Práctica 2 IA : Agentes reactivos y deliberativos

Raúl Castro Moreno
Grupo B2

Nivel 1-B: Búsqueda en Anchura.

Empezando la práctica nos encontramos que se pide realizar la búsqueda en anchura.

La gran ventaja a la hora de pensar este método es que se nos había proporcionado previamente el algoritmo de búsqueda en profundidad, donde a la hora de la implementación, es muy sencillo de cambiar de una a otra ya que solo hay que cambiar la estructura de datos en la cuál se van almacenando y borrando una vez utilizados, los nodos no visitados, es decir, se cambia la estructura de datos de los nodos abiertos.

Para la lista de **nodos cerrados(visitados)** donde más que los nodos, lo que almacenamos son los estados de los nodos, vemos este metodo (y en todos los demás como se verá más adelante) que usamos como estructura de datos un **SET** el cual lleva asociado un functor para ordenar los estados de los nodos según el criterio que hemos escogido.

Y ahora lo importante, y lo que nos hace obtener la búsqueda en anchura, para la lista de **nodos abiertos (no visitados)** usamos como estructura de datos una **COLA (QUEUE)** , diferente al de profundidad que usaba una **PILA(STACK).** Con este cambio conseguimos ir recorriendo la lista de nodos abiertos en el orden por el cual recorremos por niveles el árbol de nodos. Esto es porque añadimos los nuevos nodos al final de la cola, y el siguiente que miramos es el primero que había entrado, es decir , el que más tiempo lleva en la cola. Se diferencia con profundidad al que al usar una pila, el siguiente nodo que cojemos es el ultimo que se a añadido a la pila.

La búsqueda en anchura es **más eficiente** que la de profundidad, pero hay casos en los que puede ser que profundidad sea más rápido que anchura. Por ejemplo, si el nodo que se busca esta en niveles muy inferiores, y da la casualidad de que profundidad lo encuentra de los primeros.

Por último añadir, que con este método siempre encontramos el camino **más corto** hacia el objetivo , pero **no el más óptimo en cuanto a gasto de batería**.

Nivel 1-C: Búsqueda con Costo Uniforme.

Para este método, vamos a reutilizar el método de la búsqueda en anchura, ya que nos proporciona en el camino más corto pero ahora vamos a conseguir que sea el más corto con menor coste.

Para ello tenemos que cambiar varias cosas, ya que aquí entran el cálculo de los costes y con ello, el que se reduzcan algunos de esos costes al llevar equipados, tanto las **zapatillas** como el **bikini.**

Primero vamos a crear un nuevo nodo en el cual, aparte del **ESTADO** y de la secuencia, le añadimos una variable que guarde el coste de ir a ese nodo.

También vamos a añadir a la estructura del **ESTADO** dos nuevas variables booleanas llamadas **zapatillas** y **bikini,** las cuales se inicializan a **false** y se ponen a **true** cuando pasamos por encima de la casilla la cual nos equipe con ello.

Continuamos creando un nuevo functor para **SET** de los estados de los nodos cerrados, donde usamos lo que teníamos en el de la búsqueda por anchura, y además le añadimos que los compare por si tienen zapatillas y en caso de tenerla los 2 estados, compare por si tienen bikini.

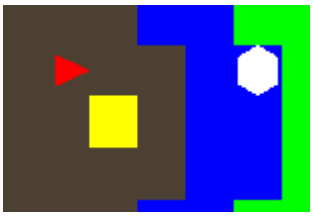
Ahora vienen los 2 pasos más importantes:
En primer lugar tenemos que crear un método para que nos devuelva el coste de cada movimiento, para ello usamos la siguiente tabla.

Tipo de Casilla	Gasto Normal Batería	Gasto Reducido Batería
'A'	100	10 (con Bikini)
'B'	50	5 (con Zapatillas)
'T'	2	2
Resto de Casillas	1	1

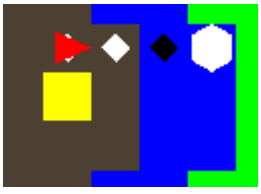
Siendo ‘A’ agua, ‘B’ bosque y ‘T’ suelo arenoso.

El siguiente paso importante, es a la hora de usar la estructura de datos para los nodos abiertos, ya que ahora nos interesa que el siguiente nodo a escoger, sea el que menos coste conlleve. Por lo que vamos a usar una **COLA DE PRIORIDAD(PRIORITY QUEUE)** , para la cual tenemos que definir un functor que los ordene según el coste, teniendo así un functor para cada una de las listas. **(“ComparaNodos” y “ComparaNodosCoste”).**

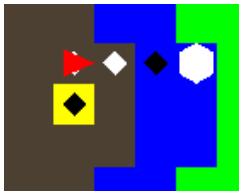
A continuación vamos a ver la diferencia entre anchura y coste uniforme de forma más clara con ejecuciones de la interfaz gráfica. Queremos llegar al siguiente objetivo:



Plan con anchura



Plan con Coste Uniforme

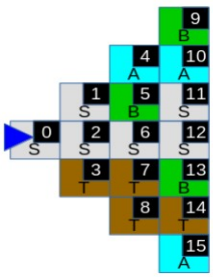


Se observa que anchura va por el camino más corto, mientras que coste uniforme, se gira primero a por el bikini para que le cueste menos cruzar el agua.

Nivel 2: Cambios en el método think.

Para el nivel 2, he utilizado el método de **Búsqueda por Costo Uniforme** como método para planificar el camino hacia el objetivo, aunque le he aplicado unos leves cambios que explicaré más adelante.

Lo primero que nos encontramos en el nivel 2, es que vamos a ciegas, es decir, no sabemos el mapa, por lo que el método de búsqueda no funcionara correctamente si no sabe por que terreno esta planificando el camino. Para ello creamos un método (**“Pintar”**), al cual pasandole el mapaResultado y los sensores, podemos ir pintando en el mapa las casillas que detecta el personaje. Esto es gracias a que tiene los sensores de terreno en esta disposición:



Gracias a esos sensores, podemos rellenar la variable de mapaResultado de 16 en 16.

Una vez pintamos el mapa, tenemos que empezar a recalcular planes, ya que puede que cuando se calculo al principio sin saber el mapa, pusiera que el camino fuera por encima de arboles o agua, que le hagan tener un coste de bateria mucho más alto que otro posible camino.

Para ello en mi código, he tenido en cuenta a la hora de recalcular, **si la casilla de enfrente es un arbol o es agua y el siguiente paso es hacia esa casilla**, que recalcule el plan. Ademas le he añadido que lo haga **solo cuando no tenga las zapatillas y o el bikini respectivamente**, ya que estaria recalculando cada vez que se encuentre uno, pero por ejemplo, con el mapa casi entero descubierto, los caminos que se van a obtener van a ser con coste uniforme y probablemente les haga pasar por arboles o agua, entonces sabiendo que ese es el camino óptimo no tiene sentido estar recalculandolo cada vez que pasemos por ahi.

Para saber la si la casilla siguiente a la que voy esta en el camino, he creado un método llamado **“siguientePaso”.**

También he tenido en cuenta, que si la casilla que tengo enfrente es un muro **‘M’** o un precipicio **‘P’** y la siguiente casilla del camino va hacia esa casilla, pues que se recalcule el plan, cosa que también he puesto que se realize el **sensor de colsiones** esta en **true**.

Otra cosa que ocurría es que si se “chocaba” con un aldeano ‘a’ , se consumian pasos del plan sin que este avanzara, por lo que he implementado que si la casilla de enfrente tiene en la superficie un aldeano, y la casilla siguiente de mi camino esta donde el aldeano, que la siguiente accion sea **“actIDLE”,** donde el personaje se queda quieto esperando a que el aldeano se quite, pero sin consumir pasos del plan.

A continución, me doy cuenta de que una vez llega a un objetivo, sale el siguiente, pero el personaje se queda quieto en el objetivo, y no se planifica nada. Para ello he creado **“copiaDestino”** que es un **struct de estado**. Esto lo hago para poder hacer una condición en la que poner a **false** la variable de **hayplan** para que replanifique el nuevo camino hacia el siguiente objetivo. **“copiaDestino”** se inicializa con los valores del struct de **“destino”,** la diferencia entre los 2 es que **“copiaDestino”** solo se actualiza al nuevo destino cuando los valores de este no coinciden alguno con los valores del siguiente destino mientras que **“destino”** se actualiza cada vez que usamos el método think.

He añadido también que se actualize el estado actual, en caso de pasar por encima de la casilla de **zapatillas ‘D’** o de **bikini ‘K’ ,** poniendo a **true** los valores booleanos del estado tanto de zapatillas o bikini según sea la casilla.

Finalmente, he añadido todo lo relativo a **la casilla de recarga ‘X’**, para que guarde la casilla, para que vaya hacia ella con un plan y para que se quede en ella recargando.
Para ello:

1º- Implementación guardar coordenadas Casilla Recarga

En primer lugar he creado un booleano inicializado a **false** llamado **“recargaEnc”** el cual me permite usarlo como condición para saber si se ha encontrado la casilla de recarga. Entonces, cada vez que entra al think, si no la ha encontrado, con un pequeño bucle de 16 iteraciones compruebo si alguna de la casilla del sensor es la de recarga. Si no está, no ocurre nada, pero en caso de ser alguna la casillad de recarga, pongo **“recargaEnc”** a **true**.

Al tener esa variable en **true ,**me meto en un bucle con un bucle anidado el cual recorre el mapa entero buscando la fila y columna de la casilla recarga, las cuales asigno a **“filRec”** y **“colRec”** respectivamente, siendo estas dos variables enteras declaradas antes, guardando así las coordenadas de la casilla recarga.

2º- Implementación planificar camino hacia Casilla Recarga

Creamos una variable booleana llamada **“irRec”** inicializado a **false.** Después de esto, nos vamos a la parte del código del método think, dónde haciamos mediante una condición, que cambiara los valores de **“copiaDestino”** y pusiera **“hayplan”** a **false**.

Justo después de esas sentencias y dentro de la misma condición, vamos a poner otra en la cuál vamos a poner un **umbral de batería, otro de tiempo**, y el booleano de **“recargaEnc”**. En mi caso he puesto que cuando **la batería sea menor de 250** y el **tiempo sea mayor que 200**(el tiempo que me refiero aquí, es el que nos proporciona el sensor de vida), con el booleano de la **casilla recarga encontrada en true**, copiamos en el struct **“destino”** , en su fila y columna, los valores de **“filRec”** y **“colRec”** respectivamente indicando que el siguiente destino es la casilla recarga, ponemos **“irRec”** a true, **“hayplan”** lo ponemos a false para que recalcule y **“contRecarga”** lo ponemos a 0(explico en el siguiente paso que es).

Finalmente, establecemos que mientras **“irRec”** esté en **true**, no se pueda actualizar el destino a los que dicen los sensores.

3º- <u>Implementación para recargar en Casilla Recarga</u>

En primer lugar he creado 2 contadores inicializados a 0 llamados **“contRecarga” y “contPaso”** , los cuales voy a usar para que cuando esté el personaje en la casilla de recarga, se mantenga en ella un número de iteraciones. Creo 2 porque he estipulado 2 momentos de recarga: cuando pasa por encima porque está dentro del camino, y cuando mi destino es la casillade recarga.

Para el primer caso, cuando **“hayplan”** este a **true** es decir, haya un camino, y pase por encima de la casilla recarga, hago 10 iteraciones quieto en la recarga, recargando 100 de batería y se controla con **“contPaso”** la cual se reinicia a 0 cuando ya no esta el personaje en la casilla de recarga,

Para el segundo caso, que es cuando la casilla recarga es el destino, recargo el mismo numero de iteraciones que tamaño sea el mapa, es decir, en un mapa de tamaño 30, hago 30 iteraciones de recarga , recargando así 300 de batería. Esto lo he hecho así para que sea proporcional al tamaño del mapa la recarga que hago. A parte de lo anterior mencionado, tambien se pone **“irRec”** a **false**, puesto que ya no voy a recargar, permitiendo así que el destino se actualize al de los sensores, y **“hayplan”** a **false** para que recalcule el camino al nuevo destino, continuando por ese camino una vez termina de recargar.

Todo lo anterior mencionado han sido los cambios que he implementado en el método think.

Añadir un último cambio que he implementado, y es en el método del coste, donde he considerado que el coste de pasar por una casilla recarga es 0 y que pasar por un precipicio es 150 siendo super caro, para que nunca planifique por ahí.

A continuación pongo una tabla con los objetivos que he obtenido en algunos mapas usando la plantilla de comprobación (que especifica la semilla,coordenadas y orientacion) que se ha subido a PRADO para el nivel 2.

Mapas	**Número de Objetivos**
mapa30.map	112
mapa50.map	67
mapa75.map	37

Decir que no he ejecutado los de 100, porque aún sin interfaz gráfica, es decir, usando BelkanSG, me tarda muchísimo, y no se si es por usar Windows , por la eficiencia del coste uniforme.

FIN