente:

Estado actual = estado bloques de ladrillos * posiciones posibles del agente + posición del agente = estado bloques de ladrillos * 64 + posición del agente

Donde “estado bloques de ladrillos” ϵ [0,15) y “posición del agente” ϵ [0,63)

Consiguiendo que “Estado actual” sea un número del intervalo [0, 1024) haciendo posible recorrer todos los estados.

Cálculo del estado actual con enemigo:

Para calcular el estado actual en la versión se añaden dos elementos:

- Las posiciones posibles del enemigo que son las mismas que las del agente. Posiciones posibles enemigo = columnas * filas = 8*8 = 64
- La posición del enemigo que se convierte en un único igual que la del agente.

Posición del enemigo = fila actual * número de columnas + columna actual

Donde "fila actual" ϵ [0,8), "número de columnas" = 8, y "columna actual" ϵ [0,8)

El cálculo del estado actual es el siguiente:

Estado actual = estado bloques de ladrillos * posiciones posibles del agente * posiciones posibles del enemigo + posición del agente * posiciones posibles del enemigo + posición del enemigo =

= estado bloques de ladrillos * 64 * 64 + posición del agente *64 + posición del enemigo =

= (estado bloques de ladrillos * 64 + posición del agente) *64 + posición del enemigo

Donde "estado bloques de ladrillos" ϵ [0,15) y "posición del agente" y "posición del enemigo" ϵ [0,63)

Consiguiendo que "Estado actual" sea un número del intervalo [0, 65.536) haciendo posible recorrer todos los estados.

Código del cálculo del estado actual:

El cálculo del estado actual se a modificado como se ha explicado en la función ComputeState de la clase Game.

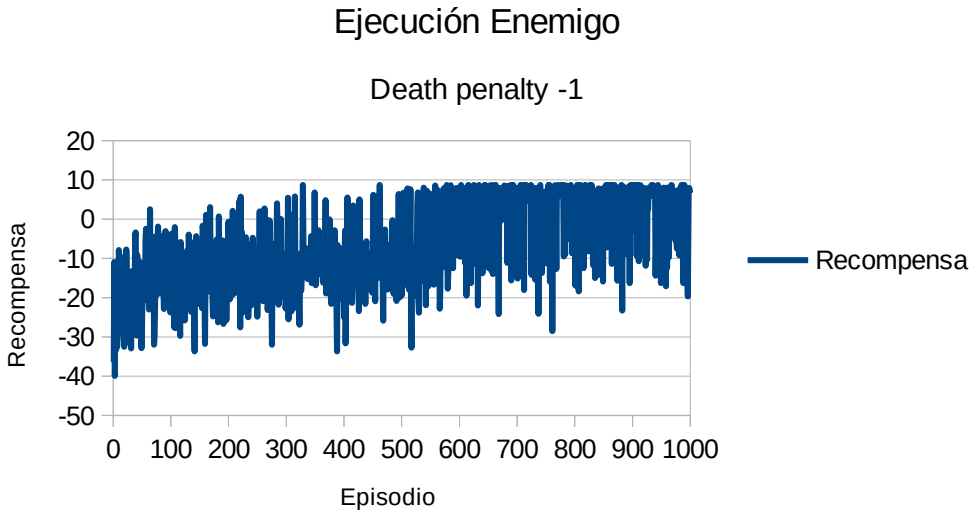
```
private int ComputeState() {

        UpdateBreakables();

        int stateBreakables = System.Convert.ToInt32 (breakables, 2);
        int agentPosition   = agentRow*COLS + agentCol;
        int enemyPosition = enemyRow*COLS + enemyCol;

                int  state  =  stateBreakables*ROWS*COLS*ROWS*COLS  +
agentPosition*ROWS*COLS+ enemyPosition;
        return state;
}
```

Curva de aprendizaje del nuevo entorno:



Tiempo de ejecución: 784,6326 segundos = 13 minutos y 4,6326 segundos

Análisis:

A diferencia del entorno anterior la curva de aprendizaje tiene muchas más bajadas y subidas, ya que se ha añadido una nueva recompensa negativa cuando el agente choca con el enemigo.

Observando la evolución de la curva se puede observar como, a partir del episodio 600, la curva comienza a estabilizarse. Aunque el agente sigue sin alcanzar la salida en algunas ocasiones.

Se aprecia que en este nuevo entorno el agente tiene muchos más estados que llenar, ya que después de comenzar a estabilizarse la curva aparecen muchos picos negativos por no ser capaz de llegar a la salida.

En comparación con el entorno anterior al agente le cuesta mucho más aprender a llegar a la salida del laberinto.

Actividad 5: visualización

En esta actividad visualizaremos la tabla Q de forma intuitiva. Para conseguirlo simplificaremos un poco el juego. Realiza las siguientes tareas:

a) Elimina los bloques de ladrillo del juego (deja casillas vacías en su lugar)

b) Modifica la codificación de estados para ajustarla a las nuevas reglas

c) Sobre cada casilla dibuja una flecha o equivalente que indique la dirección preferida por el agente si se encontrara en esa casilla, es decir la dirección (acción) con valor más alto en la tabla Q en la fila correspondiente a esa casilla. Estas flechas irán cambiando a medida que el agente aprenda, así que se tendrían que actualizar (cada turno o cada episodio)

Incluye un paquete de Unity con tu proyecto y un vídeo del resultado.

Flechas:

Para representar las flechas en la escena se ha creado el prefab de un sprite en forma de triangulo que se coloca sobre las casillas vacías apuntando en la dirección más deseada por el agente en el turno actual.

Código:

```
board_int = new int        [ROWS, COLS] {
        {0, 2, 0, 0, 0, 2, 0, 0},
        {0, 2, 0, 2, 0, 0, 0, 0},
        {0, 0, 0, 2, 2, 0, 2, 0},
        {2, 2, 0, 2, 0, 0, 0, 0},
        {0, 0, 0, 2, 0, 2, 2, 2},
        {0, 2, 2, 2, 0, 0, 0, 0},
        {0, 0, 0, 2, 0, 0, 2, 0},
        {0, 2, 0, 0, 0, 0, 2, 3}
};
```

Se cambian los bloques de ladrillo por casillas vacías, cambiando los valores 1 de board_int por valores 0.

```
public void GameSetup(int nRows, int nCols) {

// Number of states has two components:
// a) Position of the agent on the board (rows x cols)
// b) State of the breakables (intact/broken) -> 2^# of breakables
//  The number of states is the product of a) and b) (all possible
combinations)

rows        = nRows;
cols        = nCols;
```

```
states = rows * cols;

// 4 actions in this game: right, up, left, down
actions = 4;
qTable  = new float[states, actions];

for (int i = 0; i < states; i++)
        for (int j = 0; j < actions; j++)
                qTable[i, j] = 0.0f;

}
```

Se ha modificado la función GameSetup de la clase Agente, quitando de los parámetros el número de bloques y ajustando el cálculo del número de estados.

Número de estados = Posiciones posibles del agente = Filas * Columnas =

= 8 * 8 = 64

```
private int ComputeState() {

        int agentPosition   = agentRow*COLS + agentCol;

        int state = agentPosition;
        return state;
}
```

Se ha modificado el cálculo del estado actual de la función ComputeState teniendo solamente en cuenta la posición del agente, que es transformada en un número del intervalo [0, 64).

```
void FillBoardArrow()
{
// 0: rigth; 1: up; 2: left; 3:

// Instantiate the arrows
// NOTE: rows -> movement along z; cols -> movement along x
board_arrow = new GameObject[ROWS, COLS];
for (int i = 0; i < ROWS; i++)
        for (int j = 0; j < COLS; j++)
                if (board_int[i, j] == 0)
                {
                        Vector3 pos = new Vector3(j, -0.4f, i);
                        int arrowPosition = i * COLS + j;
                        int bestaction = agent.BestAction(arrowPosition);
                        switch (bestaction)
                        {
                        case 0:
```

```
                                                board_arrow[i, j] =
Instantiate(prefabArrow, pos, Quaternion.Euler(90,90,0));  // Right
                                break;

                            case 1:
                                                board_arrow[i, j] =
Instantiate(prefabArrow, pos, Quaternion.Euler(90, 0, 0));   // Up
                                break;

                            case 2:
                                                board_arrow[i, j] =
Instantiate(prefabArrow, pos, Quaternion.Euler(90, -90, 0));  // Left
                                break;

                            case 3:
                                                board_arrow[i, j] =
Instantiate(prefabArrow, pos, Quaternion.Euler(90, 180, 0));  // Down
                                break;
                            }
                    }
}
```

Se ha creado la función FillBoardArrow en la clase Game. Esta función instancia los prefabs de las flechas sobre las casillas vacías, mirando en dirección de la mejor acción. Se llama a FillBoardArrow en la función Start de la clase Game, después de iniciar la tabla Q mediante StartAgent.

```
void UpdateBoardArrow()
{
// 0: rigth; 1: up; 2: left; 3:
// Update arrows
// NOTE: rows -> movement along z; cols -> movement along x
for (int i = 0; i < ROWS; i++)
  for (int j = 0; j < COLS; j++)
    if (board_int[i, j] == 0)
    {
      int arrowPosition = i * COLS + j;
      int bestaction = agent.BestAction(arrowPosition);
      switch (bestaction)
      {
      case 0:
          board_arrow[i, j].transform.rotation = Quaternion.Euler(90, 90,
0);  // Right
        break;

      case 1:
          board_arrow[i, j].transform.rotation = Quaternion.Euler(90, 0,
0);   // Up
```

```
            break;

        case 2:
            board_arrow[i, j].transform.rotation = Quaternion.Euler(90, -90,
0);  // Left
            break;

        case 3:
            board_arrow[i, j].transform.rotation = Quaternion.Euler(90, 180,
0);  // Down
            break;
        }
    }
}
```

Se ha creado la función UpdateBoardArrow en la clase Game. Esta función actualiza la rotación de las flechas de acuerdo a la dirección deseada por el agente. UpdateBoardArrow se utiliza en la función NewTurn de la clase Game, después de actualizar la tabla Q mediante la función GetReward de la clase Agente.

Recursos

Manual de teoría de la asignatura, en especial el apartado 4.1.

Paquete de Unity con el juego Maze.

Criterios de valoración

Las actividades de la PEC tendrán la siguiente valoración asociada:

Actividad 1:	2 puntos
Actividad 2:	1 punto
Actividad 3:	2 puntos
Actividad 4:	2 puntos
Actividad 5:	3 puntos

Formato y fecha de entrega

La PEC se ha de entregar antes del próximo 10 de enero (a las 24h).

La solución que hay que entregar consiste en un informe en formato PDF usando este enunciado como plantilla y, para cada actividad en la que se pida: un vídeo demostrativo donde se vea el uso de lo que pide el enunciado y una carpeta con la implementación. El informe ha de contener la descripción y el código relacionado a lo que pide el enunciado. Todos estos ficheros se han de comprimir en un archivo ZIP.

Adjuntad el fichero al apartado de **Entrega y registro de EC (REC)**. El nombre del fichero debe ser ApellidosNombre_IA_PEC3.zip.

Para dudas y aclaraciones sobre el enunciado, dirigíos al consultor responsable de vuestra aula.

Nota: Propiedad intelectual

A menudo es inevitable, al producir una obra multimedia, hacer uso de recursos creados por terceras personas. Es por tanto comprensible hacerlo en el marco de una práctica de los estudios del Máster de Ingeniería Informática, siempre que esto se documente claramente y no suponga plagio en la práctica.

Por lo tanto, al presentar una práctica que haga uso de recursos ajenos, se presentará junto con ella un documento en el que se detallen todos ellos, especificando el nombre de cada recurso, su autor, el lugar donde se obtuvo y el su estatus legal: si la obra está protegida por copyright o se acoge a alguna otra licencia de uso (Creative Commons, licencia GNU, GPL ...). El estudiante deberá asegurarse de que la licencia que sea no impide específicamente su uso en el marco de la práctica. En caso de no encontrar la información correspondiente deberá asumir que la obra está protegida por copyright.

Deberán, además, adjuntar los archivos originales cuando las obras utilizadas sean digitales, y su código fuente si corresponde.