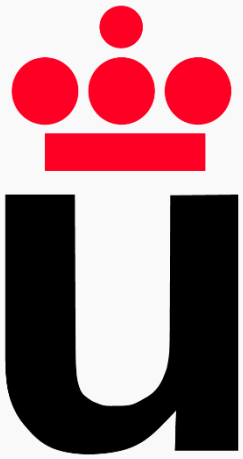


GRADO EN DISEÑO Y DESARROLLO DE VIDEOJUEGOS

DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL

Práctica grupal 1: Algoritmo Q-Learning

Profesores
Alfredo Cuesta
Dan Casas

Autores: Grupo X
Pedro Casas Martínez
Adrián Vaquero Portillo

Fecha de entrega
18 de noviembre de 2019

Índice

1. Descripción del algoritmo empleado para solucionar el problema.

El algoritmo empleado para la práctica es el Q-Learning, un algoritmo de aprendizaje automático por refuerzo. Esto quiere decir que para decidir la mejor acción en función del entorno el algoritmo recibe una cierta recompensa predefinida dependiendo de las decisiones que tome durante su etapa de aprendizaje.

En concreto, con Q-Learning se genera una tabla de valores que indican la calidad de cada acción posible para todos los estados del entorno donde por defecto se inicializan todos los valores al mismo. Después, se simulan una serie de episodios en los cuales se asigna un estado inicial aleatorio y se intenta llegar al objetivo mediante acciones tomadas al azar. Conforme se van tomando estas acciones aleatorias se modifica el valor de calidad de la acción en ese estado ($Q(s,a)$) en función de la recompensa obtenida (r) y del valor de calidad máximo del estado siguiente ($\max\left(Q(s',\forall a')\right)$). El cálculo del nuevo valor de calidad también depende dos parámetros:

- Alfa (α): ratio de aprendizaje, define cuánto se conserva del valor Q actual y cuánto de la modificación calculada.
- Gamma (γ): ratio de descuento, define la importancia del máximo valor de calidad del estado siguiente.

Por todo esto, la fórmula de la regla de aprendizaje para calcular el nuevo valor de calidad en el proceso de aprendizaje tras tomar una acción al azar queda definida como:

$$Q(s,a) = (1 - \alpha) * Q(s,a) + \alpha * (r + \gamma * \max\left(Q(s',\forall a')\right))$$

2. Características de diseño e implementación

Descripción general

En nuestro programa hemos definido un array tridimensional llamado *tablaQ*, que almacena los valores de calidad mediante los cuales se deciden las acciones del agente, en nuestro caso hacia qué dirección se mueve nuestro personaje. Los dos primeros índices de este array se corresponden con las coordenadas *x* e *y* del agente en el entorno, y el tercero se corresponde con la acción. Por lo tanto, sus dimensiones son el ancho por el alto del entorno por las acciones posibles en cada estado, es decir, 14x8x4.

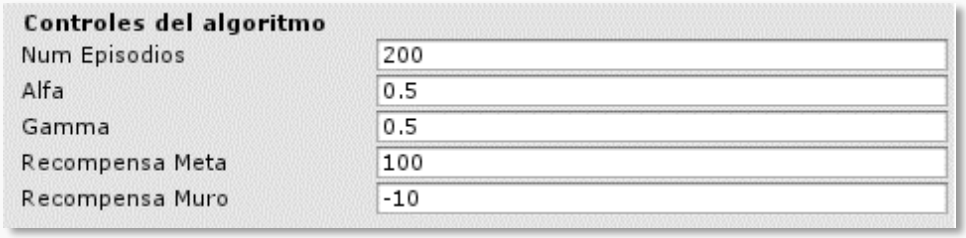
La primera vez que se llama a la función que define el siguiente movimiento del personaje, se comprueba si existe el fichero en el cual se encuentra la *tablaQ* de la semilla actual. Si no existe, se ejecuta nuestra función *AlgoritmoQ*, que genera la *tablaQ*, y se crea un fichero con el nombre de la semilla en el cual se escribe la tabla generada. Si el fichero ya existía, se leen sus valores y se guardan en la *tablaQ*. Posteriormente, se emplean estos valores en la función cada vez que es llamada para decidir cuál será el siguiente movimiento del personaje.

Controles

Nuestro algoritmo posee variables que se usan como control del algoritmo y que pueden ser modificadas en el inspector de Unity. Estas variables tienen un valor por defecto definido por nosotros con el que sabemos que el programa funciona. Las variables son las siguientes:

- Número de episodios: indica cuántas veces el algoritmo tratará de llegar hasta la meta tomando decisiones al azar desde una posición inicial aleatoria. Cuántos más episodios más precisa será la tabla Q.
- Alfa: ratio de aprendizaje, funciona tal y como está explicado en la descripción del algoritmo.
- Gamma: ratio de descuento, funciona tal y como está explicado en la descripción del algoritmo.
- Recompensa de meta: qué valor se usa como recompensa para el estado meta.
- Recompensa de muro: qué valor se usa como recompensa en un estado donde haya un muro.

Los valores por defecto se pueden ver en la siguiente imagen, sacada directamente del inspector de Unity:



La función *AlgoritmoQ*

La función *AlgoritmoQ* recibe como parámetros la información del entorno (*boardInfo*) y la posición de la meta (*goals*). Primero definimos las dimensiones de nuestro array *tablaQ* en función del parámetro *boardInfo*. Tras esto, comienza un bucle for que se ejecuta tantas veces como episodios definamos en nuestra constante global *numEpisodios*. En cada iteración del bucle, se calcula un estado inicial aleatorio, y comienza un bucle while en el cual se simulan acciones aleatorias hasta que se llega a la meta, y se actualiza la *tablaQ* tras cada acción mediante la regla de aprendizaje especificada en el apartado de descripción del algoritmo.

Como apunte, al principio este algoritmo se desarrolló de forma que la posición inicial fuera siempre la del personaje cuando se inicia el juego, pero consideramos que con una posición inicial aleatoria se considerarían más casos, haciendo que la *tablaQ* resultante fuera más precisa para cualquier caso posible.

Las funciones *LeerFichero* y *EscribirFichero*

Usamos estas dos funciones para leer el fichero y almacenar sus valores de calidad en nuestro array *tablaQ* y para escribir en un nuevo fichero los valores de calidad almacenados en nuestro array respectivamente. La nomenclatura de nuestros ficheros es el número de la semilla seguido de ".txt".

En la primera ejecución de la función *GetNextMove*, que se llama para definir el próximo movimiento del personaje, se genera el nombre del archivo en una string y se comprueba si existe un fichero con ese nombre:

- Si existe dicho fichero, se ejecuta *LeerFichero*, que recibe *boardInfo* y la string *nombreArchivo*, comienza a leer el archivo con ese nombre línea a línea. Por cada línea se llama a la función *obtenerValoresQDeString* que recibe una línea y un array de floats y guarda en ese array los números que haya en la línea separados por espacios. Finalmente, se introducen los datos en el array *tablaQ*.

- Si no existe, se ejecuta *AlgoritmoQ* que calcula los valores del array *tablaQ*. Tras esto, se llama a la función *escribirFichero* que guarda los datos de *tablaQ* por líneas en un fichero, de forma que en cada línea del fichero se encuentren los valores Q de cada estado separados por espacios.

La función *GetNextMove*

En el resto de las ejecuciones de la función *GetNextMove*, los valores del array *tablaQ* ya están calculados, por tanto, se llama a la función *obtenerMejorAccion* para encontrar, dada la posición del personaje, la acción con mayor valor de calidad, que se devuelve como un entero de 0 a 3 (N, S, O, E). Tras esto, dependiendo del valor obtenido, la función *GetNextMove* devuelve el movimiento correspondiente.

3. Dificultades y obstáculos en el desarrollo del algoritmo

Salida del dominio del problema

Tuvimos un problema que no supimos identificar al principio, puesto que no se sacaba ningún mensaje de error. El problema consistía en que, en el bucle del *AlgoritmoQ*, a la hora de tomar una acción al azar en los extremos del dominio, era posible tomar una acción que dejaría al personaje fuera del entorno. También estábamos intentando acceder a una coordenada -1 del array *tablaQ*, tras lo cual el programa se salía del bucle y no llegaba a calcular los valores Q correctamente. Tras descubrir que este era el problema, decidimos hacer un clamp de la posición para que no pudiera estar fuera del dominio.

Mejora de la precisión de *AlgoritmoQ*

Durante el desarrollo del algoritmo, decidimos que se saliera del bucle while cuando se llegara a un muro o a la meta. Esta implementación suponía que la simulación apenas llegaba a la meta, puesto que era mucho más probable que se llegara a un muro antes de llegar a la meta. Consideramos que sería suficiente con la recompensa negativa al llegar al muro, y así nos aseguramos de que en todos los episodios se llega a la meta. Con este cambio se mejoró mucho la precisión del Algoritmo, que ahora nos daba un camino muy optimizado.

Semillas que imposibilitan el camino a la meta

Tras terminar el algoritmo, reparamos en que hay ciertas semillas (por ejemplo, la semilla 81194) en las cuales el mapa que se genera no facilita ninguna ruta posible para llegar a la meta, dejando al personaje bloqueado. Pensamos que lo mejor sería, por lo

menos, notificar por la consola de Unity del problema con el mensaje: “La posición a la que se dirige el personaje no es andable, es posible que no sea posible llegar a la meta. Pruebe con otra semilla.” si el personaje está andando hacia un muro, y el mensaje “La posición actual no es andable, es posible que el personaje no pueda continuar, pero se intentará de todos modos” si el personaje aparece dentro de un muro.

4. Discusión sobre los resultados obtenidos.

En conclusión, estamos muy satisfechos con los resultados obtenidos, puesto que, tras probar con diferentes semillas, hemos observado que la tabla Q resultante siempre ofrece un camino óptimo hacia la meta, además de calcularlo en muy poco tiempo. También estamos satisfechos con el sistema de gestión de ficheros, que ha quedado muy organizado. Nos ha quedado más claro cómo funciona este algoritmo de aprendizaje tras tener la oportunidad de programarlo por nuestra cuenta.