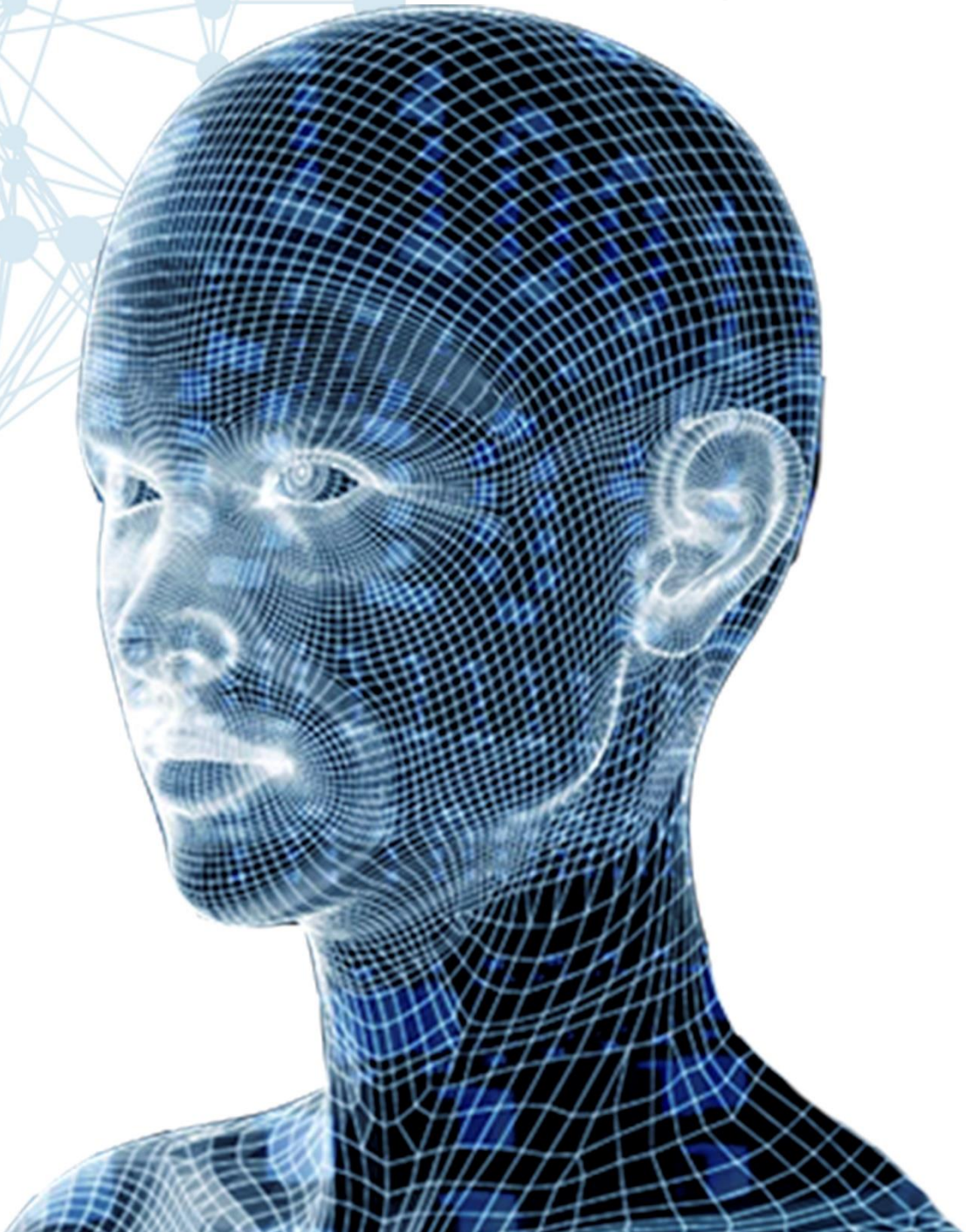
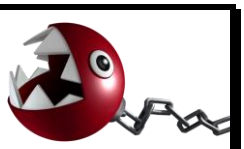
DISEÑO DE JUEGOS CON INTELIGENCIA ARTIFIAL

GRUPO 14:
Ana María Alcaide Recio
Sergio Sánchez-Uran López
Carlos Marques Sánchez

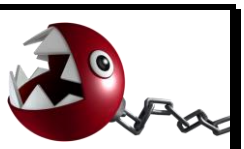


Universidad
Rey Juan Carlos



Contenido

Universidad Rey Juan Carlos



DESCRIPCIÓN DEL ALGORITMO UTILIZADO.
CARACTERÍSTICAS DE DISEÑO E IMPLEMENTACIÓN.

Clase Nodo

Es una clase auxiliar que nos sirve para almacenar mediante variables **Cellinfo** la posición actual del personaje y la posición anterior que se guardará como **NodoPadre** y de tipo **Cellinfo**

También tenemos creada una lista que guarda los nodos que vamos expandiendo y sus correspondientes padres.

Tenemos una función que expande los nodos del mapa partiendo de dónde esté el personaje.

Para ello del nodo actual estudiamos todos los nodos posibles en los que se puede avanzar y seleccionamos el que tenga menor coste respecto de la meta, esto se usa en el método A* para calcular el camino a la meta (en el método calculatePath2 de la clase **OfflineRandomMind** y **OnlineRandomMind**).

Tenemos el método ordenar que se encarga de ordenar los nodos expandidos antes de retornar la lista.

OFFLINE

Aplicamos el algoritmo A* que expande todos los nodos hasta que encuentra la meta, encuentra el camino óptimo.

Tenemos una función que se llama **CalculatePath**, que aplica el algoritmo anterior. Para ello en cada iteración expandimos el nodo correspondiente a la última posición de la lista **fathers *(es el array que utilizaremos para ir guardando el camino hacia la meta y del que más tarde iremos extrayendo las posiciones para recorrerlo)*** y guardamos estos nodos en la lista de Nodos **expandedNodos** para más tarde guardar en la lista **fathers** el nodo con menor coste.

El resto de nodos no seleccionados los guardamos en la lista de **abiertos** exceptuando al padre que lo borramos para evitar ciclos simples. Y ordenamos los nodos expandidos de esa iteración en función del coste.

Al terminar las iteraciones del algoritmo exploramos la lista de padres(fathers) de final al principio y hacemos que el personaje recorra la lista.

También está el método **GetNextMove** que mueve el personaje cambiando la posición actual de éste a una posición futura dependiendo de las acciones que se desarrollen en el método **calculatePath**.

Por ejemplo, si el nodo que vamos a elegir en la siguiente posición es el de arriba, la posición en x no variará, pero la posición en y incrementará en 1.

ONLINE

En este paso aplicamos el algoritmo de **horizonte** o **hill climbing** con un horizonte variable según la distancia del personaje al enemigo.

Por ejemplo: si la distancia de la celda es 12 hasta la meta, el horizonte será 10, restamos 2 al total porque es mayor a 3, si fuera menor a 3 el horizonte sería 1.

Primero vamos a por el enemigo más lejano de la meta y luego a por el siguiente enemigo que éste más lejano de la meta, así hasta que se acaben los enemigos que será cuando nuestro personaje ejecutará el método **calculatePath2** e irá a la meta.

Tenemos el método **calculatePath2** que hace la misma operación que **calculatePath** del método offline.

En el método **selectEnemy** que selecciona el enemigo más lejano que vamos a coger. Para ello calculamos la distancia a todos los enemigos que haya en el nivel y guardamos la posición del enemigo que esté más lejos de la meta.

También tenemos el método **checkEnemiesCatched** que lo que hace es checkear la cantidad de enemigos que tenemos activos, si el número se ha reducido actualizamos el estado del número de enemigos activos guardados en la variable **numenemies** y volveríamos a seleccionar el siguiente enemigo a coger, hasta que el número de enemigos es 0 con lo que la variable **allcatched** valdría true y calcularíamos el **path** desde la posición donde nos encontramos hasta la casilla de meta.

En el método **getNextMove**, sigue igual que en el anterior, pero hemos incluido una serie de instrucciones if. Tenemos varias variables boolenanas que nos ayudan con estos if. Por ejemplo, tenemos una variable llamada **allcatched** que comprueba si se han cogido todos los enemigos, y hasta que esta variable no se true, no entra en el método **calculatePath** y no va hacia la meta.

Nota: La velocidad por defecto fijada en este método para que resulte más rápido de ver y no se haga tan lenton por el número de iteraciones que se producen, se ha fijado mediante Time.TimeScale = 3 lo que supone triplicar la velocidad normal.

Q- LEARNING

Tenemos una clase llamada Q-Learning que inicializa las tablas que iremos actualizando en cada iteración. También según se vaya actualizando (método **learn**) las tablas tendremos la posibilidad de elegir entre la casilla con más valor cercana o elegir al azar los movimientos del avatar (método **randomStep**).

Tenemos un constructor que inicializa todas las tablas utilizadas. **QTable** la inicializa a 0 y la tabla de recompensas(**rewardTable**), todas las casillas son inicializadas a -1 menos la meta que tiene un valor de 100 y -5 las paredes (aunque nunca va a llegar a ellas).

La variable épsilon (**e**) controla si elegimos una casilla al azar o con el máximo valor de la **Q-Table**.

Dentro de la clase **QLearningRandomMind**, encontramos métodos como **Save**, **Load**, **deleteFile**, **reloadGame** y **exitQlearning**.

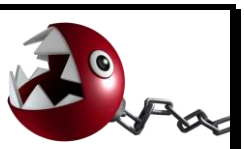
Como detalle después de ejecutar el método Load, se reinician el número de iteraciones a 600 de nuevo.

En **getNextMove** tenemos un if que si existe el archivo lo carga y si no crea un objeto de tipo **Qlearning** y lo inicializa.

Para calcular el siguiente paso usamos la función **nextStep** del objeto creado que nos dirá el siguiente movimiento dependiendo de un número **Random** generado entre 0 y 1 y dependiendo del valor de épsilon(e) elegiremos entre movimiento al azar o el máximo valor de los valores cercanos de la tabla **QTable** y antes de retornar el valor seleccionado invocamos a la función **learn** del objeto para actualizar el valor de la **Qtable** con la posición y la acción seleccionada.

Grabaremos la escena en cada iteración y en caso de que lleguemos a la meta o al máximo de iteraciones (fijado a un máximo de 600 iteraciones), recargaremos la escena para volver a empezar.

Se han añadido un botón y un marcador. El botón **ExitButton** sirve para parar la ejecución y el marcado de **épsilon** nos marca el límite entre elegir al azar o seleccionar la siguiente acción por **Qtable**.



DISCUSIÓN SOBRE LOS RESULTADOS OBTENIDOS.

En el método online hemos encontrado problemas ya que, en primer lugar la selección de los enemigos se realiza por distancia de la posición del enemigo a la meta seleccionando el que está más alejado, una vez que el primer enemigo ha sido seleccionado el siguiente enemigo seleccionado sería el siguiente más lejano a la meta sin tener en cuenta la posición de nuestro Personaje, por lo que creemos que la solución sería buena pero no óptima en cuanto a que si tuviéramos muchos enemigos en pantalla llevaría más tiempo eliminar los enemigos y encontrar la meta.

En el método Q-learning, nosotros lo hacemos un número de veces pequeño, desde 1 a 0,1 con un paso de 0.05, para encontrar el camino por lo que no siempre encuentra el mismo ya que se necesitarían más iteraciones para que los resultados de la tabla estuvieran lo más ajustado posible y variasen lo menos posible. Por lo que si lo hiciéramos mil veces el camino óptimo seguramente tuviera mejores valores y menos variaciones.

Podemos concluir también que, los métodos **OnlineRandomMind** y **OfflineRandomMind**, para grandes mapas consume mucha memoria al tener que guardar todos los nodos y tener que recorrer todas las posiciones del mapa.