INTELIGENCIA ARTIFICIAL

Memoria de proyecto

Estudio de Algoritmos de Búsqueda

Jose Miguel Acitores

Pablo Darnaude

Alberto Soutullo

CONTENIDO

El problema se nos presenta con 4 semillas, las cuales proporcionarán 4 escenarios diferentes para poner a prueba tanto los algoritmos Offline como Online. En el apartado OnLine, también se incluirá un numero de enemigos, cuyas posiciones también son generadas a raíz de la semilla.

SEMILLAS

SEMILLA 4186

SEMILLA 5875

SEMILLA 8685

SEMILLA 476

BÚSQUEDA OFFLINE

El objetivo de este apartado será hacer que nuestro robot encuentre el mejor camino a la salida (GOAL) mediante algoritmos Offline si existe.

La búsqueda offline es aquella que encuentra el resultado y posteriormente establece una serie de acciones fijas en cada posición.

Vamos a realizar el estudio de los resultados de 2 algoritmos ligeramente diferentes. En uno de ellos, además, se realizará un estudio a mayores para explicar las dos posibles variantes que se han encontrado.

ALGORITMO DE BÚSQUEDA EN AMPLITUD

La búsqueda en amplitud se trata de un algoritmo de búsqueda no informada, es decir sin ninguna información a priori, completo y óptimo, siempre encuentra la solución si existe una y además encuentra el mejor camino.

Este algoritmo expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. En nuestro caso, el árbol está formado por todos los caminos existentes, siendo la raíz el camino de longitud 0, la primera capa el camino de longitud 1 y así sucesivamente.

A la hora de implementarlo hemos optado por amplitud puro, es decir, no eliminamos bucles de exploración, sino que exploramos todos los nodos hijos, pero si limitamos que un nodo hijo no expanda su propio padre. De esta manera se obtiene una alta cantidad de nodos en todas las semillas comparándolo con los otros algoritmos y también comprobamos que se lleva a cabo en un periodo de tiempo mayor.

Para implementar este algoritmo hemos seguido el siguiente pseudocódigo:

abierta ← S_0

Repetir

Si vacía? (abierta) **entonces**

devolver (negativo)

Nodo ← primero (abierta)

Si meta? (nodo) **entonces**

devolver (nodo)

sucesores ← expandir (nodo)

Para cada n∈ sucesores **hacer**

n. padre ← nodo

ordInsertar (n, abierta, fin)

Fin {repetir}

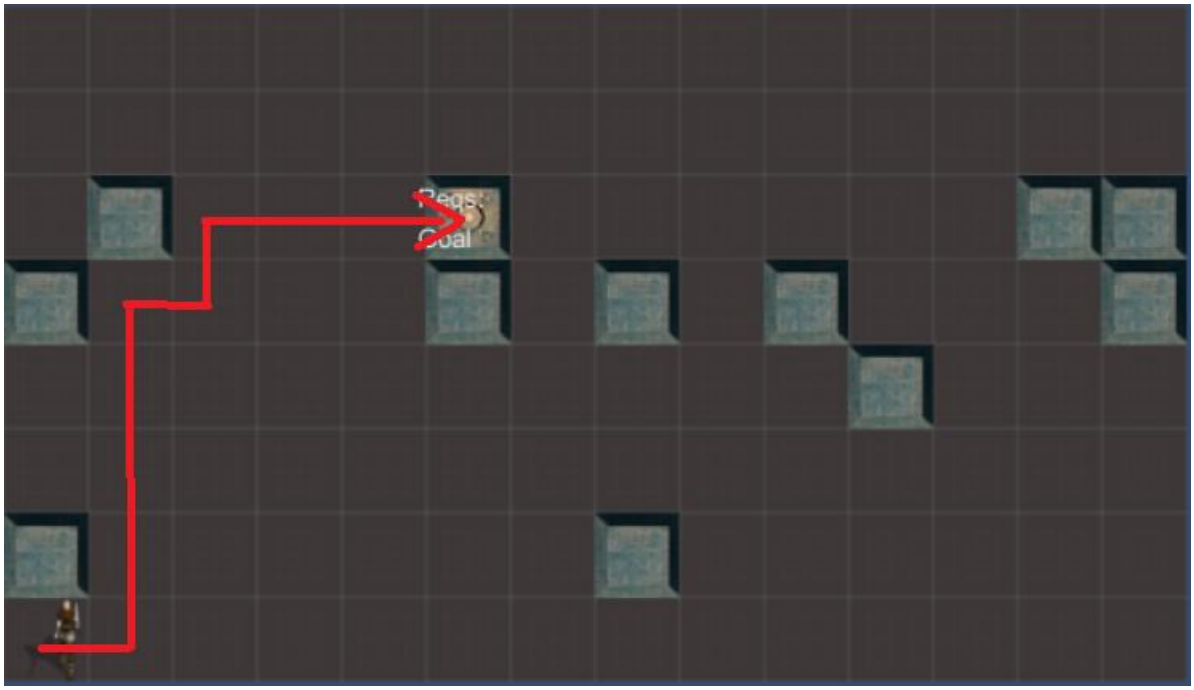
SEMILLA 4186

Mediante el algoritmo de búsqueda en amplitud, en la semilla 4186 se han expandido un total de **1985** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:

Como se puede observar, de esta manera se expanden una cantidad de nodos muy elevada, ya que se está haciendo una búsqueda completa sin eliminación de bucles.

CAMINO:



SEMILLA 5875

Mediante el algoritmo de búsqueda en amplitud, en la semilla 5875 se han expandido un total de **9** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:

En este caso, el número de nodos es mucho menor ya que GOAL se encuentra muy cerca de nuestra casilla de salida.

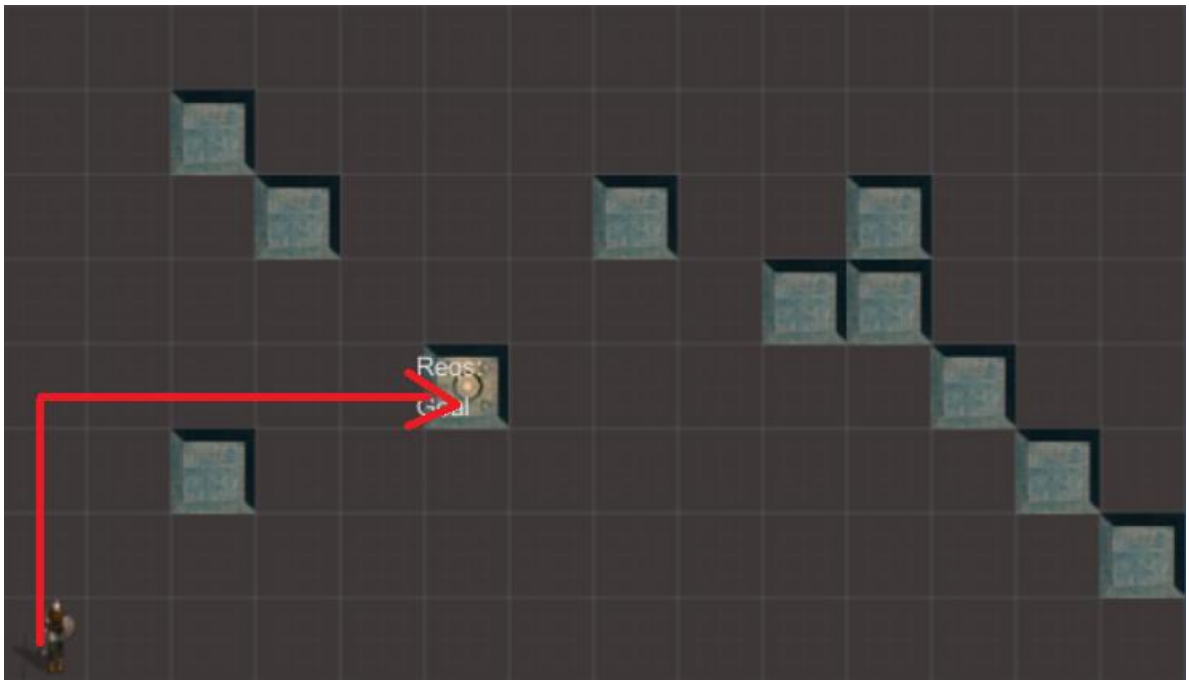
CAMINO:

SEMILLA 8685

Mediante el algoritmo de búsqueda en amplitud, en la semilla 8685 se han expandido un total de **503** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:

CAMINO:



SEMILLA 476

Mediante el algoritmo de búsqueda en amplitud, en la semilla 476 se han expandido un total de **169** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:

CAMINO:

Con los anteriores escenarios se comprueba que el requerimiento de recursos de una búsqueda en amplitud es exponencial. También se comprueba que siempre que existe un nodo meta se encuentra, y el camino tomado es el óptimo.

El orden de expansión de este algoritmo viene dado por el método **`WalkableNeighbours()`** que se encuentra en el script CellInfo.cs. Dicho orden es Arriba, Derecha, Abajo, Izquierda.

ALGORITMO DE BÚSQUEDA A *

El algoritmo A* se trata de un algoritmo de búsqueda informada, es decir, poseemos información a priori para tomar las decisiones, completo y si la heurística es admisible, óptimo. Este algoritmo trata de seleccionar el mejor camino a través del coste mediante una heurística. Es un algoritmo basado en la búsqueda por amplitud, pero priorizando el camino con menos coste para encontrar la ruta más óptima.

Para implementar este algoritmo hemos seguido el siguiente pseudocódigo:

abierta ← s_0

Repetir

Si vacío? (abierta) **entonces**

devolver(negativo)

nodo ← primero(abierta)

Si meta? (nodo) **entonces**

devolver(nodo)

sucesores ← expandir(nodo)

Para cada n∈sucesores **hacer**

n. padre ← nodo

ordInsertar (n, abierta, f *)

Fin {repetir}

PROBLEMAS DE IMPLEMENTACIÓN Y SOLUCIONES:

Uno de los primeros problemas que nos encontramos a la hora de implementar este algoritmo era el cómo devolvíamos los movimientos necesarios para llegar a GOAL. En un principio optamos por una estructura tipo COLA, ya que al inicio pensábamos en llenar la cola según avanzábamos cuándo realmente había que incluir los movimientos una vez llegada a la salida, haciendo uso del atributo padre que ya teníamos añadido en la clase Nodo. Siendo por tanto lo que necesitábamos una estructura tipo PILA. De esta manera, una vez habiendo llegado a GOAL, lo único que teníamos que hacer era un bucle para deshacer el camino mientras el padre no fuese null (el nodo inicial), e ir añadiendo los Movimientos. De esta manera, el primer movimiento en insertarse sería el último en emplearse.

No nos dimos cuenta que se necesitaba una lista de nodos ya visitados. Cuando vimos que se expandía una gran cantidad de nodos, intentamos buscar una explicación a ello. Haciendo el algoritmo paso a paso, nos dimos cuenta de que el problema era que se estaban expandiendo nodos que ya se habían expandido anteriormente. Para solventarlo creamos una Lista de Nodos ya expandidos, en la cual hacemos una búsqueda del Nodo que se va a expandir antes de hacer la expansión. Si éste se encuentra en ella, lo obviamos.

Dentro de la implementación del A*, decidimos que lo que devolviese fuera un booleano de si se encontraba GOAL o no. De esta manera, si se encuentra GOAL se va a rellenar de movimientos una variable de clase, mientras que, si GOAL no es encontrado, se devolverá un false, haciendo que el personaje no realice movimientos.

Para implementar el algoritmo A* hemos tenido que decidir entre dos variantes del mismo, A* en amplitud y A* en profundidad. Hemos incluido una variable pública en Unity llamada **Deep Preference.** Esta selecciona el tipo de A* que se va a realizar (siendo true el A* en profundidad).

ALGORITMO A* EN AMPLITUD

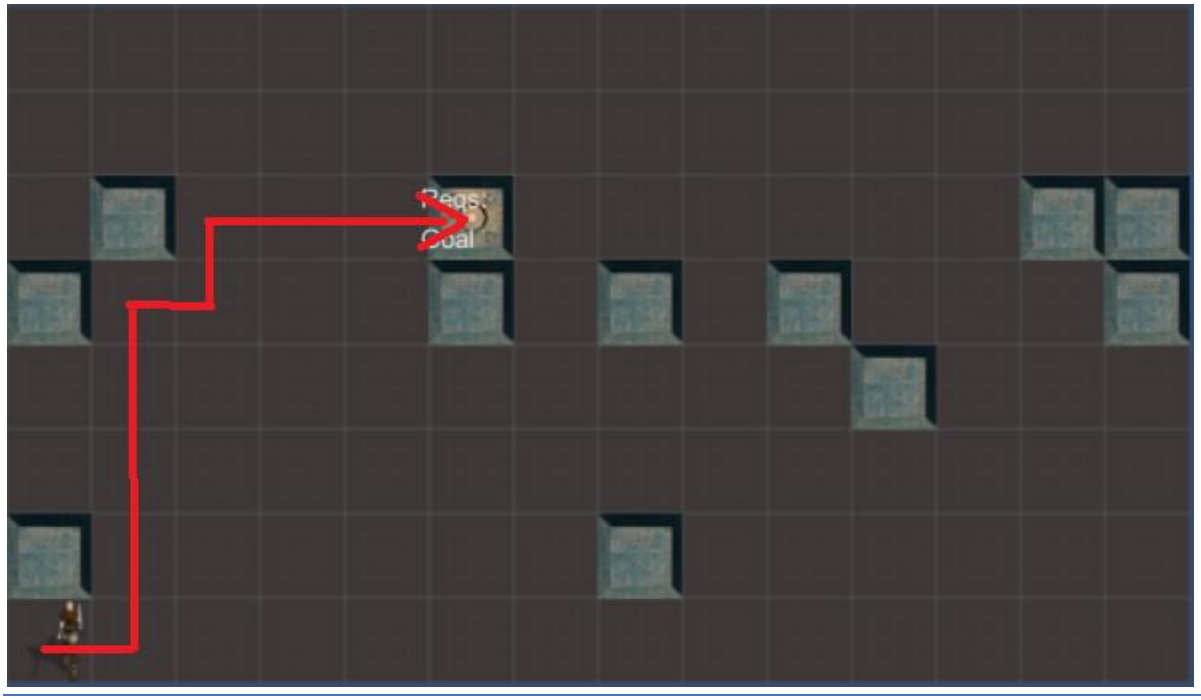
Este algoritmo es una variante del A* en la que, al insertar en la lista de nodos para explorar, los nuevos nodos se insertan detrás de aquellos que tengan el mismo coste total, así se expandirán primero los nodos que ya estaban en la lista, siendo entonces este algoritmo más similar a la búsqueda por amplitud.

SEMILLA 4186

Mediante el algoritmo de búsqueda A* en amplitud, en la semilla 4186 se han expandido un total de 28 nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:

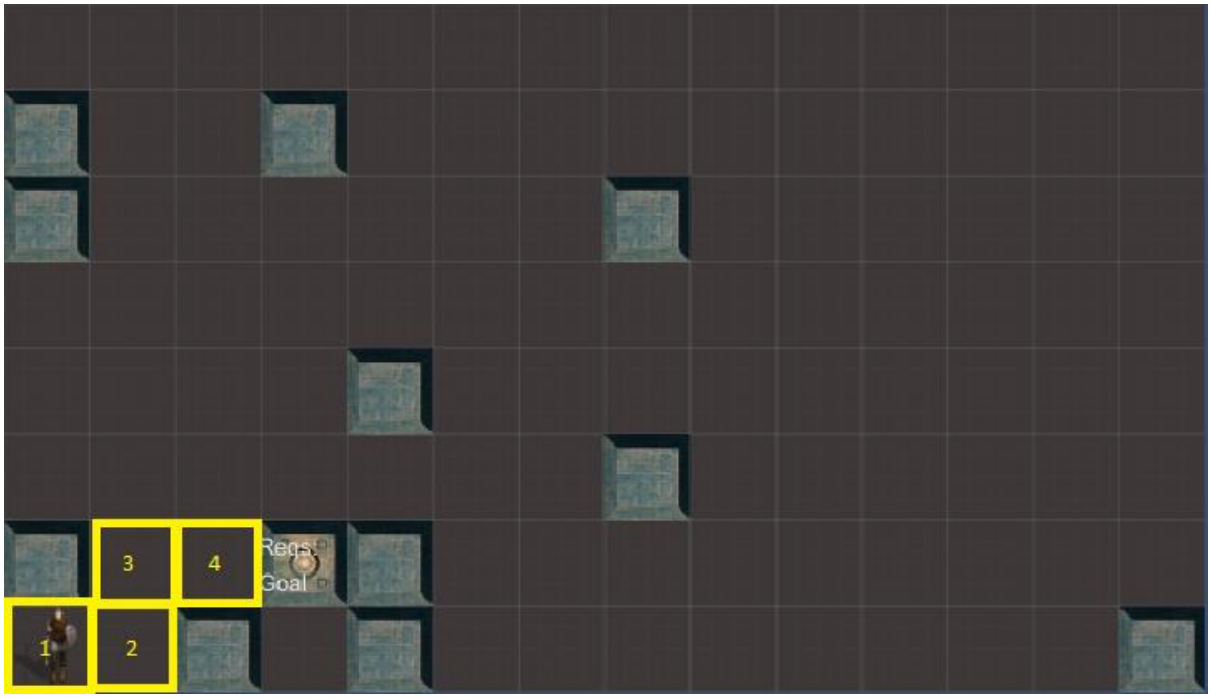
CAMINO:



SEMILLA 5875

Mediante el algoritmo de búsqueda A* en amplitud, en la semilla 5875 se han expandido un total de **4** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:

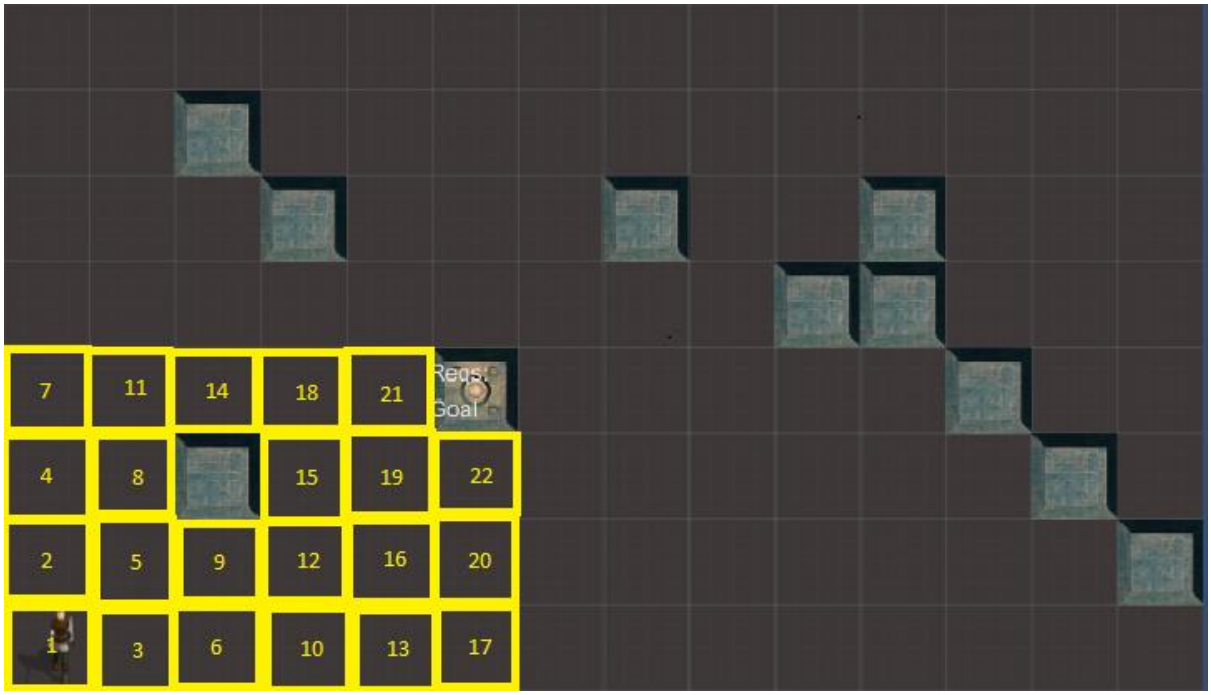


CAMINO:

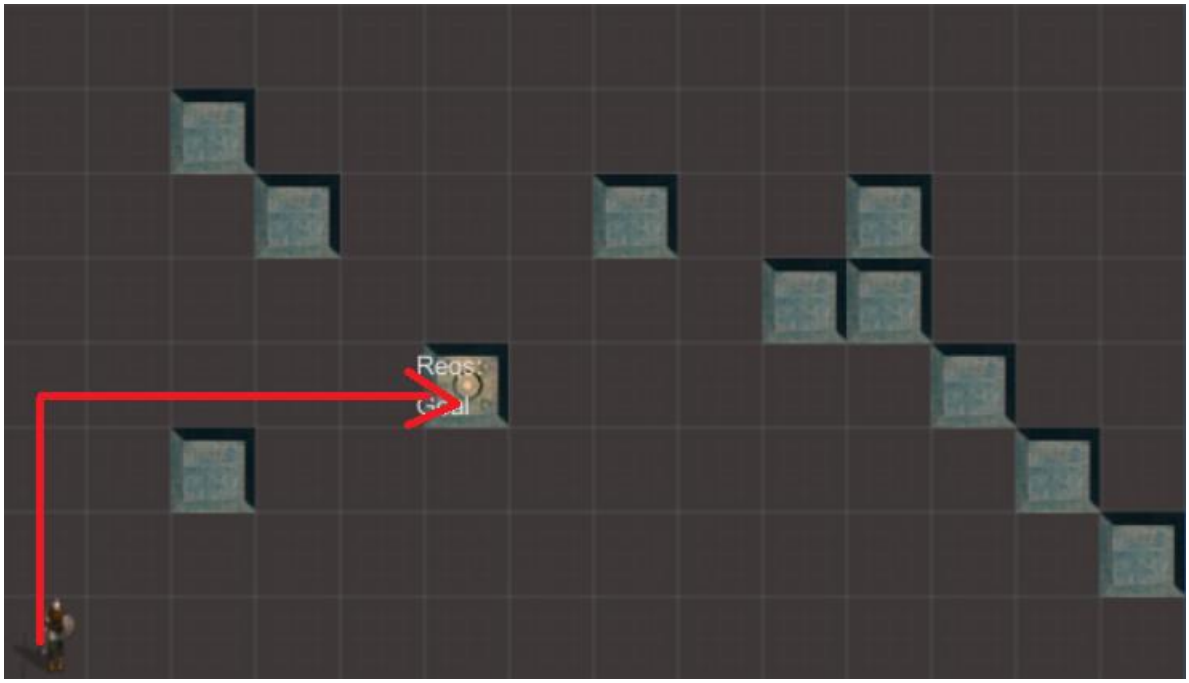
SEMILLA 8685

Mediante el algoritmo de búsqueda A* en amplitud, en la semilla 8685 se han expandido un total de **22** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODO:



CAMINO:



SEMILLA 476

Mediante el algoritmo de búsqueda A* en amplitud, en la semilla 476 se han expandido un total de **10** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODO:

CAMINO:

ALGORITMO A* EN PROFUNDIDAD

Este algoritmo es una variante del A* en la que, al insertar en la lista de nodos para explorar, los nuevos nodos se insertan delante de aquellos que tengan el mismo coste total, así se expandirán primero los nodos que acaban de ser añadidos a la lista de nodos, siendo entonces este algoritmo más similar a la búsqueda por profundidad dentro del propio A*.

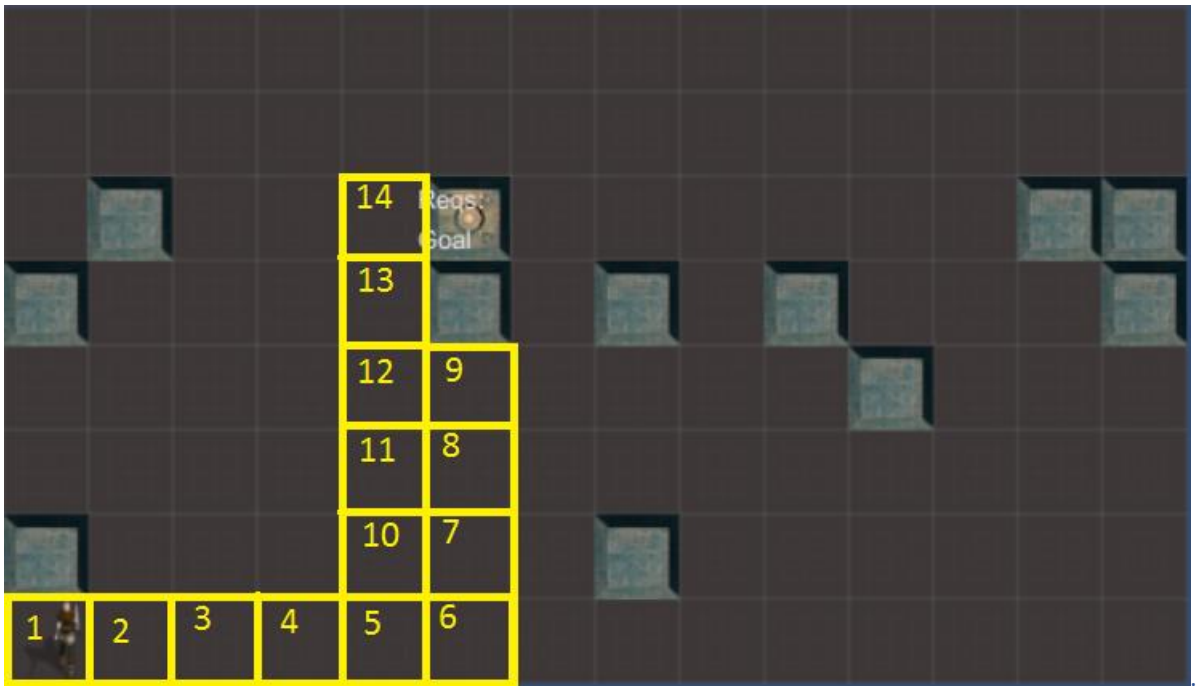
Este algoritmo genera un menor número de nodos en nuestro caso ya que al ser un mapa creado con cuadrículas y sin movimiento en diagonal directo, se evita explorar dos caminos cuando queremos movernos en diagonal ya que al tener la nueva casilla el mismo peso que la anterior, esta entrará por delante en la lista.

Por ello seleccionamos esta variante del algoritmo A* para nuestro proyecto, gracias a ello podríamos evitar crear una lista de nodos explorados para comprobar si el nodo que visitamos ha sido expandido ya o no, ya que, si se repiten nodos en la lista, se priorizarán aquellos que tengan menor coste y por ello se encontrará el camino óptimo sin necesidad de la misma y ahorrando tiempo de ejecución. Aun así, decidimos incluirla ya que esta ahorrará repetir un camino erróneo varias veces ahorrando espacio en la memoria al crear los nodos y como en nuestro caso es un mapa de poco tamaño no influye prácticamente en el tiempo de ejecución.

SEMILLA 4186

Mediante el algoritmo de búsqueda en amplitud, en la semilla 4186 se han expandido un total de **14** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:



CAMINO:

SEMILLA 5875

Mediante el algoritmo de búsqueda en amplitud, en la semilla 5875 se han expandido un total de **4** nodos. El orden de expansión está reflejado en la imagen de abajo.

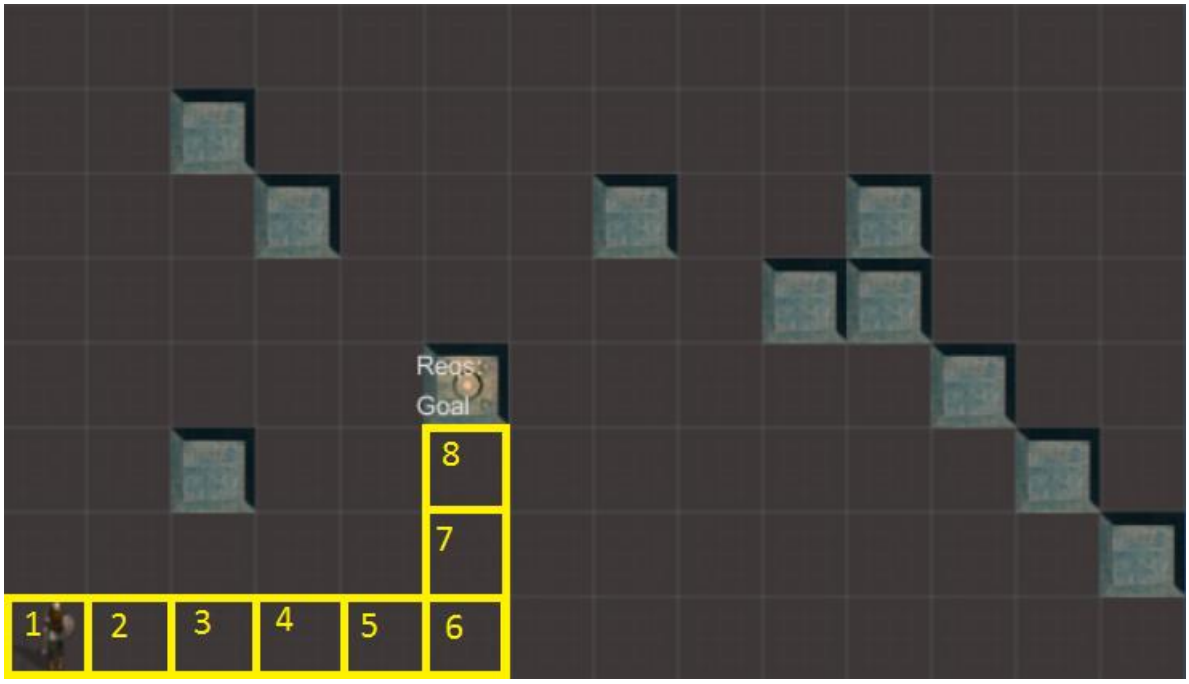
EXPANSIÓN DE NODOS:

CAMINO:

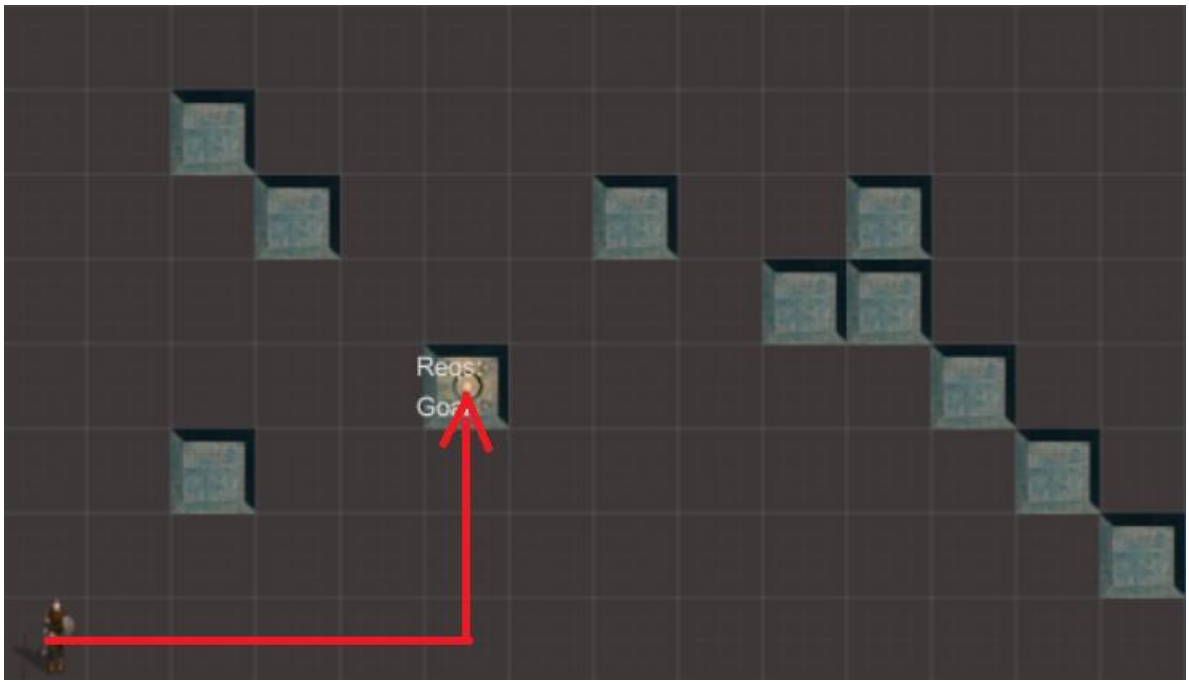
SEMILLA 8685

Mediante el algoritmo de búsqueda en amplitud, en la semilla 8685 se han expandido un total de **8** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:



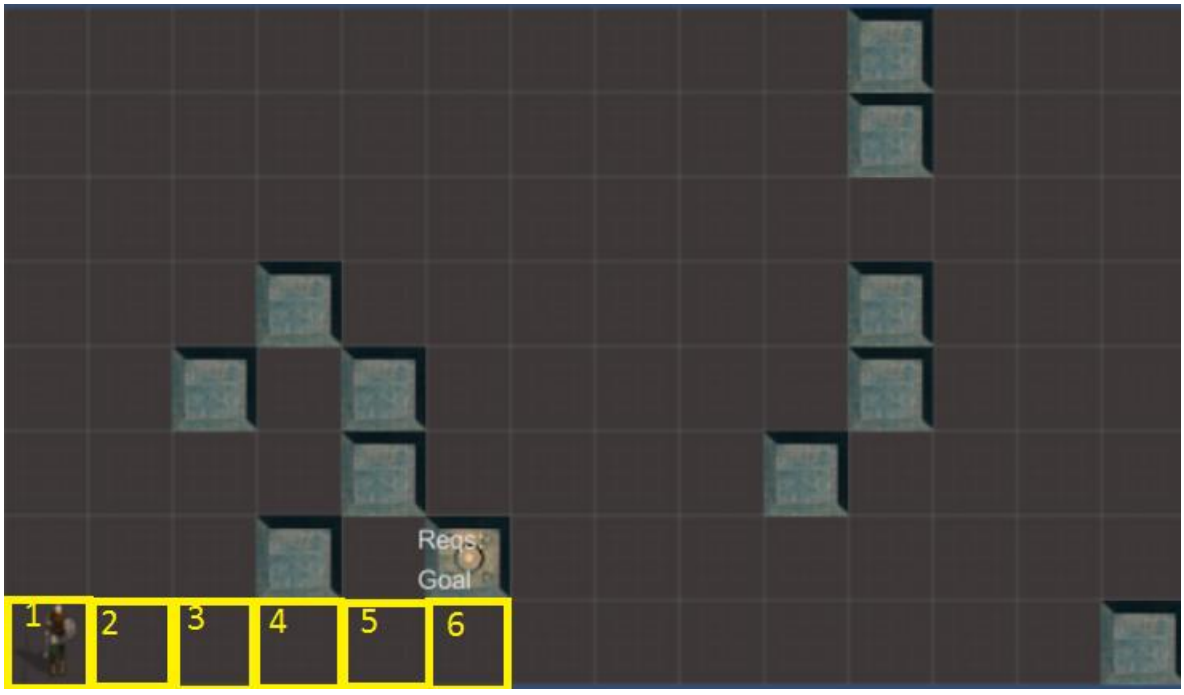
CAMINO:



SEMILLA 476

Mediante el algoritmo de búsqueda en amplitud, en la semilla 476 se han expandido un total de **6** nodos. El orden de expansión está reflejado en la imagen de abajo.

EXPANSIÓN DE NODOS:



CAMINO:

Podemos llegar a la conclusión que, para este tipo de problemas, como anteriormente hemos indicado, se expanden menos nodos en profundidad. También se aprecia que cuanto mayor sea la distancia a GOAL mayor será la diferencia entre los 3 algoritmos, especialmente entre amplitud y A*, ya que gracias a la heurística se evitan caminos potencialmente innecesarios.

BÚSQUEDA ONLINE

Es también llamada búsqueda en línea y a diferencia de las búsquedas offline, esta observa el entorno y hace el cálculo de la acción que debe realizar.
Este tipo de búsquedas es muy utilizado cuando el agente se encuentra en ambientes desconocidos.
Después de cada acción, un agente online recibe una percepción al decirle que estado ha alcanzado; gracias a esta información, el agente decide que acción realizar.

ALGORITMO DE BÚSQUEDA CON HORIZONTE

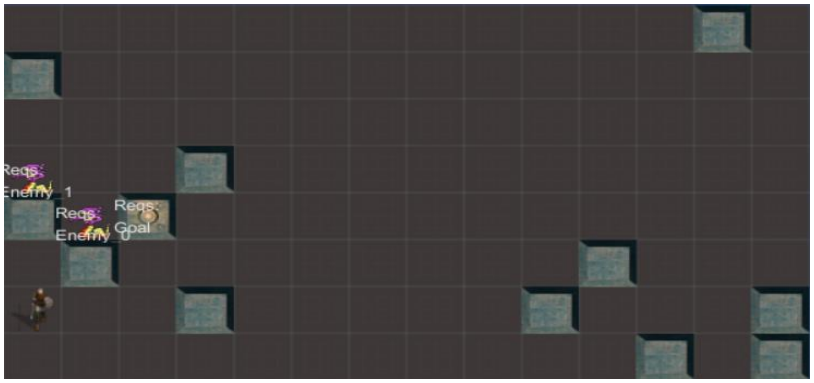
Este algoritmo de búsqueda es similar al A* pero a menor escala, ya que tiene un horizonte o límite, es decir, explora las capas del árbol hasta un límite y devuelve la mejor solución de la última capa sin necesidad de que esta sea el final o resultado esperado. Para ello, igual que el A* emplea el coste para evitar la reordenación y la poda de hojas en el proceso, así, si llega a una hoja y esta tiene menor coste que cualquiera de los nodos abiertos anteriormente, este se queda directamente con ella. Cuando sigue expandiendo y encuentra que, por ese camino, alpha será menor, directamente no lo expande, optimizando así el tiempo del algoritmo.

En nuestro algoritmo hemos decidido que vaya siempre por el mismo enemigo, ya que pensamos que, si tiene que estar cambiando entre diferentes enemigos por distancia se puede dar la casualidad que vaya a por uno, y luego cambie, haciendo que esto sea menos eficiente. El orden de los enemigos es proporcionado por **BoardInfo.Enemies**. De esta manera nuestro robot siempre va a por un objetivo fijo, y una vez que lo alcance irá a por el siguiente. Cuando no queden enemigos en el mapa, se dirigirá a la salida si es posible.

También se ha implementado una opción llamada **Exit Loops.** Mientras se hacían pruebas se ha observado que, si el enemigo se queda de tras de un muro de gran longitud, el robot será incapaz de tomar un camino debido a que el horizonte no es lo suficientemente grande. Según la variable **Max Loop Movements** establecemos el número de movimientos que consideramos bucle. El bucle lo hemos establecido de la manera de que si en 20 movimientos, solo se han hecho 2 movimientos iguales (ejemplo arriba-abajo, derecha-izquierda), se considera que hemos entrado en un bucle, por lo que el horizonte se amplía en uno. Tras diferentes pruebas hemos encontrado que una vez alcanzado el horizonte necesario para traspasar el muro, nuestro robot sale sin problemas y alcanza al enemigo.

Debido a los cambios mencionados anteriormente, nuestro algoritmo funciona con cualquier valor de horizonte, siempre y cuando este no sea 1, ya que no tendría sentido.

También se ha decidido, de la misma manera que en los algoritmos Amplitud y A*, que, si el enemigo atraviesa un muro, durante el momento en el que no se puede alcanzar (no es Walkable), el personaje va a esperar.



Ejemplo de situación de bucle.

Para implementar este algoritmo hemos seguido el siguiente pseudocódigo:

Repetir Si estado actual != meta

abierta ← s_0

α ← +∞

Repetir

Si vacío? (abierta) **entonces**

devolver(negativo)

nodo ← primero(abierta)

sucesores ← expandirEnProfundidad(nodo)

Para cada n∈sucesoresHoja **hacer**

α ← min (α, f*(n)

Si meta (encontrado)

n* ← meta

Si no

n* ← nodo(f*(n) = α)

Fin {repetir}

devolver nodo n*

Fin {repetir}

BÚSQUEDA DE CAMINO:

En la imagen anterior se explica visualmente el funcionamiento de nuestro algoritmo con un horizonte de 3.

En el nodo 0 abrimos el 1 y el 2, con el nodo 1 abrimos el 3 y 4 el, y con el 2 no se llega a abrir el 5, ya que obtenemos un alpha mayor que el que obtenemos con 3. Observamos que el nodo 3 es el que se halla más próximo del enemigo, por lo que nos moveremos hacia la derecha.

Uno de los problemas que nos hemos encontrado a la hora de implementar este algoritmo fue el hacer que funcionase con un horizonte variable. Al final acabamos dándonos cuenta que esto se solucionaba siempre de la misma manera, que es volver por los padres hasta el primer hijo del Nodo resultado. Lo que hicimos fue un bucle recursivo, de manera que subiese por el árbol hasta dicha condición.

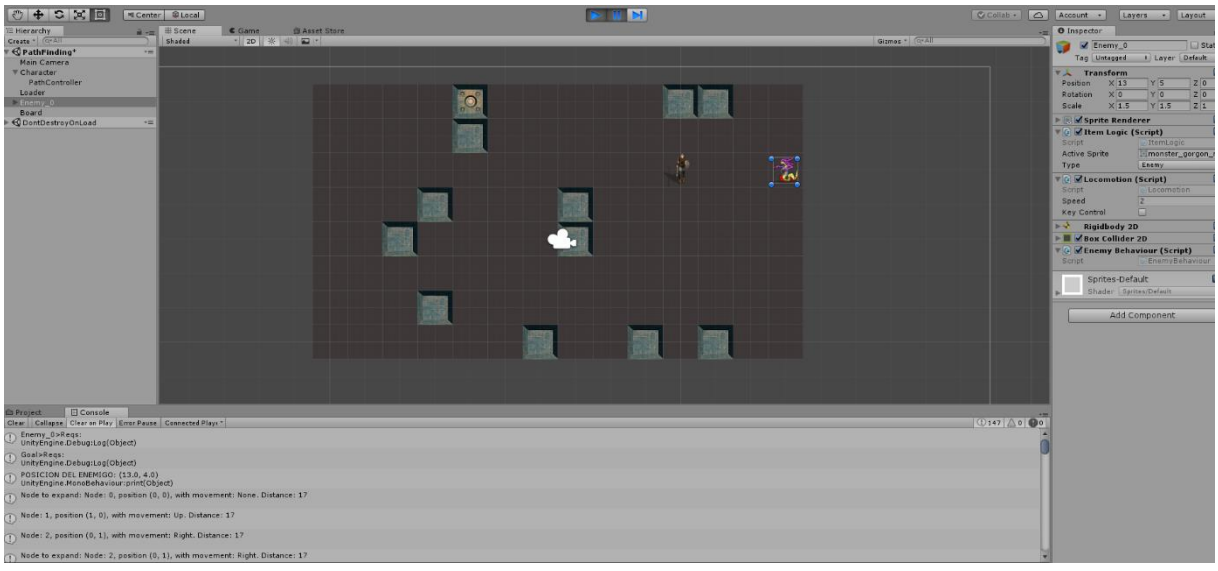
ENEMIGOS CERCANOS

Si encontramos al enemigo mientras expandimos cesaremos la expansión e iremos directos a por el mismo.

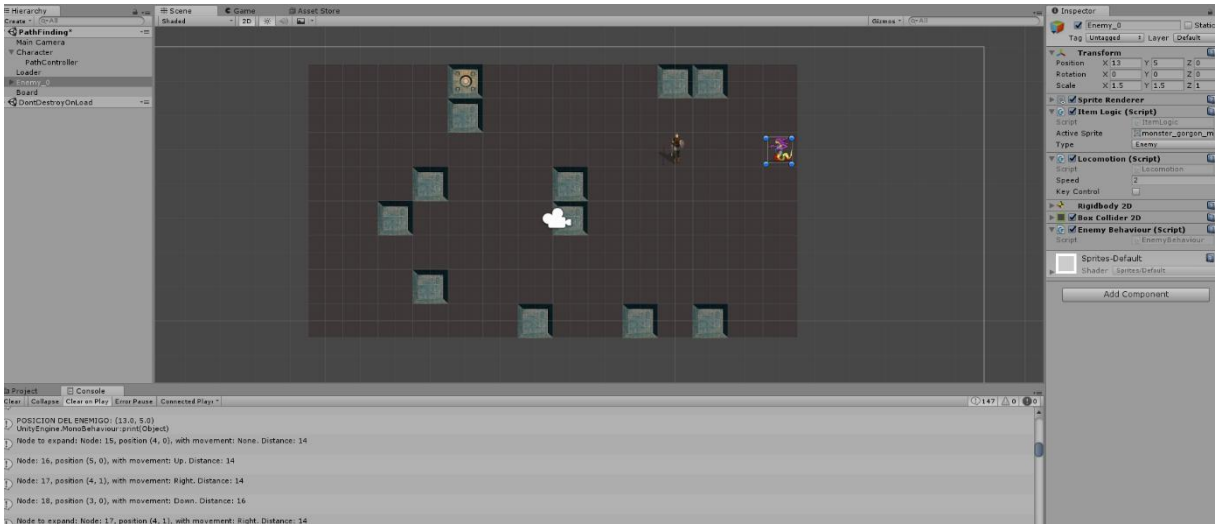
Al encontrar el enemigo en el nodo 5 no abriremos nuevos nodos y nos moveremos inmediatamente hacia el nodo 1.

MENCIÓN DE ERRORES DE CÓDIGO FACILITADO

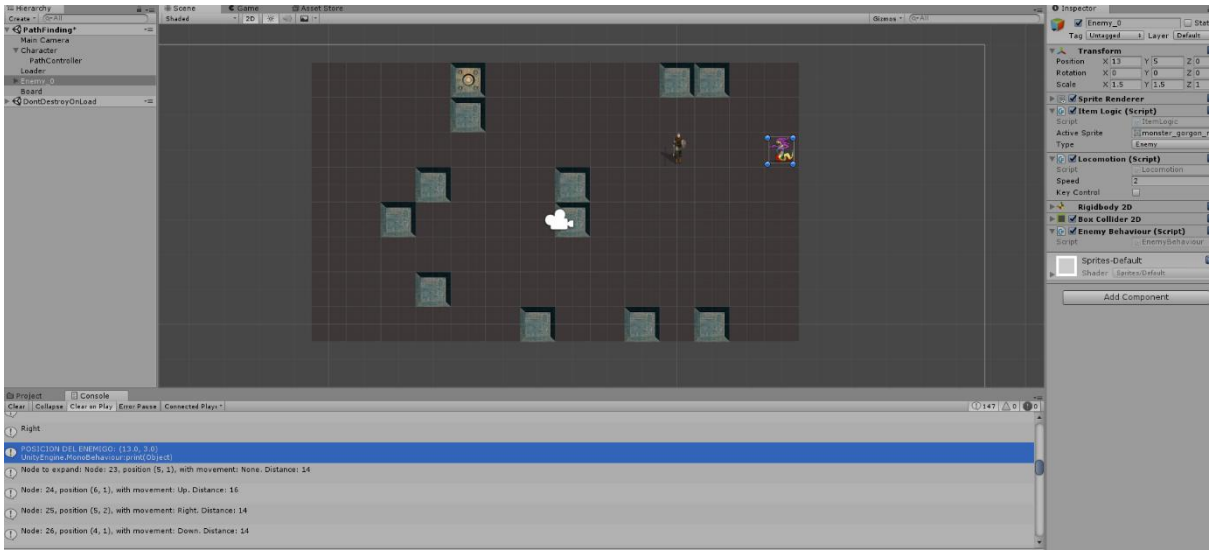
Cuando estábamos realizando el algoritmo de búsqueda por horizonte nos dimos cuenta de que, por alguna extraña razón, cuando el personaje se encontraba directamente pegado al enemigo, no iba hacia él. Después de probar a cambiar nuestro código, ya que pensábamos que se debía a algún error, decidimos imprimir la posición del enemigo. Resulta que ésta no correspondía con la del enemigo. Después de barajar algunas posibilidades, decidimos quitar el movimiento a los enemigos modificando el script de RandomMind para que siempre devolviese un movimiento None. Este fue el resultado:



Como se puede apreciar al inicio del Script, nos dice que la posición del enemigo es (13, 4), lo cual es falso, ya que la posición del enemigo es (13, 5). Pero de alguna manera esto cambia:



Ahora resulta que la posición del enemigo es la correcta (13,5). No entendemos como esto es posible a pesar de revisar todos los scripts, ya que en ningún momento ni nosotros ni otro script (que hayamos visto) está modificando la posición del enemigo. Sin embargo, pasado un rato la posición vuelve a cambiar a (13,3), de nuevo errónea.



Nuestro personaje en este caso alcanza perfectamente el enemigo, mientras que otras veces va hacia la posición que nos indica **BoardInfo.Enemy.** Algunas veces la posición no coincide con el Sprite, por lo que el Sprite desaparece, aunque el nuestro robot no haya colisionado con el enemigo.